GENETIC ALGORITHMS
and other “natural” optimization method to solve hard problems
— a tutorial review

Jarmo T. Alander
Department of Electrical Engineering and Energy Technology
University of Vaasa, P.O. Box 700 FIN-65101 Vaasa, Finland

e-mail: etunimi sukunimi miuku uwasas fi www: http://www.uva.fi/~j al
phone (2012-): +358-29-449 8270

DRAFT April 30, 2013

This report is updated every now and then. The most recent and complete version of this report is available at site http://www.uva.fi/ TAU/reports/report96-1/English.pdf. A somewhat similar and partly more complete tutorial in Finnish as file Finnish.pdf in the same directory.
Trademarks

Product and company names listed are trademarks or trade names of their respective companies.

Warning

While this report has been compiled with the utmost care, the author takes no responsibility for any errors, missing information, the contents or quality of the methods or references, nor for the usefulness and/or the consequences of their application. The fact that a method or reference is included in this publication does not imply a recommendation. The use of any of the methods is entirely at the user’s own responsibility.
# Contents

1 Introduction
   1.1 Biology = computing? ................................................. 1
   1.1.1 Evolution in economics ............................................. 2
   1.2 Biological evolution .................................................. 4
      1.2.1 Breeding .......................................................... 4
      1.2.2 Selection in nature ................................................ 5
      1.2.3 Dangerous as a tiger .............................................. 9
      1.2.4 Problems .......................................................... 10
      1.2.5 Genotype vs. phenotype ......................................... 10
      1.2.6 Diversity .......................................................... 12
      1.2.7 Game theory ....................................................... 13
      1.2.8 More mathematics ............................................... 13
      1.2.9 Origin of life ..................................................... 13
      1.2.10 Evolutionary models of computing .............................. 14

2 Proteins & DNA
   2.1 Structure of proteins .................................................. 19
      2.1.1 Amino acids ...................................................... 19
      2.1.2 Primary structure ................................................ 20
      2.1.3 Gene coding ...................................................... 21
      2.1.4 Secondary structure ............................................. 22
      2.1.5 Tertiary structure ............................................... 23
      2.1.6 Protein families ................................................ 25
   2.2 Protein folding problem ............................................... 25
      2.2.1 Thermodynamic hypothesis = energy minimum .................. 26
      2.2.2 Kinetic i.e. folding pathway hypothesis ....................... 27
      2.2.3 Folding in vivo and chaperones ................................ 27
      2.2.4 Misfolding ....................................................... 28
      2.2.5 Inverse protein folding problem ................................ 29

3 Origin of Life
   3.1 Related work .......................................................... 32
   3.2 Building blocks of life ............................................... 32
   3.3 Peptides ............................................................... 33
   3.4 Discussion ............................................................. 35
## CONTENTS

3.5 Peptides first .......................................................... 35
3.6 Molten globules ......................................................... 36
3.7 Optimality of proteins .................................................. 36
3.8 Conclusions and future ............................................... 36

4 **Soft computing** .......................................................... 39
4.0.1 The immune system ................................................... 39
4.0.2 Neural systems ........................................................ 40
4.0.3 Quantum computing .................................................. 42

5 **Optimisation methods** .................................................. 45
5.1 Parallelism vs. Computing .............................................. 45
5.2 Search efficiency ........................................................ 45
5.2.1 Processing speed ....................................................... 46
5.2.2 Fitness landscape ....................................................... 46
5.2.3 Complexity theory ...................................................... 47
5.2.4 Phase transitions ...................................................... 48
5.3 Classification of optimisation algorithms ............................ 49
5.4 Some popular optimisation methods .................................. 51
5.4.1 Deterministic methods ................................................ 51
5.4.2 Interval methods in optimisation ................................... 57
5.4.3 Logic programming etc ................................................. 58
5.4.4 Stochastic methods .................................................... 58
5.4.5 Replica method ........................................................ 61
5.4.6 Renormalisation group method ..................................... 61
5.5 Quantum computing .................................................... 62
5.5.1 Solving SAT and TSP using QC ...................................... 63
5.6 Outline for a search algorithm ......................................... 65

6 **Genetic algorithms** ..................................................... 67
6.1 Basics of genetic algorithms .......................................... 67
6.1.1 Genetic operators ...................................................... 68
6.1.2 A more realistic example ............................................. 77
6.1.3 Comparisons .......................................................... 79
6.2 GA implementations ..................................................... 79
6.2.1 Distribution ............................................................ 80
6.3 Genetic programming .................................................... 80
6.4 Artificial life .............................................................. 80
6.5 GA literature ............................................................ 81

7 **Optimal GA** ............................................................... 85
7.1 Introduction ............................................................... 85
7.1.1 Related work .......................................................... 86
7.2 Definitions ............................................................... 86
7.3 Operations of GAs ....................................................... 87
7.3.1 Random generator ..................................................... 87
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.3.2</td>
<td>Crossover</td>
<td>88</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Multiparent breeding</td>
<td>88</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Mutations</td>
<td>88</td>
</tr>
<tr>
<td>7.3.5</td>
<td>Fitness</td>
<td>88</td>
</tr>
<tr>
<td>7.4</td>
<td>Program</td>
<td>89</td>
</tr>
<tr>
<td>7.5</td>
<td>Optimization</td>
<td>90</td>
</tr>
<tr>
<td>7.5.1</td>
<td>Test arrangement</td>
<td>90</td>
</tr>
<tr>
<td>7.5.2</td>
<td>Test problems</td>
<td>91</td>
</tr>
<tr>
<td>7.5.3</td>
<td>Test case</td>
<td>94</td>
</tr>
<tr>
<td>7.6</td>
<td>Observations</td>
<td>95</td>
</tr>
<tr>
<td>7.6.1</td>
<td>Random numbers</td>
<td>95</td>
</tr>
<tr>
<td>7.7</td>
<td>Conclusions</td>
<td>95</td>
</tr>
<tr>
<td>7.7.1</td>
<td>Population size</td>
<td>96</td>
</tr>
<tr>
<td>7.7.2</td>
<td>Initial population and mutations</td>
<td>97</td>
</tr>
<tr>
<td>7.7.3</td>
<td>Finetuned crossing</td>
<td>97</td>
</tr>
<tr>
<td>8</td>
<td>Analysing GA more</td>
<td>99</td>
</tr>
<tr>
<td>8.1</td>
<td>Background</td>
<td>99</td>
</tr>
<tr>
<td>8.1.1</td>
<td>Related work</td>
<td>100</td>
</tr>
<tr>
<td>8.2</td>
<td>Genetic algorithm</td>
<td>100</td>
</tr>
<tr>
<td>8.3</td>
<td>Theory</td>
<td>101</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Risk estimation</td>
<td>101</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Number of generations</td>
<td>102</td>
</tr>
<tr>
<td>8.4</td>
<td>Problem set</td>
<td>103</td>
</tr>
<tr>
<td>8.4.1</td>
<td>All are ones</td>
<td>104</td>
</tr>
<tr>
<td>8.4.2</td>
<td>Maximum sum</td>
<td>105</td>
</tr>
<tr>
<td>8.4.3</td>
<td>Tiling polyominoes</td>
<td>105</td>
</tr>
<tr>
<td>8.4.4</td>
<td>3-SAT problem</td>
<td>105</td>
</tr>
<tr>
<td>8.4.5</td>
<td>Folding snake</td>
<td>106</td>
</tr>
<tr>
<td>8.5</td>
<td>Experiments</td>
<td>107</td>
</tr>
<tr>
<td>8.5.1</td>
<td>Fitness distributions</td>
<td>107</td>
</tr>
<tr>
<td>8.5.2</td>
<td>Combinatorial explosion</td>
<td>108</td>
</tr>
<tr>
<td>8.5.3</td>
<td>Fitness landscape</td>
<td>109</td>
</tr>
<tr>
<td>8.5.4</td>
<td>Search speed</td>
<td>110</td>
</tr>
<tr>
<td>8.5.5</td>
<td>Population size</td>
<td>111</td>
</tr>
<tr>
<td>8.5.6</td>
<td>Search efficiency model</td>
<td>111</td>
</tr>
<tr>
<td>8.6</td>
<td>Discussion and recommendations</td>
<td>114</td>
</tr>
<tr>
<td>8.6.1</td>
<td>Comparisons to other results</td>
<td>114</td>
</tr>
<tr>
<td>8.6.2</td>
<td>Autocorrelation analysis</td>
<td>114</td>
</tr>
<tr>
<td>8.6.3</td>
<td>Sensitivity</td>
<td>116</td>
</tr>
<tr>
<td>8.6.4</td>
<td>Recommendations</td>
<td>116</td>
</tr>
<tr>
<td>8.7</td>
<td>Conclusions</td>
<td>117</td>
</tr>
</tbody>
</table>
## 9 Fitness function

9.1 Fitness landscape .................................. 119
9.2 SAT landscape .................................... 120
  9.2.1 Related work .................................. 120
  9.2.2 Criticism ...................................... 121
  9.2.3 Notations ...................................... 121
  9.2.4 SAT fitness function ......................... 121
  9.2.5 Deduction of subexpressions ................. 122
  9.2.6 Weighted fitness function ................. 124
9.3 The maximum satisfiability problem ............... 125
  9.3.1 Impulse response ............................. 126
  9.3.2 Exhaustive MAXSAT testing .................. 128

## 10 Applications

10.1 Most popular subjects ............................ 133
10.2 Finnish GA projects .............................. 135
10.3 Some application examples ....................... 136
10.4 Optics ........................................... 136
  10.4.1 Optical filter design ......................... 136
10.5 Telecommunications .............................. 138
  10.5.1 Ring loading problem ......................... 138
10.6 Layout design .................................... 140
  10.6.1 Cable harness routing design ............... 140
  10.6.2 Truss ......................................... 142

## 11 Discussion

11.1 Comparing GA and QC ............................ 145
  11.1.1 Automatic design of quantum circuits ....... 146
  11.1.2 Conclusions and future ...................... 146

## 12 Conclusions and future

12.1 Research ......................................... 147
12.2 Implementations .................................. 147
12.3 Applications ..................................... 147
12.4 Distribution ...................................... 147

Problems .............................................. 147

A Problems ............................................ 149
  A.1 Small problems .................................. 149
  A.2 Small projects .................................. 150

B Bibliography ........................................ 150

B.1 Glossary .......................................... 171
INDEX

173
List of Figures

1.1 Life on Mars? .................................................. 2
1.2 The annual number of GA papers. .......................... 3
1.3 Different varieties of ornamental pumpkins ................. 5
1.4 An Ichneumon sp. parasitoid wasp laying her egg. ......... 6
1.5 A fly or bumble bee? .......................................... 8
1.6 Original wasp brand or a pirat copy? ....................... 11
1.7 A poor man’s tropical rainforest? ............................ 13

2.1 A generic amino acid. ......................................... 20
2.2 Protein folding and inverse protein folding problems. .... 29

3.1 A hunting wasp ................................................ 34

4.1 Quantum computing analogy ................................. 43

5.1 An adaptive walk in a rugged landscape .................... 47
5.2 A search tree .................................................... 49
5.3 The basic classes of optimisation algorithms ............... 50
5.4 A resistor circuit with tolerances. .......................... 55
5.5 Refraction of light ray on a spherical surface .............. 57
5.6 Simulated annealing .......................................... 60
5.7 Renormalisation group method ............................... 61

6.1 A genetic algorithm. .......................................... 68
6.2 Crossover ....................................................... 69
6.3 A rusty leave of maidenhair tree (Ginkgo biloba L), ....... 70
6.4 Random number distribution for a linear congruence generator. 71
6.5 A LFSR to produce random bit sequence. .................... 71
6.6 Indirect chromosome coding .................................. 73
6.7 A toy genetic algorithm core ................................ 74
6.8 A network of four serial resistors. .......................... 76
6.9 A network of 16 resistors. .................................... 76
6.10 The evolution of population when searching the solution of a four resistor problem .................. 78
6.11 The number of function calls when solving the 16 resistor problem .................................. 79

7.1 fig:classes ...................................................... 87
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>fig:random</td>
<td>87</td>
</tr>
<tr>
<td>7.3</td>
<td>fig:crossover</td>
<td>88</td>
</tr>
<tr>
<td>7.4</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>7.5</td>
<td></td>
<td>92</td>
</tr>
<tr>
<td>7.6</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>7.7</td>
<td></td>
<td>93</td>
</tr>
<tr>
<td>8.1</td>
<td>The toy snake folding problem of 27 bricks:</td>
<td>107</td>
</tr>
<tr>
<td>8.2</td>
<td>Distribution of problem fitness values</td>
<td>108</td>
</tr>
<tr>
<td>8.3</td>
<td>(a) Correlation of fitness values. (b) A close-up view of (a) at short</td>
<td>110</td>
</tr>
<tr>
<td></td>
<td>correlation lengths.</td>
<td></td>
</tr>
<tr>
<td>8.4</td>
<td>Distribution of evaluations needed for finding the solution to the</td>
<td>111</td>
</tr>
<tr>
<td></td>
<td>polyomino layout problem</td>
<td></td>
</tr>
<tr>
<td>8.5</td>
<td>Distribution of evaluations needed for finding the solution to the</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>34 bits One-max problem</td>
<td></td>
</tr>
<tr>
<td>8.6</td>
<td>Distribution of evaluations needed for finding the solution to the</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>maximum sum problem</td>
<td></td>
</tr>
<tr>
<td>8.7</td>
<td>The average number of generations $n_G$ at different population sizes $n_P$</td>
<td>115</td>
</tr>
<tr>
<td></td>
<td>and test problems</td>
<td></td>
</tr>
<tr>
<td>9.1</td>
<td>The distribution of the unweighted and the weighted fitness function values</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>of the expression $E_1$</td>
<td></td>
</tr>
<tr>
<td>9.2</td>
<td>The distribution of the MAXSat fitness function of the expression $E_4$</td>
<td>128</td>
</tr>
<tr>
<td>9.3</td>
<td>The distribution of the maximum (●) and the minimum (◦) number of true</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>subexpressions of all two variable MAXSat problems. Figures give the number</td>
<td></td>
</tr>
<tr>
<td></td>
<td>of erroneous estimates</td>
<td></td>
</tr>
<tr>
<td>9.4</td>
<td>The distribution of the maximum (●) and the minimum (◦) number of true</td>
<td>131</td>
</tr>
<tr>
<td></td>
<td>subexpressions of all three variable MAXSat problems.</td>
<td></td>
</tr>
<tr>
<td>9.5</td>
<td>The range of the number of true subexpressions ($n_{true}$) of $n$ variable</td>
<td>132</td>
</tr>
<tr>
<td></td>
<td>MAXSat problem as the function of the number of subexpressions ($n_s$).</td>
<td></td>
</tr>
<tr>
<td>10.1</td>
<td>A woodland brown ($Lopinga achine$ Scopoli, 1763)</td>
<td>135</td>
</tr>
<tr>
<td>10.2</td>
<td>A dielectric thermal solar heat mirror.</td>
<td>137</td>
</tr>
<tr>
<td>10.3</td>
<td>A 8 node ring.</td>
<td>139</td>
</tr>
<tr>
<td>10.4</td>
<td>A shematic diagram of two different wireharness configurations satisfying the</td>
<td>141</td>
</tr>
<tr>
<td></td>
<td>same pin-to-pin wiring specifications.</td>
<td></td>
</tr>
<tr>
<td>10.5</td>
<td>Greenlee’s truss</td>
<td>143</td>
</tr>
<tr>
<td>10.6</td>
<td>Postiporstua truss</td>
<td>143</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Some proteins and their properties. ........................................... 18
4.1 Some figures, that describe genome information quantities. .............. 41
6.1 Main good and bad features of genetic algorithms. .......................... 67
6.2 Glossary of the key terms in GAs. ........................................... 72
6.3 The classes used in our examples. ............................................. 76
6.4 Special GA subbibliographies .................................................. 82
6.5 The geographical distribution of GA researchers according to Vaasa GA bibli-
ography database (22.10.2012). .................................................. 83
7.1 Test problems, ................................................................. 91
8.1 The parameters of the genetic algorithm used in the experiments. ........ 101
8.2 Population size estimates $n_P$ .............................................. 101
8.3 The number of samples $n_f$ needed on the average to exceed $x$, ........ 104
8.4 An example of a polyomino tiling (fitness $f = 12$) and the parameters of the
polyominos: ................................................................. 105
8.5 Approximate parameter values $n_G^\infty$, $n_P^\infty$ and $n_0$ ................. 113
8.6 The average number of generations $n_G$ needed for finding the solution .... 113
8.7 The the problems and the success rate $P$ at different population sizes $n_P$, .. 114
9.1 How to select a proper optimisation method for a given type of cost function. 119
9.2 The goal vector $V$ and the fitness functions of the expression $E_1$. ........ 123
9.3 The distribution of the fitness functions $H$ and $H_w$ ......................... 123
9.4 The MAXSAT fitness function $H_W$ and the number of true subexpressions $n^1$
of the expression $E_4$. .................................................... 127
9.5 The MAXSAT fitness function of the expression $E_4$ and the two best solutions. 127
9.6 The number of MAXSAT problems as the function of the total number of
variables $n$. ................................................................. 129
9.7 All the by the simple occurrence method unsolved two variable MAXSAT prob-
lems. ................................................................. 130
9.8 The results of the exhaustive MAXSAT estimation tests for 1, 2, and 3 variables130
9.9 The estimation error distribution of all two variable MAXSAT problems using
linear and exponential weights. ............................................. 130
9.10 The results of the exhaustive MAXSAT estimation tests for 1, 2, and 3 variables.130
## LIST OF TABLES

10.1 The most popular subjects of GA papers (22.10.2012). ........................................ 134
10.2 Greenlee’s truss .................................................. 142
10.3 Postiporstua truss .................................................. 142

11.1 Comparison of some properties of quantum computing (QC) and genetic algorithms (GA). .................................................. 145
Abstract

The apparent billion years triumph of biological evolution has inspired computer scientists to implement, analyze, and utilize similar methods to solve e.g. optimization and search problems that are already known or seem to be computationally hard and/or complex. This has led to the study of genetic algorithms (GA) and other evolutionary methods for global optimization and search to be applied in various branches of engineering and sciences. Perhaps the most attractive features of genetic algorithms are on one hand their simplicity, which makes it easy to implement and tailor them to practical problems and on the other hand their ability to solve hard problems. The other prominent feature is that GAs are general i.e. they are quite independent of the details of the object problems.

Most applications of genetic algorithms are related to more or less real-world problems of engineering, science and economics. Genetic algorithms are one type of the so-called soft computing methods, which include neural networks and fuzzy logic among other methods that more or less have their model in nature—organic or inorganic. In addition to combinatorial and numerical problems, GAs are able to handle also more complicated data structures, even programs, when we have a method called genetic programming. By using genetic programming, we can totally automatically solve problems by generating e.g. LISP functions.

This GA tutorial is based on the author’s research in this fascinating area in general and guest lectures at University of Jyväskylä, University of Kuopio, Helsinki University of Technology, University of Salzburg, and Tampere University of Technology in particular. The author has used this report also as the core material of GA tutorials at the 2nd and 3rd Nordic GA Workshops [Ala96, Ala97], the Finnish Artificial Intelligence Conference STeP-98, and ECCTD’01 conference Aug 28 2001 at Espoo. There are also much material based on the author’s published papers on genetic algorithms and especially their applications. And last but not least this tutorial contains a lot of references to recommended reading. A partly more complete tutorial in Finnish was published by Tekes [Ala98].

A brief list of topics covered by this tutorial (appr. half an hour lecture each):

- basics of Darwinian evolution
- optimisation methods in general
- fitness landscape
- basics of genetic algorithms: population, selection, crossover, and mutation
- an analysis and example of GA optimization
- a short review of typical applications
- an overview of GA literature and further reading
- a demonstration of GA optimization

The reader of this tutorial is assumed to have basic knowledge of the following subjects and skills: programming\(^1\), calculus, linear algebra\(^2\), probability calculus and basic genetics.

\(^1\)C++ and/or Java preferably

\(^2\)vectors and matrices
Preface

The current version of this report is still more like a short literature review containing a whole lot of references to GA papers from our GA literature database than a comfortable armchair tour tutorial or a cartoon slide show. That is why all comments and suggestions concerning this tutorial or GAs in general are more than welcome.

The author wants to thank all those who have been involved with this tutorial. The work was partly supported by Finnish Technology Development Centre (Tekes), for which the Finnish report [Ala98] was done.
Chapter 1

Introduction

In this tutorial report and overview, we are going to look how the basic principles of biological evolution can be applied in computing, to find optimal solutions to engineering and scientific problems and even find new technical contructions. The principles of biological evolution have been studied and applied to computing in tens of thousands of scientific papers [?]. Therefore we already have some idea when and how to use these methods called genetic algorithms, evolutionary optimisation, evolution strategies, etc to solve important technical, economic, and scientific problems. Evolution is not just a theory, it is a working algorithm.

In this introductory chapter we will briefly review some of the basics of biology and especially evolution in order to get some background for better understanding of the key concepts and operations used in evolutionary algorithms. We will encounter such problems as why wasps are black-yellow striped as do some other small creatures, why apples are so tasty, etc. It is even anticipated that the engineering success of evolutionary principles will in turn provide some inspiration for biological research.

In the second chapter we will briefly introduce the building blocks of biology, proteins, and how they are coded in DNA.

In chapter 3 and on we will finally go into the computational methods which are based on genetics and evolutionary models from nature.

1.1 Biology = computing?

It is easy and sometimes surprising to see that computing and biology or life have so much in common. First of all biological systems have memory as DNA, neural networks, and immune systems, which all contain vital information for organisms. Memory in DNA is surprisingly stable. Some parts of the genome, the most vital, may contain information that has remained nearly intact over millions of years of organic life on earth. Information is valuable and that is why there have evolved many ways to protect DNA against harmful noise i.e. mutations. This results in so permanent a memory that it easily outperforms modern computer memories that may last only for some years or decades but definitely not for millennia, not to speak about millions or even billions of years as some parts of DNA seem to have survived.

The second elementary information processing operation in computing is selection. It can be realised by a simple if-then-else construction in computer programs. In nature selection was the key operation, which was recognised by Charles Darwin in his famous work
"On the Origin of Species by Means of Natural Selection" [Dar79]. Selection is most natural operation in both computing and nature. It is easy to implement but hard to be analysed mathematically because of its nonlinearity. Not surprisingly selection, and nonlinearity, are also the key information processing operations that can be found in neural networks and immune systems, too.

The third elementary operation is iteration, which means that operation sequences are repeated over and over again, perhaps millions of times. In computing this is realised e.g. by such control structures as for or while -loops. In nature generations of organisms are following each other and myriads specimens of each species are living and interacting with each others and other organisms. The number of currently living species is so high, over millions, that it can only be estimated, the exact number is still unknown.

The above three things, memory, selection, and iteration are the three basic operations needed for any computing system realisations may it be created by an engineer or evolution. We will see that they are also the core operations of the genetic and evolutionary algorithms.

Problems
Is there any devices on Mars (fig. 1.1) that has been designed, or tested by using evolutionary algorithms?

1.1.1 Evolution in economics
Phenomena that are much like natural selection can be easily found in areas like economics, engineering, technology and product development. Best products are selling better than not
Figure 1.2: The annual number of GA papers (●). Observe that after year 1998 the collection is quite incomplete. From late 70’s to mid 80’s the growth was exponential with appr. 40%/year increase. Data taken from Vaasa GA bibliography. GA papers related to VLSI design and electronics shown by ◦, simulation by ‘S’ and Nordic papers by ‘N’.
so good ones so that finally the less good products tend to disappear one by one from the markets. New products are developed by applying and combining already known components and methods (memory) resulting often better products than ever existed before. In a way products are struggling for survival in the market much like species are struggling for survival in nature. In both cases struggling does not necessarily mean an easy to observe open arena fight but more likely a much more subtle, often indirect, rival for limited resources in a very complex ecological or economic network that can be observed only by statistical methods. Sometimes products are so successful that they increase their market share exponentially until the market is saturated or an even better product appears. Examples of exponential growth in technology need not to be searched far away: It can be seen even in the number of published GA papers (fig. 1.2).

**Problems**

Can you find saturation in the GA research in ‘research ecology’?

**References**

[Far93].

### 1.2 Biological evolution

Charles Darwin represented the theory of the origins of species by gradual natural selection over a century ago [Dar79]. Among other things, this theory explained the apparent relationship between different species and families, which were formalized by Swedish Carl Linné already a century before Darwin. Darwin’s theory raised much discussion immediately for and against. This discussion, in spite of the great progress e.g. in molecular biology, which has strongly supported the theory, has not totally ceased.

Darwin’s great contribution was that he was able to reveal the slow but inevitable evolution of species in biological, paleontological and especially in breeding evidence. In nature evolution is so slow, that we are not able to see it happen immediately. There might be some evidence of development that is detectable within a few generations [Wei95, Wei97]. These results are quite questionable, however.

#### 1.2.1 Breeding

In breeding animals and plants by man, the situation is quite different: Man has intentionally selected the best crop and species for breeding and the results have been quite astonishing. Let us only see how different looking are the different dog races. An imaginary (alien) spectator from outer space could easily be mislead to consider different dog races as totally different species. The natural variation of cats is much lower, correspondingly the differences between different cat races are less striking. Perhaps the most striking breeding examples are found among the cultivated plants like cabbages and pumkins. All the various cabbage (Brassica oleracea) types ranging from leavy cabbage to cauliflower and kohlrabi are actually breed from the same wild species during thousands of years.
1.2. Biological Evolution

1.2.2 Selection in nature

Darwin’s work claims that selection is the primary driving force of evolution. However, it is much more difficult to see evolution in action. The outcomes of the selectionist processes are primarily of statistical nature and can be seen only after a number of generations.

Body size is one of the primary traits that evolution is working on. Body size is also easy to measure and correlated with many physiological and fitness characters. Thus we could expect to see plenty of studies analysing selectionistic processes. However, the actual shortage of such studies reflects the difficulty of selection studies in biology [Bla00].

Viruses are the smallest and most primitive life forms and should thus provide some illustrative examples of evolutionary principles. The RNA viruses have smaller genetic memory than the DNA viruses therefore the latter (e.g. herpes) have better ability to evolve and cooperate with the host organisms than the previous often causing more serious illness (like HIV) of the host. Observe that a host soon death is not the best host for a virus struggling for survival. Thus selection favours the less virulent pathogens and parasites. For a comparison of simple and more complex viruses (RNA viruses vs. DNA viruses) from the view point of genetics and evolution potential see [CL01].

An example of a fruit having high diversity not only in gardens but also in nature is domestic apple (Malus pumila (Juniper and Mabberley 2006) formally known as Malus domestica). Its relatives in nature have only small and bitter tasting fruits, except this one species that still grows in the Tian Shan mountains of Western Asia. The fruits of the wild M. pumila are large and diverse, almost like those that you can find at the local market squares.
It has been proposed that this exceptional species was actually evolved due to the harvest of fruits by such large herbivores as bears and horses, which selected the biggest and most tasty, sugar rich, apples, the seeds of which they then naturally spread better than those of the more bitter specimens.

Figure 1.4: An *Ichneumon sp.* parasitoid wasp laying her egg by drilling a hole to a log by her inch long but hair thin ovipositor (in the image you can actually see the shelter of the ovipositor, whithin which the actual ovipositor is located). Her antennas are carefully locating the host caterpillar where the egg is laid. Her six reddish legs are firmly gripping the bark on the log. How the drilling actually happens is still somewhat unclear.

Was Darwin wrong?

By providing counter examples many famous scientists have claimed that Darwin’s evolution theory cannot be right. One such claim was presented by the famous French entomologist J. Henri Fabre, who studied the behaviour of several insect species by ingenious experiments. One such careful observation and experimentation revealed the unbelievable life style of a digger wasp (*Scolia maculata*). It finds its prey, a big caterpillar of a large beetle, deep underground. How can a winged bumble bee like wasp be evolved to behave like that. Fabre concluded that it is totally impossible, hence Darwin must be wrong. However, a simple reasoning reveals that this time Fabre might have been far too hasty in his conclusion. Namely, we can easily see how the evolution could have happened in this case: In the ancient prewaspian eon quite many caterpillars of various insects were simply crawling on the ground looking for all kinds of debris to eat without a worry of any wasps. When parasitoid wasps later evolved the caterpillars were really easy targets even for these newly evolved prewasps.
1.2. BIOLOGICAL EVOLUTION

That is why some of the host caterpillars evolved to crawl deeper under the debris, where they were in better shelter against not only wasps but also drying, UV, other predators like birds, etc dangers. However, some wasps also evolved to dig a little bit deeper. Thus finally only a few caterpillars were evolved to feed deep enough to escape from nearly all wasps so that only the most fittest digger wasps got deep enough to forage them. This was roughly how the digger wasps probably coevolved with their prey; little by little without any dramatics violating the principles of gradual evolution by selection. The main limit of the depth is likely the shortage of fresh enough edible underground debris than the limits of the evolutionary process itself. Observe, that model of coevolution has also been applied to evolutionary computing [?].

Another astonishing wasp way of living is given by the *Ichneumon sp.* parasitoid wasp that drills her hair thin ovipositor deep to a living hard tree to be able to lay eggs on her larval stage host (fig. 1.4). Obviously at the evolutionary beginning the wasp had only a very short ovipositor to pier the skin of their host only. So that a prey eating timber just under a thin bark, was already in quite a good safe from most, but not all, wasps. There are numerous such wasp species having only short ovipositors even today. Wasp species are having ovipositors of different lengths, as required by the site types of their different host species.

It is interesting to notice that there are also parasitoid flies, but they have to come along without any hard ovipositors. They just lay their egg on the host, which might get rid of it mechanically, while a wasp egg injected inside a host can only be eliminated by the host’s immune system [?, ?, ?, ?]. That is why those host caterpillars that live in good shelter like under tree bark does not have much to worry about parasitoid flies, unless there are flies that attack just when the prey is also laying her eggs. Some parasitoid flies have evolved to fly and lay eggs very fast... The diversity i.e. the number of parasite and parasitoid species and the variation of their biology is extremely rich. This is by no means limited to only wasp, flies, and other insects but apply to even plants.

Sexual selection

Another counter example claimed to show that the principle of the survival of the fittest does not always apply, is the ashtonishing ornamental displays that some birds, insects, even mammals, show. Perhaps the most famous example are the paradise birds, the males of which seem to be quite unfit to their environment. A biological explanation for this has been coined by the term *sexual selection*. It means that the females somehow like those ornaments and that the poor male can nothing but always evolve to be so ornamental that it has finally difficult to survive at all. When we know the key role of selection in nature, artificial breeding, and also as a factor greatly influencing the efficiency of evolutionary algorithms, could it be that the Mother nature relies just on pure random selection in the evolutionary process. Could it be that in the evolutionary process by sexual selection, it is actually the Mother nature i.e. selection that is selecting the best males for the females and not so much the other way round. Namely, in order to be very ornamental the ability of the male to survive, i.e. its fitness, must be high, higher than that for a less ornamental male having easier living. Obviously the males of paradise birds and similar species must have very good genes and that is also beneficial for the offspring of the female. In this way the evolution process is actually influencing, i.e. amplifying, itself. In evolutionary algorithms this kind of selection
CHAPTER 1. INTRODUCTION

Figure 1.5: a) A male hover fly (Volucella pellucens) feeding on a flower of thistle (Cirsium arvense). The male of this species can often be seen hovering on sunny places in gardens and forests. Observe his extremely large compound eyes. The offspring of this species grow up in a (Vespula sp.) social wasp (b) nest. Observe that the fly and its host does not much resemble each other. However, the fly usually have no problems to enter the wasp nest, which usually is not an easy task for an insect or even a larger animal. Remember that these wasps actually forage insects for their offspring. It seems that it is not the visual appearance of the fly that fools the wasps usually rigorously protecting their nest. Indeed, visually this hover fly species does not closely remind any other common hymenopteran species. See text for a possible explanation of the coloring.
that takes the very best individuals is called elitism. It is known to work well as we will see later. We will also see how to use evolution to find faster evolution.

Another example of sexual selection given by insects that can be found in a typical backyard, namely the hover flies (fly (Diptera) family Syrphidae, in America known as flower flies, as also in Finnish ‘kukkakärpänen’, which literally means exactly flower fly): The male hover flies of some species (like that shown in fig. 1.5) like to hover in places where they can be easily seen, such as on open sunny areas of paths within a dense forest, garden, or even a backyard. It seems that the male must be in a quite good condition, i.e. fit, e.g. to avoid predation by birds when waiting the female to breed. In the case of hover flies the ornaments also mimic wasps, bees, and bumble bees, which might give them some protection from predators. However, no hovering behaviour is found among the hymenopteran species or other insects, except for dragon flies, so that the predator might easily tell the hover flies from all other non-dipteran insects. Actually the hovering might be a way to stabilize the visual field by keeping the head in a fixed position and direction in order to easier recognize movements (of predators) in the visual field. It might not be a mere coincidence that the eyes of male hover flies are usually larger than those of the females (fig. 1.5). There are also observations that birds might not be fooled by the hymenopteran like appearance of hover flies [?]. Thanks to its striking ornamental appearance and display way of attracting breeding females—and predators—the male hover fly must be good in observing and escaping predators, both good features for survival in general i.e. for high fitness. Selection has done its job well and many hover fly species are quite abundant and regular visitors of flowers in meadows and gardens. A more ordinary looking and better hiding greish brown fly might be much less fit due to the lesser selection pressure it encounters.

1.2.3 Dangerous as a tiger

In the above section we discussed the reasons why flies can look like wasps. But how in the first place wasps invented their yellow-black striped brand? This must have happened by the gradual evolution, but how. Yellow and black stripes means some kind of danger, an animal like a tiger, an insect like a wasp, and so on. However, a tiger might not have any need to advertise her dangerousness. The reason for the striped appearance of a tiger is more likely to hide a tiger in a striped environment. The flowering plants and insects have coevolved. Let us think a situation where a prewasp is collecting nectar and pollen on a preflower: a black or dark colored insect on a shining yellow flower. What a contrast to be recognised by also those who are interested in having an insect meal. Hence a flower giving energy and nourishment in return to the transport of pollen is at the same time also a dangerous place because here the pollen collecting insects can be most probably and easily found by their predators. For an insect visiting a flower yellow coloration perhaps with some darker stripes is a much better fashion than a dark uniform color. And indeed, try to catch a house fly by your bare hands. That is not usually so easy. Then try to catch a bright colored hover fly on a flower. Perhaps after only a few failures you should be able to do that. However, be careful of not confusing a hover fly to a wasp. Is the hover fly thinking that he or she is fooling you by mimicking a wasp or relying on being unrecognisable from the flower. It might not be a mere coincidence that the wasp collecting pollen can be attracted by yellow objects. So actually the coloring of the hover fly might be more to hide it on a flower than to mimick wasps, which also have predators. Therefore the wasps have invested on hard skin protecting
them much better than the soft skin of flies. Only a few flies have hard skin, not any hover fly. Hence, it is a bonus to the fly but not to the wasp to have similar appearance. Now you might understand why the hover fly in fig. 1.5 does not so much resemble a wasp or bee, even if its close relatives do so very well. However, it still has the high contrast black and white pattern, which may make it less easy to be recognised as a tasty fly—but not a wasp.

1.2.4 Problems

1. Observe flowering medow and how birds hunt insects on flowers, or do they?

2. Think how the flowers have evolved to protect their pollen transporters, or haven’t they.

Ants

The evolution of social hymenopterans, ants, bees, and wasp was a hard puzzle for Darwin. How could it be that the worker ants (bees, or wasps) that are not reproducing have evolved at all? However, modern genetics has revealed that actually the worker ants are closer relatives to the offspring of the nest than the queen ant, who only lays the eggs. This is because of the peculiar genome system of not only ants but shared by other hymenopterans alike honey bees and other social wasps. Thus outsourcing of even reproduction is sometimes beneficial and found by the evolutionary process. We will briefly return to ants when showing how the idea of simulated foraging worker ants can be used to solve search and optimisation problems by the algorithms called ant systems.

Gradual evolution: wasp brand mimickry

Figure 1.6.a shows two flies, the other of which is more like a wasp than a typical fly. This is called mimickry—a textbook example of gradual evolution. A fly looking like a wasp may not be as tempting or easy a prey as the more typical looking flies.

Gradual evolution also applies e.g. on the evolution of eye from a very weak sensor of ambient light only into a most sensitive and delicate organ. Naturally the evolution of eye did not happen in a few step but may have taken at least some $10^5$ or so generations [NP94]. However, this time is so short that actually the eye has been evolved not only once but several times based on different histological processes. Compare e.g. the compound eye of insects (fig. 1.5) to the eye of vertebrates.

1.2.5 Genotype vs. phenotype

Not all diversity found in nature is due to genetical differences. Even identical twins are not exactly identical by their finger prints, iris patterns, blood vein architectures, etc., which can be used e.g. for indentification. This is because the genetic code does not give blueprints for the finer anatomical details but only rough quidelines how the organism grows—evolves from an embryo into an adult. Much of the details and even macroscopic features are influenced by the growing process and its interaction with the environment. The term genotype means the genetic information of the given organism, mainly DNA, while phenotype is used to mean the characteristics of the organism as a whole. A good example of a genotype vs. phenotype is given by the blood circulation system. Only the rough architecture of the blood circulation system is given by the genetic information. This must be so already because the genome does
Figure 1.6: Original wasp brand or a pirat copy? a) A wasp and a fly? Both insects are actually flies. The upper one is a hoverfly mimicking wasps. b) A beetle (Trichius fasciatus L.) mimicking bumblebees.
not contain enough explicit information for the details of the blood circulation system. So, how do the fine details of the system develop? In the case of the blood circulation system the basic principle is simple: the angiogenesis i.e. blood veins are grown where they are needed: in parts of the tissue needing more oxygen and other molecules transported by blood. I.e. the genetic code gives the control system by which the details of the blood circulation system develops during the growth of the animal. This also applies in such pathological situations as cancer tumor growth which also means angiogenesis i.e. growth of new blood veins to serve the needs of the growing tumor. Thus identical twins can be identified by e.g. their blood vein details, which are as unique as e.g. finger prints.

Observe, that the natural evolution is always working on phenotypes not directly on genotypes, which are only indirectly effected.

1.2.6 Diversity

There exist so many species that in spite of the work done by biologists to define and record every species, only rough estimates on the number of species can be given.

Even in the most simple biotype like sea water there exists a large number of species. This is called the plankton paradox [?]:

From Wikipedia: "In aquatic biology, the paradox of the plankton describes the situation where a limited range of resources (light, nutrients) supports a much wider range of planktonic organisms. The paradox results from the competitive exclusion principle (sometimes referred to as Gause’s Law), which suggests that when two species compete for the same resource, ultimately only one will persist and the other will be driven to extinction. Phytoplankton life is diverse at all phylogenetic levels despite the limited range of resources (e.g. light, nitrate, phosphate, silicic acid, iron) for which they compete amongst themselves."

Another diversity maintaining habitat is the Norwegian forest soil, the poor man’s tropical rainforest [?].

There are about 70,000 known species of fungi. Based on the rate of discovery of new fungi species, it can be estimated that the total number of fungi can be millions and that with the current identification speed it would take about 1000 years to discover all fungi [?].

It is illustrative to look what kind of factors are producing this high diversity also in the most simple habitats like sea water or soil. Tamás Czárán has used simple spatial rock-paper-scissors game to show how the arm’s race between the different microbials can do this [CHP02]. The idea is that each microbe has a set of antibiotics that can kill the microbials which are not resistant to those antibiotics. In order to produce an antibiotic a microbe has to invest more and also have the resistance for that antibiotic. It is therefore not reasonable to produce all kinds of antibiotics just in case but a minimum set that is enough for probable survival. It is actually the diversity and combinations of antibiotics that seems to be one simple factor explaining much of the biodiversity in such a simple habitat as the Norwegian forest soil.

An interesting question after the sexual selection hypothesis is the following. Might there be also some kind of sexual selection amongst microbes e.g. are there microbes that display clearly more than average number of resistances and antibiotics in the nature and not only in clean hospitals.

For more information on biodiversity see e.g. [CC99, ED04]. Already the cover of [CC99] is interesting: it seems to display a variety of sea shells—actually they all are the same species.

1The motivation behind this paragraph is that the author once claimed that the soil is an example of not so diverse habitat at all. Thanks to Timo ’the mythbuster’ Mantere who was opposing this claim, the author then checked the true situation, which was rather surprising also to the biologists.
1.2. BIOLOGICAL EVOLUTION

Figure 1.7: A poor man’s tropical rainforest? All the taller trees are of one species only, Norway bruce (*Picea abies* L.)! However, the soil under moss may contain a lot of different species. This habitat was located in Viinijärvi, Liperi, North Carelia, Finland.

1.2.7  Game theory

Game theory is studying games, how to play them successfully and how they behave in general as processes. Also evolutionary processes have been studied by game theoretical approaches. In the context of evolution and biology, an essential object of study is the interaction and selection caused by different species. An evolutionary stable strategy (ESS) is a strategy which, if adopted by a population of players in a given environment, cannot be invaded by any alternative strategy that is initially rare. An ESS is a refinement of the *Nash equilibrium* being stable in evolution: once it is fixed in a population, natural selection alone is sufficient to prevent alternative (mutant) strategies from invading. [Daw76] What applies to all simple mathematical models apply also to game theoretical and ESS approaches: they have their limits when analysing real situations, but in any case it gives understanding of the basic dynamics of interactions and behaviour of species. It can give answers to such problems as how there can coexist strategies that seem to be mutually excluding like why some frog males are noisy while some others are silent. Why the silent ones are not yet extinct? In a way the silent frogs are social parasites.

1.2.8  More mathematics

For a nice review of *mathematical genetics* see [Edw00].

1.2.9  Origin of life

Darwin was totally unaware of the biochemical basics of genetics. Now we know how the genetic inheritable information is coded in *DNA*, *RNA* and *proteins* and that the coding principles are actually digital much resembling the information storage in computers.
CHAPTER 1. INTRODUCTION

Building blocks of life

Here we give a brief introduction to DNA and especially proteins, which are coded by DNA and are the building blocks of every organism. A protein molecule is a chain of amino acids linked together by peptide bonds (see fig. 2.1) i.e. it is an organic heteropolymer. In total there are 20 different occurring amino acids (see appendix ??) in nature consisting of a common main chain part (backbone) and different side chains of proteins. The main chain consists of atoms N, C\(\alpha\), two O and two H atoms (fig. 2.1). The side chain is bound to the C\(\alpha\) atom. The two dihedral angles often denoted by \(\phi\) and \(\psi\) on either side of the C\(\alpha\) atom are the main degrees of freedom allowing the main chain to adopt different 3D conformations.

By structural criteria proteins can be divided into structural and globular ones. The role of globular proteins in cells is mainly to function as highly specific and efficient chemicals e.g. enzymes such as insulin [?], pheromones, hormones, bacteriocins (antibiotics), venoms, and antibodies of the immune system of vertebrates [Kle90]. About half of the proteins of Escherichia coli are thousands of enzymes, some 20 % form the ribosomes, while the rest about 30 % are used to construct the cell envelope [?]. The rate enhancements produced by enzymes can be extremely high, in excess of 10\(^{10}\) [?, ?].

According to Emil Fischer’s lock-and-key hypothesis enzyme specificity involves complementarity in shape and electric charge of the protein molecule [?]. The induced fit hypothesis assumes further some flexibility of the active site of the protein so that actually the substrate induces some conformation change to facilitate a “perfect” match.

Typically a globular protein consists of 1,000–20,000 atoms and has a diameter of about 35-100Å. This is comparable e.g. to the thickness of lipid bilayer and the diameter of tRNA [?]. One of the largest proteins is pyruvate dehydrogenase consisting of over 60,000 amino acids in 60 subunits (see table 2.1). In small proteins about half of the atoms are actually located at the surface while in larger proteins the number falls below 20 %.

The globular shape of globular proteins is due to hydrophobic interactions of the nonpolar atoms tendency to create a hydrophobic core covered by polar residues in contact with the solvent (water). The shape deviates usually considerably from the spherical, however. This indicates that there are forces opposing the hydrophobic effect [?] and the effect is not as strong as that observed in the interior of monomers [?].

General references on evolution:

References

[Dar79, Wri32, Fis58, Dun65, Rob70, Daw76, SS79, Fal81, Kim83, Gri85, Cro86, End86, Fut86, LSV86, Daw82, Daw86, Doe00, LST86, Far88, dJ88, Daw89, Gou89, Gou90, Gou91, PK91, Wei94]

1.2.10 Evolutionary models of computing

“Computer analysis of complete genomes of unicellular organisms shows that protein sequences are in general highly conserved in evolution, with at least 70% of them containing ancient conserved regions.”

Eugene V. Koonin, ??
1.2. BIOLOGICAL EVOLUTION

Biological information processing is in many ways totally different, however. The magnificent phenomenon called the evolution of species can also give some insight into information processing methods and optimisation in particular. According to Darwinism, inherited variation is characterised by the following properties [Sin98][p. 65]:

1. **Memory** Variation must be copying because selection does not create directly anything, but presupposes a large population to work on.

2. **Gradual evolution** Variation must be small-scaled in practise. Species do not appear suddenly.

3. **No direction** Variation is undirected. This is also known as the blind watch maker paradigm [Daw86].

While the natural sciences approach to evolution has for over a century been to analyse and study different aspects of evolution to find the underlying principles, the engineering sciences are happy to apply evolutionary principles, that have been heavily tested over billions of years, to attack the most complex technical problems, including also protein folding, which is a key phenomenon from the perspective of biology in general and evolution in particular.
CHAPTER 1. INTRODUCTION
Chapter 2

Proteins & DNA

“Virtually every property that characterizes a living organism is affected by proteins. Nucleic acids, also essential for life, encode genetic information — mostly specifications for the structures of proteins — and the expression of that information depends almost entirely on proteins (though some RNA molecules with catalytic activity have been discovered recently).”

Thomas E. Creighton, [?]

Here we give a brief introduction to DNA and especially proteins, which are coded by DNA and are the building blocks of every organism, from the smallest bacteria to the largest mammals.

A protein molecule is a chain of amino acids linked together by peptide bonds (see fig. 2.1) i.e. it is an organic heteropolymer. This was independently concluded in 1902 by Emil Fischer and Franz Hofmeister [?]. The name protein (from Latin proteios the first one) was given already in 1838 by Gerhard Johannes Müller, who was the first one to find a protein molecule [?][p. 98]. The name clearly reflects the primary importance of proteins in all living cells and thus for life in general. Already the smallest living organisms, parasitic respiratory system disease causing mycoplasmas, contain approximately 1,000 different proteins and the much studied Escherichia coli bacteria several thousands of proteins. [?] Over half of the dry mass of cells\(^1\) is composed of proteins [?, ?].

By structural criteria proteins can be divided into structural and globular ones. The former being collagen and keratin type construction materials of cells and further tissues. Structural proteins also include such substance as silk fibers produced by many insects especially moths and spiders [?]. Table 2.1 shows examples of typical proteins and some of their elementary properties.

The role of globular proteins in cells is mainly to function as highly specific and efficient chemicals e.g. enzymes [?], pheromones, hormones, bacteriocins (antibiotics), venoms, and antibodies of the immune system of vertebrates [Kle90]. About half of the proteins of Escherichia coli are thousands of enzymes, some 20 % form the ribosomes, while the rest about 30 % are used to construct the cell envelope [?]. The rate enhancements produced by enzymes can be extremely high, in excess of \(10^{10}\) [?, ?]. Many enzymes require additional

\(^1\) e.g. Escherichia coli
non-protein components, called cofactors, in order to function as catalysts. Cofactors may be other organic molecules or metal ions.

A special and important class is membrane proteins comprising about 40% of all known globular proteins. They locate themselves in cell membranes giving them special chemical properties.

According to Emil Fischer’s lock-and-key hypothesis enzyme specificity involves complementarity in shape and electric charge \[?\]. The induced fit hypothesis assumes further some flexibility of the active site so that actually the substrate induces some conformation change to facilitate a “perfect” match.

Several membrane proteins are related to the photosynthesis carried out in most plants, algae and various bacteria. The chain of reactions of photosynthesis starts in the chlorophyll pigment molecule which is bound to a membrane protein. The most abundant enzyme on Earth is RuBisCo (ribulose 1,5-bisphosphate carboxylase) another protein involved with photosyntheses by converting every year about 100Pg carbon dioxide into sugars while releasing oxygen as a byproduct vital for us more complex organisms. \[?, \]

Typically a globular protein consists of 1,000–20,000 atoms and has a diameter of about 35-100Å. This is comparable e.g. to the thickness of lipid bilayer and the diameter of tRNA \[?\]. One of the largest proteins is pyruvate dehydrogenase consisting of over 60,000 amino acids in 60 subunits (see table 2.1). In small proteins about half of the atoms are actually located at the surface while in larger proteins the number falls below 20%.

<table>
<thead>
<tr>
<th>abbr</th>
<th>mass</th>
<th>name</th>
<th>type</th>
<th>function</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFP</td>
<td>10-25 kDa</td>
<td>α-crystallin[?]</td>
<td>glob</td>
<td>major eye lens constituent</td>
</tr>
<tr>
<td>BR</td>
<td>248 res</td>
<td>antifreeze proteins[?]</td>
<td>helical</td>
<td>polar fish serum antifreeze</td>
</tr>
<tr>
<td>Cyt c</td>
<td>82-134 res</td>
<td>cytochrome c[?]</td>
<td>memb</td>
<td>light-driven proton pump</td>
</tr>
<tr>
<td></td>
<td>51 res</td>
<td>insulin</td>
<td>glob</td>
<td>respiratory enzyme</td>
</tr>
<tr>
<td></td>
<td>G-protein[?, ?]</td>
<td></td>
<td>glob</td>
<td>membrane signaling</td>
</tr>
<tr>
<td></td>
<td>Mb</td>
<td>myoglobin</td>
<td>glob</td>
<td>dioxygen binding</td>
</tr>
<tr>
<td></td>
<td>153 res</td>
<td>hemoglobin</td>
<td>glob</td>
<td>dioxygen binding</td>
</tr>
<tr>
<td></td>
<td>576 res</td>
<td>luciferase[?]</td>
<td>glob</td>
<td>bioluminescence</td>
</tr>
<tr>
<td></td>
<td>≈100 kDa</td>
<td>melittin</td>
<td>glob</td>
<td>amphophilic bee venom</td>
</tr>
<tr>
<td></td>
<td>≈500 res</td>
<td>luciferase[?]</td>
<td>memb</td>
<td>body of rhodopsin</td>
</tr>
<tr>
<td></td>
<td>≈28 kDa</td>
<td>opsin</td>
<td>glob</td>
<td>photosensoring</td>
</tr>
<tr>
<td>PYP</td>
<td>125 res</td>
<td>photoactive yellow protein[?]</td>
<td>glob</td>
<td>DNA repair / UV damage</td>
</tr>
<tr>
<td></td>
<td>PYP</td>
<td>photolyase[?, ?, ?]</td>
<td>glob</td>
<td>mad cow disease agent</td>
</tr>
<tr>
<td></td>
<td>PrP</td>
<td>prion[?]</td>
<td>α/β</td>
<td>multi-enzyme (60 chains)</td>
</tr>
<tr>
<td></td>
<td>27-30 kDa</td>
<td>pyruvate dehydrogenase[?]</td>
<td>glob</td>
<td>enzyme in photosynthesis</td>
</tr>
<tr>
<td></td>
<td>4.6 MDa</td>
<td>RuBisCo</td>
<td>memb</td>
<td></td>
</tr>
</tbody>
</table>

Table 2.1: Some proteins and their properties. For references see index (chapter ??) at the corresponding protein name.

\[2\] the abundance is partly due to the inefficiency of RuBisCo!
2.1. STRUCTURE OF PROTEINS

2.1 Structure of proteins

"The folded conformation of globular proteins is a state of matter peculiar in more than one respect. The density is that of a condensed phase (solid or liquid), and the relative position of the atoms is, on the average, fixed; these are the characteristics of the solid state. However, solids are either crystalline or amorphous, and proteins are neither…"

A. Hansen et al., [?] 

In total there are 20 different occurring amino acids (see appendix ??) in nature consisting of a common main chain part (backbone) and different side chains of proteins. The main chain consists of atoms N, Cα, two O and two H atoms (fig. 2.1). The side chain is bound to the Cα atom. The two dihedral angles often denoted by φ and ψ on either side of the Cα atom are the main degrees of freedom allowing the main chain to adopt different 3D conformations. It has been observed that the values of the dihedral angles are generally restricted to a few distinct domains in the φ – ψ space [?].

The globular shape of globular proteins is due to hydrophobic interactions of the nonpolar atoms tending to create a hydrophobic core covered by polar residues in contact with the solvent (water). The shape deviates usually considerably from the spherical, however. This indicates that there are forces opposing the hydrophobic effect [?] and the effect is not as strong as that observed in the interior of monomers [?].

K. M. Andersson and S. Hovmöller have studied the structure of 21 varying size3 and well defined4 protein structures, which contain in total over 100,000 atoms [?]. Their study indicated a mean density of proteins is $1.22 \pm 0.02$ g/cm$^3$. The precise measurement of protein density is difficult in practice because of the residual water and salt contained in protein crystals. After their statistical study the empirical formula, i.e. the relative proportion of atoms, of a protein is always very close to

$$C_{0.313n}H_{0.502n}N_{0.0856n}O_{0.0959n}S_{0.0032n},$$

where $n$ is the total number of atoms in the protein. We can say that half of the atoms of a protein are hydrogen, the mean content is namely 50.2% while the standard deviation is 0.8%, which even if it is the largest among all atom types, is not so high.

In addition to amino acids, many proteins contain inorganic ions, water or other inorganic or organic molecules. These may be intrinsic parts of the structures, like zinc ions in insulin, or substrates, inhibitors, effectors, or antigens in the case of immune system antibodies. [?]

The active site is usually a groove on the surface, but it can also be located surprisingly deep within the globule. It takes from fractions of seconds to minutes for a protein to fold from its unfolded conformation to the folded native conformation.

2.1.1 Amino acids

In table ?? (in the appendix) are shown the 20 amino acids found in natural proteins together with some of their basic properties. Altogether there are more than 150 different amino acids

---

310kDa - 380kDa  
42.8Å resolution or better
in cells, but only the shown 20 or some of their derivatives occur in proteins.

Several experiments have shown that among other molecules, several amino acids or their precursors are easily synthesized from N$_2$, CO$_2$ and H$_2$O vapor mixtures under conditions expected to have existed on Earth before organic life emerged [?].

Why in the first place are there 20 different amino acids occurring in natural proteins, why not more or less? A simple reasoning based on the principle of minimum set of design parameters is as follows. There are several parameters by which the different amino acids can be classified. The most important are the side chain volume $V$, hydrophobicity $h$ and charge $C$, which can be grouped as $V = \{\text{small, medium, high}\}$, $h = \{H,P\}$, and $C = \{-, 0, +\}$. This classification gives $VhC = 2 \times 3 \times 3 = 18 \approx 20$ amino acid parameter value combinations. Some experiments with reduced amino acid sets seem to indicate that at least $\alpha$-helices and $\beta$-sheets can be created using only a few amino acids (three for $\alpha$-helix and five for $\beta$-sheet) [?, ?, ?]. In the experiment by David S. Riddle et al the reduction did not much affect the folding time of the test protein (src SH3 domain) and was in one experiment even faster than that of the wild type [?]. For an analysis of the information contents of amino acids see [?].

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{generic_amino_acid.png}
\caption{A generic amino acid. R = side chain radical, bonds between residues (peptide bonds having torsion angle $\omega_i$ in proteins) are shown by dotted line (\cdots) and torsion (dihedral) angles $\phi_i$ and $\psi_i$ by $'$).}
\end{figure}

\subsection{Primary structure}

“The genetic code is an algorithm that relates triplets of nucleotides called codons — ATG, CGG, CAA, for example, — to specific amino acids that, in turn, are linked with one another by peptide bonds to make proteins. Because all of life depends on this algorithm, its chemical basis and that of protein synthesis can tell us something about how life arose on Earth.”

Paul Schimmel and Rebecca Alexander, [?]\

The amino acid sequence of a protein is known as its primary structure. It is uniquely determined by the genetic code: three consecutive nucleic acids (codon) of DNA determine one amino acid. The mapping is not one to one, but has some redundancy, because there are a total $2^6 = 64$ codes expressed by the codons and only 20 different amino acids. This
(uneven) redundancy has the potential to further play some role in biasing evolution of protein sequences [?]. For robustness and changeability of genetic codes, both a standard genetic code (SGC) and deviant codes exist in different cell organelles having their own DNA, but the difference between them is not great, however (see e.g. [?]). For codon usage statistics see [?] and for protein sequence statistics methods see e.g. [?].

The ends of the protein sequence are called the N terminus and C terminus according to the corresponding exposed main chain atom (nitrogen or carbon).

The synthesis of proteins happen in a cell organelle called a ribosome consisting of over 50 different proteins. Protein synthesis (PS) seems to be going on in cells all the time and at least for mammals it seems to be essential to sustain life. The details of the process are unfortunately not yet fully known and thus it is not possible to synthesize longer protein sequences (> 50 residues) without living cells.

Frederick Sanger was first to determine the primary structure of a protein in 1953. The protein was insulin, which is a really small enzyme molecule consisting of only 51 residues. [?]

### 2.1.3 Gene coding

While there are methods to reveal DNA sequences in “industrial” fashion (see e.g. [?]), there are much less and slower methods to define the 3D structure of native conformations of proteins.

The DNA sequences provide much information, also indirect, that can be used to deduce some features of the evolution of proteins. One important feature is the redundancy of genetic information in eukaryots. E.g. mRNA and DNA regulating proteins called histones have multiple (10-200) copies in DNA [?]. There is also much DNA, which does not seem to have any function, except perhaps, to make copies of themselves.6

**Introns and exons**

“Multiple introns, and the prospect that these occur within several genes in the same metabolic pathway, suggest a possible regulatory role for splicing . . .”

Victoria Derbyshire and Marlene Belfort, [?, ?]

In eukaryots the DNA coding protein primary structure (called exon) is not usually without noncoding segments called introns. There seems to be only a weak correlation of foldons i.e. structure elements to exons [?, ?, ?, ?, ?]. An analysis based on a larger set of sequences seems to indicate that there is a stronger correlation between introns and the structure of ancient proteins, however [?]. The strong correlation is carried by introns that lie between codons and are thus believed to be of ancient origin and related to protein assembly by exon shuffling. In addition the length distributions of exons and foldons are very similar. [?]. In any case the relation between exons and protein structures remains somewhat obscured.7

---

5It has been observed that PS is inhibited in some organs of some hibernating mammals, however. [?

6This so called selfish DNA, some of which, called transposons are able to jump from one place in the DNA into another, may have been the basis of the vertebrate immune system [?].

7Perhaps the explanation of introns comes from the protein synthesis: introns seems to have the role of the production and transportation of proteins through RNA into which the introns are translated together with
CHAPTER 2. PROTEINS & DNA

Some amphipathic peptides have been observed to have a self-replication ability \([?, ?, ?, ?]\). This finding may solve the “chicken-and-egg” paradox \(i.e.,\) of which was first: RNA or protein at the beginning of evolution.

2.1.4 Secondary structure

“The ability of a polypeptide chain to fold up into a unique, highly ordered structure is the most important feature that distinguishes a biologically active protein from an inert polymer.”

Heinrich Roder & Gülnur A. Elöve, \([?]\)

The chain of the protein molecule typically forms a secondary structure the most common types of which are the so called \(\alpha\)-helices and \(\beta\)-strands (50%-60% of residues \([?]\)). These correspond to regions of the \((\phi, \psi)\) Ramachandran map near point \((-60^\circ, -40^\circ)\) (\(\alpha\)-helix) and point \((-120^\circ, +135^\circ)\) (\(\beta\)-sheet), respectively. The length of these structures range typically from 5 to 15 amino acids, which is also the correlation length that can be found among average proteins \([?]\) and is of the order of the axis of the inertial ellipsoid.

According to experimental and theoretical evidence the secondary structure is formed, at least partially, rapidly at the beginning of the folding process, which is not a great surprise because of the local interactions involved. D. K. Klimov and D. Thirumalai have estimated that a four turn \(\alpha\)-helix topology forming takes about 500ns whereas for \(\beta\)L-sheet the time is about \(10\mu s\) \([?]\). Secondary structures are hydrogen bonded, but hydrogen bonding is a weak driving force, however. Obviously this is one of the main reasons why the prediction of secondary structure as found in the final native structure has turned out to be unexpectedly difficult. \([?, ?, ?]\)

\(\alpha\)-helix

An \(\alpha\)-helix is a right-handed spiral shape having 3.6aa \((5.4\text{Å})\) per one turn. The helix conformation is the preferred state of the backbone that compacts the chain expelling water while allowing intrasegment hydrogen bonds \([?]\) and van der Waals contacts. It is the most stable arrangement of a polypeptide backbone resulting from a regular pattern of hydrogen bonding between the consecutive carbonyl and imino groups. Also the atoms of the backbone pack together perfectly. \([?]\) The \(\alpha\)-helix is also a natural building block of fibrous structural proteins, in which they can form long coiled coils. Typically \(\alpha\)-helices consist of about 12 residues \([?, ?]\). A quite common helix type in proteins is one that has polar and apolar residues on opposite helical faces. These are called amphipathic helices.

A typical example of an amphipathic peptide is melittin. The strong asymmetry causes melittin helices to align themselves on a membrane surface, the hydrophobic part immersed into the lipids forming the membrane while the polar residues are exposed to solvent (water). \([?]\)
2.1. STRUCTURE OF PROTEINS

β-sheet

In β-sheets the adjacent amino acids are related by 180° rotations. From this results its planar sheet like shape often denoted by planar arrows. The structure is also stabilized by hydrogen bonding between the carbonyl and imino groups, but the groups are not consecutive but belong to adjacent chains. There are two possible arrangements, parallel and antiparallel β-sheets depending on whether the adjacent chain run in the same or the opposite direction. In parallel β-sheets an extra mid-sequence, usually an α-helix, is needed to bring the halves together. In practice β-sheets have the tendency to twist right-handedly, although also left-handed conformations exist.

Energetically β-sheets are approximately equal to α-helices (see prion protein in ch. 2.2.4). After K. M. Andersson and S. Hovmöller there does not seem to be any significant difference in atomic volume between protein with mainly α-helices and those with mainly β-strands [?].

Together with α-helices β-sheets consist of most secondary structure motifs in structural proteins. In contrast to α-helices, β-sheets involve long range interactions. Therefore the success rate of β-strand predictors has been modest, only about 60 %, i.e. actually only slightly better than purely random guessing [?].

Turns

While α-helices and β-sheets contribute much of the residues of typical globular proteins, sharp turns are needed between them to keep the structure highly compact. From this it follows that turns are located at the surface in contact with solvent molecules. One class of turns, called β-turns [?] consists of four (or five) consecutive residues, of which the first and last are hydrogen bonded, while the rest are highly polar due to unpaired N–H and C=O groups and tend to be less hydrophobic in general. Thus the turn is highly polar and hence likely to be found at the surface. There are also some β-turns inside globular proteins, however [?]. These buried β-turns seem to make complexes with water molecules, thus burying water molecules inside the globule and making water molecules integral parts of the protein structures.

Random coils

Most, about 53 %, of all residues in globular proteins are in irregular structures making protein 3D structure prediction actually really enigmatic [?].

2.1.5 Tertiary structure

“There are many globular proteins in a living cell, and they play a key role... However, the theory of such systems is extremely hard; a protein globule is perhaps one of the most complex objects in modern physics. What is most striking and unusual is that proteins have a strictly defined tertiary structure.”

Alexander Yu. Grosberg & Alexei R. Khokhlov, [?]

The three-dimensional native (active) conformation of protein is called tertiary structure. Sometimes the term quaternary structure is also used to denote the native conformation of the most complex (oligomeric) proteins consisting of several subunits.
The modularity *i.e.* subunits of proteins and building blocks in general are vital for reliability: a complex system consisting of a number of identical or nearly identical small modules is much easier to construct than a corresponding system consisting of a few complex subunits.

The longer the sequence the more difficult is its folding. Periodicity analysis reveals, in addition to α-helicity periodicity, 350bp periodicity for eukaryots and 440 bp periodicity for prokaryotes giving approximately 120 and resp. 150 aa subunit lengths [?].

Especially in many small proteins disulfide bridges largely contribute to the stability by binding distant parts of the protein together.

In this tutorial tertiary structure is used as a synonym to the native conformation in all cases.

**Stability and uniqueness**

The native conformation seems to be quite stable (albeit marginally) and unique. There seems to be only a few exceptions to uniqueness [?, ?]. In view of the protein function this stability, uniqueness and compactness is practically always most natural. Perhaps the most stable proteins are those, of which the vertebrate eye lens mostly consists of.\(^8\) Once laid down these proteins remain for the lifetime of the animal, which can exceed a hundred years. [?] This is even more remarkable, when we remember, that these proteins are exposed to rather hard environmental stress. It might not be a coincidence, that at least one of these proteins, α-crystallin, is related to heat shock proteins *i.e.* molecule chaperones [?, ?]. It seems that the lens proteins have descended from ancient stress-induced enzymes that got their current special optical task but perhaps retained some of their original functions, too.\(^9\) [?] Their evolutionary stability is high: only three amino acid changes per 100 residues every million years. This is comparable to cytochrome c (two changes), which is considered a highly conserved protein [?]. A highly conserved primary structure is a clear sign of strong constraints laid on protein structure and thus its functions. The stronger the constraints the less likely a beneficial mutation is. One of the least varying protein seems to be histone H4, which is a DNA binding protein controlling transcription of genetic information and is thus most strictly constrained. It also gives a clear indication of the ruggedness of the protein evolutionary landscape: one mutation correlation length is very short, thus effectively pruning mutations and in this way slowing evolution. But speed of evolution and metabolism is relative. While the current enzyme catalyzed reactions seem to be both highly specific and fast, there is no reason why the situation should have been so in the primordial environment, where all organisms were much simpler and the absolute selection pressure was thus much lower than today.\(^10\) The thermophilic bacteria appear to be the most primitive organisms known.\(^11\) [?, ?, ?]

---

\(^8\)While the eye lens consists of mostly globular proteins, the *corpus vitreus*, the major portion of the vertebrate eye, is rich in structural proteins of collagen type [?].

\(^9\)This fact nicely supports the much debated gradual evolution(s) of the eye(s).

\(^10\)In harsh extreme environments such as in hot ponds (geyshirs) or several kilometers deep within the (nearly) solid rock of the lithosphere or oil wells, arctic ice, salt marshes, Finnish forest humus [?] etc., where some primitive *"extremophile"* archaebacteria, *Archaebacteria*, (Woese 1977) live, the ecosystem may be quite similar to the ancient one.

\(^11\)An obvious opposite explanation is that thermophilic bacteria have lost (got rid off) all other than extremely vital and robust functions while conquering their current extreme environments [?]. In any case the
2.2. PROTEIN FOLDING PROBLEM

While a single residue mutation can be fatal it is easy to understand why there is so much repetition in the genetic information. By having e.g. 100 times one gene coding a key enzyme effectively prevents point mutations from destroying the vital information. In addition it gives the opportunity to have 100 times more of the enzyme in synthesis and last but perhaps not least the repetition allows much freedom to evolution. The evolutionary benefit of repetition comes from diversity: due to the vital enzyme forms some inefficient or even slightly harmful forms can co-exist and evolve away from the vital form. Some day a mutation can turn some of these actually more or less parasitic enzyme variants into the most effective ones. This might be one mechanism by which evolution performs local search in the protein sequence space since ancient progenotes [?].

2.1.6 Protein families

"Enzymes are high precision catalysts whose specificity and efficiency have been optimized by the processes of trial and error conducted through many millions of years of evolution. The specificity and efficiency of enzymes are of crucial importance to metabolism. Their ability to carry out a whole network of interrelated reaction chains producing only the desired products is an achievement never surpassed in any process designed by Man."

T. Palmer, [?]

It has been estimated that there exist about 1,000 different protein families and thus also their representative folds [?]. The structure of proteins has evolved together with organisms over billions of years. It is therefore natural and instructive to look at protein structures also from the evolutionary point of view. The evolution of proteins can and should be seen as a process similar to general evolution in nature.

Analysis of the sequences and structures shows that we can indeed construct evolutionary trees based on molecular data from related molecules in different species. These trees have been shown in most cases to be equivalent to the evolutionary trees constructed from classical comparative anatomy and taxonomy. [?]

2.2 Protein folding problem

The folding of protein was first proposed by Hsien Wu in 1929 in Peping, China:

"The protein molecule is not to be regarded as a long straight chain but rather as a compact structure. Besides the peptide linkage by which the amino acids are joined “end to end” there are other kinds of linkages which unite different portions of the chain “laterally.” These lateral linkages are very labile. The chain may be conceived to fold repeatedly at short intervals forming a three-dimensional network somewhat resembling a crystal lattice in which the atoms are replaced by molecules of amino acids. Denaturation is the breaking of these labile linkages."

Hsien Wu, [?]

origin of thermophilic bacteria seems to be quite ancient according to biochemical evidence.
Levinthal’s paradox: “If a protein is to find its functional conformation by wandering randomly through conformational space, in excess of $10^{50}$ years would be required for folding.”

The number of conformations is approximately $8^n$, where $n$ is the number of residues \textit{i.e.} each residue has on the average 8 possible conformations. It has been estimated that the number of proteins that could possibly have existed during evolution on Earth is somewhere between $10^{40}$ and $10^{50}$.

Włodek Mandecki has nicely compared the complexity of protein design and the game of chess: selecting amino acids for a 100 residue long protein have the same order of possibilities ($\approx 10^{130}$) as a typical chess game of 40 moves. The success of the computer in chess gives us some hope that also the protein folding problem could be solved for practical purposes by computing methods.

2.2.1 Thermodynamic hypothesis = energy minimum

“This hypothesis states that the three-dimensional structure of a native protein in its normal physiological milieu (solvent, pH, ionic strength, presence of other components such as metal ions or prosthetic groups, temperature, and other) is the one in which the Gibbs free energy of the whole system is lowest; that is, that the native conformation is determined by the totality of interatomic interactions and hence by the amino acid sequence, in a given environment.”

Christian B. Anfinsen, [?]

Briefly said, the native conformation is thermodynamically stable, \textit{i.e.} it corresponds to the (global) minimum of free energy. This can be shown by demonstrating that the native structure is only a function of state and does not depend on the process or initial conditions leading to it. A large number of experiments done show that the thermodynamic hypothesis seems to be valid (see review in [?] and references therein). Remember that while simple systems like single atoms or small molecules have discrete energy levels, macromolecules have highly degenerate states, which are best characterised by a continuous energy landscape. It can be said that it is the highly multimodality or ruggedness of this landscape that makes the detection of the minimum conformation an extremely challenging problem.

The so called weak thermodynamic hypothesis assumes that the folding process deals with the easy free energy optimization instances.

It must be remembered that the thermodynamic hypothesis and the minimum energy assumption is only a physico-mathematical model, not a proved theorem, which indeed seems to be more or less valid for most proteins. We cannot prove it, perhaps only give some falsification examples. In other words mathematics and theory provide some simple tools for approximate modeling of complex systems. Free energy is a state variable not having any special chemical meaning, except indirectly for the usually very few special cases in which the active site is involved directly or indirectly. We cannot measure free energy and thus the immediate evolutionary pressure on it cannot be high. The result of evolution we see as protein structures is not the result of a process primarily trying to minimize free energy, but to optimize the functions of the proteins or still more correctly and generally the fitness of species [Dar79]. Due to the stability aspect the free energy minimum does not contradict too much this goal, however. Proteins are rather complicated structures that may have several
2.2. PROTEIN FOLDING PROBLEM

diverse and peculiar properties related to their structure. All that matters is actually how they function together with other proteins and molecules in cells, not whether or not being exactly at some local free energy minimum.

2.2.2 Kinetic i.e. folding pathway hypothesis

The refolding of denatured protein back into a biologically active form takes from about 1ms to 100s or longer. During this short folding period of time the protein molecule undergoes perhaps some $10^{11} - 10^{13}$ conformational changes. Two consequences can be directly drawn from this time scale: on the one hand it is much too short for an exhaustive random search for the minimum free-energy structure; on the other hand it is clearly much longer than a simple collapse into a minimum free-energy hole.

It is thus reasonable to believe that there are some kind of folding pathways to the native state that can effectively avoid exhaustive search of potentially all possible alternatives. The simplest assumption is perhaps that this path goes through independent local motif folding i.e. parts of secondary structure elements are rather easily attainable and if they are stable enough finally we end up with the native conformation.

The protein folding mechanism may be encoded in the amino acid sequence, analogously to the genetic code of a DNA chain, but the code is still unknown, if it exists at all. Revealing this code would have a profound effect on our understanding of the basic chemistry and evolution of life. An argument against this hypothesis is that the DNA does not contain neither theories nor models of physical chemistry, but the results of these processes. Finding a formula in DNA might be as hopeless as finding the gas equations by viewing a chamber filled with gas or theories of solid state physics by weighting a crystal. Statistical analysis has not revealed any significant deviations of certain sequence characteristics, such as hydrophobicity pattern, from a random distribution. This is slightly controversial since, a more refined analysis indicate some deviation from random distribution, however.

In any case it can be said that solving the protein folding problem is equivalent to deciphering of the second half of the genetic code. Thus the importance of protein folding can hardly be overestimated and it was the reasons for selecting it as one of the key topic of this tutorial to give some background of evolution and the building blocks it uses.

2.2.3 Folding in vivo and chaperones

It takes from fractions of seconds to minutes for a protein to fold from its unfolded conformation to the folded native conformation. In vitro, i.e. in test tubes, most globular proteins seem to spontaneously fold to their native state if certain conditions hold. In vivo, i.e. in living cells, there are also special enzymes called chaperones that aid proteins to avoid messing together in the crowded cell environment and instead to get their right globular native shape. These proteins are produced especially in stress situations in which cells contain exceptionally high concentrations of misfolded proteins. A typical situation is heat shock, after which chaperones are also called heat shock proteins (Hsp). Some chaperones keep the protein unfolded until its synthesis is finished or it has arrived at its cellular destination organelle or membrane, i.e. they assist protein transport. In short, chaperones are proteins that take much care of the folding process in cells from the very beginning of the protein synthesis until the recycling of old (misfolded) proteins. Suzanne L. Rutherford and Susan Lindquist
have proposed that Hsp90 may be a key factor in rapid morphological evolution in a stress situation, while normally controlling morphological regulation by folding the corresponding rather unstable proteins [?, ?].

According to [?] there are the following three main strategies that the cell can use in order to prevent aggregation in the extremely concentrated environment of the cell:

1. blocking of exposed hydrophobic residues and thus protecting them,
2. isolation of the protein molecule from the cell environment until folding is completed, and
3. acceleration of folding pathway conformation evolution.

The first two strategies are related to molecular chaperones, while protein synthesis via so called preproteins belong to the third strategy class. Examples of the above strategies are e.g.:

1) activities of molecular chaperone Hsp70, 2) activities of Hsp60 in combination with ATP, and 3) α-lytic protease [?, ?, ?]. [?] It is interesting that the second example involves about 130 hydrolyses of ATP molecules for refolding of one rhodanese molecule. As so much energy is needed for folding of this special protein, it is no surprise that it seems to be unable to fold properly in vitro into its native conformation without the help of molecular chaperones. [?]

2.2.4 Misfolding

“What are the key features of the amino acid sequence that are critical to protein folding and generalized prediction of protein structure?”

Mark S. Johnson, [?]
with them to new cells and generations thus apparently causing Creutzfeld-Jacob’s disease and its bovine equivalent bovine spongiform encephalopathy, also known as mad cow disease and some other, luckily, mostly rare diseases of both animals and humans.

It seems that the prion protein PrP has two different conformations: normal cellular form denoted by PrP\(^C\) (\(C = \text{cellular}\)) and the abnormal denoted by PrP\(^S\) (\(S \approx \text{scrapie-like}\)). The difference is that PrP\(^C\) is mostly \(\alpha\)-helical, while PrP\(^S\) consists mostly of \(\beta\)-sheets. [?] The conformation difference is thus profound, concerning also secondary structure and not only tertiary structure. Prion-related diseases can be characterised as ‘protein folding diseases’ [?].

### 2.2.5 Inverse protein folding problem

The problem of designing a polymer sequence that folds to a given target conformation is called the inverse protein folding problem (IPF, see fig. 2.2). It seems that the concept inverse protein folding was introduced in 1986 in an E.M.B.O. course [?, ?] To evaluate the effectiveness of the solution, three criteria have been proposed [?, ?]:

- The protein sequence should fold to the target conformation i.e. the energy of the solution conformation is not greater than any other conformation.
- The ground state should not be degenerate.
- There should be a clear energy gap between the energy of the solution conformation and any other conformation.

![Figure 2.2: Protein folding and inverse protein folding problems.](image)

The inverse protein folding problem can be solved by a so called threading algorithm: a score (potential energy) is computed for a sequence and the amino acid sequence - 3D protein structure alignment, which minimizes the score is the likely folding [?, ?, ?, ?, ?, ?]. The minimization problem is called the threading problem and the corresponding alignment a threading. The protein threading problem has been proven to be NP-hard\(^{12}\) [?, ?].

\(^{12}\)see chapter 5.2.3
Chapter 3

Origin of Life

What do the bees, wasps, and other small creatures tell us about the possible routes towards the emergence of life.

The emergence of life has interest man from the rise of human civilisation, probably even longer, and still today, after much progress in science, we are deeply confused about the possibility of emergence of organic life out of inorganic matter. How was it possible to have the very first functioning cell, even an extremely primitive one, having all the vital aspects of life encoded? The probability of the emergence of such functioning cell can be calculated to be practically zero even during all the life time of our Universe. Even the probability of a single protein molecule to fold into its native conformation can be calculated to be so low that there is not any possibility for any protein to be in its active, structure dependent, native state. [?, ?] This Levinthal’s paradox, and similar “mathematical proofs” can be said to indicate that there must be something more than pure change working behind the scenes for the origin of life. Another equally correct interpretation of this kind of “mathematical deductions” is that they must be incorrect. This is not because of the mathematics used but the assumptions made before the, as such correct, calculations. Hence, the assumptions must be totally wrong. In the protein folding case the latter interpretation is valid because we can easily make an experiment in which the protein really folds within a short period of time, typically in a fraction of second. In a well equipped biochemistry laboratory we can even synthesise new proteins that, if we are lucky enough, will also fold. This is so because the protein molecule does not go through every possible conformation randomly, which is the assumption of the simple exhaustive search model, but only a limited subset of possibilities because of many physical and chemical restrictions that are quite complicated and thus well beyond the assumptions of the simple mathematical models used in the “mathematical proofs”.

Similarly the origin of life models need careful consideration before any definitive conclusions can be drawn. In this paper we will discuss about the emergence of life as a problem of finding suitable building blocks having properties that make gradual evolution from inanimate matter into a primitive ancient cell possible.

The Darwinian evolution is extremely slow, there are not any great jumps to be observed. On the contrary, it took a long time before scientists had gathered enough evidence so that Charles Darwin was able to recognise this slow evolution [Dar79].
mordial soup cannot be much different. An observer on the shore of an ancient ocean would not observe anything dramatic happening, even when observing for millions of years. Even today the direct observation of evolution in nature is difficult, even if not impossible [?]. In spite of its seemingly slow progress, the evolutionary approach can be used in breeding of plants and animals, even in engineering design and optimisation [?]. In both cases evolution has been sped up by heavy selection. E.g. in some plant breeding experiments there may be selection ratio 1:10,000 or even higher. Similar figures can be found in genetic programming, where the evolutionary principle is used to create computer programs [?].

The key phenomena behind biological evolution are

- **diversity** of entities behaving at least a little different in their environment,
- **selection** in the form of chemical and physical interactions between both the entities and environment, and
- **reproduction**, which means that the entities are able to increase their number *e.g.* by harvesting more material from the environment until they break into smaller pieces.

It seems that any candidate for the ancient attempt for organic life must fulfill these. In addition some sort of **metabolism** *i.e.* input and processing of energy and material is vitally beneficial.

### 3.1 Related work

Other works on the origin of life and the peptides first hypothesis include [? , ?]. For further references on peptides and proteins see *e.g.* [?].

### 3.2 Building blocks of life

The striking feature of all organic life is that it is based on biochemical structures and processes that are in common to all living beings from the simplest bacteria to plants and animals. In this way an organism is like a computer, all computations are based on a simple character set, the codons encoded in DNA, and the construction materials and fine chemicals are mainly proteins, which further are strings, *i.e.* polymers, of amino acids, the type and order of which are given by the DNA memory.

Practically the chemistry of life can be said to be chemistry of thousands, even millions of different protein molecules encoded by DNA. The other feature in common to all organisms is that they are composed of one or more cells.

This kind of system, even if it seems to be well organised and more and understandable while the science progresses, is far too complex to emerge out of the building blocks by change. The obvious conclusion is thus that there must be something simpler between the primordial soup and the very first ancestor cell. This view is the topic of this chapter. We should be able to find the building blocks and processes that made it possible the first cell-like structure to emerge out of the primordial soup.

Having two main building block types, DNA (RNA) memory and protein (enzymes) tools, we have the problem to decide which one of them is the most important and thus the candidate for the older one in the chain of the processes leading to the emergence of life.
3.3. PEPTIDES

Because of its stability DNA is good for memory. Together with the processes keeping DNA in condition in cells, it easily beats the modern computer memories when speaking of stability. DNA and RNA have also some catalytic properties but they cannot be compared to the extremely specific and powerful catalytic properties of proteins. The weak point of proteins is their low stability. Too high or low temperature are among the many environmental conditions that easily denature or even decompose protein molecules.

Frankly speaking, both alternatives seem to be too delicate to survive to be successful candidates for the building blocks of the very first cells in the harsh environment on the ancient Earth.

3.3 Peptides

An ideal candidate for a building block in an ancient ocean could be a molecule that has both memory and catalytic capabilities and is not too complex.

Such molecules exist, they are small proteins, called peptides. The difference between a protein and a peptide is the size. The peptides are much smaller than the modern proteins. So, in practice there is not any clear difference between the two. Great. This means that there is not any barrier between the peptides and the proteins, except the problem of biochemistry vocabulary. That might not be the first concern of the first ancient cells, however. Actually a protein can be seen to be composed of a secondary structure, like α-helices resembling peptides [?]. After Christian de Duve the first proteins might have been less than 20 amino acids long [?].

How about the catalytic properties of peptides? Actually many peptides have strong effect on cell functions. E.g. the toxins of many snakes, spiders, cones, scorpions, wasps, etc. contain peptides that are extremely poisonous [?]. Even a small amount of snake venom can be lethal. Venoms are poisonous usually because they have a strong influence on nervous cell functions e.g. by blocking the nerve signals causing paralysis of the victim. In this respect the hunting wasps, e.g. Sphecidae (fig. 3.1), may be an extreme: they paralyse their pray to keep it alive while their larva is feeding it for days. See [?] for description of behaviour of different hunting wasps. Another nice example is the group of sea cones hunting live fish. They also have venoms in their harpoon like organ to help to catch fishes that come too near of this extremely slowly moving animal that paradoxically is a predator of fast moving fishes [?]. Observe also that e.g. neurotransmitters include amino acids, derivatives of amino acids and peptides [?].

In addition quite many animals, like insects, use peptides as antibiotics [?].

The property shared by both venoms and antibiotics is that they cannot be extremely specific to be useful. This relatively low specificity makes them opportunistic e.g. capable to paralyse and kill new prey species and to prevent attack of new micro organs. Being much shorter than proteins, peptides can be expected to evolve faster, which indeed seems to be the case [?]. This is beneficial in “communication” between different specimens. The conditions within cells are much more stable allowing higher specificity with the cost of longer sequences needed to construct proteins. Hormones are chemicals used in intercellular communications.

\footnote{A dead insect soon decomposed into a poisonous menu unsuitable for the maggot like wasp larva in its comfortable nest.}
Figure 3.1: *Mellinus arvensis* (Linnaeus, 1758), a common hunting wasp (*Hymenoptera: Sphecidae*) species in the Southern part of Finland upto and including the region around Vaasa (Ostrobothnia australis (Oa); the author’s observations during Summers 2003 and 2004). It hunts and paralyses flies (*Diptera*) with its venom for her offspring, which are waiting their mother in an underground nest in soft sandy soil. The specimens in the image were collected 18th August 2004 at Oa, Vaasa, Öjberget on a bank of fresh loose fine sand. The original image was made by using Hewlett Packard Scanjet 5550c. The final image was contrast enhanced by digital image processing using program GIMP.
One class of them is peptide and polypeptide hormones, which thus nicely display evolution from simple peptides towards more complex and specific proteins (e.g. oxytocin 9 amino acids, *human growth hormone* (*HGH*) 191 amino acid protein).

Thus peptides seem to be a tempting group of chemicals having strong effect on cells and cell membranes in particular [?, ?]. The properties of the cell membranes are vital for all cells.

Thus, we have seen that the peptides can be extremely powerful chemicals having dramatic effect on cells and their interactions. The other key property that is needed for an ancestor molecule group is coding and replication. This also belongs to the properties of peptides [?, ?, ?].

### 3.4 Discussion

How does the peptides fulfill the three key phenomena, diversity, selection, and reproduction that are vital for evolution? Peptides being amino acid polymers easily fulfill *diversity*. Having about 20 or so amino acids available, even a short peptide chain has an enormous number of possible combinations. In practise some amino acids are more likely to polymerise than others but in any case there seems to be no problem with the diversity issue.

Selection is also available in the form of the environment, speed of polymerisation *etc.*. Again we do not have any problem to imagine that some peptides were selected more often than some others.

The *reproduction* is more problematic. Here we will assume the following scenario. In order to have a suitable environment for chemical reactions, a cell like structure is needed. Fortunately cell like structures, small vesicles, are formed by the self-assembly of lipid molecules. Once there was lipids in the ancient ocean, there was soon also cell like structures.

Sooner or later some of these cells were carrying molecules inside them and on the lipid membrane that were able to harvest more molecules into the cells. Some of these further grew also larger until they were not mechanically stable resulting a set of smaller cells.

Peptides are not so specific, but anyhow quite efficient and still simple. When evolution needed more specific chemicals, it was quite easy to use these peptides as building blocks to create more complex proteins, which are composed of peptide like secondary structures. This approach can be called as the “peptides first hypothesis”.

An obvious way to evolve proteins out of peptides is to join peptides into longer sequences. Eldon G. Emberly *et al* have used stacking of given secondary structures to analyse all possible α-helical structures and their designability [?]. Stacking here means to combine pieces loose of α-helices in all possible ways. Adding turns to these stacks gives the possible native conformations.

### 3.5 Peptides first

It is reasonable to assume that during the evolution the first building blocks were simple like peptides more than complex multi-domain proteins. Nowadays peptides still have certain roles in organisms. They are quite abundant as important components of such substances as, *hormones*, *venoms*, and *antimicrobials* or hybrids of them (see e.g. [?, ?, ?]). These substances are characterised by two important factors: fast and strong influence and wide
CHAPTER 3. ORIGIN OF LIFE

spectrum. A venom should kill or at least paralyse its target fast, within a few minutes, irrespective of the target species. Correspondingly an antimicrobial should recognise and stop invasion by any foreign micro organ before it is too late from the victim point of view. Among peptides there seems to be a lot of suitable ones, which are highly efficient and have broad spectrum. The influence is usually, especially for amphipathic peptides, by interfering the functions of cell membranes. It is easy to imagine that these kind of substances were important also in the inter-species communications at the rise of organic life. It is also easy to imagine that the evolution used these simple peptides as building blocks to construct more complex proteins that are more specific and perhaps even more efficient than the simple peptides they first consisted of (for an example of a helical protein folding model fitting this scenario see [?]). And indeed most of the current proteins still contain $\alpha$-helix and $\beta$-sheet, which are the main motifs of membrane bound peptides. In a similar way we can assume that these secondary motifs are important as intermediates and final structural motifs of folding proteins. Observe that some peptides can form both stable $\alpha$-helices and $\beta$-sheets. The type depends on conditions like temperature or pH [?].

3.6 Molten globules

It is interesting and instructive to notice that there are some proteins, like clusterin that contain molten globule like disordered regions. It is supposed that the chaperone like activity and the high affinity of clusterin to a wide array of ligands is based on this structural feature [?]. This is also promising for the emergence of life: extreme specificity is not any strict property of proteins also some more flexible structures are beneficial. One peptide could have been involved with a set of reactions rather than one specific reaction.

An example of structural stability in evolution is provided by the venom of Conus cone snails, which is used to paralyse the lively prey, like small fish, of these slowly moving marine gastropods. The venom consists of several highly variable small peptides. The variability is not related to structure but the amino acid residues. The only conserved residues are Cys residues providing the many structure stabilising disulphide bridges. [?, ?, ?] A much more well known hypervariable gene families can be found in the immune systems or their challengers the surface glycoproteins of parasites [?].

3.7 Optimality of proteins

It is widely assumed that the native conformation is also the free energy minimum. It is also known that the native conformation is well defined, unique, and rigid and that the native conformation is essential for the proper functioning and specificity of proteins. On the other hand to be unique the conformation must be at the minimum energy. This is so because otherwise there are typically a large number of possible conformations at the same energy level.

3.8 Conclusions and future

We have here speculated about the key problem of biology, the origin of life. Do this discussion have anything to do with more practical problems like engineering? The answer is definitely,
yes. Engineering is based on sciences, which are applied in order to create something useful as economically as possible. In the case of the peptides first hypothesis we can refer to the new science called peptide or protein engineering, which means a branch of engineering trying to create new peptides and proteins for engineering and other artificial purposes [?]. One goal of this new engineering field is to create nanoscale machines for the benefit of mankind. [?]
Chapter 4

Soft computing

Lotfi Zadeh has given the name **soft computing** to cover methods such as

- **artificial neural networks**,  
- **fuzzy logic**,  
- **immune systems**, and  
- **evolutionary algorithms** like GAs.

The name reflects the fact that these methods are all in clear contrast to the deterministic i.e. 'hard' **artificial intelligence** methods, that are based on e.g. formal **mathematical logic** and its applications like **Prolog language** [?]. The other feature in common to all soft computing methods is that they have been successful in solving such difficult real world problems as **classification** and **pattern recognition** (**neural networks**), **nonlinear control** (**fuzzy systems**), and **global optimization** and complex problems (**genetic algorithms**).

4.1 The immune system

“No truth is of course permanent. It is in the nature of immunological investigation, or indeed any investigation, that even principles are modified by subsequent discoveries.”

Jan Klein, [Kle90]

The **immune system** of higher animals is based on the generation of a broad spectrum of chemical structure, molecules called antibodies, and the selection of these structures by the immune system.\(^1\) This principle has been successfully applied also to some computational experiments. The ideas presented in this and the following section are mainly due to Gerald Edelman, who received a Nobel prize for his immunological studies. Edelman has also done much work on computer simulations of the **nervous system** based on sound physiological and histological assumptions [Ede87, ER90] and his approach has even been used to demonstrate a **mobile robot control** [?].

\(^1\)Also such lower organisms like plants and insects have immune mechanisms including antimicrobial peptides called **defensins**.
The human body can successfully recognize a large set of chemicals foreign to the body, even chemicals that have never existed before on the planet Earth. So how can they be successfully handled? The system works after the selectionist principle proposed by Macfarlane Burnet [ER90]. The development of the immune system is controlled by the genomic information. However, the specific functioning antibodies cannot be found directly coded in the genome. The sheer number of different antibodies simply prevents any direct one-to-one coding.

The immune system works so that every single cell makes only one kind of antibody. A special mechanism diversifies different cells so that in early childhood the body develops a huge repertoire of different antibodies, each on an individual cell. When a foreign molecule or antigen comes in, it binds to a particular subset of those antibody shapes. There is a mechanism to increase the number of those cells bearing antibodies, i.e. selection, that match the given antigen.

In general, in no two individuals are there the same binders. From the mechanism it also follows that there are a lot of nonsense antibodies. It is also easy to understand that the different antibodies cannot be created directly by evolution as that would be too slow a process and would require too much genomic information. The genome only tells how antibodies are generated in general, not the exact structure of any specific antibody.

The immune system is perhaps the best example of selection working in a short time scale in nature. The other important selectionist process can be found in the development of the nervous system.

Algorithms mimicking the functioning of the immune system has been developed for solving optimisation problems. For references on immune systems as a model of computational methods see bibliography [Alai].

4.2 Neural systems

“... in order to control behaviour, the biological brain must be able to form internal models of the sensory environment and its history. Such a “miniature environment” to which all decisions are related is provided by memory. ”

Teuvo Kohonen, [Koh89]

The human cerebral cortex, which is responsible for the higher mental processes in the brain, has approximately $10^{10}$ neurons and some $10^{15}$ connections between neurons. Brain development is guided by human genome, which contains only approximately 20,000 genes (c.f. table 4.1) so that the exact neural structure which seems to be quite irregular cannot be determined directly only by the genetic information. [AR90] The situation is similar to that of the blood circulation systems. However the proper functioning of the nervous system is much much more complicated.

Actually, how this extremely complex system functions at all! E.g. how are pattern recognition skills implemented in these neural networks. A really interesting thing is the capability of animals, including human beings, to generalize on a very small set of samples. This is a property that the artificial classifying systems are unfortunately mostly lacking. It
has been shown that pigeons can be conditioned e.g. on leaves of Quercus alba (white oak). After being conditioned the pigeons recognize that all white oak leaves are in the same class and reject all other kinds of leaves, such as maple leaves [Cer79]. The conditioning is on leaves of white oak in general and not on specific leaves used in teaching, which cannot usually be recognized from one another, i.e. there seems not to be much overlearning in nature unlike in machine learning. The same conditioning can be done using trees, faces, fishes etc, or even artificial figures e.g. such as Charlie Brown in the “Peanuts” cartoons [Cer80]. In a way it seems as if the Linnean like taxonomy detection is somehow built-in as classifiers in the brains of higher animals. A practical feature when trying to classify your edible and inedible conspecies.

4.2.1 Selection in the brain

“Our knowledge, our attitudes, and our actions are based to a very large extent on samples. This is equally true in everyday life and in scientific research.”

William G. Cochran, [Coc63]

The evolution of species is assumed to be based on natural selection [Dar79]. Gerald Edelman suggests that the same selection principle is also responsible for neural functions [ER90]. Selection requires pre-existing diversity. In the case of the brain this means structural diversity.

The neural networks of the brain are primarily created by epigenetic processes which also create many random variations. After most of the nervous system is built, synapses in various combinations are selected by strengthening or weakening them, instead of increasing the numbers of cells in these circuits by growth or cell division as in the immune system. In order to match the spatio-temporal continuity of the world, which doesn’t have any symbolic labels, the nervous system evolves in such a way that it organizes into certain kinds of maps [Koh89]. These maps are further connected in a special way that allows their properties to be linked for higher level connections.

The brain is too random to be like any man-made digital computer. It is also too complex to be entirely and exactly defined by the genetic code. After Edelman it seems that both the brain formation and functions are based on selection, the same selection principle that is behind evolution and can be found also in the immunological system.
For further references on artificial neural networks combined with genetic algorithms see bibliography [Alab].

4.3 Quantum computing

Lately there has been much interest in using quantum mechanics based computing, *quantum computing* to solve hard problems. The key property of quantum computing is to be able to represent several alternatives by using *qubits*, which can be equal to 0 and 1 at the same time by using superposition of two quantum states

$$|\Psi > = a|0 > + b|1 >,$$

where $|\Psi >$ is the qubit wave function, while $|0 >$ and $|1 >$ represent the 0 and 1 states, $a$ and $b$ are complex numbers such that $|a|^2 + |b|^2 = 1$. By using $n$ qubits it is, in principle, possible to represent up to $2^n$ bit combinations at the same time. This is a feature not possible in classical computing and it is one of the key properties of quantum computing that can be tried to solve problems suffering from *combinatorial explosion*.

A quantum computer can be compared e.g. to the *roller coaster* shown in Figure 4.1. The state of computing is the position and velocity of the car defined by the shape of the track. The preparation of state is analogous to putting the car to a certain starting point, say A, and giving it a certain starting velocity. The result of computation is analogous to measuring the state of the car at a certain track position, say C. However, because of the Heissenberg’s uncertainty principle we cannot measure *both* location and speed precisely. That is why the computation is usually repeated to get precise enough result. We cannot continue the calculation after the measurement, which stops the car i.e. the wave function is collapsed due to the measurement—we must start again from the point A. Stopping the car at point C obviously collapses our computation to point B. Ideally a quantum computer is like a pendulum *i.e.* *adiabatic* meaning that no energy is lost during computation, so that after the computation it returns to the starting point, *i.e.* D=A, like in a practical roller coaster. The problem is to know where the car (state) was during the computation.

For more references on quantum computing see bibliography [Alaj].

A closely to quantum computing related area of computing is *molecular computing* where single molecules or populations of molecules are used to implement computing.
Figure 4.1: A quantum computing analogy: roller coaster.
Chapter 5

Optimisation methods

"Test everything. Hold on to the good. Avoid every kind of evil."

1 Thess. 5:21-22.

The topic of this chapter is a brief introduction to the basic properties of optimisation problems and some most popular optimisation i.e. search methods, their classification, and applications.

5.1 Parallelism vs. Computing

From the computational point of view parallelism of all physical and chemical systems makes them difficult for rigorous mathematical analysis and simulation by computers. More and more powerful computers are needed and luckily also provided by computer engineers. The opposite aspect is that this massive parallelism of all physical systems could facilitate extremely efficient novel computing machines. They could be so efficient that only a slight deviation from pure random processing would be needed and they could solve the most difficult optimization problems much faster than any current computer.

The problem with parallel processing is its inherent locality propensity caused by practical von Neumann architecture bottlenecks. The more global information is needed the more processing overhead it causes due to the management requirements of the shared information. Parallel computer architectures have made it possible to solve large applications. Some applications like genetic algorithms lend themselves quite easily to be evaluated by parallel computers like DECmpp (MasPar) (see [PS98] and references therein). → Vaasa GA bibliography: http://www.uva.fi/~TAU/reports/report94-1/gaPARAbib

5.2 Search efficiency

In this chapter we will briefly analyse the factors affecting search efficiency.

\(^1\)E.g. DNA and quantum computing [?, BF98]
5.2.1 Processing speed

In order to be able to compare different search approaches we must have ways to measure search efficiency. The most straightforward approach is to measure the processing (CPU) time needed to find the solution conformation. Statistics analyses of the processing times can be further used to get a more precise image. A measure, that is more machine architecture and instruction set independent, is to count the number of steps or parameter value trials evaluated. In this tutorial we have used the number of trials as the primary search efficiency measure. Observe that this measure not only recognises the steps needed to proceed but also those taken in order to sense the best directions to proceed e.g. when using a local hill climbing phase.

Search time can be analysed by e.g. the following hierarchy of operations:

1. **primary measurement**: measuring (processing) time until solution found,
2. **basic statistics**: average time, median, variance, risk involved, etc., and
3. **comparisons** to different random and natural distributions.

5.2.2 Fitness landscape

“Nihil esse certi”

Gaius Plinius Secundus, [?] 

Manfred Eigen has emphasised the fractal nature of landscape and the high dimensionality of sequence space facilitating evolution. This can be expressed as the so called fractal island model: in infinite dimensional space no filling is necessary for on an island poured water to reach the ocean. This model explains why it is crucial to have a high dimensional space for a successful search. The other benefits include [?]

- practically an unlimited coding capacity (combinatorial explosion),
- unique points and well defined environments (discrete structure),
- short distances (*Hamming distance*) between points, and that
- gradient search does not easily trap in these kinds of landscapes.

In order to estimate and compare the difficulty of optimisation problems and the efficiency of the corresponding optimisation methods we need measures also for the problems. One important property is the shape of the function to be optimized. The more local extremes it has *i.e.* the more rugged it is the more difficult the problem tends to be to solve. Ruggedness, unfortunately, while being a very concrete concept, lacks good formal definition. A number of empirical measures have been proposed including the number of local optima and the average length of up- or downhill walks. One method is to evaluate the correlation of the *fitness landscape*. Most of the empirical correlation functions \( \hat{r}(s) \) are more or less exponential

\[
\hat{r}(s) = \exp(-s/L),
\]
where $L$ is the \textit{correlation length} and $s$ is the distance \textit{i.e.} number of mutations. There are at least the following three methods to estimate $L$ from $\hat{r}(s)$ by solving [Sta95a]:

$$
\hat{r}(L) \overset{\text{def}}{=} \frac{1}{e}, \quad \text{(definition of } L) \\
- \ln \hat{r}(s) \overset{\text{def}}{=} \frac{1}{Ls + c_0}, \quad \text{and} \quad \text{(linear regression)} \\
L \overset{\text{def}}{=} - \frac{1}{\ln \hat{r}(1)}. \quad \text{(nearest neighbour correlation)}
$$

It seems that the correlation length $L$ usually depends linearly on the system size $n$. This means that we can use the so called \textit{scaled correlation length} $L_0 \overset{\text{def}}{=} L/n$ as a more or less problem size independent measure, by which to compare landscapes of various problems [Sta95a].

Gregory Sorkin has defined landscape to be \textit{fractal} as follows [Sor88].

**Definition 5.2.1 (Fractal fitness landscape)** A landscape $f$ is fractal if

$$
\left\langle \left\| f(x) - f(y) \right\|^2 \right\rangle \propto d^{2h}(x,y),
$$

where $d()$ is the distance between points $x$ and $y$ and $f()$ is normally distributed.

Peter F. Stadler has reformulated this definition in terms of the correlation $r()$ and auto-correlation function $\hat{\rho}()$:

$$
1 - \hat{\rho}(s) = (1 - r(1))d^{2h}. 
$$

\textit{E.g.} RNA landscapes are known to be fractal according to this definition, when $h \approx 1/2$. [Sta95a]

Peter F. Stadler \textit{et al} have further used \textit{Fourier transform} to analyse fitness landscapes and optimisation operators like \textit{mutation} and \textit{crossover}. According to their results, it seems that the efficiency of operators depend on the landscape \textit{i.e.} the problem to be solved [Wei91, HS98, SW98, SH99].

\textbf{Sorry, this figure is still missing. Please see the figure in the original reference.}

\textbf{Figure 5.1:} An adaptive walk in a rugged landscape \textit{(Adapted from [?](fig. 5)).}

\textbf{5.2.3 Complexity theory}

"In this chapter we take a close look at ... problems that do not seem inherently difficult but for which only exponential-time algorithms are known."

Reingold, Nievergelt & Deo, [RND77]

In this section we briefly review some main theoretical concepts concerning optimisation methods and their complexity from the theoretical point of view.
NP-completeness

Definition 5.2.2 A problem is NP-hard if every problem in NP is transformable to it, and a problem is NP-complete if it is both NP-hard and in NP.

Here NP means the set of problems for which there exists a nondeterministic algorithms that run in polynomial time. The concept nondeterministic in this context is only an abstraction. No physical device is capable of unbounded nondeterministic behaviour—unfortunately. [RND77]

Definition 5.2.3 Hamilton cycle: A closed path visiting each node of the graph exactly once.

Whether a given digraph has a Hamiltonian cycle is proved to be NP-complete. [GJ79]. All the Hamilton cycles can, at least in principle, be generated e.g. by using a simple backtracking algorithm, however.

In complexity theory the term NP-complete problem is used to formally denote decision problems for which the answer is either ‘yes’ or ‘no’. An optimisation problem can be transformed to the framework of NP-completeness by introducing a threshold $B$ and asking “whether there exist a solution with value less than or equal to $B$”. The optimisation problem will be at least as hard as the decision problem, since finding the optimum would answer the corresponding decision problem for every value of $B$. An optimisation problem is said to be NP-hard if its corresponding decision problem is shown to be NP-complete.

The first problem that was shown to be NP-complete was the Boolean satisfiability problem (Boolean SAT): how to assign the variables of a Boolean expression so that the value of the expression is true. [GJ79]

In the 2D case the Hamilton cycle problem in a given lattice area can be seen as a polyomino tiling problem: given an area of connected squares find all possible tilings of (ordered set of) dominoes [Gol94b]. The ordering of the dominoes is needed to keep the chain connected as a single sequence. We will see some polyominoes later in this review.

Currently the perhaps to most efficient method to solve SAT and related problems with Boolean functions is to code them as compact Ordered Boolean Decision Diagrams, which are actually acyclic graphs [?, ?]. The key to efficiency is to find a proper order of the Boolean variables. That is a problem that has been also solved by genetic algorithms [?].

5.2.4 Phase transitions

Lately there has been a growing interest in experimental algorithm analysis. This has been partly because of some experiments that show large change in response to small change of some input parameters. Due to the similarity of these responses to phase transitions of materials they have been named accordingly.

A simple model of phase transitions given by Bernardo A. Huberman and Tad Hogg is as followings [HH87]: Let us consider a (heuristically guided) search process in a tree of depth $d$ (e.g. the number of cities in TSP) and (uniform) branching ratio $b$ (fig. 5.2). The problem is to find a solution by finding a route starting from the root to a leaf and solving the given problem. If we could somehow at each node select the right branch, it would require examining $d$ nodes to find the solution. In the other extreme case when we do not have any information to choose the branch, but trial and error exhaustive search, it would take, on
the average, examine one half of the nodes i.e. \( n \approx b^d/2 \). In a more realistic case we have a method (heuristic) at each node to eliminate the wrong alternatives with probability \( 1 - p \).

The effective branching ratio \( z \) is thus \( bp \). It can be shown [HH87] that the average number of nodes per step \( n = N/d \), when the tree is very deep and \( z < 1 \) is given by

\[
n = \frac{2 - z - p}{2(1 - z)}.
\]

This quantity has singularity (critical point) at \( z \geq 1 \), indicating a sudden transition from linear to exponential complexity. In practice this means that a small change in the effectiveness of the search method has a major impact on the overall performance.

As Bernardo A. Huberman and Tad Hogg point out, this transition behaviour occurs largely independent of the details of the model heuristics just like critical phenomena seem to be independent of the physical details of the materials.

The other point of view to phase transitions is the problem to be solved and its parameters while using fixed search heuristics. A heuristic performing well in most cases might have problems with certain problem cases perhaps characterised by certain parameter values. This kind of behaviour has indeed been found in many problem classes including the fundamental problem class of satisfiability problems [HHW96].

\[\text{Figure 5.2: A search tree (only 3 levels shown).}\]

\[\text{Figure 5.3: Classification of optimisation algorithms}\]

"It is easier to understand the creation of all the celestial bodies and the cause of their motion, in other words the origin of the whole present-day organization of the universe, than to find out by means of mechanics how a little blade of grass or caterpillar appeared"

Immanuel Kant, [?]

Figure 5.3 shows the main criteria used to classify optimisation algorithms: continuous / discrete, constrained / unconstrained and sequential / parallel. There is a clear difference between discrete and continuous problems. Therefore it is instructive to notice that continuous
methods are sometimes used to solve inherently discrete problems and *vice versa*. Parallel algorithms are usually used to speed up processing. There are, however, some cases in which it is more efficient to run several processors in parallel rather than sequentially. These cases include among others such, in which there is high probability of each individual search run to get stuck into a local extreme.

Irrespective of the above classification, optimisation methods can be further classified into deterministic and non-deterministic methods. In addition optimisation algorithms can be classified as local or global.

In terms of energy and entropy local search corresponds to *entropy* while global optimisation depends essentially on the fitness *i.e.* energy landscape.

**Figure 5.3:** The basic classes of optimisation algorithms and some example methods in the continuous-discrete – unconstrained-constrained plane (Adapted from [Gu97]).

In a way we can say that a search method $S$ is a mapping $F_S$, which maps the “wild” or free, unconstrained, distribution of fitness $I_0(E)$ into a distribution $I_s(E)$ of the trial fitnesses generated during the search:

$$I_s(E) = F_S(I_0(E)).$$

The mapping is from the original, usually nearly Gaussian distribution (*cf.* e.g. fig. ??), into a new distribution. We call this mapping the *characteristic mapping* of the method $S$ at problem distribution $I_0(E)$. It is a complex mapping, some properties of which can be used
to characterise the search method and its efficiency, when trying to solve the given problem or problems. In principle it has potential to give a more reliable picture of the average behaviour of the method than \textit{e.g.} the typical evaluation to solution figures, which only give one sample search processing time.

\textbf{Algorithm completeness}

An algorithm is said to be complete if it definitely determines whether an input has a solution or not. A complete algorithm may either \cite{Gu97}

1. determine whether or not a solution exists (existence),
2. give the variable settings for one solution (example),
3. find all solutions or an optimal solution (exhaustive), or
4. prove that there is no solution (theory).

An incomplete algorithm may find a solution in a favourable case, but give up or not terminate in other cases. In the strict sense incomplete algorithms are not algorithms at all. However, in practise we do not have much better methods than a set of more or less incomplete heuristic algorithms when trying to solve the most difficult optimisation tasks either.

For decomposable systems see \cite{?}.

\section{Some popular optimisation methods}

In this chapter we will introduce some of the most popular optimisation methods that might be used instead of GAs to solve certain problems. Remember that according to the famous \textit{“no free lunch”} -theorem, no optimisation method is guaranteed to be good to solve every given problem \cite{WM95}.

\subsection{Deterministic methods}

Methods producing everytime the same result are called deterministic. They can be further classified as exhaustive and non-exhaustive ones. Typically classical numerical analysis methods, such as the famous Newton’s method \cite{PFTV88}, belong to this class. Others include such generally used schemes as depth first (DFS) and breadth first search (BFS), which have got their names after the way they scan the tree of all alternatives \textit{i.e.} the search tree \cite{AHU74}. Other much used deterministic methods include branch and bound, backtracking, and dynamic programming.

As already stated the numerical analytic methods are mainly deterministic and thus they are in serious difficulties when encountering a multimodal optimisation problem. One way to somewhat avoid this problem is to use some smoothing of the object function to transform it into a less multipeaked one. One method of this kind is the diffusion equation method (DEM) \cite{PHP98}. Another method group is multilevel programming \cite{MPV98}. In a way these methods fall quite close to nondeterministics methods. To randomly sample an object function is more or less analogous to smoothing.
CHAPTER 5. *OPTIMISATION METHODS*

**Backtracking**

Backtracking is a recursive way to exhaustively scan all alternatives. The name backtracking comes from the fact that the method, when encountered a dead end, backtracks to the next previous alternative (decision point). *Prolog* programming language is based on backtracking.

**Branch and bound**

Branch and bound (*B&B*) resembles backtracking but uses a measure to cut those paths that can be estimated not to contain the solution.

**Dynamic programming**

Sorry, this chapter is still empty. Working on ...

**Interval methods**

R. E. Moore [Moo66] provides some simple methods to approximate the range of rational functions. The main application area of interval methods was originally, and still strongly is, the control of the rounding and approximation errors of numerical algorithms. There has been, however, some interval activity in optimisation.

One application of interval arithmetics is related to the famous *Kepler’s conjecture* i.e. what is the most efficient way to pack spheres. The known good solution, the *face centered cubic packing*, was proved to be optimal in 1998 by Thomas C. Hales. The 282-page long proof relies on long computer verifications, including interval arithmetics. [?, ?]

**Operations of interval arithmetics**

We will now give a brief overview of interval arithmetics. For further reference of interval arithmetics see e.g. [Moo79]. Moreover, Bierbaum, Schwiertz, and Garloff have gathered bibliographies on interval arithmetics and its applications [BS78, GS80, Gar85].

In the following, we will denote real scalar values by lower case letters (*a*, *b*, *c*, … *z*), while intervals are represented by the corresponding upper case letters (*A*, *B*, *C*, … *Z*). Let *f* be a real function, *x* a real number, and *X* and *Y* intervals. The following notations will be used:

\[
\begin{align*}
    f^*(X) & = \{ f(x) \mid x \in X \}, \\
    \text{mid}(X) & = (\inf(X) + \sup(X))/2, \\
    w(X) & = \sup(X) - \inf(X), \\
    X \cup Y & \quad \text{and} \quad X \cap Y 
\end{align*}
\]

(*5.1*)

(*5.2*)

(*5.3*)

(*5.4*)

*X ∪ Y* and *X ∩ Y* means the union and intersection of *X* and *Y* correspondingly.

Interval operations are defined for ordered pairs of real numbers, which represent closed intervals on the real line. The first element of the pair is the minimum value of the interval, often denoted by *inf*(*X*), while the second element is the maximum, accordingly denoted by
\textbf{5.4. SOME POPULAR OPTIMISATION METHODS}

sup(X). Real numbers \(x \in IR\) can be identified with degenerate intervals \([x, x]\). The basic interval arithmetic operations can be defined as follows:

\[
X + Y = [a, b] + [c, d] = [a + c, b + d], \tag{5.5}
\]
\[
X - Y = [a, b] - [c, d] = [a - d, b - c], \tag{5.6}
\]
\[
X \times Y = [a, b] \times [c, d] = [\min(ac, ad, bc, bd), \max(ac, ad, bc, bd)], \tag{5.7}
\]
\[
X/Y = [a, b]/[c, d] = [a, b] \times [1/d, 1/c], \text{ if } 0 \not\in [c, d]. \tag{5.8}
\]

Division can also be generalized to the case where zero belongs to the denominator [Han78]. Ratschek has shown that it is not possible in any coordinate system to evaluate all operations separately component by component [Rat80].

Interval operations resemble real operations:

\[
0 + X = [0, 0] + [a, b] = [a, b] = X, \tag{5.10}
\]
\[
1 * X = [1, 1] * [a, b] = [\min(a, b), \max(a, b)] = [a, b] = X, \tag{5.11}
\]
\[
X + X = [a, b] + [a, b] = [2a, 2b] = 2X, \text{ but} \tag{5.12}
\]
\[
X - X = [a, b] - [a, b] = [a - b, b - a] \neq 0 = [0, 0]. \tag{5.13}
\]

Interval addition and multiplication are associative and commutative:

\[
A + (B + C) = (A + B) + C,
\]
\[
A \times (B \times C) = (A \times B) \times C,
\]
\[
A + B = B + A,
\]
\[
A \times B = B \times A.
\]

However, interval arithmetics is subdistributive:

\[
A \times (B + C) \subseteq A \times B + A \times C
\]

The distributive law holds only in the case \(B \times C > 0\).

From the subdistributivity it follows e.g. that the Horner scheme is usually better than the power sum form when evaluating interval polynomials.

\textbf{Example 5.4.1} Let \(A = [-2, -1]\), \(B = [1, 2]\) and \(C = [-1, 3]\). Then

\[
AB + AC = [-10, 1] \neq A(B + C) = [-10, 0]
\]

The value of interval function thus depends on its form.
Inclusion monotonicity

The most valuable property of interval arithmetics in numerical applications is its inclusion monotonicity. Let \( A, B, X \) and \( Y \) be intervals such that \( A \subseteq X \) and \( B \subseteq Y \), then

\[
A + B \subseteq X + Y, \tag{5.15}
\]
\[
A - B \subseteq X - Y, \tag{5.16}
\]
\[
A \cdot B \subseteq X \cdot Y, \tag{5.17}
\]
\[
A / B \subseteq X / Y, \quad \text{if} \quad 0 \notin Y. \tag{5.18}
\]

In general, for any interval rational function extension \( F(X_1, X_2, \ldots, X_n) \): we have, if \( X'_1 \subseteq X_1, \ldots, X'_n \subseteq X_n \), then

\[
F(X'_1, X'_2, \ldots, X'_n) \subseteq F(X_1, X_2, \ldots, X_n)
\]

for all values \( X_1, X_2, \ldots, X_n \), for which \( F \) is defined. Thus, for degenerate intervals \( x_1, \ldots, x_n \) we have for \( \forall x_1 \in X_1, \ldots, \forall x_n \in X_n \):

\[
\begin{align*}
f(x_1, \ldots, x_n) & \in F(X_1, \ldots, X_n) \tag{5.20} \\
\text{resulting} & \\
f^*(X_1, \ldots, X_n) & \subseteq F(X_1, \ldots, X_n) \tag{5.22}
\end{align*}
\]

i.e. the interval extension is rather a good upper bound for the range of \( f \) over the given interval hyperbox \( X_1 \times X_2 \times \cdots \times X_n \). It is only in the special case when every variable \( x_i \) occurs textually exactly once, and to the first power in \( f \) that the interval generalization gives the exact range \( f^* \) [Moo79].

Example of interval arithmetics

Let us consider the resistor circuit of figure 5.4. The problem is to find the voltage \( U_1 \) over the resistor \( R_1 \), when the values of the components are:

\[
\begin{align*}
R_1 & = [100, 110] \ \Omega, \\
R_3 & = [2000, 2200] \ \Omega, \\
U & = [9, 11] \ \text{V}, \\
R_2 & = [1000, 1100] \ \Omega, \\
R_s & = [5, 15] \ \Omega \quad \text{and}
\end{align*}
\]

First we must formulate the optimal interval equations. It turns out that the only problem is the formula of the parallelly connected resistors \( R_2 \) and \( R_3 \) for which the total resistance is according to Ohm’s law:

\[
\frac{R_2 R_3}{R_2 + R_3},
\]

where both variables \( R_2 \) and \( R_3 \) appear two times. The formula can be optimized by writing it in an equivalent form:

\[
\frac{1}{1/R_2 + 1/R_3}.
\]
So, the voltage $U_1$ over resistor $R_1$ can be written in form:

$$U_1 = \frac{R_1 U}{R_s + R_1 + \frac{1}{1/R_2 + 1/R_3}}.$$

The interval arithmetic evaluation gives:

$$U_1 = [1.05, 1.57] \, \text{V}.$$

The evaluation of with the unoptimal formula $R_2 R_3 / (R_2 + R_3)$ gives:

$$U_1 = [0.97, 1.70] \, \text{V}.$$

The difference in this case is approximately 40%.

![Resistor Circuit with Tolerances](image)

**Figure 5.4:** A resistor circuit with tolerances.

**Optimal form**

A representation of interval function $F$ is said to be in **optimal form** (or simply **optimal** ) if every interval variable $x_i$ occurs textually exactly once, and to the first power in $F$. The optimal form of interval function $F$ is denoted by $\hat{F}$. \(^2\)

We will now define a couple of concepts to facilitate comparison of different interval forms. Let us denote by $\text{occ}(F, X)$ the number of occurrences of variable $X$ in the interval formula $F$. By $F\{X_1^{n_1}, \ldots, X_m^{n_m}\}$ we denote the set and the number of occurrences ($n_i = \text{occ}(F, X_i)$ and $m = \text{dim}(F)$) of variables used to evaluate $F$. E.g. if $F = A + B^2 + B$, then $F\{\} = A\{\} + B^2\{\} + B\{\} = \{A, B^3\}$. We can further define function $\text{deg}(F)$ be the sum of occurrences of all variables in the evaluation of $F$ i.e.

$$\text{deg}(F) = \sum_{i=1}^{\text{dim}(F)} \text{occ}(F, X_i).$$

\(^2\)The reader may wonder, if optimal form and linear functions are the same thing. Well, they have much in common and the difference between them is not very great, still it exists.
For an optimal form $\hat{F}$ we have simply [Moo79]:
\[ \text{deg}(\hat{F}) = \text{dim}(\hat{F}). \]

**Example 5.4.2** From CAD (optical ray tracing): Let us examine refraction of light on spherical surfaces (see fig. 2). The idea is to use intervals to represent ray bundles instead of normally used idealized infinitely thin rays. In the special case when the centers of the spheres are colinear and the refracting ray intersects this line i.e. the optical axis, the geometrical optic equations for the reflections are (see e.g. [Ala80, Ala82]):

\[
\begin{align*}
    b &= \arcsin(k \sin a), \\
    b' &= \arcsin(g \sin b), \\
    a' &= a + b - b', \\
    x'_0 &= (\sin b' + \sin a')/\sin a' = 1 + \sin b'/\sin a',
\end{align*}
\]

where the incident ray is represented by the linear equation
\[ y = (x - x_0) \tan(a), \]
while the equation of the refracted ray is:
\[ y = (x - x'_0) \tan(a') \]

and
\[ k = (x_0 - r)/r \]
\[ g = n_1/n_2. \]

If we now suppose, that $a$ and $x_0$ are intervals, instead of simple real values, then we have:

\[
\begin{align*}
    a' &= a + b + b' = \{a\} + 2(\{k\} + \{a\}) = \{a^3, x_0^2\}, \\
    x'_0 &= a' + b' = \{a^3, x_0^2\} + \{a, x_0\} = \{a^4, x_0^3\},
\end{align*}
\]

and so $\text{deg}(a') = 5$ and $\text{deg}(x'_0) = 7$, while $\text{dim}(a') = \text{dim}(x'_0) = 2$.

Already these reasonably low $\text{deg}$ values have bad influence on results of interval evaluations of these equations. This is but the begin, because if we continue refractions through next surfaces the situations becomes rapidly even worse. For values $a''$ and $x''_0$ after second refraction we have:

\[
\begin{align*}
    a'' &= \{a^3, x_0'^2\} = \{a^3, x_0^2\}^3 + \{a^4, x_0^3\}^2 = \{a^{16}, x_0^{12}\}, \\
    x''_0 &= \{a^3, x_0^2\}^4 + \{a^4, x_0^3\}^3 = \{a^{24}, x_0^{17}\},
\end{align*}
\]

And after ten (10) refractions, not an unusual case in optics or in high quality geometric modelling ray tracing, we have:

\[
\begin{align*}
    a^{(10)} &= \{a^{131836323}, x_0^{93222358}\}, \\
    x_0^{(10)} &= \{a^{186444716}, x_0^{131836323}\}.
\end{align*}
\]

In iterative process the degradation of accuracy can thus be very rapid, if non-optimal forms must be used. Something can be gained by simplifying expression, but non-optimal
expressions still remain inconvenient: If in the above example we had reduced $a'$ and $x'_0$ so that:

$$a'\{\} = \{a^2, x_0\} \text{ and } x'_0\{\} = \{a^3, x_0^2\},$$

then $a^{(10)}$ and $x_0^{(10)}$ would go down, but still remain as high as:

$$a^{(10)} = \{a^{978122}, x_0^{564719}\} \quad \text{(5.27)}$$

and

$$x_0^{(10)} = \{a^{1694157}, x_0^{978122}\} \quad \text{! (5.29)} \quad \text{! (5.30)}$$

Although being much better than the previous case, they are still astronomically far from optimal.

Lesson from the above lengthy example: never use interval aithmetics in formulas involved in iteration.

### 5.4.2 Interval methods in optimisation

The usage of interval methods in optimisation is based on their inclusion property. Interval methods are usually used with subdivision scheme.
Interval arithmetic implementations

For practical application of interval arithmetics in approximation of more general polynomial and rational functions see [AIH+84, Ala84, Ala85, Ala87, Ala95a].

For more information on interval arithmetics see e.g. [Moo66, Moo79]. Complex valued intervals are dealt with in [PP98] and interval methods in global optimization in [Moo91, Hen97].

5.4.3 Logic programming etc

For general logic programming languages see e.g. Prolog [CM84, Coh88, Bra86].

For a combinatorial optimisation programming language OPL see [Hen99].

Heuristics

A lot of heuristic, i.e. sound tactics, optimisation methods exist. In this group we will—finally/include also genetic algorithms.

5.4.4 Stochastic methods

“stochastic [from the greek stochastikos: lit., skillful in aiming; also stochazesthai: to aim at]”

Webster’s New World College Dictionary, [?]  

Most of the successful methods to solve large and difficult multimode optimisation problems are stochastic i.e. non-deterministic. Also the methods focused on in this tutorial belong to this class.

Random walk

The simplest non-deterministic search method is pure random walk, but it is also known to be extremely inefficient, once there are some regularities to be utilised. It is obvious that also the non-deterministic part should have some memory about where it seems to be beneficial to do some local search trials. In this respect a careful stochastic analysis would be the ideal way to judge the most promising search areas. This is a recommended approach for linear or nearly linear systems with stochastically independent variables. Unfortunately there are usually also quite many variables, for which usually only a limited number of observations is available. In addition the most advanced statistical methods tend to be quite complex. It is therefore uncommon to see statistical methods, such as experimental design, or analysis of variance (ANOVA), heavily used in search problems even if, in theory, they should have some potential.

There are however methods that try to do some sort of simple stochastic analysis of the most promising search areas. The most straightforward and obvious is to concentrate on areas where already good solutions have been found. This can be done by returning to already found solutions and continuing the search non-deterministically from those points. These are called restart methods. It should be obvious that this type of method is potentially superior
5.4. SOME POPULAR OPTIMISATION METHODS

to the simple random walk, which soon forgets entirely where it has already encountered good solutions.

There are also some methods that do utilise memory in a bit more developed way like simulated annealing [MRR+53], tabu search [Glo89, Glo90, ?], genetic algorithms [Hol73], and especially neural networks [Koh89]. These will be briefly reviewed next.

Example 5.4.3 A gedanken experiment: Our Sun shines because of nuclear fusion processes running in its core. When a nuclear fusion happens it produces high energy photons. These photons are then traveling with the speed of light and they are scattering every now and then—actually quite often. In terms of computing these photons are doing search by random walk by which they finally reach the surface of the sun after having lost most of their energy on the way up to the surface of the sun. The question is now: how long does it take for these photons or their descendants to reach a sunny beach on our Earth? It can be estimated that it takes about one million years or something like that. Why on earth such a long time for the photons traveling at the speed of light? That is because random walk is a very inefficient way to find the surface. The same applies to any search algorithm that is basically random walk. Think twice before using it, wheather or not the sun is shining.

Another gedanken experiment with the Sun: Let us assume that there is somewhere in the universe a switch that can switch off the photons inside the Sun. Again we ask: how long does it take until we on Earth can see the result of switching off the sun by looking for the photons emergencing from the sun? This time the answer is in the order of what you might have, like most of us, in mind for the previous question: ten minutes or something. Are the photons now using a more efficient search method to find the surface of our Sun? Definitely not, because: The sun is in an equilibrium in which gravity is compressing it while the pressure by the photons inside it works in the opposite direction. When the photons are switched off then the gravity will implode the sun. This happens very fast until the core of the sun would be so dense that new nuclear processes will start and the sun will explode. This would be clearly seen not only on Earth, but even by observers in other solar systems—even in the near-by galaxies. The sun would have exploded as a supernova—luckily this was only a gedanken experiment. The lesson learned for search algorithms is that in this latter case there is a global function, the gravity, that is scanning the whole search space at the speed of light without any scanning i.e. random walk and the particles are falling in the gravity field literally in parallel to each others.

Simulated annealing

Simulated annealing or thermodynamic or Metropolis algorithm [MRR+53] has a build-in variable $T$, temperature, which guides selection of new search directions: at the beginning the temperature has a high value thus allowing practically all alternatives, for the long run it is lowered and less and less possibilities are left until, if we are lucky, the global optimum has been encountered. In practice simulated annealing methods give quite good solutions, especially if it is rerun several times. In theory it solves any optimisation problem. But as the reader might guess, theory and practice have a slight difference, however small. In this case it is the time needed to find the optimum, which is inversely proportional to the temperature lowering rate. Unfortunately to guarantee the optimum solution, the temperature should be lowered extremely slowly! In any case simulated annealing is much used to solve complex optimisation problems either as such or combined e.g. with well performing heuristics.
CHAPTER 5. OPTIMISATION METHODS

The simulated annealing algorithm applied to a minimisation problem can be as shown in figure 5.6 [?].

- **initialisation**: generate starting trial \( C_s \)
- **random move**: generate a new trial \( C_i \) by randomly perturbing \( C_s \)
- **energy**: evaluate \( \Delta E = E_{C_i} - E_{C_s} \)
- **acceptance**: if \( \Delta E \leq 0 \) or \( N(0, 1) < \exp(-\Delta E/RT) \), then set \( C_s = C_i \)
- **termination**: if solution found then stop
- **iterate**: go back to step 2,

where \( N(0, 1) \) is the normal distribution \((\mu = 0, \sigma = 1)\).

**Figure 5.6**: Simulated annealing (Adapted from [?].)

Gary A. Huber and J. Andrew McCammon have used a population based simulated annealing method they call weighted-ensemble annealing (WEA). They have experimented with several population sizes \( n_P \) and it seems that \( n_P > 200 \) gives the highest rate of search success. [?]

For a Markov chain analysis of SA see e.g. [DSC98].

Tabu search

Tabu search, as its name already suggests, tries to avoid re-evaluating already evaluated trials. In order to do that it maintains a list of already evaluated points and checks this list while generating new trials. It is obvious that by doing this prepruning the tabu search gains some extra efficiency, while obviously having to pay something for the managing of the tabu list. [Glo89, Glo90]

Neural networks

While genetic algorithms try to mimic evolution, neural networks (NN) try to model human (or animal) brain functions, at least for the lowest levels. Like genetic algorithms NNs are also based on a few simple operations one of which is nonlinear thresholding. The resulting behaviour looks surprisingly intelligent. Neural networks have memory and they can recognise repeatedly encountered patterns. This pattern recognition property has been utilised in many technical information processing problems, including many prediction and pattern recognition problems. Because of this pattern recognition property, neural networks have potential in complex optimisation. In this tutorial neural networks are mentioned only because of completeness, however. An interested reader is suggested to get familiar with neural network methods e.g. in [Koh89].
5.4. SOME POPULAR OPTIMISATION METHODS

Monte Carlo hill climbing

The Monte Carlo methods used in optimisation are actually doing stochastic hill climbing also closely related to simulated annealing (SA) or Metropolis algorithm. Stochastic resonance [GHJM98] can also be linked to this group. It is known to reveal faint signals.

5.4.5 Replica method

It has turned out that the replica method is actually a simple but powerful method to model complex properties of systems ranging from such diverse areas as spin glasses to neural networks. The principle of the method is as follows: compute the average of $f$ by some analytic continuation procedure from the average of the partition function of $n$ uncoupled replicas of the initial system. [MPV88]

5.4.6 Renormalisation group method

Kenneth G. Wilson has generalised the renormalisation group\(^3\) transformation method (RG) introduced already in 1940’s as part of the development of quantum electrodynamics and further developed in the 1950’s by several workers, among them Murray Gell-Mann and Francis E. Low, who proposed a version that is now called the original renormalisation group [Wil79].

The idea of renormalisation is to subdivide the problem into small ones and then to rejoin the solutions in a certain way as the solution of the original problem. This is actually the way problems are often solved when using computer programming: a large problem must be subdivided into a small (trivial) subproblems, which are then joined somehow into the solution of the complex problem. RG also resembles wavelet transforms used in signal processing and vice versa wavelets have been used instead or with RG in analysis of critical phenomena [BS95]. One can further easily recognise features in common to RG and fractals [Man83].

The renormalisation group method as introduced by Kenneth G. Wilson consists of three steps and applied to a lattice type problem is shown in figure 5.7.

1. **subdivide** lattice into blocks of a few nodes each
2. **evaluate** the blocks
3. **rescale** the result of the above steps and imbed it in a new lattice having the blocks as nodes
4. **iterate** from step 1 until the result is precise enough or the given processing time elapsed

**Figure 5.7:** Renormalisation group method (Adapted from [Wil79]).

The real significance of the renormalisation group method has been in evaluation of the properties of physical systems arbitrarily close to the critical points, where the range of interactions grows without a limit. No other practical and direct method to analyse such cases has been available until recently [Wil79].

After the first success [Wil75]\(^4\) there was a series of failures when applying Wilson’s RG method. What went wrong? It seems that S. R. White and R. M. Noack have found the reason

\(^3\)actually a semigroup, because there is no inverse transformation

\(^4\)solution of Kondo problem
and luckily also a remedy to these unexpected and somewhat depressing results. Namely they show that at least in some cases good results can be gotten by selecting proper boundary conditions for the blocks. [WN92]

5.5 Quantum computing

During the last few years quantum computing has raised much enthusiasm among both computer scientists and physicists. In quantum computing the basic information unit is called qubit. Like the classical bit is holds one bit of information, but can be in any superposition of 0 and 1 bits, while the classical bit can be in either 0 or 1 state, any other value causing normally a fatal error. The zero qubit value is often denoted by the quantum mechanical symbol $|0>$, which is actually a so called ket vector, a normalised vector of a complex Hilbert space and $<0|$ is correspondingly called bra vector. The one bit is correspondingly denoted by vector $|1>$. These basic bit vectors are orthonormal (dot product):

$$<i|j> = |i><j| = \delta(i,j),$$

where $\delta(i,j) = 1$, if $i = j$, 0 otherwise. A qubit $|q>$ can be in any superposition of $|0>$ and $|1>$:

$$|q> = a|0> + b|1>,$$

where $a, b \in C$ so that $<q|q> = 1$ i.e. $|a|^2 + |b|^2 = 1$. This inherent information parallelism of qubit gives quantum computing its unique processing potential. A qubit register of $n$ qubits is able to represent simultaneously upto $2^n$ bit combinations i.e. all possible bit combinations that can be represented by $n$ bits. This gives quantum computing potential to challenge the combinatorial explosion and thus might solve problems that are in the NP class. The most important drawback of quantum computing is that the result of computation can be revealed only by making physical measurements, which destroy the superposition by projecting the result on the basic vectors. Due to the statistical nature of quantum mechanical measurements, to reveal the coefficients $a$ and $b$ above we need repeated measurements and corresponding statistical analysis. The quantum information processing system must be isolated from the environment during the whole processing interval. E.g. thermal noise can destroy quantum information. This makes practical quantum processing really challenging. Until now processing using upto 7 qubits have been demonstrated in laboratory [Suo02]. One implementation possibility is to use photons, which are quite immune to environment. The coding can be e.g. such that horisontally polarised photons represent $|0>$, while vertically polarised photons represent $|1>$. As is known from optics, a photon can be in any superposition of horisontal and vertical polarisation. When measuring such a superposition we observe always only either horisontally or vertically polarised photons. It is only their relative number that reveals the superposition coefficients.

An example of information coding somewhat resembling that of quantum computing can be found in basic physiological optics, color vision. Any purple hue between red and blue can be represented by a CRT by using only two basic spectral lines, red and blue, let use denote them $|red>$ and $|blue>$, as a superposition

$$|purple> = a|red> + b|blue>,$$
5.5. QUANTUM COMPUTING

where \( a, b \geq 0 \). If we now make a spectroscopical analysis (measurement) of \(|\text{purple}\rangle\) we are only able to detect either or both the basic lines \(|\text{red}\rangle\) and \(|\text{blue}\rangle\) but nothing in between them. The coefficients \( a \) and \( b \) can be revealed by measuring the intensities of the basic lines and that presupposes more than a few photons falling on our detector. The difference between hue and qubit is that in qubit both “lines” are coded by the same photon while in our semiclassical example they are coded by different photons and thus obey classical logic.

5.5.1 Solving SAT and TSP using QC

This section is based on the work done by Tad Hogg and Dmitriy Portnov, who have applied simulated quantum computing to solve two famous search or optimisation problems namely overconstrained satisfiability (SAT) and asymmetric traveling salesman problems (TSP). Both problems are known to be NP-hard i.e. the time required for the solution grows exponentially with the number of data points, because no known methods exists to solve them in polynomial time. Both problems are heavily used as test problems in optimisation studies.

The approach taken in the study by Tad Hogg and Dmitriy Portnov is as follows: The quantum algorithm mixes amplitudes among different states of qubits so that the bulk of amplitude gradually shifts towards states with relatively low costs. Tad Hogg has previously applied the same technique to a decision problem [Hog00]. The approach, like e.g. genetic algorithms, is a heuristics i.e. there is e.g. no guarantee that it will really end up with the global optimum. In more details the algorithm is shown below [HP00]. For further details see [HMPR99].

1. **Initialisation:** initialise the amplitude equally among the states, giving \( \psi_s^{(0)} = 2^{-n/2} \) for each of the \( 2^n \) states \( s \).

2. **Mixing:** for steps 1 through \( j \), adjust amplitude phases based on the costs associated with the states and then mix them. These operations correspond to matrix multiplication of the state vector, with the final state vector given by:

\[
\psi^{(j)} = U^{(j)} P^{(j)} \ldots U^{(1)} P^{(1)} \psi^{(0)},
\]

where, for step \( h \), \( U^{(h)} \) is the mixing matrix and \( P^{(h)} \) is the phase matrix, both to be described in more detail in the text.

3. **Fitness:** measure the final superposition, giving state \( s \) probability \( p(s) = |\psi_s^{(j)}|^2 \).

Compare this to the genetic algorithm shown in figure 6.1. There are some obvious similarities.

The mixing matrix \( U^{(h)} \) in the above quantum algorithm is defined as follows:

\[
U^{(h)} = W T^{(h)} W,
\]

where

\[
W_{rs} = 2^{-n/2} (-1)^{|r \land s|}
\]
is the Walsh transform and $s$ and $r$ refers to states. The expression $|r \land s|$ denoted the number of 1-bits the states $s$ and $r$ have in common. The matrix $T^{(h)}$ is diagonal with elements depending on $|s|$, the number of 1-bits that the state $s$ contains:

$$T^{(h)}_{ss} = t^{(h)}_{|s|} = e^{i \pi \tau_h |s|},$$

where $\tau_h$ is a constant depending on the class of problems and the number of steps. Observe, that it is not dependent on the particular problem instance being solved. From the above definitions it follows that the elements $U^{(h)}_{rs}$ depend only on the Hamming distance $d(s, r)$ between the states i.e.

$$U^{(h)}_{rs} = u^{(h)}_{d(r,s)} = (-i \tan(\pi \tau_h / 2))^{d(r,s)}.$$

Hamming distance $d(s, r)$ is number of bits differing between $s$ and $r$. The problem to be solved is implemented as the phase adjustment matrix $P^{(h)}$, which is a unitary diagonal matrix determined by the costs $c(r)$ associated with each state $r$:

$$P^{(h)}_{rr} = p^{(h)}_{c(r)} = e^{i \pi \rho_h c(r)},$$

where $\rho_h$ is a constant. The cost $c(r)$ of the search state is a measure of the fitness of the solution represented by $r$ i.e. it implements the objective function of the optimisation problem.

Observe that for optimisation problems the minimum cost is not usually known a priori, and thus amplitude amplification can not give any enchancement.

In order to be able to run the proposed quantum search algorithm the following values must be defined: $j$ the number of steps, and the phase parameters $\tau_h$ and $\rho_h$ for $h = 1, \ldots, j$. Tad Hogg and Dmitriy Portnov give two principal ways of optimising these parameters in addition they refer also to quantum computing as one alternative to do rigorous optimisation. From the view point of genetic algorithms the optimisation of these parameters would be at least worth considering. Here one must look the problem of algorithm fine tuning also from the view point of computing. Generally the optimisation of optimisation algorithm may sound great but the danger is that it may lead to a problem, well known in machine learning, called overlearning. It means that the finetuned algorithm is extremely fast to solve the problem that it was finetuned to solve to—the problem is that for other problems it may perform much worse.

Tad Hogg and Dmitriy Portnov have finetuned the above parameters and solved two famous problems, three variable satisfiability (3-SAT) problem and a six city asymmetric traveling salesman problem (TSP).

The SAT problem means the following. Given a Boolean expression of $n$ variables $B(v_1, \ldots, v_n)$ assign values true and false to all $v_i$ so that $B(v_1, \ldots, v_n) = true$. This problem seems to be quite easy. However, the problem is that the number of cases is $2^n$, which grows exponentially and is intractable for values exceeding roughly fifty. There is not any known algorithm that can solve general SAT problems in polynomial time. Actually, it can be shown that any NP complete problem can be transformed into a SAT problem in polynomial time. Thus if one succeeds to solve SAT in polynomial time, also other NP complete problems can be solved in polynomial time. If quantum computing can dramatically speed up processing of SAT problems, then it can speed up the processing of all NP complete problems. The perhaps best way of solving Boolean SAT problems computationally is to
code Boolean expressions as Boolean Decision Diagrams, which are efficient to process by traditional computer algorithms [?].

Tad Hogg and Dmitriy Portnov solve the so-called 3-SAT problem, which is the following: Given a set of Boolean expressions $B_i(v_{i1}, v_{i2}, v_{i3})$ of three variables $v_{i1}, v_{i2},$ and $v_{i3}$ selected for each expression independently from a set of $K$ variables, $1 \leq i \leq N$ and $1 \leq i1, i2, i3 \leq K$. Find the assignments for variables $v_i$ so that the number of true expressions $B$ is maximum. It can be shown that also this SAT problem class is NP complete. The results obtained by Tad Hogg and Dmitriy Portnov when solving rather difficult ($N = 80, K = 20, \mu = N/K = 4$) random samples of 3-SAT problems is comparable to, actually even somewhat better than those got using a known efficient local search heuristics GSAT [SLM92]. However, it is possible that part of the good performance is due to overlearning. That is difficult, if not impossible, to judge based solely on the information given in [HP00].

The six city asymmetric TSP problem is really a modest test problem. The state of the art in this field has been to attack problems consisting of millions of cities using linear programming during already over a decade [?]. In any case the results got were less striking for TSP than for the 3-SAT problem.

Because both problems are frequently used in testing search and optimisation methods, it is not any surprise that also the author has used them in some tests of genetic algorithms [Ala92b, Ala92d, Ala99]. The results seem to indicate that a TSP type problem where the permutation of parameters is essential is difficult for genetic algorithm based optimisation while a 3-SAT type problem is considerably easier. Actually most of the problem cases of SAT problems are even trivial to solve by just counting the number of occurrences of variables and their negations, when the Boolean expressions $B_i$ are disjunctions. It is near the phase transition value $\mu \approx 4.25$, where the really difficult degenerate problem instances can be found. By phase transition we mean here a dramatic change in the average difficult of a problem. While most SAT problem cases seem to be easy, the cases near the phase transition tend to be extremely difficult causing exponential processing time consumption [HH87].

5.6 Outline for a search algorithm

Before going to the implementation and experimentation with GAs we will give briefly some guidelines to design a general search algorithm i.e. a global optimisation algorithm based on the properties of the fitness landscape and the most common optimisation method types:

- **determinism**: A purely deterministic search may have an extremely high variance in solution quality because it may soon get stuck in worst case situations from which it is incapable to escape because of its determinism. We can try to avoid this, but it is a well known fact that the observation of the worst case situation is not guaranteed to be possible in general.

- **nondeterminism**: A stochastic search method usually does not suffer from the above potential worst case “wolf trap” phenomenon. It is therefore likely that a search method should be stochastic, but it may well contain a substantial portion of determinism, however. In principle it is enough to have as much nondeterminism as to be able to avoid the worst case wolf traps.
• **local determinism**: A purely stochastic method is usually quite slow. It is therefore reasonable to do as much as possible efficient deterministic predictions of the most promising directions of (local) proceedings. This is called local hill climbing or greedy search according to the obvious strategies.

Having the above general properties of search methods in mind we are now ready to have a closer look on GAs.
Chapter 6

Genetic algorithms

6.1 Basics of genetic algorithms

The principle of GAs is simple: imitate genetics and natural selection by a computer program.

In spite of the seemingly simple processing, the GAs are known to be good in solving some problems that are known to be computationally hard. Some apparent good and bad properties of GAs are shown in table 6.1.

<table>
<thead>
<tr>
<th>good news</th>
<th>bad news</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple</td>
<td>massive processing</td>
</tr>
<tr>
<td>parallel in nature</td>
<td>no convergence proofs</td>
</tr>
<tr>
<td>robust</td>
<td>stochastic</td>
</tr>
<tr>
<td>general</td>
<td>not well known theoretically</td>
</tr>
<tr>
<td>works in nature</td>
<td>relative new in technology</td>
</tr>
</tbody>
</table>

Table 6.1: Main good and bad features of genetic algorithms.

The parameters of the problem are coded most naturally as a DNA-like linear data structure, a vector or a string, in the simplest case as a vector of bits. Sometimes, when the problem is naturally two or three dimensional also corresponding array structures are used.

A set, called population, of these problem dependent, parameter value, vectors is processed by GA. To start we usually have a totally random population, the values of different parameters generated by a random number generator. Typically the population size range from a few dozens to thousands, the “magic” number 50 being the most popular and usually quite nicely working, too. An analysis of population size as a function of problem complexity is given e.g. in [Ala92f, Jul96].

To do optimization we need a cost function or fitness function as it is usually called by genetic algorithms users. By a fitness function we can detect the best solution candidates from the population and delete the not so good “specimens”.

The nice thing when comparing GAs to other optimization methods is that the fitness function can be nearly anything that can be evaluated by a computer or even something that cannot! In the latter case it might be a human judgement that cannot be stated as a crisp program, like in the case of eye witness [CJ91], where a human being selects among the facial
1. **Initialisation** create a (random) starting population and evaluate fitness of each individual

2. **Recombination** recombine best individuals

3. **Evaluation** evaluate fitness

4. **Selection** select best ones for the next generation

5. **Termination** if solution found stop, otherwise continue at step 2

![Figure 6.1: A genetic algorithm.](image)

images generated by a GA [Han00]. So, there are not any definite mathematical restrictions on the properties of the fitness function: it may be noncontinuous, discrete, multimodal etc.

### 6.1.1 Genetic operators

The information i.e. parameter values are processed in a GA by the selection and recombination operators. The recombination means that we take, usually two, specimens and recombine their information by an operator called crossover. It has a counterpart in biology where the chromosomes do crossover, usually at a couple of locations. By this simple mechanism new combinations of the existing building blocks are generated randomly.

There is usually also mutation involved in recombination. Its role is to create new building blocks. The role of mutation is important, but the crossover is usually much more important because it utilized effectively existing good building blocks. The mutation usually only spoils them, except every now and then when it helps to escape local optima.

The basic GA algorithm is shown in figure 6.1. There are differences how to implement the different steps and details, but it seems that GAs are quite immune to most implementation details and parameters, however [Ala92e].

#### Recombination

The recombination of parameter values is done using the so called crossover (cf. 6.2), which has as its model the corresponding crossover of chromosomes in breeding organisms. Usually two chromosomes (parents) are more or less randomly selected from the population and their parameter are shuffled randomly using crossover operation. This results in two new trial combinations of parameters (offspring), of which one or both are taken to the population. Computationally recombination can be done e.g. by indexing (C-language):

```c
for (i=0;i<n; i++) B[k][i] = P[(X[i]?p1:p2)][i];
```

where \( B \) is the new and \( P \) is the old trial population. \( X \) is the so called recombination index vector (elements randomly generated 0 or 1), \( i \) the (gene) index, \( p1 \) and \( p2 \) parent indices, \( k \) offspring index and \( n \) the length of the chromosome i.e. trial vector. Often also the other offspring, a kind of complement of the first one is generated with only a minimum extra work:
6.1. BASICS OF GENETIC ALGORITHMS

In practise a separate $B$ array is not necessary, because the result will be put back to the population array $P$. The best new trials replace the worst old trials.

In nature the chromosomes are formed by crossover, but there is usually only a couple of crossover points, while in genetic algorithms there are usually many crossover points. In the above case the crossing probability was 0.5 and which is called uniform crossover [Sys89].

The reason for this discrepancy is that in nature the evolution seems to be quite slow, while in computing we aim at as fast evolution as possible. Engineering has clear directions and ultimate goals while evolution has not. To scan a high number of diverse alternatives as fast as possible presupposes a high crossover rate.

When using crossover, or any other random operator, we should take into consideration the possibility, that the result of the operator might not be a valid trial. E.g. when solving permutation type problems, like the famous traveling salesman problem, a random recombination can create an invalid tour by deleting and/or duplicating some cities. When solving problems like these the encoding of the problem information and the genetic operations are essential for efficient processing. Evaluation of invalid trials is simply a waste of time. Crossover operators for permutation type problems are given e.g. in [?, Kli93, ?, PC95, CCL96, ?, ?].

**Mutation**

The other vital genetic operation is mutation. It is used to inject randomness and diversity to population. Without mutation selection soon prunes all but the building blocks of the most fit specimens and the search will slow down and eventually stop at a local extreme.

It is a common mistake that evolution is primarily based on mutation. There are many processes in living cells to repair mutations. Only a few mutations will actually survive and most of them are fatal to the specimen. The striking feature of genetic information is its conservation, not evolution. Actually DNA is by far longer lasting memory than any created by man [Cox01]—including even ancient petroglyphs. Figure 6.3 shows a leave of *Ginkgo biloba* L. maidenhair tree, a living fossile tree, similar fossil trees dating back 270 million years.
years, about 70 million years before the Jurassic period.

In GA low mutation rate will slow search, which has a high probability to stuck to a local extreme. A too high value of mutation rate will prevent search from finding the solution by “breaking” the building blocks too fast. Information contained in good trials is lost by destroying noise. Claus O. Wilke et al. have analysed mutation rate vs. the properties of the fitness landscape and reproduction efficiency. It seems that lower mutation rate is more effective than higher reproduction rate. The obvious reason for this at first sight controversial result is that low mutation rate provides a more solid ground i.e. lower risk for search. [WWO+01]

Figure 6.3: A rusty leave of maidenhair tree (Ginkgo biloba L.), a living fossil tree having similar fossil ancestors dating back 270 million years.

Randomness & noise

Genetic algorithms heavily rely on randomness. Mutations are actually simply random numbers, or bit combinations, on a given interval. Basically randomness is strange to digital processing, which tries to avoid randomness so that a given computation always gives the same result. How to have randomness in digital processing? There are the following basic ways to realise random numbers:

- a precomputed table of random numbers,
- a simple computational algorithm to create random numbers, and
- a physical process like radioactive decay or simply thermal noise.
All are poble with GAs and also used. The most convenient way is to use a simple random number generator. Most programming languages and/or packages provide such.

Let us see one such generator type called congruence generator. It is based on multiplication and modulus operations on integer numbers:

$$X_{n+1} = (aX_n + b) \mod m$$

By selecting suitable integer constants $a$ and $b$ more or less good random number sequence $X_i$ results (see Fig. 6.4). However, the properties, like the length of the sequence before stating over again, of this sequence really depend on the parameters. This kind of generator gives a fine number of samples. When a number that already has appeared in the sequence appears then the sequence will be repeated exactly. That is why these generators are called pseudorandom number generators because the sequence is defined once the parameters are given. In general, it is good to have some consideration of the random number generation method used when designing a stochastic algorithm like GA.

![Random number distribution for a linear congruence generator](image)

Figure 6.4: Random number distribution for a linear congruence generator $R = (37181 \times R + 7117) \mod U$ using random numbers $N = R \mod 100$, where $U = 1234567$. Y-axis scale means percentage when compared to a uniform distribution (solid line). 1,000,000 samples were drawn.

When using hardware, like FPGA, a simple way to generate random bit sequences is to use linear feedback shift register LFSR, which consists of a shift register and a few XOR-gates (fig. 6.5). It is really simple to generate pseudorandom bit sequences with digital hardware.

![A LFSR to produce random bit sequence.](image)

Figure 6.5: A LFSR to produce random bit sequence.

The third alternative is to use e.g. thermal noise to create truly random numbers. However, also here we need some analysis of the distribution of the numbers actually generated.
by the underlying physical process. In a way this way is the most natural and in parallel to the stochastic processes involved in natural evolution. In engineering terms random processes are often called simply noise.

**Fitness function**

The basic genetic algorithm is simple and easy to implement. The real challenge is usually to implement the *fitness function*, which practically contains all the knowledge of the problem to be solved. In many cases there already exists functions or *simulation* routines that can be used to implement the fitness function. The role of the fitness function is to describe the trial quality by one number, usually a float point number. In case of multi-objective or *Pareto optimisation* there is not any one number to describe the quality but several different qualities. The user is supposed to choose the proper quality combination that best fit to his/her purposes. For further references on Pareto optimisation see bibliography [Alaf].

**A GA glossary**

The terminology of genetic algorithms was inspired by biology. In order to facilitate understanding of various concepts, a brief glossary of the most frequent terms used in the context of genetic algorithms is provided in Table 6.2. As can be seen, most of them have familiar equivalent engineering or mathematical terms.

<table>
<thead>
<tr>
<th>GA term</th>
<th>computing/math term</th>
</tr>
</thead>
<tbody>
<tr>
<td>allele</td>
<td>value of parameter</td>
</tr>
<tr>
<td>chromosome</td>
<td>usually equal to specimen</td>
</tr>
<tr>
<td>fitness</td>
<td>value of function; cost function</td>
</tr>
<tr>
<td>gene</td>
<td>one parameter of solution</td>
</tr>
<tr>
<td>generation</td>
<td>one iteration round</td>
</tr>
<tr>
<td>genotype</td>
<td>problem parameter values</td>
</tr>
<tr>
<td>phenotype</td>
<td>result of fitness function evaluation</td>
</tr>
<tr>
<td>population</td>
<td>vector of trials</td>
</tr>
<tr>
<td>specimen</td>
<td>trial <em>i.e.</em> problem parameter values</td>
</tr>
</tbody>
</table>

**Examples and success stories**

Here we will briefly look what kind of applications has been solved using genetic and evolutionary computation. Genetic and evolutionary algorithms are at their best biotope when solving combinatorial problems for which there are not any know good method known. Also in such cases when there already is a good method, it might be possible to get even a better method by combining the two approaches.

WORKING ON ....

A resistor circuit example ...
6.1.  BASICS OF GENETIC ALGORITHMS

General structure
The simplest form of genetic algorithm involves three types of operators: Selection, Crossover and Mutation [Gol89].

Coding and Operations
The problem to be solved by a genetic algorithm is encoded as two distinct parts: the genotype called the chromosome and the phenotype called the fitness function. In computing terms the fitness function is a subroutine representing the given problem or the problem domain knowledge while the chromosome refers to the parameters of this fitness function.

Chromosome  Traditionally the genotype is coded using a programming language vector, array, or record-like chromosome consisting of the problem parameters. Binary (integer) and real (floating point) codings are the most frequently used basic data types to represent genes in this immediate coding approach.

Here a more indirect and general data structure will be used. The chromosome consists of genes that are pointers to valid values of the gene i.e. alleles in biological terms. This indirect gene value structure is better suited especially for combinatorial problems than the commonly used immediate coding scheme. It allows to represent efficiently arbitrary allele sets as will be see in our introductory examples, where standard resistance values are used as alleles. In the indirect coding there is a vector of possible gene values the gene is actually pointing to (Figure 6.6). In our example of a genetic algorithm (Figure 6.7) the gene value is an index of the allele array containing all possible values of the gene.

Fitness function  The purpose of the chromosome is to provide information, parameter values, for the problem encoded as a fitness or cost function, the phenotype. Usually the user needs only to worry about the fitness function and its implementation and to select reasonable parameter values, like population size, for the core genetic algorithm. Here the fitness function is simply the distance from the goal.

Mutation  The basic genetic operation is mutation. It means that the gene value i.e. allele is replaced by another, usually a random value. In our indirect coding scheme the gene is assigned a random valid index value. A mutation operator is easy to implement using any well behaving
void toyGA(int generations)
{
    int i,j,k; // indexes
    Gene[] S0 = newChromosome(Population[0]),
            S1 = newChromosome(Population[1]);
    for (i=0; i<Population.length; i++)
        for (j=0; j<Population[i].length; j++)
            Population[i][j].mutate(UX);
    for (k=1; k<=generations; k++) {
        i = 0;
        while (i<Population.length) {
            // 25% probability for mutation
            if ((UX.next(4)==0) ||
                (i==(Population.length-1)) ) {
                mutate(Population[i]);  i++;
            } else { // do crossover:
                crossover(Population[i],
                           Population[i+1],S0,S1);
                selectionByTournament(Population[i],
                                       Population[i+1],S0,S1);
                i+=2;
            }
        }
        if (i<Population.length) {
            mutate(Population[i]);
        }
    }
}

Figure 6.7: A toy genetic algorithm core toyGA. UX is a random number generator object.
random number generator able to generate valid gene values. In our indirect scheme the values must be in the range \([0, n_i - 1]\), where \(n_i\) is the size of the allele vector.

**Crossover** Crossover is a more complex genetic operator that combines two chromosomes (parents) into new ones by swapping genes of the parents randomly. The most common crossover types are one-point, two-point, and uniform crossovers. In one- and two-point crossovers there are one respective two points where the roles of genes are changed in the swapping while in the uniform crossover the probability to choose a gene from either parent is equal to 0.5. For most problems the uniform or multipoint crossover results in faster convergence than the more conservative few-point crossovers.

**Selection** Charles Darwin’s great and far reaching observation was that due to limited resources there is a continuous hard selection process among the living organisms in nature. This selection combined with genetic heritage inevitably causes gradual evolution that finally creates astonishing new organisms. In genetic algorithms the nonlinear selection is the crucial operator to maintain a search of better solutions in those points of the search space where the best solution candidates have been found so far. In other words selection is screening the search space and thus accumulates information of the most useful search areas and thus the building blocks i.e. parameter values of the best solutions. It is assumed that by combining parts of good solutions, building blocks, still better solutions can be found. If this building block hypothesis is valid, genetic algorithm is a reasonable approach to solve a given problem. It is commonly believed, based mainly on the success of genetic algorithms in solving practical problems, that most of the practical optimisation problems more or less satisfy this building block hypothesis.

**Population** A genetic algorithm maintains a set of trials called population. It is usually implemented as a fixed length vector of chromosomes. A popular population size is \(n \approx 50\), which is often a reasonable compromise between fast processing and premature convergence risk. A round updating the population array is called generation. It is also possible to update the population incrementally as shown in our toy example.

**An implementation**

The most important parts of genetic algorithms have been described. It is now time to make a synthesis, to reveal our simple genetic algorithm example core called `toyGA` written in Java\(^T\!M\)\(^1\), without the output routine calls and a couple of simple subroutines, used to solve the toy problem shown in Figure 6.7:

First a random initial population is generated by mutating every gene of every chromosome. Chromosomes are stored in the `Population` array.

After this in every generation either mutation (25%) or crossover (75%) operations are applied to each member of the population. Crossover is done between the neighbouring chromosomes. Tournament selection is used to select members for the next generation: the parent chromosome(s) are replaced by the best of the original chromosomes and the new ones created after each operation.

\(^1\)Java is a trademark of Sun Microsystems, Inc.
Table 6.3: The classes used in our examples. The source codes can be found in ftp.uwasa.fi/cs/report2003/...

<table>
<thead>
<tr>
<th>class</th>
<th>contains</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>random number generators</td>
</tr>
<tr>
<td>Gene</td>
<td>the allele structure of gene</td>
</tr>
<tr>
<td>GeneticAlgorithm</td>
<td>the genetic algorithm core</td>
</tr>
<tr>
<td>Resistor</td>
<td>simple resistor circuit</td>
</tr>
</tbody>
</table>

*toyGA* is actually one method of class called *GeneticAlgorithm*. The classes used in our examples are shown in Table 6.3.

**A toy example**

To demonstrate how a genetic algorithm functions it is applied to a toy problem shown in Figure 6.8: connect four resistors $R_i \in \{10, 20, 40\}$ Ω serially so that the total resistance $R_{\text{tot}} = \sum_{i=0}^{3} R_i$ is as close as possible to a given value $R_{\text{goal}}$. The natural fitness function for this problem setting is $f = -|R_{\text{goal}} - R_{\text{tot}}|$. The minus sign in the front of $|\cdot|$ is used here because the genetic algorithm tries to find the maximum value of the given fitness function. Finding the minimum of a function $f$ is always equivalent to finding the maximum of function $-f$.

![Figure 6.8: A network of four serial resistors.](image)

There are four resistor positions $R_i$, $i = 0, \ldots, 3$ so that the natural coding of the chromosome is such that the chromosome consists of four genes each gene representing one possible resistor value *i.e.* an allele. In total there are 3 possible values to be selected from the allele set $A$. Thus this combinatorial optimisation problem has in total $3^4 = 81$ possible solution
candidates i.e. resistor value combinations, giving in total 12 different possible values for the total resistance of the circuit.

The generationwise evolution of the population consisting of 8 chromosomes i.e. the solution search by a GA is shown in Figure 6.10. Let there be a randomly generated initial population of resistance values. The population size i.e. the number of trials in each generation is thus \( n_P = 8 \), which should be a reasonable value for the tiny toy problem. Let the goal be having \( R_{\text{tot}} = R_{\text{goal}} = 40 \, \Omega \) i.e. in the solution all resistors are equal to 10 \( \Omega \). The solution is found after 4 generations of steady increase of the average population fitness, after 15 crossovers and 5 mutations, which means that about half of the search space was scanned before the solution was found. In this case the use of genetic algorithm is not of much use. The problem is simply too small and easy. This example was introduced to demonstrate how a simple genetic algorithm functions and the possibility to illustrate the whole search process easily. The next example will show that a genetic algorithm is able to find the solution for a much more difficult problem having a huge search space.

6.1.2 A more realistic example

Let us consider a more difficult and thus more interesting and realistic resistor example shown in Figure 6.9. The resistance of each resistor can be chosen from a set of the following set\(^2\) of values \( A = \{10, 12, 15, 18, 22, 27, 33, 39, 47, 56\} \, \Omega \). There are 16 resistor positions, so that the chromosome consists of 16 genes each gene representing one possible resistor value i.e. an allele. In total there are 10 possible values to be selected from the allele set \( A \). Thus this combinatorial optimisation problem has in total \( 10^{16} \) (ten million billion) solution candidates i.e. resistor value combinations.

Figure 6.11 shows the dependence of the average number of function calls \( n_f \) needed for the GA to find the minimum resistance of the circuit as the function of the population size \( n_P \). Using a small population size, the unique solution can be found on the average in less than 2,000 function calls. This means that the genetic algorithm has explored only \( 2 \times 10^3/10^{16} \times 100\% = 2 \times 10^{-11}\% \) of the total search space. As can be seen, the number of function calls increases with increasing population size: in a large population it takes time for the building blocks to find each other. The monotonicity of the \( n_P \) graph is a sign of an easy problem. For more difficult problems having an involved fitness landscape topology the risk of sticking to local extremes tends to increase \( n_f \) dramatically for the smallest population sizes. The resistor problem is such that choosing a small resistor always drives the search to the right direction without the fear of sticking to a local minimum. A rule of thumb in selecting the population size \( n_P \) is to have \( n_P \) proportional to the number of the parameters of the problem [Ala92e]. More often than not researchers have set \( n_P = 50 \), with usually good success. The heavier the fitness is to evaluate the more important it is to try to find a reasonable population size.

Text books etc

Good general overviews of genetic algorithms are among others [Bou87, For93a, BBM93a, BBM93b, Gol94a, SP94]. There are also some text books [Rec73, Hol75, Sch77, Buk79, Buk81, Sch81, StIG92, Gol89, BMS91, Dav91, Hol92, Koz92b, Mic92, Kit93, Ree93a, Ste93b, BP93, Ano93b],

\(^2\)standard E12 series
Figure 6.10: The evolution of population when searching the solution of a four resistor problem (fig. 6.8). The fitness $f(c_i) = -|R_{goal} - R_{tot}|$ for each chromosome $c_i$ is shown on top of 4 gene values shown within a frame; $f(c_i) = 0$ means that solution $c_i$ is found. Notations: the solution is shown in \textbf{bold}, $g$ = generation, $ave$ = average fitness, $\times$ = crossover, and $\downarrow$ = mutation.
of which perhaps David Goldberg’s book [Gol89] is the most referenced, even if it according to Goldberg himself, is already a bit old fashioned. We are waiting to see the second edition.

The popularity of GAs is mostly based on Goldberg’s book and his practical application [GK87] while John Holland’s book [Hol75] has laid ground for more theoretical work. For further references on basic genetic operators and their analysis see bibliography [Alah].


6.2. GA IMPLEMENTATIONS

The principle of GA is so simple, that its implementation is not a difficult task for any programmer. In addition there is plenty of implementations available to those who want to start solving their problems immediately using GA. The internet news group comp.ai.genetic FAQ (Frequently Asked Questions) contains a rather complete set of references to so called public domain GA programs.
CHAPTER 6. GENETIC ALGORITHMS

References

[Ano93a, RTA94, Haa93]

6.2.1 Distribution

GA is easy to distribute at least in those cases, in which the fitness function is complex, when
the relative communication overhead is low. There are plenty of work on distributing GA and
the most popular platform was once Transputer, which suits well in fine grained distribution
also. Less fine distribution can be done easily on a PC- or UNIX network using e.g. PVM
(Parallel Virtual Machine) software.

One way of distribution is to have one GA running, while a set of slave computers are
evaluating fitness function. There is not any strict synchronization requirements, which makes
this kind of operation mode easy: a slave can be even stopped without much effect on the
overall proceedings of the optimization process.

A more fine grained distribution might not, in general, be worth the trouble.

6.3 Genetic programming

A separate group of GA methods is genetic programming, in which the population is a
set of computer programs or functions in practise. Especially LISP has been a popular
implementation language in these experiments due to the simple syntax of LISP functions.
John Koza from Stanford University has been the leading researcher in this field.

References

[KR92, Koz92a, LFS92, Han93]

6.4 Artificial life

Lately the simulation of life and similar systems, the so called artificial life has aroused the
interest of several researchers. The interest of biologists is obvious, because it is so easy in
principle to make experiments, that are totally impossible in nature or laboratory and which
are also difficult to be mathematically modelled and analysed.

It is also possible to simulate totally artificial and theoretical creatures and their ecosystems
by computer programs and study e.g. their evolutionary population phenomena. There
are endless set of alternatives and only the laborous programming and imagination sets limits
to this kind of work. From AI point of view artificial life is seen as a way to produce some
simple “basic” intelligence by starting from the very basic signal processing. This way is
supposed to lead to “real” artificial intelligence.
6.5 GA literature

During the last few years we have been collecting systematically, at the beginning all, now only samples. Literature and references to GA literature. In figure 1.2 there is shown the annual distribution of GA literature, from which we can see that the growth of the number of references has been approximately 40%/year during nearly the last 20 years up to and including year 1997. This means that the research was doubling every second years. Literature up to year 1993 has been collected to the booklet [Ala94a]. Nowadays we maintain a set of about 50 special GA bibliographies (table 6.4).

In table 6.5 is shown the geographical distribution of authors having GA contributions (currently over 85% of the references are classified by the country of the author).
<table>
<thead>
<tr>
<th>file</th>
<th>contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>ga90bib.ps.Z</td>
<td>GA in 1990</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>ga02bib.ps.Z</td>
<td>GA in 2002</td>
</tr>
<tr>
<td>gaAIbib.ps.Z</td>
<td>GA in artificial intelligence</td>
</tr>
<tr>
<td>gaALIFEBib.ps.Z</td>
<td>GA in artificial life</td>
</tr>
<tr>
<td>gaARTbib.ps.Z</td>
<td>GA in art and music</td>
</tr>
<tr>
<td>gaAUSBib.ps.Z</td>
<td>GA in Australia and New Zealand</td>
</tr>
<tr>
<td>gaBASICSBib.ps.Z</td>
<td>Basics of GA</td>
</tr>
<tr>
<td>gaBIObib.ps.Z</td>
<td>GA in biosciences including medicine</td>
</tr>
<tr>
<td>gaCADbib.ps.Z</td>
<td>GA in Computer Aided Design</td>
</tr>
<tr>
<td>gaCHEMPhysBib.ps.Z</td>
<td>GA in chemistry and physics</td>
</tr>
<tr>
<td>gaCIVILbib.ps.Z</td>
<td>GA in civil and structural engineering (new)</td>
</tr>
<tr>
<td>gaCONTROLbib.ps.Z</td>
<td>GA in control</td>
</tr>
<tr>
<td>gaCSbib.ps.Z</td>
<td>GA in computer science (incl. databases, data mining and GP)</td>
</tr>
<tr>
<td>gaECObib.ps.Z</td>
<td>GA in economics and finance</td>
</tr>
<tr>
<td>gaESbib.ps.Z</td>
<td>Evolution strategies</td>
</tr>
<tr>
<td>gaFAR-EASTbib.ps.Z</td>
<td>GA in the Far East (excl. Japan)</td>
</tr>
<tr>
<td>gaFRAbib.ps.Z</td>
<td>GA in France</td>
</tr>
<tr>
<td>gaFTPbib.ps.Z</td>
<td>GA papers available via web (ftp and www)</td>
</tr>
<tr>
<td>gaFUZZYbib.ps.Z</td>
<td>GA and fuzzy logic</td>
</tr>
<tr>
<td>gaGERbib.ps.Z</td>
<td>GA in Germany, Austria, and Switzerland</td>
</tr>
<tr>
<td>gaGPbib.ps.Z</td>
<td>genetic programming</td>
</tr>
<tr>
<td>gaIMPLEbib.ps.Z</td>
<td>implementations of GA</td>
</tr>
<tr>
<td>gaISbib.ps.Z</td>
<td>immune systems</td>
</tr>
<tr>
<td>gaJAPANbib.ps.Z</td>
<td>GA in Japan</td>
</tr>
<tr>
<td>gaLATINbib.ps.Z</td>
<td>GA in Latin America, Portugal &amp; Spain</td>
</tr>
<tr>
<td>gaLOGISTICSbib.ps.Z</td>
<td>GA in logistics</td>
</tr>
<tr>
<td>gaMANUbib.ps.Z</td>
<td>GA in manufacturing</td>
</tr>
<tr>
<td>gaMEDITERbib.ps.Z</td>
<td>GA in the Mediterranean</td>
</tr>
<tr>
<td>gaNNbib.ps.Z</td>
<td>GA in neural networks</td>
</tr>
<tr>
<td>gaNORDICbib.ps.Z</td>
<td>GA in Nordic countries</td>
</tr>
<tr>
<td>gaOPTICSbib.ps.Z</td>
<td>GA in optics and image processing</td>
</tr>
<tr>
<td>gaOPTIMIbib.ps.Z</td>
<td>GA and optimization (only a few refs)</td>
</tr>
<tr>
<td>gaORbib.ps.Z</td>
<td>GA in operations research</td>
</tr>
<tr>
<td>gaPARAbib.ps.Z</td>
<td>Parallel and distributed GA</td>
</tr>
<tr>
<td>gaPARETObib.ps.Z</td>
<td>Pareto optimization (new)</td>
</tr>
<tr>
<td>gaPATENTbib.ps.Z</td>
<td>GA patents (new)</td>
</tr>
<tr>
<td>gaPATTERNbib.ps.Z</td>
<td>GA in pattern recognition incl. LCS (new)</td>
</tr>
<tr>
<td>gaPOWERbib.ps.Z</td>
<td>GA in power engineering</td>
</tr>
<tr>
<td>gaPROTEINbib.ps.Z</td>
<td>GA in protein research</td>
</tr>
<tr>
<td>gaQCbib.ps.Z</td>
<td>quantum computing (new)</td>
</tr>
<tr>
<td>gaROBOTbib.ps.Z</td>
<td>GA in robotics</td>
</tr>
<tr>
<td>gaSAbib.ps.Z</td>
<td>GA and simulated annealing</td>
</tr>
<tr>
<td>gaSCHEDULINGbib.ps.Z</td>
<td>GA in scheduling</td>
</tr>
<tr>
<td>gaSIGNALbib.ps.Z</td>
<td>GA in signal and image processing</td>
</tr>
<tr>
<td>gaSIMULAbib.ps.Z</td>
<td>GA in simulation (new)</td>
</tr>
<tr>
<td>gaTHEORYbib.ps.Z</td>
<td>Theory and analysis of GA</td>
</tr>
<tr>
<td>gaTHESESbib.ps.Z</td>
<td>PhD etc theses</td>
</tr>
<tr>
<td>gaUKbib.ps.Z</td>
<td>GA in United Kingdom</td>
</tr>
<tr>
<td>gaVLSIbib.ps.Z</td>
<td>GA in electronics, VLSI design and testing</td>
</tr>
</tbody>
</table>

Table 6.4: Special GA subbibliographies at site ftp.uwasa.fi in directory cs/report94-1. The files are updated every now and then. Most of them have been available on line since 1994.
### Table 6.5: The geographical distribution of GA researchers according to Vaasa GA bibliography database (22.10.2012).

<table>
<thead>
<tr>
<th>Country</th>
<th>#</th>
<th>%</th>
<th>Country</th>
<th>#</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td>20653</td>
<td>100.00</td>
<td>United States</td>
<td>5590</td>
<td>27.07</td>
</tr>
<tr>
<td>Japan</td>
<td>2463</td>
<td>11.93</td>
<td>United Kingdom</td>
<td>2075</td>
<td>10.05</td>
</tr>
<tr>
<td>Germany</td>
<td>1392</td>
<td>6.74</td>
<td>China</td>
<td>1083</td>
<td>5.24</td>
</tr>
<tr>
<td>Finland</td>
<td>789</td>
<td>3.82</td>
<td>Italy</td>
<td>589</td>
<td>2.85</td>
</tr>
<tr>
<td>France</td>
<td>533</td>
<td>2.58</td>
<td>Australia</td>
<td>509</td>
<td>2.46</td>
</tr>
<tr>
<td>South Korea</td>
<td>463</td>
<td>2.24</td>
<td>Taiwan</td>
<td>463</td>
<td>2.24</td>
</tr>
<tr>
<td>Spain</td>
<td>414</td>
<td>2.00</td>
<td>India</td>
<td>365</td>
<td>1.77</td>
</tr>
<tr>
<td>Canada</td>
<td>333</td>
<td>1.61</td>
<td>Brazil</td>
<td>215</td>
<td>1.04</td>
</tr>
<tr>
<td>The Netherlands</td>
<td>206</td>
<td>1.00</td>
<td>Switzerland</td>
<td>183</td>
<td>0.89</td>
</tr>
<tr>
<td>Poland</td>
<td>180</td>
<td>0.87</td>
<td>Belgium</td>
<td>171</td>
<td>0.83</td>
</tr>
<tr>
<td>Singapore</td>
<td>168</td>
<td>0.81</td>
<td>The Czech Republic</td>
<td>151</td>
<td>0.73</td>
</tr>
<tr>
<td>Mexico</td>
<td>136</td>
<td>0.66</td>
<td>Austria</td>
<td>126</td>
<td>0.61</td>
</tr>
<tr>
<td>Greece</td>
<td>124</td>
<td>0.60</td>
<td>Israel</td>
<td>123</td>
<td>0.60</td>
</tr>
<tr>
<td>Sweden</td>
<td>121</td>
<td>0.59</td>
<td>Turkey</td>
<td>107</td>
<td>0.52</td>
</tr>
<tr>
<td>Portugal</td>
<td>105</td>
<td>0.51</td>
<td>Russia</td>
<td>98</td>
<td>0.47</td>
</tr>
<tr>
<td>Romania</td>
<td>77</td>
<td>0.37</td>
<td>Slovenia</td>
<td>62</td>
<td>0.30</td>
</tr>
<tr>
<td>Denmark</td>
<td>59</td>
<td>0.29</td>
<td>Hungary</td>
<td>58</td>
<td>0.28</td>
</tr>
<tr>
<td>Iran</td>
<td>57</td>
<td>0.28</td>
<td>Saudi Arabia</td>
<td>50</td>
<td>0.24</td>
</tr>
<tr>
<td>Ireland</td>
<td>48</td>
<td>0.23</td>
<td>Thailand</td>
<td>44</td>
<td>0.21</td>
</tr>
<tr>
<td>The Slovak Republic</td>
<td>42</td>
<td>0.20</td>
<td>Norway</td>
<td>36</td>
<td>0.17</td>
</tr>
<tr>
<td>New Zealand</td>
<td>32</td>
<td>0.15</td>
<td>Croatia</td>
<td>30</td>
<td>0.15</td>
</tr>
<tr>
<td>Venezuela</td>
<td>28</td>
<td>0.14</td>
<td>South Africa</td>
<td>25</td>
<td>0.12</td>
</tr>
<tr>
<td>Malaysia</td>
<td>24</td>
<td>0.12</td>
<td>Egypt</td>
<td>21</td>
<td>0.10</td>
</tr>
<tr>
<td>Argentina</td>
<td>20</td>
<td>0.10</td>
<td>Yugoslavia</td>
<td>20</td>
<td>0.10</td>
</tr>
<tr>
<td>Algeria</td>
<td>19</td>
<td>0.09</td>
<td>Bulgaria</td>
<td>19</td>
<td>0.09</td>
</tr>
<tr>
<td>Kuwait</td>
<td>18</td>
<td>0.09</td>
<td>Ukraina</td>
<td>18</td>
<td>0.09</td>
</tr>
<tr>
<td>Colombia</td>
<td>16</td>
<td>0.08</td>
<td>Chile</td>
<td>11</td>
<td>0.05</td>
</tr>
<tr>
<td>Pakistan</td>
<td>10</td>
<td>0.05</td>
<td>Marocco</td>
<td>9</td>
<td>0.04</td>
</tr>
<tr>
<td>Jordan</td>
<td>8</td>
<td>0.04</td>
<td>Byelorussia</td>
<td>7</td>
<td>0.03</td>
</tr>
<tr>
<td>Cuba</td>
<td>7</td>
<td>0.03</td>
<td>Lebanon</td>
<td>7</td>
<td>0.03</td>
</tr>
<tr>
<td>Cyprus</td>
<td>6</td>
<td>0.03</td>
<td>Tunisia</td>
<td>5</td>
<td>0.02</td>
</tr>
<tr>
<td>Bosnia and Herzegovina</td>
<td>4</td>
<td>0.02</td>
<td>Latvia</td>
<td>4</td>
<td>0.02</td>
</tr>
<tr>
<td>Lithuania</td>
<td>4</td>
<td>0.02</td>
<td>Iceland</td>
<td>3</td>
<td>0.01</td>
</tr>
<tr>
<td>Macau</td>
<td>3</td>
<td>0.01</td>
<td>Puerto Rico</td>
<td>3</td>
<td>0.01</td>
</tr>
<tr>
<td>Armenia</td>
<td>2</td>
<td>0.01</td>
<td>Azerbaithzan</td>
<td>2</td>
<td>0.01</td>
</tr>
<tr>
<td>Bahrain</td>
<td>2</td>
<td>0.01</td>
<td>Bangladesh</td>
<td>2</td>
<td>0.01</td>
</tr>
<tr>
<td>Fiji</td>
<td>2</td>
<td>0.01</td>
<td>Indonesia</td>
<td>2</td>
<td>0.01</td>
</tr>
<tr>
<td>Nigeria</td>
<td>2</td>
<td>0.01</td>
<td>U Arab Emirates</td>
<td>2</td>
<td>0.01</td>
</tr>
<tr>
<td>Zimbabwe</td>
<td>2</td>
<td>0.01</td>
<td>Estonia</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>Kazakhstan</td>
<td>1</td>
<td>0.00</td>
<td>Mauritius</td>
<td>1</td>
<td>0.00</td>
</tr>
<tr>
<td>Oman</td>
<td>1</td>
<td>0.00</td>
<td>Philippines</td>
<td>1</td>
<td>0.00</td>
</tr>
</tbody>
</table>
Chapter 7

Optimal GA

In this chapter we describe a C++ package used to analyze a class of genetic algorithms to find the optimal genetic algorithm. The parameters of the best genetic algorithms have been searched by a genetic algorithm. A traveling salesperson type problem is used as a test and an example of applications of genetic algorithms. The sequence coding problem is solved by using link sets, which allows a pure genetic algorithm approach. This preserves a clear and to the genetic algorithms usually so peculiar separation between the problem to be solved and the genetic algorithm itself. This chapter is based on the proceedings paper [Ala91a].

7.1 Introduction

One attractive feature of genetic algorithms is their simplicity, which makes it easy to implement and tailor them to practical problems. The operation of a genetic algorithm searching an optimal solution to a given problem can be divided into the following steps used in the GAs of this chapter:

**Genetic Algorithm:**

- [initialization] generate the initial (e.g. random) set of potential solutions
- [selection] calculate the distance of each solution to the goal and take the best ones
- [breeding] generate new solutions by exchanging solution parameters (crossing) between the selected good solutions and possibly add some noise (mutations)
- [repetition] if the goal is not reached, go back to the step [selection].

The implementation of genetic algorithms includes determination and optimization of a number of details, procedures and parameters. The parameters include population size, crossover type and rate, and mutation type and rate. They are analysed in this chapter.

The idea has been to use GAs to analyze themselves by searching and optimizing the parameters of the best GAs. By best we here mean algorithms that find the solutions of the given test problem set in the least number of specimens generated. The package is written in C++ [Str85] and currently contains about 4000 lines of code, most of which is related to testing and test data analysis and output as various graphs etc.
7.1.1 Related work

The fundamentals of genetic algorithms can be found in David Goldberg’s book [Gol89]. In addition, Lawrence Davis gives a good overview of applications [Dav91]. He also suggests a practical procedure to attack new problems by combining genetic algorithms with existing methods into, as he calls, hybrid GA: The pure basic genetic algorithm is usually not the best possible algorithm to solve any specific problem if a good specific algorithm already exists. There is, however, a good chance to create a better algorithm by combining the best features of existing conventional and genetic algorithms.

John J. Grefenstette has used genetic algorithms to find the optimal parameters of genetic algorithms [123]. He has tried to find the optimal values of six parameters: population size ($n$), crossover rate ($c$), mutation rate ($m$), renewal rate ($r$), scaling window ($w$) [generations], and elitist/pure selection ($s$). The total number of parameter combinations, i.e. the number of different possible genetic algorithms was $2^{18}$. The “standard” genetic algorithm based on De Jong’s work [Jon75] has parameter values: $GA_s(n,c,m,r,w,s) = GA(50, 0.6, 0.001, 1.0, \text{no scaling}, \text{elitist})$. After Grefenstette the best genetic algorithm with respect to his test function environment is $GA_{JJG} = GA(30, 0.95, 0.01, 1.0, 1, \text{elitist})$.

K. Shahookar and P. Mazumder have also applied this meta-genetic method to fine tune their integrated circuit standard-cell placement genetic algorithm [381]. They had three parameters: mutation, crossover and inversion rate. The last parameter gives the probability that some part of the chromosome flips around. Intuitively the inversion operation seems to be advantageous in geometrical problems. The best values they selected were: mutation 0.005, crossover 0.33, and inversion 0.15.

Our genetic algorithms somewhat differ from that of the previously mentioned work. That is why our first step towards the application of genetic algorithms is to try to find out what are the optimal values of the different parameters.

7.2 Definitions

Here we will give a set of informal definitions of the concepts used in the following treatment of the subject. The definitions are more or less artificial but they have some obvious similarities with concepts used in biology.

Definition 7.2.1 (Gene) A gene is a string of bits holding the inherited information of the specimen.

In our implementation the length of the gene is at most 32 bits simply because we have used 32 bit integers to represent genes. Genes are further subdivided into fields.

Definition 7.2.2 (Chromosome) A chromosome is a string of genes.

Chromosomes are represented by integer arrays.

Definition 7.2.3 (Specimen) A specimen is a more or less independent functional unit interacting with its surroundings, other specimens and having a more or less good ability to produce offspring by using its genetic information usually together with other specimens.
7.3. OPERATIONS OF GAs

Figure 7.1: The class system

Specimens are represented by objects of the specimen class (fig. ??).

Definition 7.2.4 (Species) A species is the set of specimens capable of breeding together and producing fertile offspring.

Definition 7.2.5 (Variety) A variety is a clearly distinguished subset of specimen.

Species and variations are not explicitly implemented but represented by isolated specimen sets, which are objects of the population class.

In our system the set, i.e. an array of specimens in a population, is equally divided into \( n \) subsets. The subsets can be thought of containing some kind of varieties of the specimen. A somewhat similar scheme is also used by Tanese [Tan89], who calls this the partitioning of genetic algorithm.

Definition 7.2.6 (Fauna) Fauna is the set of all current specimens.

A class fauna is used to contain all populations under test.

7.3 Operations of GAs

7.3.1 Random generator

As a random integer number generator we have used the following simple procedure:

```c
int random(int& u, int limit)
{
    int a=1237, b=1331, c=1337713;
    u = (a*u+b) % c;
    return u % limit;
}
```

In figure ?? you can see the distribution of the first 1.000.000 random numbers generated by \texttt{random(u,101)} and starting from \( u=13 \).

Figure 7.2: Distribution of 1.000.000 random numbers generated by the procedure \texttt{random(u,101)}
7.3.2 Crossover

The bit strings of the parents’ breed can be combined in many ways. The principle is that we break parent sequences in two or more subsequences and recombine these sequences into one or more new offspring combinations. In the early works of GA, much work has been done with two and three subsequences i.e. with one or two point crossover. In our work we have used crossover rate as a free parameter ranging from zero, i.e. no crossing, to the number of bits used in genetic information, i.e. uniform crossover [Sys89].

A popular way of representing chromosomes is to use fixed sized arrays. Two descendants are then created by swapping every even or odd subrange of the parent chromosomes determined by the crossover operator as shown in fig. 6.2, e.g. by using binary masks. If variable length chromosomes are used the parent chromosomes may not have the same crossover indexes and so we cannot use binary masks, but construct the descendant chromosomes by linking the pieces one after another just as we do when processing linear lists. However, certain care must be taken in selecting the more or less independent crossover points of both parent chromosomes [?].

Figure 7.3: fig:crossover Crossover operator

7.3.3 Multiparent breeding

In nature there exist two possible breeding mechanisms: parthenogenesis or cloning and sexual reproduction i.e. we have either one or two parents. At least in artificial systems we can imagine that there could be more than two specimens involved in breeding. In computations where we do not have most of the limitations of biological systems, we can have any number of parents. However, here we will only use two parents.

7.3.4 Mutations

Mutation and crossover are considered to be the key phenomena in genetics and genetic algorithms. In nature the mutation rate varies much but is typically rather small ranging from $1 : 100,000$ to $1 : 300,000,000$ [Far88, p. 521]. In genetic algorithms mutation rate varies even more but is typically much greater than the natural one. E.g. Schaffer et al [SCED89] have used mutation rates ranging from $1 : 100$ to $1 : 500$.

In our system mutations are handled by “injecting” constant chromosomes into the population. The same chromosomes are also used as the initial population chromosomes. Intuitively it is clear that the selection of the starting chromosomes have a profound effect at least on the first few steps.

7.3.5 Fitness

The survival of specimens depends both on its abilities to fit into the environment and the actual environment confronted with. Both of these factors are more or less statistical and contain randomly varying uncertainties. In genetic algorithms the fitness is usually a scalar value evaluated from chromosomal and environmental parameters.
7.4. PROGRAM

In our tests we have used fitness functions $f$ returning values in the half open interval $(-\infty, |C|]$, where $|C|$ is the length of the chromosome in bits and represents the best possible fitness. In case of variable length chromosomes we may normalize this by using values $f/|C|$.

In our GAs the fitness values of specimens are sorted in decreasing order. The breeding starts from the fittest specimen $S_i$ and goes towards the less fit end of the array ($i = 1, \ldots, n-1$). The other parent $S_j$ is selected randomly so that $j = \min(i + 1 + \text{random}(m), n)$, where $\text{random}(m) \in [0, m]$ is a pseudorandom number and $m$ is the mobility of the population. All the fitness values are relative to the fittest specimen, which is always selected (elitism). The drawback of this arrangement is that the sorting causes some extra overhead and it also makes the GAs less parallel. Because the specimens do not need to be in strict order we could sort only roughly [IW91] in order to save some time and to make sorting better suited for parallel hardware.\(^1\)

7.4 Program

The test system has been implemented in C++ programming language [Str85]. The basic object of the system is called specimen:

```cpp
class specimen{
friend class populationSpecimen;
    int Genes[GeneLength];
    population *P;
public:
    .
    .
};

class population{
};

class hiihuu {
}

class fauna{
}
```

\(^1\)Actually there was a bug in the sorting routine so that the sorting was not perfect allways, but that does not seem to influence the results too much.
CHAPTER 7. OPTIMAL GA

7.5 Optimization

7.5.1 Test arrangement

The test arrangement was as follows: The tested genetic algorithms were represented by the population class. The optimized parameters were packed into a 29 bit long subchromosome of the population class. The parameters were:

- size $[0, 127]$ step size 1 (7 bits),
- mutation rate $[0, 15]$ step size 1 (4 bits),
- crossing rate $[0, 31]$ step size 1 (5 bits),
- mobility $[0, 15]$ (4 bits),
- minimum Hamming distance between parents $[0, 15]$ step size 1 (4 bits),
- number of variations i.e. population subdivision $[1, 8]$ step size 1 (3 bits), and
- elitism rate $[1, 4]$ step size 25% of the total population size (2 bits).

Size is the population size. Mutation rate denotes the number of specimens that are taken from the initial specimen pool during each generation. Crossing rate stands for the average number of crossing points per chromosome. Mobility controls the maximum allowed random distance between the breed specimens in terms of the number of intervening sorted specimens. Number of variations is the number of equal sized subpopulations.

As a test problem we used the below explained traveling salesperson type problem of finding the shortest route through seven points.

A metalevel genetic algorithm was used to search for the best parameter combination. The metalevel algorithm itself was similar to the tested algorithms having parameter values that were somewhat varied from test to test. In addition to the optimized parameters there were also some other parameters that could be set manually for each test run. These parameters included maximum number of specimen and population generations ($N_{s,\text{max}}$ and $N_{P,\text{max}}$), samples per population at start ($S_{P}^{0}$), number of fauna test rounds ($N_{F}$), sampling increment per round ($\Delta S_{P}$), and error penalty ($E$). $N_{s,\text{max}}$ and $N_{P,\text{max}}$ limit the depth of the search. $S_{P}$ gives the number of randomly chosen test cases for each candidate genetic algorithm and $N_{F}$ gives the number of restarts. The test loop is the following:

```markdown
for (j=0, $S_{P} = S_{P}^{0}$; j<$N_{F}$; j++){
    for (i=0; i<$N_{P,\text{max}}$; i++){
        \text{<one generation step>}
    }
    $S_{P} = S_{P} + \Delta S_{P}$;
}
```

Population fitness function $F_{P}$ i.e. the fitness function of the candidate genetic algorithm was chosen so that

$$F_{P} = \begin{cases} 
-gn_{P}, & \text{if solution was found} \\
-gn_{P}E(|C| - f_{\text{best}}) & \text{if solution was not found}
\end{cases}$$

\(^{2}\text{by fitness}\)
where \( g \in [1, N_{s,\text{max}}] \) is the number of GA generations tested, \( n_P \in [0, 127] \) is the size of the population of GAs, \( |C| = 32 \) is the length of the chromosome (in bits), and \( f_{\text{best}} \in (\infty, |C|] \) is the fitness of the best specimen found.

The initial genes for both levels were not randomly chosen but consecutively all zero or all one bits, so that the initial population contains 0 and 1 bits at all bit positions.

In the following figures you can see some of the results of the optimization runs. From these we can draw e.g. the following conclusions:

- The convergence of fitness is rather steady as expected.
- The more crossing we have the better, i.e. uniform or nearly uniform crossing seems to be the best in every test case.
- High mobility gives the best results.
- Mutation rate decreases rapidly as expected.
- Optimum population size ranges from under 20 to over 60 specimens. Size \( \approx 30 \) seems to be reasonable.
- In our test cases it seems that subdivision of populations is not beneficial.
- 25\% elitism seems most beneficial. Our test arrangement is such that the rest of the specimen table is also rather well sorted (bubble sort).

The number of all possible GAs in our test arrangement is \( 2^{29} \). The number of GAs generated in the largest test run was 20544, so that at most only \( 20544/2^{29} \approx 0.004\% \) of all the possible GA parameter combinations was actually tested. It is therefore easy to understand why the results of the different test runs somewhat differ from each other. However, in practical optimisation problems it is often not so vital to reach the global optimum, but to find reasonable solutions, which are not too far away from the global optimum.

### 7.5.2 Test problems

Test problems are shown in Table 7.1.

<table>
<thead>
<tr>
<th>code</th>
<th>problem</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>010101...01</td>
</tr>
<tr>
<td>1</td>
<td>101010...10</td>
</tr>
<tr>
<td>2</td>
<td>001100...0011</td>
</tr>
<tr>
<td>3</td>
<td>110011...1100</td>
</tr>
<tr>
<td>4</td>
<td>field(0,5) = 010101</td>
</tr>
<tr>
<td>5</td>
<td>field(16,21) = 010101</td>
</tr>
<tr>
<td>6</td>
<td>field(0,5) = ( \sqrt{256} )</td>
</tr>
<tr>
<td>7</td>
<td>field(0,10) = ( \sqrt{1024} )</td>
</tr>
</tbody>
</table>

Table 7.1: Test problems,
Figure 7.4: The evolution of population fitness. [Ala91b].

Figure 7.5: The evolution of population size. [Ala91b].
Figure 7.6: The evolution of mutation rate. [Ala91b].

Figure 7.7: The evolution of crossover rate i.e the number of crossover points. [Ala91b].
7.5.3 Test case

Route planning

Let us have a set of $n$ fixed points (cities). The problem is to visit all $n$ points $P_i$ in minimum total time $t$. There are at least $n - 1$ possible movements (links) between the different points each having a traveling time $t_{i,j} \geq 0$.

Genetic algorithms have been applied in solving the salesperson or similar problems (see e.g. [?, ?, Alag]). Much work has been done in order to fit crossing operators to these sequence type problems. It is quite clear that if we have two relatively different routes, it is difficult to combine a new route out of random pieces of the parent routes. Some additional more or less ad hoc heuristics, resulting in a more or less distant solution due to implicit mutations, have been used. The other obvious drawback is that the clear separation of the genetic algorithm part and the problem part of the system is confused. Especially this applies to the crossing operator. This is unfortunate because the separation is typical of genetic algorithms and hard to find in any other problem solving approach.

Here we show how the sequence coding and crossing can be solved in a simple way. Let us assume that the traveling times $t_{i,j}$ between every pair have been given and in addition that $t_{i,j} = t_{j,i}$ so that the number of possible bidirectional links is $n(n-1)/2$, where $n$ is the number of points $P_i$. The chromosomal representation of the solution can now be simply a binary string where 1’s denote that a link is available for route and 0’s that the corresponding link is not available. It is enough that the length of the chromosome, by which we can represent every possible route or Hamilton cycle between $n$ points, is equal to $n(n - 1)/2$. The chromosome does not represent any given route but a set of allowed links out of which we can construct route candidates. Usually the genetic algorithms solving the traveling salesperson problem go in this respect a step further and require that the chromosomal representation is already a solution. From the biological point of view our solution is more natural because the natural genetic code is never the solution of any problem itself but just the instructions how to construct a solution and the crossing over seems to be independent of as well the genotypes as phenotypes involved. In computing a given application program is not the solution but only the instructions how to construct a solution. In the computer environment, however, it is easy to confuse the code and the result of coding especially in simple applications.

In our approach the knowledge of the problem is coded into the fitness function which in our link set representation case is the following: Because we do not get the route trivially from the chromosomal representation, we must construct one. If a route covering all points can be constructed the fitness is proportional to the length of that route. If a route does not exist a value representing a not-so-good-solution is returned. The traveling salesperson and unfortunately also the Hamilton cycle problem are known to be NP-hard. For the case of too many extra links, the procedure Hamilton has a slack variable that tells the number of maximum allowed alternative links that are left to process. If slack is equal to zero also a not-so-good-solution is returned.

The procedure Hamilton takes the first existing link when constructing a route. This resembles the allele system of polypl oid cells in which there are two or more alternative values for each gene. Some alleles are dominant and have thus top priority in cell functions, while others are recessive and active only when there is not any dominant allele present. We could also utilize a similar system in GAs by arranging values by some priority. In case of the traveling salesperson, this would naturally be the increasing order of links by lengths.
7.6. OBSERVATIONS

When we use the link set representation and the given, take-the-first-possible-route, fitness procedure Hamilton the crossing operator, mutation etc. genetic algorithm functions can be any of the basic ones operating on binary strings.

The role of genetic algorithm in finding the best route is to drop the poor links until, in the best case, only the \( n - 1 \) best links remain. The best solution can in principle have also excess links, that are not included in the route evaluated by Hamilton, but that does not do any harm because we already know the best solution anyway. The drawback of the link set approach is that we can have a number of combinations that do not represent any valid route. The other problem is as we have already stated that the extra links decrease execution speed to some extent. In practise there are also many other obvious points of development but they go beyond the scope of this paper.

The matrix \( E \) giving the distances between the points of the test problem is given below:

\[
E = \begin{bmatrix}
0 & 6 & 5 & 4 & 3 & 2 & 1 \\
6 & 0 & 7 & 6 & 5 & 4 & 3 \\
5 & 7 & 0 & 8 & 7 & 6 & 5 \\
4 & 6 & 8 & 0 & 9 & 8 & 7 \\
3 & 5 & 7 & 9 & 0 & 10 & 9 \\
2 & 4 & 6 & 8 & 10 & 0 & 11 \\
1 & 3 & 5 & 7 & 9 & 11 & 0
\end{bmatrix}
\]

7.6 Observations

7.6.1 Random numbers

The genetic algorithm heavily relies on random or pseudo random numbers. One popular set of method to generate pseudo random numbers is congruential methods that use remainder function. It was also our first method. When the first test runs were monitored, it was surprising to notice that about half of the processing time was devoted to remainder evaluations although there were only two remainder operations per generated number.

After this observation several attempts were made in order to eliminate unnecessary remainder operations. Some of the random number ranges were of form \( 0 \ldots 2^n - 1 \) where were can use left- and rightshift operations instead of slow arithmetic division and multiplication operations. The other enhancement was to use array of 8 bit random numbers to rapid random bit string generation.

7.7 Conclusions

In this paper we have described a class of genetic algorithms and shown how genetic algorithms can be used to search their optimal parameters. A traveling salesperson type movement optimization is used as a test and example problem, which is solved applying pure genetic algorithm approach. The approach puts some effort on coding and the fitness function design but it helps to clearly separate the genetic algorithm from the problem at hand, the separation of which is paramount and peculiar to the genetic algorithm problem solving approach in general.
The pure genetic algorithm approach seems promisingly beneficial from the software design and economical point of view and thus deserves further elaboration.

The selection of the initial population seems to have some influence on GA performance, but this subject seems to be rather less worked on in GA research.

Finally it took hours and hours of CPU time to process the test cases on a SUN-4 workstation. In spite of the millions of specimens and thousands of GAs generated during testing, the exact optimal values of some of the parameters of our genetic algorithms still remain somewhat unclear. More thorough testing is needed.

```cpp
class specimen{
friend class population; // for Fitness

int Ordinal,Mark;
int Born; // generation
float Fitness;
int Genes[ChromosomeLength]; // chromosome
population *P;

public:
char* Name;

specimen();
specimen(population*,int*);
specimen(specimen&);
~specimen(){DeletedSpecimens++;}

int breed(specimen**, int, specimen**, int);
int ordinal() {return Ordinal;}
population* popul() { return P; }
char* name() { return Name; }
int* genes() { return Genes; }
int operator[](int i) {return (Genes[i]);}
int field(int i, int j){return Field(Genes[0],i,j);}
int field_set(int,int,int);
void setPopulation(population* p) { P=p; }
int born() { return Born; }
void refresh(){ Fitness = -INF; }
void mark() { Mark=1; }
void umark() { Mark=0; }
int marked(){ return Mark; }
int equals(specimen**, int);
int distance(specimen**, int);

// class specific methods:
virtual float fitness(float);
friend ostream& operator<<(ostream&,specimen&);
}
```

### 7.7.1 Population size

It seems that about 50 specimens is usually a reasonable size of population. Is this some kind of natural constant, or is the size $n$ actually a function $n = n(L_c, ...)$ of different parameters of the algorithm chosen. Intuitively it seems that the length of the chromosome $L_c$ should
have some influence on $n$. This deserves more analysis and we will return to this problem later.

### 7.7.2 Initial population and mutations

The widely used random initial population does not seem to be a very good idea, because of the *Building block hypothesis*. Again it seems intuitively that a more carefully tuned starting population would be better. One property that should be useful is the completeness of the set, which means that every possible chromosome can be generated from the starting set without resorting to mutations. If the starting set is not complete, which may be easily the case when using random population, we must have mutations in order to get all possible alternatives. This may be one reason why high mutation rate is considered reasonable during the first generations. In a way we start evolution from scratch instead of carefully setting the set of specimens beforehand.

An analogy to breeding is clear: before we start our experiment we usually design the starting population so that it already contains all the desired properties. Mutations are used only occasionally.

### 7.7.3 Finetuned crossing

Usually the crossing over distribution is independent of the fitness function. Should we in some cases finetune the crossing distribution by taking into account the contributions of different schemata into the fitness value.

E.g. we have a gene that is responsible for the total error. It would be wise to try to change this gene instead of the others. Let us call this as gene manipulation, which is known to be the fastest way to develop given properties.
Chapter 8

Analysing GA more

This chapter is based on author’s published paper [Ala99].

8.1 Background

Eventhough the principle of GA is very simple, the mathematical analysis of GA has turned out to be difficult, and there exist not much profound results relating GA performance and problem difficulty, which would be interesting to those who are applying GA to real world problems.

In this chapter we analyse empirically genetic algorithm search efficiency on several combinatorial optimisation problems in relation to building blocks and fitness landscape. The test set includes five problems of different types and difficulty levels all with an equal chromosome length of 34 bits. Four problems were quite easy for genetic algorithm search while one, a folding problem, turned out to be a very hard one due to the uncorrelated fitness landscape.

The results show that genetic algorithm is efficient in combining building blocks if the fitness landscape is well correlated and if the population size is large enough. An empirical formulae for the average number of generations needed for optimization and the corresponding risk level for the test set and population sizes are also given. The results seem to show that GAs are efficient in combining building blocks, if the fitness landscape is correlated especially around the solutions and the population size is large enough to provide shelter to all “hibernating” but scattered building blocks of the solution.

The purpose of this study is to shed some more light on the basic fitness function properties with respect to the functioning of genetic algorithms, and thereafter give some advice for future applications. The emphasis is put on population size, building blocks and fitness landscape, which further influence search efficiency and success rates.

The optimisation efficiency will be empirically studied using population sizes ranging from 25 to 1600 and five different type and complexity combinatorial problems. The problem set consists of 1) Onemax, 2) maximum sum, 3) boolean satisfiability, 4) polyomino tiling, and 5) a folding problem. The fitness landscape has been analysed by evaluating autocorrelation along random one-bit mutation paths.
8.1.1 Related work

In our previous study we have optimized some parameters of genetic algorithm including population size and mutation rate. The optimization was done by a genetic algorithm using a tiny travelling salesman problem (TSP) as a test function [Ala91b]. In a later study we analysed the effect of population size on genetic algorithm search [Ala92d]. The object problem was an Onemax type with the chromosome length $n_g$ varying in the range $[4, 28]$. $n_g$ was used as a measure of problem complexity. The empirical results suggested that the optimal population size $n_P$ seems to be included in the interval $[\log(N), 2\log(N)]$, where $N$ is the size of the search space [Ala92d]. For the Onemax problem $N = 2^{n_g}$ and thus the optimal population size interval is approximately $[n_g, 2n_g]$. Perhaps the most important conclusion from the above empirical studies was that the search efficiency does not seem to be too sensitive to either population size or other parameters, but more to the properties of the object problem fitness function. [Ala92d]

In [Jul96] Bryant Julstrom derived a formula for the population size when solving TSP. Julstrom’s estimate gives a somewhat smaller upper bound for the population size than the our estimate especially for larger search spaces.

De Jong and Spears have analysed the interacting roles of population size and crossover [JS91]. David E. Goldberg et al have analysed statistically the selection of building blocks [GDC92].

Colin Reeves has analysed a small population using an experimental design approach [Ree93b]. Hahner and Ralston have noticed that small population size may be the most efficient in rule discovery [HR95].

Also infinite populations have been theoretically analysed [Sta95b].

Weinberger has analysed autocorrelations both in Gaussian and in the so called NK-landscapes [Wei90].

Other studies with fitness distributions include [Boo92, FK94, Gre94, MBV96, MP96].

Work on fitness landscapes includes [ES96, ECS89, Jon95, Kin94, Kol97, MdWS91, MS94, MW92, MFH91, PNPO97].

References to other studies on parameters of genetic algorithms and search efficiency can be found in bibliography [Alah].

References

[GKD89, MFH91, Rad92, Bäc93, Bäc92, FM92, Kar93, KQF93, MPGL93]


8.2 Genetic algorithm

The genetic algorithm used in this study was coded in C++ specifically for this study keeping in mind possible further studies e.g. on other fitness functions. In practise this means e.g. that the results of this study should be as much as possible independent of the results of the previous studies [Ala91b, Ala92d] done by another program, also coded in C++.

A somewhat simplified body of the genetic algorithm used is shown on page ??, while its parameters are shown in Table 8.1.
8.3. Theory

Here we will deduce a stochastic search dynamics model for both selection efficiency and genetic algorithm search risk in relation to population size. In spite of the simplicity of the models, we will see that they fit quite nicely to the empirical results.

8.3.1 Risk estimation

<table>
<thead>
<tr>
<th>$n_a \backslash P$</th>
<th>0.99</th>
<th>0.999</th>
<th>0.9999</th>
<th>0.99</th>
<th>0.999</th>
<th>0.9999</th>
<th>0.99</th>
<th>0.999</th>
<th>0.9999</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>12</td>
<td>15</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>23</td>
<td>31</td>
<td>39</td>
<td>28</td>
<td>36</td>
<td>44</td>
</tr>
<tr>
<td>8</td>
<td>34</td>
<td>52</td>
<td>69</td>
<td>49</td>
<td>66</td>
<td>84</td>
<td>61</td>
<td>78</td>
<td>95</td>
</tr>
<tr>
<td>16</td>
<td>71</td>
<td>107</td>
<td>143</td>
<td>102</td>
<td>137</td>
<td>173</td>
<td>126</td>
<td>162</td>
<td>197</td>
</tr>
<tr>
<td>32</td>
<td>145</td>
<td>218</td>
<td>290</td>
<td>206</td>
<td>279</td>
<td>351</td>
<td>256</td>
<td>329</td>
<td>401</td>
</tr>
<tr>
<td>64</td>
<td>292</td>
<td>439</td>
<td>585</td>
<td>416</td>
<td>562</td>
<td>708</td>
<td>516</td>
<td>663</td>
<td>809</td>
</tr>
<tr>
<td>128</td>
<td>587</td>
<td>881</td>
<td>1174</td>
<td>835</td>
<td>1129</td>
<td>1422</td>
<td>1037</td>
<td>1330</td>
<td>1624</td>
</tr>
<tr>
<td>1024</td>
<td>4713</td>
<td>7070</td>
<td>9427</td>
<td>6705</td>
<td>9062</td>
<td>11418</td>
<td>8323</td>
<td>10679</td>
<td>13036</td>
</tr>
<tr>
<td>10240</td>
<td>47155</td>
<td>70732</td>
<td>94309</td>
<td>67080</td>
<td>90657</td>
<td>114234</td>
<td>83263</td>
<td>106840</td>
<td>130417</td>
</tr>
</tbody>
</table>

Table 8.2: Population size estimates $n_P = (\log(1 - P) - \log(n_g))/\log(1 - 1/n_a)$ at several risk levels $P$, allele probabilities $p = 1/n_a$ and number of genes $n_g$.

In order to analyse proper population size with respect to the risk that the solution is not found, let us assume that the initial population contains all the building blocks necessary for the solution. These hibernating but scattered building blocks should also survive until the solution is rejoined by the selection and crossover process.

The probability $P$ (risk level) of having all necessary parameter values or alleles present in the population, is evaluated most easily by the complement: what is the probability of a missing solution building block:

$$1 - P = \sum_{i=1}^{n_g} (1 - p_i)^n_P,$$
where $n_P$ is the size of the population, $n_g$ is the number of parameters i.e. genes and $p_i$ is the probability of the solution parameter(s) at gene $i$. In case of homogenous genes, for which $\forall i: p_i = p$, this equation reduces to

$$1 - P = n_g(1 - p)^{n_P}.$$ 

In the further special case, when only one parameter value is valid for a solution, holds $p = 1/n_a$, where $n_a$ is the number of all possible values of a parameter i.e. the number of alleles. Usually $n_a = 2^{n_{bb}} - 1$, where $n_{bb}$ is the length of the building block in bits. By taking a logarithm of the above equation we get an equation for $n_P$ as function of the risk level $P$, number of genes $n_g$ and solution allele probability $p$:

$$n_P(P, n_g, p) = \frac{\log(1 - P) - \log(n_g)}{\log(1 - p)}$$

The values of this population size estimate are shown for several risk levels $P$ and number of alleles $n_a = 1/p$ in Table 8.2. From the above equation we can easily solve the risk level $P$ as function of population size $n_P$ and allele probability $p$:

$$P(n_P, n_g, p) = 1 - n_g(1 - p)^{n_P}$$

$$= 1 - n_g e^{n_P \ln(1 - p)}$$

$$\approx 1 - n_g e^{-m_P}.$$ 

This exponential behaviour can be seen more or less clearly in figures 8.4–8.5, where the histogram of the number of fitness function evaluation $n_f$ at different population sizes are quite similar in shape so that they nicely scale with the above risk level function. Another way of using the above relation is to estimate the length of the building blocks $n_{bb}$ via $p$. Solving the above equation with respect to $p$ we get

$$p = \frac{\ln(1 - P) - \ln(n_g)}{-n_P}.$$ 

Now the average building block size $\tilde{n}_{bb}$ is approximately given by

$$\tilde{n}_{bb} = 2 \log(1/p)$$

$$= 2 \log \left( \frac{-n_P}{\ln(1 - P) - \ln(n_g)} \right).$$

### 8.3.2 Number of generations

In order to estimate the speed of genetic search, let us assume that the search is at first primarily/only done by crossover and selection: The motivation behind this assumption is the empirical fact, that the role of mutation in both natural and artificial evolution seems to be much smaller than the role of crossover, when the solution should be found relatively fast. For the long run the roles interchange and mutation becomes the driving force of evolution (re)creating extinct and eventually totally new alleles.

Let us thus assume, that in each generation $i$ after the initial one the number of building blocks of the solution is increased by a factor $s_i$. This increase is continually driven by...
crossover and selection until either the hibernating but scattered solution is happily rejoined or otherwise the search transits to the next phase where the missing building blocks are primarily searched by mutation and selection. This two phase functioning can be clearly seen in figures 8.4–8.6. It also comfortably resolves the “crossover or mutate” dilemma: which one is more important in genetic algorithm search. For efficient processing both are vital: crossover for the first few starting generations and mutations there after.

Let the size of the initial population be $n_P$ and the selection efficiency (ratio) acting on the hibernating building blocks during each generation $i$ be $s_i$. Search by crossover ceases naturally with the diversity of the population i.e. when the population is filled with more or less identical specimens by the selection process. Assuming a constant selection efficiency $s$ this happens at generation $n_G$ for which

$$s^{n_G} = n_P$$

i.e., when

$$n_G = \frac{\log(n_P)}{\log(s)}.$$ 

In the somewhat idealistic case, when $s = 2$, then

$$n_G = 2 \log(n_P),$$

which is the number of steps in binary search among $n_P$ items in accordance with the assumption that genetic algorithm search can be seen as a stochastic parallel binary search process. Using the above equation we get for the expected number of fitness function evaluations $n_f$:

$$n_f = n_P n_G = n_P 2 \log(n_P),$$

which seems to be more or less empirically valid for reasonable population sizes $n_P$.

As a comparison to random search: if the fitness values are normally distributed, with mean $\mu = 0$ and standard deviation $\sigma^2 = 1$, then the expected number of evaluations $n_f$ needed to exceed value $x$ i.e. $\max\{x_i\} \geq x, i = 1, \ldots, n_f$ is proportional to $\sqrt{e^{ax^2}}$ [Wei90], which is really a fast growing function (see Table 8.3). As can be seen the problem is practically unsolvable for a pure random search approach already, when $x \approx 8$.

Using the empirical distributions shown in figure 8.2 for the Onemax, maximum sum, polyomino, 3-SAT and the snake problems the above model gives the following estimates for the expected number of evaluations before the solution is found $10^7, 30.000, 600, 11, and 2000$ correspondingly. All figures, except that for the Onemax problem, are desperately too small implying that the tails of the distributions of combinatorial problems are not extrapolated well by the peak region implied normal distribution. This can be seen already from the histograms of figure 8.2: the tails of the distributions deviate more and more from the paraboloid the further away from the peak we are.

### 8.4 Problem set

In order to analyse genetic algorithm search efficiency we have selected the following five different type and complexity combinatorial optimisation problems to be solved exactly in our experiments:


\[ n_f = \sqrt{e^{x^2}} \]

<table>
<thead>
<tr>
<th>( x )</th>
<th>( n_f )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>90</td>
</tr>
<tr>
<td>4</td>
<td>2.981</td>
</tr>
<tr>
<td>5</td>
<td>268.337</td>
</tr>
<tr>
<td>6</td>
<td>65.659.969</td>
</tr>
<tr>
<td>7</td>
<td>43.673.179.098</td>
</tr>
<tr>
<td>8</td>
<td>78.962.960.182.681</td>
</tr>
<tr>
<td>9</td>
<td>388.084.696.243.620.000</td>
</tr>
<tr>
<td>10</td>
<td>5.184.705.528.587.070.000.000</td>
</tr>
</tbody>
</table>

Table 8.3: The number of samples \( n_f \) needed on the average to exceed \( x \), when \( x \) is normally distributed with \( \mu = 0 \) and \( \sigma^2 = 1 \).

- Onemax,
- 7 fields maximum sum,
- seven polyomino block tiling on a square,
- a 233 clause 3-SAT problem, and
- a 16 degrees of freedom toy brick problem we call snake folding.

All problems share a 34 bit long chromosome vector structure consisting of seven genes (ints of C, see Table 8.4 for one example problem encoding) of lengths 4, 4, 5, 5, 6, 6, and 4 bits correspondingly. 34 bits give approx. \( 16 \times 10^9 \) possible combinations, which is quite much but not too much to allow massive “search until success” repetition of experiments to get some significance level for statistical analysis.

8.4.1 All are ones

Our first test problem is the well-known Onemax problem: for a binary string \( x \) of length \( l \) this is the problem of maximising

\[ \sum_{i=1}^{l} x_i, \quad x_i \in \{0, 1\}^l. \]

The fitness is the number of one bits in the chromosome vector (maximum = 34 = solution). This problem should be ideal in terms of building blocks and fitness landscape correlation: each bit position contributes to the fitness independent of any other position. But as we will see this simple problem is not very simple for genetic approach, perhaps because the fitness landscape is actually highly multimodal [Cul95].

Onemax has been used by many GA researchers as a test problem [Cul95, HR96, Jon95], because in many respects it is a simple and well suiting problem to genetic algorithms.
8.4.2 Maximum sum

The next problem is to find the maximum sum of the seven genes, the lengths of which vary from 4 to 6 bits (see Table 8.4 for chromosome structure). The fitness of the solution is $233 = (15 + 15 + 31 + 31 + 63 + 63 + 15)$, when all bits = 1. So that the Onemax and the maximum sum problem share exactly the same solution vector, while their fitness function values and distributions (see fig. 8.2) are clearly different.

Maximum sum is actually a linear problem, which can be best solved by linear programming.

8.4.3 Tiling polyominoes

The problem is to fill a $4 \times 4$ square by the following blocks called polyominoes [Gol94b]:

![Polyomino Blocks](image)

<table>
<thead>
<tr>
<th>$i$</th>
<th>int</th>
<th>bin</th>
<th>$\phi$</th>
<th>$x$</th>
<th>$y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>01 11</td>
<td>-</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>11 00</td>
<td>-</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>00 01</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>01 00</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>26</td>
<td>01 10 10</td>
<td>$-\pi/2$</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>44</td>
<td>10 11 00</td>
<td>$-\pi$</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>11</td>
<td>10 11</td>
<td>-</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

| bits | 34 | 34 | 6 | 14 | 14 |

Table 8.4: An example of a polyomino tiling (fitness $f = 12$) and the parameters of the polyominoes: $i =$ block index, int = parameter as an integer, bin = parameters as a binary string, $\phi =$ angle of rotation, $x$ and $y$ the x- and y-coordinates of the bottom left corner of the blocks. This chromosome structure is shared by all test problems.

The fitness of our polyomino tiling or packing problem is defined to be the number of unit squares covered by the seven polyomino blocks. The blocks have integer coordinate values (0, 1, 2, 3). In addition polyominoes 3 and 4 (dominoes) can be in vertical or horizontal orientation ($\phi = 0, \pi$) and polyominoes 5 and 6 (corner trominoes) may be in any of the four orientations $\phi = 0, \pm \pi/2, \pi$. The tiling of all the seven polyominoes is thus also encoded in 34 bits as shown in Table 8.4.

Polyominoes provide many interesting combinatorial problems. To GA research they seem to be quite new, however. The author knows only one other study of solving polyomino tiling problems using genetic algorithms [GL96].

8.4.4 3-SAT problem

The next problem is a 3 variable per clause boolean satisfiability problem (3-SAT) consisting of 233 clauses. The 3-SAT problem means the following: we have a set of three variable boolean expressions of the form

$$v_i + v_j + v_k,$$
where $v_i$ is a boolean variable or it’s negation and ‘+’ stands for conjunction i.e. boolean or-operation. The problem is now to find such a truth assignment to the variables that the maximum number of clauses evaluates to true i.e. our problem is actually of MAX 3-SAT type.

In our test case we have chosen 34 variables to meet the chromosome compatibility requirement and 233 clauses to be comparable with the maximum sum problem. The C-language routine SATinitialize that generates our 3-SAT problem instance is shown below:

```c
void SATinitialize(int t)
// Initialize SAT problem cases:
// t gives the number of clauses (=233).
// Variables and their complements are
// represented by 2D arrays VARS and NVARS,
// which are assumed to be reset.
// setBit() sets the bits in these arrays
// so as to meet the chromosome structure.
// Nbits is the number of bits or
// variables (=34).
{
    int i,j;
    for (i=0; i<3*t/2; i++) {
        j = (i+i/Nbits)%Nbits;
        setBit(len,VARS[i%t],j);
        setBit(len,NVARS[(i+1)%t],j);
    }
}
```

The fitness of our 3-SAT problem is an integer function, the maximum value of which is 233, when all the boolean clauses are true i.e. satisfied. 3-SAT is proven to be an NP-complete problem [GJ79], but only a small fraction of all possible cases are actually that hard to solve [Ala92c]. It seems that when the ratio of the number of clauses to the number of variables is approximately 4.5 then the most difficult cases appear [SML96]. In our 3-SAT problem case the ratio is $233/34 \approx 6.85$, which should not imply an extremely difficult case. It turned out that our problem was actually quite easy.

Being a basic NP-hard problem boolean satisfiability problems have been also quite popular in genetic algorithm research [Bit93, DH94, EvdH97, FPS01, Fra94, GV97, HD94, ?, JS89, LK94, MR99, Par95].

### 8.4.5 Folding snake

The last and most difficult problem for genetic algorithm is what we call the folding snake. The “toy snake” consists of 27 small cubic bricks arranged like a string of pearls, each brick able to rotate with respect to its predecessor. The problem is to fold the snake into a $3 \times 3 \times 3$ cube (see fig. 8.1). It turns out that there are sixteen degrees of freedom that are enough to solve the problem. Each degree of freedom may have rotation value 0, 1, 2, or 3 times $\pi/2$ giving total $2 \times 16 = 32$ bit long chromosome representation. This is 2 bits less than in the other problems, but that was not considered a primary factor for GA search efficiency, however. Due to symmetry reasons the solution is not unique, but there are $4 \times 2$ solutions: the first degree of freedom may have any of the 4 possible values while the fourth has 2 possibilities. In any case the author feels that this type of problem may be NP-complete (the
similar protein folding problem namely is \([\ldots]\)). At least it is the most difficult problem in our test set.

**Figure 8.1:** The toy snake folding problem of 27 bricks: the initial unfolded (top) and final folded (bottom) states. The relevant rotational degrees of freedom are shown by emphasised lines between bricks. Axis of rotations are shown for brick pairs (2,3) and (3,4).

The snake can be thought of as a simple model of a multi degree of freedom mechanism such as a robot or a macromolecule (see section ??). E.g. solving inverse kinematics can be studied by changing the fitness function to a function that measures the distance of the end brick from a given destination. Constraining the workspace would naturally lead to path planning type problems.

Sad to say to both robot engineers and chemists, the snake folding seems to be quite a hard problem at least for a pure genetic algorithm approach.

### 8.5 Experiments

The following properties of the test problems were analysed:

- distribution of fitness values in order to see how probable it is to find the solution by change (mutation alone) and
- autocorrelation of fitness values along a random one-bit mutation path in order to have a rough idea of the fitness landscape for search efficiency (crossover and selection).

In addition the effect of population size on search efficiency and success was analysed by running 1000 times optimization each time using a different random number seed (and sequence). This test was repeated on each problem of the test set.

#### 8.5.1 Fitness distributions

The first experiment was to reveal the fitness function distributions. This was done by generating \(10 \times 10^6\) random chromosomes and making a histogram of the fitness values (figure 8.2). In spite of the quite large number of trials no solution for any of the test set problem was actually found when evaluating the histograms suggesting that pure random search seems not to be efficient to solve our test set problems or at least the used random number sequence does not contain the solution pattern.
Observe that the shapes of the histograms of the Onemax, maximum sum and the poly-
ominio problems are quite symmetrical and near second order polynomials on the logarithmic
scale. That means that their distributions are quite near $e^{x^2}$ i.e. Gaussian as assumed in
many statistical analysis texts. But this is true only near the peak.

Distributions of the 3-SAT and snake problems are markedly skewed towards the solution,
but as we will see this seems to help GA only in the 3-SAT case to reveal the solution easily.

**Figure 8.2:** Distribution of problem fitness values ($1$ = Onemax, $3$ = 3SAT, $\bullet$ = maximum
sum, 2mm = polyomino and $S$ = snake folding) of $10 \times 10^6$ random samples. Observe that random
sampling has not found any solution for any problem (the best trial for 3-SAT has the fitness 232).
Solution fitness = 16/34/233/-9 is fixed and shown by a vertical arrow (↓), while x-scale varies with
the problem. All problems share $x = 0$, except the snake problem, which has negative fitness values
otherwise it has the same scaling as the Onemax problem.

### 8.5.2 Combinatorial explosion

**Sorry, still working on this section!**

Figure 8.2 shows some typical distributions of combinatorial problems. They are all more
or less similar to Gaussian distribution i.e. there are a lot of average combinations, while
the interesting extreme ones are really few in number.
8.5.3 Fitness landscape

In order to have a rough idea of the shape of the fitness landscapes, the following autocorrelation function \( \text{acorr} \) as a function of the Hamming distance \( h_{i,j} \) between chromosomes \( c_i \) and \( c_j \) was evaluated:

\[
\text{acorr}(h_{i,j}) = \frac{100\%}{n} \sum_{i \neq j} \left( \frac{2 \min(f_i, f_j)}{\max(f_i, f_j)} - 1 \right),
\]

where \( n \) is the number of samples sharing the same Hamming distance and \( f_i = f(c_i) \) is the fitness of chromosome \( c_i \). In figures 8.3 is shown this autocorrelation function for the test problems. Each histogram was evaluated by generating a \( 10^6 \) long single random mutation path and evaluating the above correlation index among 67 (= \( 2 \times 34 - 1 \)) consecutive chromosomes along this path.

Notice, that the autocorrelations were evaluated only to \( h = 29 \) because of the way how the evaluation was done: it is quite improbable that all the bits would be inverted during a relatively short (=67) single mutation path. For the same reason the values for high values of \( h \) are much more noisy than the values for shorter \( h \).

As can be seen 3-SAT and Onemax problems have the highest correlation, while the polyomino tiling and snake problems have lower correlations for short correlation distances. Observe, however, that the definition of our autocorrelation differs from the traditional definition. E.g. the skewness of the distribution implies a relatively higher overall correlation. This shows clearly in autocorrelation of the 3-SAT, snake and polyomino problems, while Onemax and especially maximum sum has quite a linear decreasing trend with increasing \( h \).

The multimodality of the Onemax problem slowing GA search [Cul95] can be seen as a small periodic component on the mainly decreasing trend of the autocorrelation function. This periodicity can be seen more clearly in the close-up view of the autocorrelations shown in figure 8.3(b).

The snake folding problem, which, as we will see, seems to be the most difficult problem in our test set has quite a high autocorrelation after all. It is nearly as high as that for the Onemax problem and mostly higher than for the polyomino problem, which as we will see, will be much easier for GA.

In basic statistical analysis (e.g. in [Wei90]) statistical isotrophy is usually assumed. That means that the autocorrelation, on the average, is independent of the point where it was evaluated. This isotropy is the basic assumption also in the NK-landscape model, where contributions of each gene position are assumed to be normally distributed and independent of each other. For an optimization problem it is not at all \( a \) priori certain that the statistical isotrophy holds, however.

If the problem is not statistically isotrophic, then the interpretation of the autocorrelation is neither straightforward. In order to test the isotrophy assumption, we have evaluated the autocorrelation in the vicinity of one of the solutions of the snake problem, which is potentially the most unisotrophic problem in our test set. The local autocorrelation, denoted by ‘L’, is shown in figure 8.3 among the “global” autocorrelations. As can be seen the difference between the local and global autocorrelation is huge. Another point is that in the random path used in the evaluation of the autocorrelation the most probable values around the mean determine the correlation thus greatly amplifying the possible non-isotrophism. This result immediately warns us not to rely too much on \( a \) priori assumptions when solving an
optimization problem, the statistical properties of which are not known, which unfortunately usually is the case.

Figure 8.3: (a) Correlation of fitness values. (b) A close-up view of (a) at short correlation lengths. (1 = Onemax, 3 = 3-SAT, M = maximum sum, 2mm = polyomino, and S = snake folding, L = local autocorrelation of snake folding around a solution).

8.5.4 Search speed

The effect of population size on search speed and success was analysed by running 1000 times optimization each time using a different random number seed (and sequence). The experiment was repeated on all test problems and on several population sizes, while the other parameters were kept fixed. No attempt was made to optimize the parameters of the genetic algorithm. E.g. genewise crossover was used in all tests, even if it is not as good as binary crossover for problems like the Onemax.

The search was continued until either the solution was found or approximately the given number of fitness functions was evaluated. For all except the snake folding problem we had limit = 50,000 fitness function evaluations. For snake folding the limit was set at 160,000
8.5. EXPERIMENTS

evaluations. The result of the search speed experiment are shown in Tables 8.6-8.7 and figures 8.4-8.6.

The parameters of the GA were: elitism = population size/2, while the other parameters were fixed for all experiments: crossover rate = 50%, single gene mutation rate = 10%, swap rate = 10%, and total chromosome mutation rate = 30%.

As can be seen the solution is found using more or less fitness function evaluations depending on the problem case and population size. Obviously the speed of search gives a rough measure of the relative complexity of the test problems with respect to GA optimization.

One hypothesis is that the difference is due to the building blockability i.e. separability of the problems. Unfortunately the estimate \( n_{bb} = 2 \log(n_P / \ln(1 - P)) \) can not be used because for all test problems the selection efficiency is monotonic without any clear asymptoticity.

8.5.5 Population size

In figures 8.4–8.6 is shown the effect of population size on the number of function evaluations needed to find the solution for the polyomino, Onemax and maximum sum problems. The histograms have been created by running GA 1000 times using different random number seeds and population sizes ranging from 25 to 1600. As can be seen the larger the population size the more function evaluations are needed. From this we could conclude that the small population sizes are most effective, but that is true only if the solution is actually found within a reasonable time. Namely the probability to find the solution by using at most a given number of evaluations (50.000) also decreases with decreasing population size (see Table 8.7).

![Figure 8.4: Distribution of evaluations needed for finding the solution to the polyomino layout problem. Parameters of the GA are the same as in the previous figure except for the population size \( n_P \), which was 50, 100, 200, 400, 800, and 1600, while elite = \( n_P / 2 \). Solid line (—) is 100(1−e\(^{-n_f/3000}\)).](image)

8.5.6 Search efficiency model

In Table 8.6 is shown the average number of generations needed to solve the test problems. It seems that the average number of generations needed can be quite precisely approximated
Figure 8.5: Distribution of evaluations needed for finding the solution to the 34 bits Onemax problem. The parameters of the GA are the same as in the previous figure, except for population size maximum = 800.

Figure 8.6: Distribution of evaluations needed for finding the solution to the maximum sum problem. The parameters of the GA are the same as in the previous figure.
by the (empirical) formula:

\[ \hat{n}_G = n_G^\infty + \frac{n_P^\infty}{n_P - n_0}, \]

where \( n_G^\infty \) is the asymptotic number of generations needed for a very large population, \( n_0 \) is the minimum population size and \( n_P^\infty \) is the size of a population that does not much reduce the number of generations i.e. “a very large population size” typically having ranged from a few hundred to over 2000 in our experiments.

In Table 8.5 is shown the model parameters for the average number of generations \( n_G \) needed. This estimation is also shown as a solid line in figure 8.7 for the polyomino problem, where the effect is most clearly manifesting itself.

<table>
<thead>
<tr>
<th>Problem</th>
<th>( n_G^\infty )</th>
<th>( n_P^\infty )</th>
<th>( n_0 )</th>
<th>ES</th>
<th>( n_P )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Onemax</td>
<td>7.25</td>
<td>520</td>
<td>9</td>
<td>1.11</td>
<td>50-1600</td>
</tr>
<tr>
<td>max sum</td>
<td>10.5</td>
<td>2000</td>
<td>5</td>
<td>1.46</td>
<td>100-1600</td>
</tr>
<tr>
<td>polyomino</td>
<td>10.5</td>
<td>2650</td>
<td>36</td>
<td>0.80</td>
<td>100-1600</td>
</tr>
<tr>
<td>3SAT/233</td>
<td>3.5</td>
<td>470</td>
<td>8</td>
<td>3.10</td>
<td>25-1600</td>
</tr>
<tr>
<td>snake</td>
<td>26</td>
<td>15000</td>
<td>72</td>
<td>0.58</td>
<td>200/800/3200</td>
</tr>
</tbody>
</table>

Table 8.5: Approximate parameter values \( n_G^\infty \), \( n_P^\infty \) and \( n_0 \) of the average generations model \( \hat{n}_G = n_G^\infty + \frac{n_P^\infty}{n_P + n_0} \) for the test problem set.

<table>
<thead>
<tr>
<th>( n_P )</th>
<th>1max</th>
<th>maxs.</th>
<th>poly.</th>
<th>3SAT</th>
<th>snake</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>415.47</td>
<td>-</td>
<td>-</td>
<td>21.98</td>
<td>-</td>
</tr>
<tr>
<td>50</td>
<td>8.89</td>
<td>30.00</td>
<td>121.57</td>
<td>28.92</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>3.05</td>
<td>19.09</td>
<td>81.37</td>
<td>2.69</td>
<td>-</td>
</tr>
<tr>
<td>*100</td>
<td>1.41</td>
<td>2.94</td>
<td>12.36</td>
<td>1.32</td>
<td>-</td>
</tr>
<tr>
<td>200</td>
<td>1.72</td>
<td>13.10</td>
<td>28.46</td>
<td>1.50</td>
<td>200.71</td>
</tr>
<tr>
<td>400</td>
<td>1.29</td>
<td>4.50</td>
<td>10.89</td>
<td>1.23</td>
<td>-</td>
</tr>
<tr>
<td>800</td>
<td>1.04</td>
<td>1.86</td>
<td>4.80</td>
<td>1.07</td>
<td>37.50</td>
</tr>
<tr>
<td>1600</td>
<td>0.90</td>
<td>1.43</td>
<td>4.08</td>
<td>0.97</td>
<td>-</td>
</tr>
<tr>
<td>3200</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>6.78</td>
</tr>
</tbody>
</table>

Table 8.6: The average number of generations \( n_G \) needed for finding the solution at different population sizes \( n_P \) and problems using unoptimized genetic algorithm and a more optimal (*100) one.
The data from Table 8.6 is also displayed in figure 8.7(a). In figure 8.7(b) is shown the corresponding average selection efficiency felt by specimens i.e. \( s/n_P \) given by

\[
\frac{s}{n_P} = \left[ n_P \right]^{(1/n_G - 1)}.
\]

As can be seen the efficiency is at its maximum at relatively small population sizes actually not that far away from the so popular value \( n_P = 50 \). Unfortunately the search success rate also rapidly decreases with decreasing population size making the definition of the exact location of the best search efficiency uncertain. It seems that the population size range \([n_c, 2n_c]\), where \( n_c \) is the length of the chromosome vector in bits, is quite reasonable as was already suggested in [Ala91b].

It must be reemphasised that no attempt to optimize the parameters of the genetic algorithm itself was done either before or during the experiments. Comparisons to other experiments seem to show that the genetic algorithm used was not that bad, however. In Table 8.6 is also shown the results of a more optimal genetic algorithm at population size 100 (denoted as *100). As can be seen the relative difference to the corresponding non-optimal results is of the order 2.

## 8.6 Discussion and recommendations

### 8.6.1 Comparisons to other results

In our experiments we have partly used the same or similar problems than in our previous studies and studies by other authors. Unfortunately in our previous experiment we measured the efficiency in CPU time units and not in the number of fitness function evaluations [Ala91b].

Höhn and Reeves used standard GA to solve the Onemax problem of length \( n_g = 9, 19, 29, 39 \) and 49 bits. Their result showed that the number of function evaluations increased linearly with length [HR96] at population size 200. An interpolation for 34 bits after their data would give \( n_f \approx 2447 \) function evaluations, while in our experiment \( n_f \approx 2014 \), when population size is 200.

### 8.6.2 Autocorrelation analysis

If the problem is not statistically isotrophic, then the interpretation of the autocorrelation is neither straightforward. The local autocorrelation for the most difficult problem, snake
Figure 8.7: The average number of generations $n_G$ at different population sizes $n_P$ and test problems ($1 = \text{Onemax}, 3 = \text{3-SAT}, M = \text{maximum sum}, 2\text{mm} = \text{polyomino}, \text{and S} = \text{snake folding}, \text{Sbin} = \text{snake folding using binary crossover}$). The solid line shows the average number of generations needed for the polyomino tiling given by the estimate $n_G \approx 10.5 + 2650/(n_P - 36)$.

folding, is shown in figure 8.3 among the “global” autocorrelations. As can be seen the difference of the local and global autocorrelation is huge. This result warns us not to rely too much on a priori assumptions when solving an optimization problem, the statistical properties of which are not known, which unfortunately usually is the case. Actually the statistical empirical algorithm analysis is one of the very neglected areas of computer science [?].

The other conclusion we can draw from the modest local autocorrelation is that it must be one major factor explaining the hardness of the snake problem.

The 34 variable MAX 3-SAT problem seems to be the easiest of the five test problems. This can be see both as a fast and reliable processing. This is somewhat surprising because 3-SAT is known to be an NP-complete problem. It must be kept in mind that the NP-completeness does not tell much about the given problem instance. A given MAX 3-SAT problem may be difficult, but not necessarily. The easiness may actually be a more general property of rule based systems for which Hahnert and Ralston found that small population sizes are the most efficient ones [HR95]. The maximum sum and polyomino tiling problems are both quite difficult, but in a different sense. If we look at what happens during the active search phase, we see that the polyomino problem is faster, but unfortunately also less reliable than the GA search on the maximum sum problem.

It is obvious that the polyomino problem better fulfills the building block hypothesis: once we have the building blocks available the rest goes more reliably than in the search for the maximum sum problem solution, where the least significant bits are easily lost during the first few generations, where the most significant bits dominate search done by selection and
crossover. When the population size is large also the least significant bits of the maximum
sum problem survive and the behaviour of both problems are quite similar.

The snake folding problem, interesting because of its relation to some important practical
optimisation problems, turned out to be far more difficult than the rest of the test problems.
It seems that the short correlation length around the solution explains this difficulty and also
warns us not to rely too much on statistical isotrophy assumptions when using autocorrelation
in estimation of problem difficulty.

8.6.3 Sensitivity

Perhaps the most positive property of genetic algorithm is its robustness i.e. non-sensitivity to
most of its parameters. It is actually difficult to find any parameter that would dramatically
affect its functioning. That is especially a nice feature in pilot projects applying genetic
algorithm optimisation to new problems. If the genetic approach is suited at all, it usually
gives quite good results. The processing time might be quite long, but that is not usually
a great disadvantage in engineering design work where part of the engineers work could be
replaced in this way by an automatic optimisation system like in [AL97].

8.6.4 Recommendations

In this chapter we will give a few recommendations for those who plan to use genetic algo-
rithms in combinatorial optimization.

Population size

In case of random initial population it is wise to use a population size that gives a high
enough probability for each solution building block to be included. In case of integer or real
number encoded problems remember to take into account also the effect of encoding.

A combinatorial problem may not be isotrophic. This has a profound effect on population
size. The shorter the correlation length the larger the population size needed.

In combinatorial problems we should also consider local hill-climbing, which effectively
makes fitness landscape correlation much longer.

The acceptable risk level of not solving the problem within a given number of function
evaluations is also an essential factor having influence on proper population size. The lower
the risk level the larger the population is needed, which unfortunately also leads to slower
processing on the average.

In many real valued problems the fitness landscape is much more strongly correlated than
in discrete combinatorial problems and this has a profound effect on the search efficiency
and optimal population size, but that is beyond the scope of this study. It must be noted,
however, that in many cases the real encoded problem can be effectively solved by using
a surprisingly small population size and local hill-climbing might even make convergence
considerably faster.

Number of generations

Easy problems should be solvable usually within a few tens of generations. Longer processing
time may be due to insufficient population size or might be caused by problems with fitness
8.7. CONCLUSIONS

function and/or short correlation length and/or non-isotrophism. The less correlated or local fitness landscape the larger the population size needed.

Fitness function

Be aware that in constrained problems the constraints themselves affect fitness values and further landscape correlation. By careless penalty values we can make landscape correlation length really short and thus prevent selection from catching those trials that are “just round the corner”.

Genetic operators

In binary encoded problems use binary uniform crossover. Be sure that during the later generations the mutation operator is capable of producing the missing building blocks while not too much disturbing the already found blocks. Use elitism but do not prevent creation of new building blocks too much either.

Summary of recommendations

The author supposes that the most important advice is: if there already exists a good solution algorithm with reasonable processing time, then it is usually just a waste of time to try to replace it with a genetic algorithm. And really, a hybrid combining the best features of genetic and traditional algorithm is always potentially superior to either one alone. For example the snake problem is quite efficiently solved by a simple backtracking approach.

8.7 Conclusions

In this study we analyse empirically genetic algorithm search efficiency on five combinatorial optimisation problems with respect to building blocks and fitness landscape. Four of the problems were quite easy for genetic algorithm search while the fifth, the folding problem, turned out to be very hard due to the uncorrelated fitness landscape around the solution.

The results show that genetic algorithm is efficient in combining building blocks if the fitness landscape is well correlated also around the solution and if the population size is large enough to shelter all necessary solution building blocks. An empirical formulae for the average number of generations needed for optimization and the corresponding risk level for the test set and population sizes was further given.

Directions for future study include

• a larger set of test problems
• parametrised and more complex test problems
• comparative studies with other methods (simulated annealing, tabu search, etc)
• analysis of other parameters of genetic algorithms
• more rigorous statistical analysis of the experimental results
• an analysis of the non-isotrophy and its relation to population size and convergence
• testing statistical isotrophy of combinatorial problems
• analysis of the effect of local hill-climbing on correlation and search efficiency
Chapter 9

Fitness function

e most important part of any optimisation is the fitness function also called cost function, or goal function. The selection of a proper optimisation method depends on the properties of the fitness function. As we have already notices, in principle there is not much restrictions for the fitness function when evolutionary optimisation methods are used.

The following list gives a number of features that are important when selecting an optimisation method.

<table>
<thead>
<tr>
<th>Property</th>
<th>Variables</th>
<th>Primary optimisation method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linear</td>
<td>1-1M</td>
<td>Linear programming</td>
</tr>
<tr>
<td>Continuous</td>
<td>1-3</td>
<td>Calculus, Gradient based</td>
</tr>
<tr>
<td>ditto, multimodal</td>
<td>1-10</td>
<td>Gradient based (Newton)</td>
</tr>
<tr>
<td>ditto, multimodal</td>
<td>1-100</td>
<td>GA with gradient</td>
</tr>
<tr>
<td>Discrete</td>
<td>1-5</td>
<td>Dynamic programming</td>
</tr>
<tr>
<td>ditto, multimodal</td>
<td>1-100</td>
<td>GA with Dynamic programming</td>
</tr>
<tr>
<td>NP</td>
<td>any</td>
<td>GA</td>
</tr>
<tr>
<td>Pareto</td>
<td>any</td>
<td>GA</td>
</tr>
<tr>
<td>Others</td>
<td>any</td>
<td>GA</td>
</tr>
</tbody>
</table>

Table 9.1: How to select a proper optimisation method for a given type of cost function.

WORKING ON ....

9.1 Fitness landscape

It is obvious that the shape or topology of the fitness function has a profound influence on the optimisation speed. A linear problem can be effectively solved by linear programming while a complex discrete problem may be NP hard.

Multimodality i.e. the number or density of local extremes is effectively preventing the usage of most traditional optimisation methods such as gradient based methods.

If a problem is discrete it often means that it needs much processing time if there is not any way to select the most promising trials. For the set of NP hard problems there is not any such efficient way to brune the alternatives, so that they are hard to solve with any method.
Next we will look some examples of fitness landscapes of problems. Let us start with the famous Boolean satisfiability problem (SAT).

9.2 SAT landscape

In complexity theory, NP denotes the set of all (decision) problems solvable by a non-deterministic polynomial time algorithm, while P denotes the set of all (decision) problems solvable by a deterministic polynomial time algorithm. NP problems are considered “hard” in the sense that they are not currently solvable in deterministic polynomial time.

The canonical example of a problem in NP is the boolean satisfiability problem (SAT): Given an arbitrary boolean expression $E$ of $n$ variables, does there exist an assignment to those variables such that the expression is true?

Every problem L in NP can be transformed into an equivalent SAT problem in polynomial time (Cooke’s theorem), the reverse polynomial time transformation may not exist. Those problems in NP which do have 2-way transformations form an equivalence class of “equally hard” problems and have been called NP-complete problems.

9.2.1 Related work

The problem of hard NP problem cases has been dealt with at least in [?, CKT91].

Genetic algorithmic solution of the SAT problem has been dealt with at least in [JS89], where the following fitness function $f$ alternatives are suggested for a boolean expression $E$:

1. $f \equiv E$, which is the trivial fitness function having the obvious drawback that it does not give any information of solutions that are “nearly” true.

2. CNF: Transform the given expression $E$ into conjunctive normal form (CNF) and define fitness function to be the total number of top level conjuncts which evaluate to true. Unfortunately one can not in general perform the transformation in polynomial time without introducing a large number of additional boolean variables.

3. MINMAX: define

$$
\begin{align*}
    f(0) &= 0 \\
    f(1) &= 1 \\
    f(-e) &= 1 - f(e) \\
    f(\land e_1 \ldots e_n) &= \min(f(e_1) \ldots f(e_n)) \\
    f(\lor e_1 \ldots e_n) &= \max(f(e_1) \ldots f(e_n))
\end{align*}
$$

4. average: the above definition of fitness function $f$ can be changed into a more continuous one by changing the definition of $f(\land e_1 \ldots e_n)$:

$$
    f(\land e_1 \ldots e_n) = \frac{\sum_{i=1}^{n} f(e_i)}{n}
$$

which rewards more “nearly true” $\land$-clauses.

5. modified average: use the above fitness function after having transformed clauses of the form $\neg(\land e_1 \ldots e_n)$ using De Morgan’s law.
6. non-linear average: the average term can be raised to some small integer power $p$:

\[ f(\land e_1 \ldots e_n) = \left( \sum_{i=1}^{n} f(e_i)/n \right)^p \]

without losing the property that $f(\land e_1 \ldots e_n) = 1$ iff $\land e_1 \ldots e_n = 1$.

### 9.2.2 Criticism

The above fitness function designs deserve some criticism: The approach has been to find a fitness function $f$ that both gives some distance measure from the solution and at the same time gives the value of the object boolean expression $E$ i.e. a mapping $f : E \to [0, 1]$. This seems to introduce too much a priori constraints for the single scalar function $f$. One may also ask if it is really reasonable to try to code the fitness of a possibly extremely complex expression into one scalar value which is obviously not able to carry all structural information of the boolean expression.

One further point is that a good SAT problem solving algorithm should be able to solve the problem at once in the trivial case of every variable occurring at most once in the expression, when the solution assignment can be deduced straightforwardly.

### 9.2.3 Notations

Boolean functions are represented in prefix notation and the most important notations used are:

- $1, 0$: true and false
- $\phi$: the so called don’t care bit; also the empty set
- $\#$: number of
- $\neg e$ or $\overline{e}$: negation of expression $e$
- $(\land e_1 \ldots e_n)$: and function of $n$ variables
- $(\lor e_1 \ldots e_n)$: or function of $n$ variables

### 9.2.4 SAT fitness function

A good SAT fitness function should have at least the following “rules of thumb” properties:

1. Evaluation of trivial and simple problems should not take much time and the evaluated solutions should be the exact ones.

2. The more occurrences a variable or subexpression has the greater its weight to the initial guess should be.

3. General subexpressions should have more weight than special ones.

By a general subexpression we mean a clause that covers a large set of variable assignment combinations. A single variable clause $v_i$ is an example of extreme generality; it is true for half of the problem space.
9.2.5 Deduction of subexpressions

The above proposed SAT and fitness function evaluation are both done starting from constants and variables and going up the expression tree. However, if the object expression does not have multioccurrence variables, the obvious evaluation order is top down by the following rules:

\[ \neg e = 0 \Rightarrow e = 1 \]
\[ \neg e = 1 \Rightarrow e = 0 \]
\[ \land e_1 \ldots e_n = 1 \Rightarrow e_1 = \ldots = e_n = 1 \]
\[ \lor e_1 \ldots e_n = 0 \Rightarrow e_1 = \ldots = e_n = 0 \]

In addition we have the following formulae for the “weaker” deductions:

\[ \land e_1 \ldots e_n = 0 \Rightarrow e_1 \approx \ldots \approx e_n \approx 0 \]
\[ \lor e_1 \ldots e_n = 1 \Rightarrow e_1 \approx \ldots \approx e_n \approx 1 \]

Using these rules the solution of the SAT problem in the trivial case of every variable occurring at most once can be solved without resorting to any further iterations, at all. This applies to subgoals and can be used to reduce the expression to be evaluated by removing obvious tautologies and contradictions.

The more important fact is that we can use the deduced values as goals and subgoals of the fitness function. Let us denote the set of deduced values of subgoals and terminals as bit vector \( G \). Now we can define a fitness function \( f_G \) as

\[ f_G = H(G,V) \]

where \( H() \) is the Hamming distance between two bit vectors i.e. the number of differing bits, and \( V \) is the bit vector of subgoals and terminals of the given variable assignment.

\[ H(X,Y) = \sum_{i=1}^{n} X_i \oplus Y_i, \]

where \( \oplus \) denotes exclusive or operation. The definition of \( f_G \) can be easily generalized by assigning different weights \( w_i \) to subgoals:

\[ H_w(X,Y) = \sum_{i=1}^{n} w_i(X_i \oplus Y_i), \]

**Example 9.2.1 (The first SAT example)** Let the boolean expression be

\[ E_1(a, b, c, d) = \lor((\land a \neg b)(\land c \neg d)) \]

Then

\[ V = [a, b, c, d, \neg a, \neg b, \neg c, \neg d, (\land a \neg b), (\land c \neg d), E_1] \]

and

\[ G = [1, 0, 1, 0, 1, 1, 1, 1]. \]

The fitness values of the expression \( E_1 \) can be found in table 9.2 for every possible assignment while in table 9.3 is shown the distribution of these fitness values together with the classification to solution and nonsolution cases. As can be seen the discrimination between these cases is clear.
Table 9.2: The goal vector $V$ and the fitness functions of the expression $E_1$. Solutions are underlined.

<table>
<thead>
<tr>
<th>abcd</th>
<th>$V_{5...8}$</th>
<th>$E_1$</th>
<th>$H$</th>
<th>$H_W$</th>
<th>$H''$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>1100</td>
<td>0</td>
<td>5</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>0001</td>
<td>1000</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>0010</td>
<td>1101</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>0011</td>
<td>1000</td>
<td>0</td>
<td>6</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td>0101</td>
<td>0000</td>
<td>0</td>
<td>9</td>
<td>12</td>
<td>0</td>
</tr>
<tr>
<td>0110</td>
<td>0101</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>0111</td>
<td>0000</td>
<td>0</td>
<td>8</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>1000</td>
<td>1110</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>1001</td>
<td>1010</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>1010</td>
<td>1111</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
</tr>
<tr>
<td>1011</td>
<td>1010</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>1100</td>
<td>0100</td>
<td>0</td>
<td>6</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>1101</td>
<td>0000</td>
<td>0</td>
<td>8</td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td>1110</td>
<td>0101</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>1111</td>
<td>0000</td>
<td>0</td>
<td>7</td>
<td>10</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 9.3: The distribution of the fitness functions $H$ and $H_w$ and the solution classification of the expression $E_1$.

<table>
<thead>
<tr>
<th>$f \equiv H(V,G)$</th>
<th>$f \equiv H_W(V,G)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f$</td>
<td>#1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

= solution, ☐ = nonsolution.
9.2.6 Weighted fitness function

The weight values of the weighted fitness function $H_w$ can be defined recursively as follows:

\[
\begin{align*}
  w(E) &= 1 \\
  w(\tau) &= w(e) \\
  w(e_i) &= \begin{cases} \\
    \frac{w(\land e_1 \ldots e_n)}{n}, & \text{if } \land e_1 \ldots e_n = 0 \\
    w(\land e_1 \ldots e_n), & \text{if } \land e_1 \ldots e_n = 1 \\
  \end{cases} \\
  w(e_i) &= \begin{cases} \\
    \frac{w(\lor e_1 \ldots e_n)}{n}, & \text{if } \lor e_1 \ldots e_n = 1 \\
    w(\lor e_1 \ldots e_n), & \text{if } \lor e_1 \ldots e_n = 0 \\
  \end{cases}
\end{align*}
\]

i.e. the strong deductions transport the weight values unmodified down to the next level, while weak deductions cause the weights of the inputs to be divided by the number of inputs.

The values $w_i$ are relative so that we can define a new integer valued weight vector $W$ that is equivalent to $w$:

\[ W = kw, \]

where $k$ is the least integer such that $kw_i$ is integer for every index value $i$. The weight vector $W$ allows us to use pure integer arithmetics instead of slower rational or floating point arithmetics.

![Figure 9.1: The distribution of the unweighted ($H_1 = \circ$) and the weighted fitness function values ($H_{W}^{1} = \bullet$) of the expression $E_1$ as the function of the Hamming distance from the nearest solution.](image)

**Example 9.2.2 (Example 9.2.1 revisited)** If we apply the above rules to the previous example, we get the weight vectors

\[ w = [\frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, \frac{1}{2}, 1] \text{ or } W = [1, 1, 1, 1, 1, 1, 1, 1, 2]. \]

The values of $H_w$ can be found in table 9.2. If you compare the different histograms of table 9.3, you can see that the weighted fitness function is somewhat better in discriminating the solutions from nonsolution assignments.

Observe that the optimization problem can be reversed so that, instead of trying to maximize subexpressions leading to the truth value 1, we are trying to minimize the changes...
9.3. THE MAXIMUM SATISFIABILITY PROBLEM

of subexpressions to yield the value 0. When applied to the function $E_1$ this symmetric approach yields

$$W^0(E_1) = [1, 1, 1, 1, 1, 1, 2, 2] \quad \text{and} \quad G^0(E_1) = [0, 1, 0, 1, 0, 0, 0, 0].$$

The corresponding fitness function $H^0_W$ can also be found in table 9.2.

**Theorem 9.2.1** For $n$ input $\lor$ and $\land$ functions $H_W$ is a linear function of the distance from the solution assignment.

1°: Let the expression be

$$E_{\land n} = \land_{i=1}^n v_i.$$

Then

$$W = [1, \ldots, 1, 1], \quad G = [1, \ldots, 1, 1], \quad \text{and} \quad H_W(V,G) = \sum_{i=1}^n 1 \cdot G_i \oplus V_i + (\land_{i=1}^n n_i \oplus 1) = \text{number of false inputs},$$

which is a linear function of the Hamming distance of the assignment from the solution. The number of different assignments having Hamming distance $d$ from the solution is given by the binomial coefficients:

$$n_d = \binom{n}{d} = \frac{n!}{d!(n-d)!}$$

2°: Let the expression be

$$E_{\lor n} = \lor_{i=1}^n v_i.$$

Then

$$W = [1, \ldots, 1, n], \quad G = [1, \ldots, 1, 1], \quad \text{and} \quad H_W(V,G) = \sum_{i=1}^n 1 \cdot G_i \oplus V_i + n(\lor_{i=1}^n n_i \oplus 1)$$

from which it follows that

$$\begin{cases} H_W(0,G) = 2n & \text{and} \\ H_W(V \neq 0, G) = n - \text{number of true inputs} \end{cases}$$

which is again a linear function of the Hamming distance of the assignment from the solution for all values $V \neq 0$.

9.3 The maximum satisfiability problem

The boolean satisfiability problem can be generalized e.g. by the following way: If the given SAT problem does not have any solution, what is the best possible assignment satisfying maximum number $n^1$ of subexpressions? This is called the maximum satisfiability problem (MAXSAT). Pierre Hansen et al have worked on this problem and compared different heuristic methods [HJ90]. They have restricted the expression to be in the conjunctive normal form i.e. to be represented by a product of sums.
Example 9.3.1 (From [HJ90]) Let the expression $E_4$ be $\wedge_{i=1}^{12} C_i$ where

$$
C = \{ (\lor ab), (\lor a\neg b), (\lor \neg a\lor b), \\
(\lor \neg ac), (\lor a\neg c), (\lor b\lor c), (\lor \neg bd), \\
(\lor \neg bc), (\lor \neg b\lor c), (\lor cd), (\lor \neg c\lor d) \}
$$

Then

$$
V = [a, b, c, d, \neg a, \neg b, \neg c, \neg d | C].
$$

The weight vector for this type of problem can be chosen as follows: The subgoals are assigned weight 1, while the variables and negations of variables are assigned weights:

$$
w_i = n(v_i) / \sum_{j=1}^{n} n(v_j)
$$

where $n(v)$ is the number of times that a variable $v$ (or negation of it) occurs in $E$ and $n$ is the total number of variables (or negations). So that

$$
w = \left[ \frac{2}{4}, \frac{4}{4}, \frac{3}{4}, \frac{2}{4}, \frac{4}{4}, \frac{1}{4}, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 \right].
$$

Goals for terminals were chosen according to most probable (most weight) value:

$$
G = [0, 0, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1].
$$

The maximum solution is given by the assignment

$$
a = 0, \ b = 0, \ c = 0, \ d = 1,
$$

for which 11 out of the total 12 possible subexpressions are true. Observe that the formulae:

$$
\text{int}(W_{1..4}) = \text{int}\{w(a), w(b), w(c), w(d)\}
$$

and

$$
\neg\text{int}(W_{5..8}) = \neg\text{int}\{w(\neg a), w(\neg b), w(\neg c), w(\neg d)\}
$$

give us in this case the best solution and another good solution. They are the “statistically” best MaxSAT estimates.

9.3.1 Impulse response

One way of studying the effect of a variable $v_i$ is to set $v_i = 1$ or reset $v_i = 0$, while keeping the rest of the variables in reverse state ($v_j = \neg v_i, \forall j : j \neq i$). The impulse response or the fitness value of the specific variable tells the relative importance of the truth value of the variable to the overall solution.

Example 9.3.2 (Examples 9.2.1 and 9.3.1 again) The impulses $[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], \text{ and } [0, 0, 0, 1]$ give the following responses for $E_1$:

$$
I_1 = [3, 10, 3, 10]
$$

while the reverse impulses $[0, 1, 1, 1], [1, 0, 1, 1], [1, 1, 0, 1], \text{ and } [1, 1, 1, 0]$ give:

$$
I_0 = [11, 4, 11, 4].
$$
### 9.3. THE MAXIMUM SATISFIABILITY PROBLEM

<table>
<thead>
<tr>
<th>$V$</th>
<th>$C$</th>
<th>$H_W$</th>
<th>$n^1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0111</td>
<td>011111101101</td>
<td>15</td>
</tr>
<tr>
<td>0001</td>
<td>1110</td>
<td>011111111111</td>
<td>4</td>
</tr>
<tr>
<td>0010</td>
<td>1101</td>
<td>011110111111</td>
<td>9</td>
</tr>
<tr>
<td>0011</td>
<td>1100</td>
<td>011111101111</td>
<td>19</td>
</tr>
<tr>
<td>0100</td>
<td>1011</td>
<td>101111101011</td>
<td>23</td>
</tr>
<tr>
<td>0101</td>
<td>1010</td>
<td>101111110111</td>
<td>16</td>
</tr>
<tr>
<td>0110</td>
<td>1001</td>
<td>101111111011</td>
<td>26</td>
</tr>
<tr>
<td>0111</td>
<td>1000</td>
<td>101111111011</td>
<td>27</td>
</tr>
<tr>
<td>1000</td>
<td>0111</td>
<td>110101101101</td>
<td>25</td>
</tr>
<tr>
<td>1001</td>
<td>0110</td>
<td>110101111111</td>
<td>14</td>
</tr>
<tr>
<td>1010</td>
<td>0101</td>
<td>110110001111</td>
<td>32</td>
</tr>
<tr>
<td>1011</td>
<td>0100</td>
<td>110110011110</td>
<td>29</td>
</tr>
<tr>
<td>1100</td>
<td>0011</td>
<td>110011101011</td>
<td>33</td>
</tr>
<tr>
<td>1101</td>
<td>0010</td>
<td>110011101111</td>
<td>26</td>
</tr>
<tr>
<td>1110</td>
<td>0001</td>
<td>111010111011</td>
<td>36</td>
</tr>
<tr>
<td>1111</td>
<td>0000</td>
<td>111010111010</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 9.4: The MaxSat fitness function $H_W$ and the number of true subexpressions $n^1$ of the expression $E_4$.

<table>
<thead>
<tr>
<th>$f \equiv H_W(V,G)$</th>
<th>$f$</th>
<th>$#$</th>
<th>2 best solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>26</td>
<td>3</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td></td>
<td>■</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>34</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>36</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

■ = the 2 best solutions, □ = others.

Table 9.5: The MaxSat fitness function of the expression $E_4$ and the two best solutions.
CHAPTER 9. FITNESS FUNCTION

the distribution of the
distance from the best solution

Figure 9.2: The distribution of the MAXSAT fitness function of the expression $E_4$.

from which we can deduce that the assignment

$$[a, b, c, d] = [1, 0, 1, 0]$$

should be a promising solution candidate. And as you may remember this is actually the best solution. The same procedure applied to the expression $E_4$ of the MAXSAT-problem yields:

$$I^1 = [25, 23, 22, 4] \text{ and } I^0 = [27, 29, 26, 36]$$

giving a solution:

$$[a, b, c, d] = [0, 0, 0, 1]$$

which again is the best possible alternative.

At least applied to these two example problems we have shown that using only the byproducts of our fitness function evaluation it is possible to precisely estimate the best solution in a very straightforward and intuitively sound way without resorting to any further iteration steps at all. In other words, we must look for more complex problems in order to be able to utilize the main topic of our work namely, solving the assignment problem using the genetic algorithmic.

9.3.2 Exhaustive MAXSAT testing

If the number of variables is $n$, then the number of different MAXSAT clauses is $3^n - 1$ (The alternatives for every $n$ positions of the clause are: $v_i$, $\neg v_i$, and $\phi$. We omit the trivial case of no variables.). From these $3^n - 1$ clauses we can choose $2^{3^n-1} - 1$ different non empty CNF-expressions. The conjunctive normal form seems rather simple, but as you can see in table 9.6 the number of MAXSAT-problems is so large that it effectively prevents us from any exhaustive method testing for $n > 3$. The last column of the table 9.6 gives an estimate of the testing time if we optimistically assume that one MAXSAT-problem is solved in $1 \mu s$. 
Table 9.6: The number of MaxSat problems as the function of the total number of variables $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$3^n - 1$</th>
<th>$2^{3^n-1} - 1$</th>
<th>$t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0µs</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3µs</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>255</td>
<td>0.255ms</td>
</tr>
<tr>
<td>3</td>
<td>26</td>
<td>$\sim 65 \cdot 10^6$</td>
<td>$\sim 1\text{min}$</td>
</tr>
<tr>
<td>4</td>
<td>80</td>
<td>$\sim 1 \cdot 10^{24}$</td>
<td>$\sim 10^{11}\text{a}$</td>
</tr>
<tr>
<td>5</td>
<td>242</td>
<td>$\sim 10^{73}$</td>
<td>$\sim 10^{33}\text{a}$</td>
</tr>
<tr>
<td>6</td>
<td>738</td>
<td>$\sim 10^{222}$</td>
<td>$\sim 10^{100}\text{a}$</td>
</tr>
</tbody>
</table>

As we can see from table 9.6 the exhaustive testing of the MaxSat-problem is possible only for $n \in \{0, 1, 2, 3\}$. We have indeed tested these cases and the results are shown in tables 9.8, 9.9, and 9.10 where the estimation error distributions are given.

In the first test (table 9.8) occurrences of variables were counted and a variable was assigned to 1 if it had more occurrences than its negation. This simple estimate gives already good results. The next step was to fine tune the method by giving different weights to variable occurrences. In the second test (table 9.9, first row) the weight was linearly proportional to the number of 1’s of the subexpression result (4, 6). In the last test (table 9.9, second row) exponential weights (1, 2) were used. If you compare tables 9.8 and 9.9, you can see that the simple occurrence count method is the best one.

The reason for this somewhat disappointing result may be that the more variables a subexpression has the more it also limits variables and leaves less freedom to other assignments. The next test revealed that this was really the case. Table 9.10 gives the estimation error distribution when variables were given weights (6, 3, 2) i.e. a one variable subexpression has two times the weight of a two variable subexpression and three times that of a three variable subexpression. In other words we hope that the false cases collapse into as few as possible multivariable subexpressions. This counts because smaller subexpressions are more difficult to satisfy and the number of subexpressions is what is maximized. As you can see this estimate is clearly the best. All two variable cases (total 255), except the 12 ones shown in table 9.7, are correctly estimated. All these cases are symmetric in such a way that the weight of the variable and its negation are equal and we must evaluate the right assignment from other than occurrence count data. Most (8) of the erroneous estimates can be correctly estimated if we substitute the right variable assignment and re-estimate the reduced subexpressions (table 9.7). Only the doubly uncertain and symmetric 4 cases are left unsolved even after this enhanced estimation.

The dependence between the number of subexpressions $n_s$ and the number of true subexpressions is given in figures 9.3, 9.4, and 9.5.
subexpressions $C_i \Rightarrow$ estimate red. $C_i \Rightarrow$ re-est.

<table>
<thead>
<tr>
<th>$C_i$</th>
<th>estimate</th>
<th>red. $C_i$</th>
<th>re-est.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \lor b$, $a \lor \overline{b}$</td>
<td>$a = 0$, $b =?\quad b$</td>
<td>$b = 0$</td>
<td></td>
</tr>
<tr>
<td>$a \lor \overline{b}$, $a \lor \overline{b}$</td>
<td>$a =?$, $b = 0$</td>
<td>$a$</td>
<td>$a = 0$</td>
</tr>
<tr>
<td>$a$, $a \lor b$, $a \lor \overline{b}$</td>
<td>$a =?$, $b = 0$</td>
<td>$a$, $\overline{a}$, $\overline{a}$</td>
<td>$a = 0$</td>
</tr>
<tr>
<td>$a$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b =?\quad \overline{a}$</td>
<td>$b$, $\overline{b}$, $b$</td>
<td>$b = 0$</td>
</tr>
<tr>
<td>$a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a = 1$, $b =?\quad \overline{b}$</td>
<td>$\overline{b}$</td>
<td>$b = 0$</td>
</tr>
<tr>
<td>$a$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b =?$</td>
<td>don’t</td>
<td>$a =?$, $b =?$</td>
</tr>
<tr>
<td>$a$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b =?\quad a$, $\overline{a}$, $\overline{a}$</td>
<td>$a = 0$</td>
<td></td>
</tr>
<tr>
<td>$b$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b = 1$</td>
<td>$\overline{a}$</td>
<td>$a = 0$</td>
</tr>
<tr>
<td>$a$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b =?\quad a$, $\overline{a}$, $\overline{a}$</td>
<td>$a = 0$</td>
<td></td>
</tr>
<tr>
<td>$b$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b =?\quad a$, $\overline{a}$, $\overline{a}$</td>
<td>$a = 0$</td>
<td></td>
</tr>
<tr>
<td>$a$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b =?\quad a$, $\overline{a}$, $\overline{a}$</td>
<td>$a = 0$</td>
<td></td>
</tr>
<tr>
<td>$a$, $a \lor b$, $\overline{a} \lor \overline{b}$</td>
<td>$a =?$, $b =?\quad a$, $\overline{a}$, $\overline{a}$</td>
<td>$a = 0$</td>
<td></td>
</tr>
</tbody>
</table>

Table 9.7: All the by the simple occurrence method unsolved two variable MAXSAT problems.

<table>
<thead>
<tr>
<th>$n$</th>
<th>estimation error distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>99%</td>
</tr>
<tr>
<td>3</td>
<td>64.18%</td>
</tr>
</tbody>
</table>

Table 9.8: The results of the exhaustive MAXSAT estimation tests for 1, 2, and 3 variables. Estimation method: variable is true, if it has more occurrences that its negation, otherwise variable is assigned to 0. If weight is equal both alternatives are evaluated.

<table>
<thead>
<tr>
<th>weight</th>
<th>error</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>linear</td>
<td>(4, 6)</td>
</tr>
<tr>
<td>exponential</td>
<td>(1, 2)</td>
</tr>
</tbody>
</table>

Table 9.9: The estimation error distribution of all two variable MAXSAT problems using linear and exponential weights.

<table>
<thead>
<tr>
<th>$n$</th>
<th>estimation error distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>2</td>
<td>95.3%</td>
</tr>
<tr>
<td>3</td>
<td>79.82%</td>
</tr>
</tbody>
</table>

Table 9.10: The results of the exhaustive MAXSAT estimation tests for 1, 2, and 3 variables. Estimation method: variable is true, if it has more weighted occurrences that its negation, otherwise variable is assigned to 0. Weight = 6, 3, and 2 for 1, 2, and 3 variable terms.
9.3. THE MAXIMUM SATISFIABILITY PROBLEM

Figure 9.3: The distribution of the maximum (●) and the minimum (○) number of true subexpressions of all two variable MAXSAT problems. Figures give the number of erroneous estimates.

Figure 9.4: The distribution of the maximum (●) and the minimum (○) number of true subexpressions of all three variable MAXSAT problems.
CHAPTER 9. FITNESS FUNCTION

Figure 9.5: The range of the number of true subexpressions ($n_{true}$) of $n$ variable MaxSat problem as the function of the number of subexpressions ($n_s$).
Chapter 10

Applications

It is difficult to find any main optimisation related engineering problem that has not been used as a target for evolutionary optimisation. If you find such, you can congratulate yourself. Let us take an example from nature.

The author found perhaps the largest habitat in Finland of the rare and endangered butterfly woodland brown (*Lopinga achine* Scopoli, 1763) (fig. 10.1) [?]. This might be a good candidate for a new application of evolutionary optimisation to estimate the potential habitats of this and similar endangered species. However, a google search revealed that a genetic algorithm based system called *GARP* [SN92] had already been used to estimate its possible habitats [RGBM06]. For more references on *GARP* and evolutionary computation in general in ecological studies see [Alad]. So, to find totally new application areas for evolutionary optimisation may be hard.

10.1 Most popular subjects

Table 10.1 shows the most popular subjects of GA literature, from it we can see that the most popular application areas are

- engineering (all branches; most popular are power and electrical engineering, electronics and structural engineering) → Vaasa GA bibliography: http://www.uva.fi/~TAU/reports/report94-1/gaENGbib.pdf

Observe that this classification like the classification shown in table reftab:anno is not restrictive, but there are subclasses and joint classes.
## TABLE 10.1: The most popular subjects of GA papers (22.10.2012)

<table>
<thead>
<tr>
<th>Subject</th>
<th>#</th>
<th>Subject</th>
<th>#</th>
</tr>
</thead>
<tbody>
<tr>
<td>control</td>
<td>1047</td>
<td>analysing GA</td>
<td>807</td>
</tr>
<tr>
<td>analysing GA</td>
<td>807</td>
<td>parallel GA</td>
<td>659</td>
</tr>
<tr>
<td>parallel GA</td>
<td>659</td>
<td>machine learning</td>
<td>541</td>
</tr>
<tr>
<td>machine learning</td>
<td>541</td>
<td>patent</td>
<td>449</td>
</tr>
<tr>
<td>patent</td>
<td>449</td>
<td>electromagnetics</td>
<td>412</td>
</tr>
<tr>
<td>electromagnetics</td>
<td>412</td>
<td>chemistry</td>
<td>382</td>
</tr>
<tr>
<td>chemistry</td>
<td>382</td>
<td>manufacturing</td>
<td>330</td>
</tr>
<tr>
<td>manufacturing</td>
<td>330</td>
<td>VLSI</td>
<td>294</td>
</tr>
<tr>
<td>VLSI</td>
<td>294</td>
<td>economics</td>
<td>259</td>
</tr>
<tr>
<td>economics</td>
<td>259</td>
<td>tutorial</td>
<td>237</td>
</tr>
<tr>
<td>tutorial</td>
<td>237</td>
<td>antennas</td>
<td>231</td>
</tr>
<tr>
<td>antennas</td>
<td>231</td>
<td>medical imaging</td>
<td>218</td>
</tr>
<tr>
<td>medical imaging</td>
<td>218</td>
<td>coding</td>
<td>199</td>
</tr>
<tr>
<td>coding</td>
<td>199</td>
<td>evolution</td>
<td>189</td>
</tr>
<tr>
<td>evolution</td>
<td>189</td>
<td>protein folding</td>
<td>178</td>
</tr>
<tr>
<td>protein folding</td>
<td>178</td>
<td>clustering</td>
<td>149</td>
</tr>
<tr>
<td>clustering</td>
<td>149</td>
<td>optimisation</td>
<td>141</td>
</tr>
<tr>
<td>optimisation</td>
<td>141</td>
<td>bibliography</td>
<td>128</td>
</tr>
<tr>
<td>bibliography</td>
<td>128</td>
<td>VLSI design</td>
<td>116</td>
</tr>
<tr>
<td>VLSI design</td>
<td>116</td>
<td>codevolution</td>
<td>112</td>
</tr>
<tr>
<td>codevolution</td>
<td>112</td>
<td>population size</td>
<td>109</td>
</tr>
<tr>
<td>population size</td>
<td>109</td>
<td>classifiers</td>
<td>107</td>
</tr>
<tr>
<td>classifiers</td>
<td>107</td>
<td>graphs</td>
<td>104</td>
</tr>
<tr>
<td>graphs</td>
<td>104</td>
<td>mutation</td>
<td>96</td>
</tr>
<tr>
<td>mutation</td>
<td>96</td>
<td>databases</td>
<td>93</td>
</tr>
<tr>
<td>databases</td>
<td>93</td>
<td>GARP</td>
<td>86</td>
</tr>
<tr>
<td>GARP</td>
<td>86</td>
<td>acoustics</td>
<td>84</td>
</tr>
<tr>
<td>acoustics</td>
<td>84</td>
<td>FEM</td>
<td>79</td>
</tr>
<tr>
<td>FEM</td>
<td>79</td>
<td>computer graphics</td>
<td>75</td>
</tr>
<tr>
<td>computer graphics</td>
<td>75</td>
<td>software testing</td>
<td>72</td>
</tr>
<tr>
<td>software testing</td>
<td>72</td>
<td>artificial intelligence</td>
<td>70</td>
</tr>
<tr>
<td>artificial intelligence</td>
<td>70</td>
<td>clusters</td>
<td>68</td>
</tr>
<tr>
<td>clusters</td>
<td>68</td>
<td>evolvable hardware</td>
<td>66</td>
</tr>
<tr>
<td>evolvable hardware</td>
<td>66</td>
<td>cancer</td>
<td>64</td>
</tr>
<tr>
<td>cancer</td>
<td>64</td>
<td>QSAR</td>
<td>59</td>
</tr>
<tr>
<td>QSAR</td>
<td>59</td>
<td>diversity</td>
<td>55</td>
</tr>
<tr>
<td>diversity</td>
<td>55</td>
<td>analysing GP</td>
<td>52</td>
</tr>
<tr>
<td>analysing GP</td>
<td>52</td>
<td>DNA</td>
<td>50</td>
</tr>
<tr>
<td>DNA</td>
<td>50</td>
<td>motors</td>
<td>47</td>
</tr>
<tr>
<td>motors</td>
<td>47</td>
<td>application</td>
<td>47</td>
</tr>
<tr>
<td>application</td>
<td>47</td>
<td>assembly</td>
<td>46</td>
</tr>
<tr>
<td>assembly</td>
<td>46</td>
<td>immune systems</td>
<td>45</td>
</tr>
<tr>
<td>immune systems</td>
<td>45</td>
<td>bin-packing</td>
<td>44</td>
</tr>
<tr>
<td>bin-packing</td>
<td>44</td>
<td>controller</td>
<td>43</td>
</tr>
<tr>
<td>controller</td>
<td>43</td>
<td>logistics</td>
<td>41</td>
</tr>
<tr>
<td>logistics</td>
<td>41</td>
<td>ant systems</td>
<td>38</td>
</tr>
<tr>
<td>ant systems</td>
<td>38</td>
<td>fitness landscape</td>
<td>37</td>
</tr>
<tr>
<td>fitness landscape</td>
<td>37</td>
<td>parameter estimation</td>
<td>36</td>
</tr>
<tr>
<td>parameter estimation</td>
<td>36</td>
<td>game theory</td>
<td>35</td>
</tr>
<tr>
<td>game theory</td>
<td>35</td>
<td>software</td>
<td>34</td>
</tr>
<tr>
<td>software</td>
<td>34</td>
<td>vehicle routing</td>
<td>33</td>
</tr>
<tr>
<td>vehicle routing</td>
<td>33</td>
<td>OBDD</td>
<td>33</td>
</tr>
</tbody>
</table>

Subject: engineering, neural networks, control, analysing GA, hybrid, implementation, signal processing, medicine, review, quantum computing, CAD, optics, telecommunications, electronics, TSP, crossover, layout design, shape design, remote sensing, agents, popular, aerodynamics, hardware, simulation, regression, ecology, quantum computer, fitness, games, machine vision, cellular automata, planning, astronomy, search, simulated annealing, tomography, path planning, laminates, transportation, stereo vision, music, inversion problems, art, chemometrics, fractals, statistics, rules.
10.2. FINNISH GA PROJECTS

Figure 10.1: A woodland brown (*Lopinga achine* Scopoli, 1763) resting on a leaf of *Betula* sp..

10.2 Finnish GA projects

Most GA research in Nordic countries has been done in Finland. The following list contains the most prominent areas and applications: (→ Vaasa GA bibliography: http://www.uva.fi/~TAU/reports/report94-1/gaNORDICbib.pdf)

- basics [Ala91a, Ala92e, Aut93, FK94, San94, San95, San96, Ala98]
- control /elevators: [ATY94, Mog96], PID: [Tör96, Tör97, AMT97]
- differential evolution [Lam99b]
- software testing [AMTV96, AMM97, AMMM98, Man99, Mog99, MA00]
- medical diagnosis [Lau97, LJ98, LJLV99]
- diesel engine cam shaft shape design [ANS95, AL96, Lam97, Lam99a]
- electric motors [Pal96]
- power engineering [Kau95, Man96]
- hardware implementation [AH95, HSOK95, HKS+96, Häm96]
- chemical reaction kinetics [AAMK94]
- chemical engineering [HH98]
CHAPTER 10. APPLICATIONS

- atomic clusters [AT00]
- filter design [MV94, Vuo94, HKA96, KHK96, Hut99]
- signal and image processing [AN95, ML95, Vuo95, Kau99] wavelets [AEH97]
- halftoning for ink jet printer [AMP98]
- scheduling in electronics manufacturing [Joh99]
- database queries [San93a, San92, San93b, San93c]
- neural networks [Nis94, Häm95a, Häm95b, HJKK96, HNK97, Fre97, Koh99]
- aerodynamics optimization [TML95]
- graph drawing [Elo96a, LMST96]
- vehicle routing [Brä99b, Brä99c, Brä99a, Brä01]
- economics [BSvW95, BLS96a, BLS96b]
- particle accelerators: undulator magnet sorting [Ryy94, Ryy96, RT96, AR97, Ryy98, Ryy99].

Next we will look more carefully on some specific applications, both Finnish and foreign.

10.3 Some application examples

10.4 Optics

10.4.1 Optical filter design

Reference: [ELL+93]

Keywords: automatic design, optics, interference filters, solar energy concentrators

Authors: T. Eisenhammer, M. Lazarov, M. Leutbecher, U. Schöffel, and R. Sizmann

Authors address: Physics Section, Ludwig-Maximilians-Universität, Amalienstr. 54, D-80799, Munich, Germany

Background

The work deals with automatic design of dielectric filters. The example is a heat mirror for thermal solar energy applications. Heat mirrors are high-pass filters with high transmittance for radiation at solar wavelengths and high reflectance for thermal radiation at longer wavelengths. Heat mirrors are used to reduce energy losses that are due to thermal infrared radiation from solar energy black body absorbers. One way to produce heat mirrors is to stack a thin metallic layer (silver, gold, or copper) between two high refractive index dielectric films. The goal of the work has been to show that it is possible to choose the optimal sequence and thicknesses of materials with widely different optical properties by using a GA.

The procedure is demonstrated for silver-based heat mirrors with a total of five metallic and dielectric layers (fig. 10.2).
Abstract

In the optimization of multilayer stacks for various optical filtering purposes not only the thicknesses of the films are to be optimized, but also the sequence of materials. Materials with very different optical properties, such as metals and dielectrics, may be combined. A GA is introduced to search for the optimal sequence of materials along with their optical thicknesses. This procedure is applied to a heat mirror in combination with a blackbody absorber for thermal solar energy applications at elevated temperatures (250 °C). The heat mirror is based on silver films with antireflective dielectric layers. Seven dielectrics have been considered. For a five-layer stack the sequence (TiO$_2$/Ag/TiO$_2$/Ag/Y$_2$O$_3$) is found to be optimal. [author’s abstract]

Results

GAs have been successfully used to design a dielectric solar heat mirror. The quality of the result is comparable to that of other design methods.

Details of GA used

- Population size: 150
- Generations: 300-500
- Mutation rate: 0.005 /bit
- Crossover: two-point
- Selection: elitist
- Function evaluations: 45,000-75,000
- Software: ?
- Program size: ?
- Processing time: ?
- Implementation time: ?
- Coding: binary
- Genome length: 55 bits
CHAPTER 10. APPLICATIONS

Discussion and comments

Alternative methods

Simulated annealing see [DK90, MK89].

Similar application and future

See next section.

Further references

The authors other papers on this subject [ELS92].

Other reference to papers using GAs in problems of optics:

diffractive elements: [Cal91a, Cal91b, MHR92]

[Yan93] IR: [AK92]

10.5 Telecommunications

10.5.1 Ring loading problem

Reference: [KC93]

Keywords: telecommunications, ring loading problem, optimization, SONET, optical networks

Authors: Nachimuthu Karunanithi and Tamra Carpenter

Authors address: Bellcore, 445 South Street, Morristown, NJ 07960, USA

Background

The work deals with sizing of SONET (Synchronous Optical Network) rings in a telecommunications network. SONET technology allows today’s telecommunications networks to accommodate extremely high data transmission rates. Ensuring survivability through fault-tolerant network design is becoming increasingly important. SONET rings are incorporated into telecommunications network architectures to provide immediate recovery from certain equipment failures. Rings consists of a set of nodes connected by high-capacity optical links to form a cycle that nowhere overlaps itself (fig. 10.3). In these rings every node pair is connected by two physically diverse paths. Thus, no single node or link failure will disconnect the nodes of a ring. All traffic, except that which terminates at a failed node, can be diverted around the failure as long as the links have enough capacity. The network planners goal is to install capacity so that it assures protection against single failures and at the same time does not make the ring too expensive. In practise this means determining the minimum link capacity that assures full single failure survivability.

The required capacity depends on the routing of the demands around the ring. The ring loading problem is now: how do we route (clockwise or anti-clockwise) the demands between nodes so that the maximum load on any link is as small as possible? This ring loading problem is NP-complete [CS92]. The binary nature of the problem provides an immediate encoding for GA.
Abstract

GA solutions to the ring loading problem are compared with optimal solutions obtained by the CPLEX mixed integer program solver and heuristic solutions that are generated by the tools in the SONET Toolkit system and the authors’ own implementation of Cosares and Saniee [CS92] two-phase greedy heuristic. 10 and 25 node rings with four different types of demands (uniform random or bimodal) are used in the comparisons.

Results

The GA finds optimal solution to at least once for each 10 node problems. The standard deviation indicates that there is little variability in the quality of the GA solution. For several problems the genetic algorithm obtains an optimal solution in every trial and in many instances the worst solution obtained by GA is at least as good as the two-pass solution.

For some of the 25 node problems, CPLEX reaches its limit before it discovers an optimal solution. The best GA solutions often are the best solutions obtained. The main drawback in using standard optimization software is the long execution time needed to examining many thousands of branch and bound nodes. Perhaps the main result of this work is that at least in this problem GA is robust not only in the quality of it’s solutions, but also in the time it takes to obtain them.
Details of GA used

Population size ?
Generations ?
Mutation rate 0.15
Crossover single point / reduced surrogate crossover
Selection rank-based
Function evaluations ?
Repetitions 50
Software GENITOR [Whi89]
Program size ?
Processing time ?
Implementation time ?
Coding binary (1 = clockwise, 0 = anti-clockwise)
Genome length ? bits
Computer Sun4m workstation

Discussion and comments

Applying a GA to this problem is simple to do and yields good results. The GA does not require much more time than the two-pass algorithm, which is currently used in the SONET Toolkit.

Alternative methods

Branch and bound see [CS92].

Similar application and future

See next section.

Further references

The authors other papers on GA [KDW92].
Other reference to papers using GAs in problems of telecommunications:

10.6 Layout design

10.6.1 Cable harness routing design

Reference: [?]  
Keywords: automatic design, cable harness, automotive engineering, layout design  
Author: Andrew B. Conru  
Authors address: Center for Design Research, Stanford University, Stanford, CA 94305, USA e-mail: conru@sunrise.stanford.edu
Background
The work deals with automatic design of automotive cable harnesses. Surprisingly the cable harness is one of the most expensive parts of ordinary personal cars. While the manufacturing of other parts of cars have been quite successfully been automatized, wire harnesses are mostly manually combined and assembled. Hence the economical pressure to design, optimize, and assemble cable harnesses is high.

The cable harness is composed of the cables used to conduct electric current between different devices of automobile. The cables usually ended by connectors and the cables are held together by ties. Usually the bundle is shielded by plastic tubes. The structure of the harness can be modelled shematically as a tree (see fig. 10.4).

Figure empty; see the original reference.

Figure 10.4: A shematic diagram of two different wireharness configurations satisfying the same pin-to-pin wiring specifications.

Abstract
The paper describes a system for automatically routing cable harnesses in three-dimensional environments using a pair of genetic algorithms. The cable harness routing problem can be formulated as a graph search problem with a large, convex search space. A genetic approach is used to intelligently and adaptively search for routings which are close to the global optimum. The problem is decomposed into two problems: generating a harness configuration (topology) and routing the harness in the environment. The paper defines the genetic operators used and suggest parameter settings which quickly find routings which match the geometry of the environment into which the harness is to be assembled. [author’s abstract]

Results
GAs have been successfully used to design a relatively simple (automotive) cable harnesses. The quality of the result is comparable to or better than that of other design methods.

Details of GA used
- Population size: 350-500
- Generations: 15-50
- Mutation rate: 0.03-0.06 /bit
- Crossover: one-point
- Selection: elitist 0.15
- Function evaluations: ?
- Software: in C++
- Program size: ?
- Processing time: ?
- Implementation time: ?
- Coding: list structure
- Genome length: varying
Discussion and comments

Alternative methods

The proposed GA method was compared with simple random search and the minimum cost path heuristic proposed by Takahashi and Matsuyama \( ? \). GA was consistently better than the minimum cost path heuristic while being also several orders of magnitude slower, however. A hybrid of these two methods might be both faster and give better results than either one.

The random search performed well in simple wire harnesses but quickly fell behind GA as the number of transitions increased.

Similar application and future

See next section.

Further references

Other reference to papers using GAs in cable harness problems:

\[ \text{\cite{ZSE94}} \]

Other related articles by the author \( ? \).

10.6.2 Truss

In structural engineering one basic object of design is various truss structures used in buildings, bridges, etc. In this section we will look this type of applications from the evolutionary optimisation view point.

WORKING ON ...

*  

Truss calculations A typical simple truss structure is given in figure 10.5. The forces in a truss can be solved by a linear equation. Therefore no global optimisation method is needed in the basic calculation when the structure is fixed. However, the situation changes when there are free parameters like the locations of the nodes or the dimensions of the links.

(find this from elsewhere)

Table 10.2: Greenlee’s truss

(do you need this?)

Table 10.3: Postiporstua truss
Figure 10.5: Greenlee’s truss

Figure 10.6: Postiporstua truss
Chapter 11

Discussion

Using GA we have once “succeeded” in finetuning the parameters of GA so successfully that
the resulting algorithm was unbeatably fast to solve the given fixed problem [?]. The bug
was that the set of parameters included also the random number generator seed. The result
was that the genetic algorithm found the random number seed that immediately lead to the
solution. As warning as this real life example is, it also demonstrates the power of genetic
algorithm based optimisation.

11.1 Comparing GA and QC

There seems to be some indication that the performance of QC and GAs have some similarity.
In table 11.1 is given a brief list of comparable properties of quantum computing versus genetic
algorithms. For more information on genetic algorithms see e.g. [Ala98] or references in [Alac].

<table>
<thead>
<tr>
<th>property</th>
<th>QC</th>
<th>GA</th>
</tr>
</thead>
<tbody>
<tr>
<td>example in</td>
<td>quantum mechanics</td>
<td>evolution</td>
</tr>
<tr>
<td>nature</td>
<td>heuristics</td>
<td>heuristics</td>
</tr>
<tr>
<td>functioning</td>
<td>stochastic</td>
<td>stochastic</td>
</tr>
<tr>
<td>building block</td>
<td>qubit</td>
<td>gene / bit</td>
</tr>
<tr>
<td>theory</td>
<td>difficult</td>
<td>very difficult</td>
</tr>
<tr>
<td>scanning method</td>
<td>entanglement</td>
<td>crossover</td>
</tr>
<tr>
<td>structural parameters</td>
<td>several</td>
<td>several</td>
</tr>
<tr>
<td>dynamics</td>
<td>iterations</td>
<td>generations</td>
</tr>
<tr>
<td>applicability</td>
<td>unknown</td>
<td>high</td>
</tr>
<tr>
<td>speed</td>
<td>extremely fast</td>
<td>slow</td>
</tr>
<tr>
<td>implementation</td>
<td>extremely difficult</td>
<td>easy</td>
</tr>
<tr>
<td>generality</td>
<td>unknown</td>
<td>wide</td>
</tr>
</tbody>
</table>

Table 11.1: Comparison of some properties of quantum computing (QC) and genetic algo-
rithms (GA).

Quantum computing has also inspired to develop new genetic algorithms that have some
quantum computing flavour [NM96, HK00, HPLK01].
11.1.1 Automatic design of quantum circuits

As at least the students of this course have perhaps noticed the design of quantum circuits is a non-trivial mental activity. This section describes a genetic algorithm based method to automatically design quantum circuits [WG99]. Other similar work is described in [GWC98, SBBS99, YI00, Rub01].

In the work by Colin P. Williams and Alexander G. Gray genetic programming has been used to design quantum circuits, more specifically quantum teleportation circuits. The resulting circuits have been simple, even simpler than those previously designed manually [WG99].

In order to be able to utilise evolutionary computing we must define the target or fitness function. Williams and Gray defined their fitness function as follows:

$$f(S,U) = \sum_{i=1}^{2^N} \sum_{j=1}^{2^N} |U_{ij} - S_{ij}|, \quad S,U \in U(2^N),$$

where $U$ is the unitary matrix of the target circuit and $S$ is the unitary matrix of the proposed circuit generated by GP. When $f = 0$, $S$ implements exactly the given target circuit.

A predefined set of elementary quantum gates are given to the GP, which is also able to simulate the functioning of the circuits consisting of the given gates and their interconnections by evaluating the resulting unitary matrix $S$.

The GP method was able to automatically create both send and receive parts of the teleportation circuit. The send circuit consisted of two XOR and L and R gates and was comparable to the known minimum send circuits designed manually. The solution was found after 26.4 generations on the average, when using population size 100. Both figures are really modest for GP, from which we can safely deduce that designing a teleportation circuit is very easy for GP.

The minimum receive circuit consisted also of four gates, which was two gates less than in the best human designed teleportation receiver gate so far.

11.1.2 Conclusions and future

From the relative easiness of automatic design of basic quantum circuits and the fact that GAs has been widely used in VLSI design [Alae], we can deduce that GA has high application potential also in design of more elaborate quantum circuits. Personally the author is interested in applying GAs to more realistic and challenging quantum circuit design problems in collaboration with those more familiar with the technical details of quantum computing. One specific topic could be a comparison study of QC and GA performance on different problem types representing different fitness landscapes types [HS98].
Chapter 12

Conclusions and future

GAs are one heuristic method group among other heuristic methods and it is at our disposal when we do not have more deterministics methods. From the trend of GA literature volume we can conclude that we will see much more GA work in the near future. There does not seem to be any really negative results that would turn researchers into other directions.

12.1 Research

From the few theoretical contributions and the simple structure of GA we might conclude that we will see some more theoretical attempts to attack GAs and their properties. This research might also be interesting to theoretical biologists and population geneticists.

12.2 Implementations

Some program package already have GA subpackages, but there might be more coming soon.

12.3 Applications

The success of GA depends on the success of GA applications. This trend is continuing at least for a few years to come.

12.4 Distribution

GA and NN are really fit for massive parallel processing, which is certainly the trend of future high performance computing.
Appendix A

Problems

A.1 Small problems

1. **Nature as teacher** Why and what kind of information processing methods can be found in nature?

2. **Genetics** Explain shortly the following concepts: a) haploid, b) diploid, c) deme, d) polymorphism, and e) recessive allele.

3. **Ichneumon** How the *Ichneumon sp.* wasp knows where to drill on the logg?

4. **Applications** Explain shortly the most important application areas of genetic algorithms in automation.

5. **Applications** Explain shortly one GA application in electronics.

6. **Applications** Explain shortly one GA application in computer science.

7. **Applications** Explain shortly one GA application in telecommunications.

8. **Power engineering** Explain shortly the most important applications areas of genetic algorithms in power engineering.

9. **Crossover** Explain shortly what is crossover and give an implementation of it e.g. in Java.

10. **Phase transition** Explain shortly what is phase transition and how it influences the complexity of a given problems.

11. **Sudoku** How would you solve Sudokus with GAs? What could be a good fitness function?

12. **Diversity** Explain shortly methods that can be used to control diversity in genetic algorithms.

13. **Diversity** Explain shortly how the plankton paradox can be explained.

14. **Permutation type problems** How do you solve permutation type problems with genetic algorithms i.e. what should be especially be considered in implementation.

15. **Pareto type problems** What is Pareto optimisation? How do you solve Pareto problems with genetic algorithms i.e. what should be especially be considered in implementation.

16. **Polynomial fit** How would you solve the following problem with a GA: Fit a polynomial of degree $n + 1$ thru given $n$ points, so that the polynomial should deviate as little as possible from the line between the points.

17. **Hull** How would you solve with a genetic algorithm the following problem: A give $5 \times 5$ unit square should be covered totally by as few unit circles as possible.

18. **Hull** How would you solve the following problem by GA: A given rectangle area should be covered as completely as possible by given $n$ smaller rectangles.

19. **Variable selection** What is the so called variable selection problem and how evolutionary computation methods can be used to solve it? Please, give an example of variable selection problem.

20. **Population size** How the size of the population influences genetic algorithm search. What kind of advice would you give for assigning a proper population size.
21. **Interval arithmetics** Let 

\[ F = \frac{(A + B)}{(C - D)} + E, \]

when \( A = [-12, -10], \ B = [0, 1], \ C = [10, 20], \ D = [-3, 3] \) and \( E = [-1, 1] \). What is the range of \( F \) given above at the above given variable intervals as evaluated by interval arithmetics?

22. **Fuzzy controller** Explain how GAs can be used for tuning of a fuzzy controller.

23. **Neural network learning** How neural networks learn? What should be remembered when training neural networks? How GAs can be used in teaching neural networks?

24. **CT/GA** How genetic algorithms can be applied in radio therapy?

25. **Genetic programming** Explain shortly what is genetic programming and what kind of problems can be solved using it.

### A.2 Small projects

1. **Random number generator** Let us have the generator \( X_{i+1} = (aX_i + b) \text{mod} 100 \). Study empirically i.e. by calculating the goodness of the generator when \( a = 17711 \) and \( b = 23171 \): How even is the distribution and how long is the sequence?

2. **Random number generator** Let us have the generator \( X_{i+1} = (aX_i + b) \text{mod} 100 \). Try to find such constants \( a \) and \( b \) that gives the best generator i.e. the longest and most even distribution.
Bibliography


BIBLIOGRAPHY


[Ala93c] Jarmo T. Alander. On robot navigation using a genetic algorithm. In Albrecht et al. [ARS93], pages 471–478. also as [Ala93b].


[Rec94] Ingo Rechenberg. Evolutionstrategie ’94. Frommann-Holzboog-Verlag, Stuttgart (Germany), 1994. (in German; includes also [Rec73]).


[San93a] Sam Sandqvist. GA and database queries. In Alander [Ala93d].


**Notation** used in bibliography:

\[ \leftrightarrow 123 = \text{this contribution was cited on page 123.} \]
Appendix B

Glossary

A short glossary of terms related to genetic algorithms and natural genetics and biology.

B.1 Glossary

A short glossary of genetic terms [?]:

<table>
<thead>
<tr>
<th>term</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>allele</td>
<td>an alternative state of a gene</td>
</tr>
<tr>
<td>autosome</td>
<td>chromosomes in both sexes</td>
</tr>
<tr>
<td>B chromosomes</td>
<td>accessory chromosomes; not vital</td>
</tr>
<tr>
<td>chromatide</td>
<td>a pair of chromatides = chromosome</td>
</tr>
<tr>
<td>chromosome</td>
<td>a linear gene containing body</td>
</tr>
<tr>
<td>clone</td>
<td>descendants of a single cell</td>
</tr>
<tr>
<td>codon</td>
<td>nucleotide triplet = genetic code unit</td>
</tr>
<tr>
<td>crossover</td>
<td>gene exchange</td>
</tr>
<tr>
<td>diploid</td>
<td>two sets of chromosomes (2n) (eukaryote organisms)</td>
</tr>
<tr>
<td>Drosophila melanogaster</td>
<td>the common fruit fly</td>
</tr>
<tr>
<td>eukaryote</td>
<td>higher organism (excl. viruses, bacteria, and bluegreen algae)</td>
</tr>
<tr>
<td>gametes</td>
<td>a sex cell, egg, or sperm</td>
</tr>
<tr>
<td>gene</td>
<td>a unit of inheritance</td>
</tr>
<tr>
<td>genetic code</td>
<td>the 64 possible triplets of mRNA</td>
</tr>
<tr>
<td>genome</td>
<td>genes inherited</td>
</tr>
<tr>
<td>haplodiploidy</td>
<td>fertilized eggs —females; unfert. eggs —males</td>
</tr>
<tr>
<td>haploid</td>
<td>one set of genes (usually in gametes)</td>
</tr>
<tr>
<td>hermaphrodite</td>
<td>male and female reproductive structures</td>
</tr>
<tr>
<td>heterozygote</td>
<td>having two different alleles</td>
</tr>
<tr>
<td>homeostasis</td>
<td>compensation of environmental changes</td>
</tr>
<tr>
<td>homeotic gene</td>
<td></td>
</tr>
<tr>
<td>homeobox</td>
<td></td>
</tr>
<tr>
<td>maternal inheritance</td>
<td>non-mendelian inheritance through the egg</td>
</tr>
<tr>
<td>mutation</td>
<td>gene alteration; possibly a new allelic state</td>
</tr>
<tr>
<td>polyploidy</td>
<td>more than two sets of chromosomes</td>
</tr>
<tr>
<td>telomere</td>
<td>DNA cap at each end of the chromosome</td>
</tr>
<tr>
<td>zygote</td>
<td>a fertilized egg</td>
</tr>
</tbody>
</table>
Appendix C

Index
Keywords found in text: take when compiled..................