Evolutionary Algorithms and Optimization

D I S S E R T A T I O N

zur Erlangung des akademischen Grades
doctor rerum naturalium
(dr. rer. nat.)
im Fach Physik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät I
Humboldt-Universität zu Berlin

von
Herr Dipl.-Phys. Axel Reimann
geboren am 28.05.1973 in Hennigsdorf

Präsident der Humboldt-Universität zu Berlin:
Prof. Dr. Jürgen Mlynek

Dekan der Mathematisch-Naturwissenschaftlichen Fakultät I:
Prof. Dr. Michael Linscheid

Gutachter:

1. Prof. Dr. Werner Ebeling
2. Prof. Dr. Heinz Mühlenbein
3. PD Dr. Dr. Frank Schweitzer

eingereicht am: 20. August 2001
Tag der mündlichen Prüfung: 5. Dezember 2002

Zusammenfassung

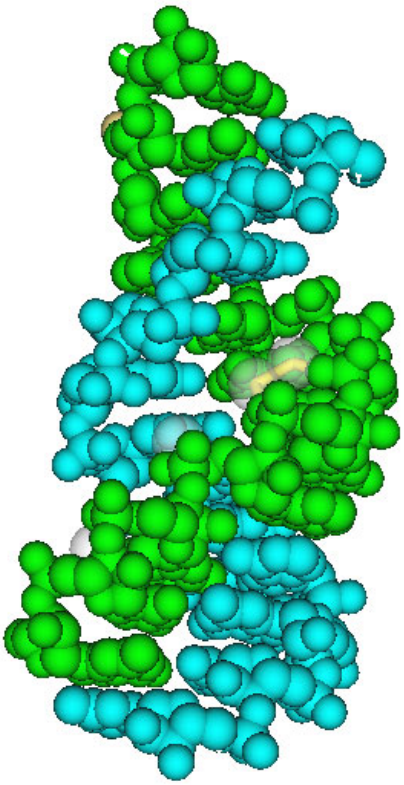
Diese Arbeit beschäftigt sich mit dem Thema *Evolutionäre Algorithmen* und deren Verwendung für Optimierungsaufgaben. Im ersten Teil der Arbeit werden die theoretischen Grundlagen ausführlich dargelegt, die zum Verständnis der Problemstellung und der vorgeschlagenen Lösungsmöglichkeiten notwendig sind. Dazu gehören die Einführung des Konzeptes von Fitneßlandschaften, deren Eigenschaften sowie die kurze Darstellung bekannter stochastischer Optimierungsverfahren wie z.B. Simulated Annealing. Im Anschluß daran wird auf neue Verfahren – insbesondere gemischte Strategien – eingegangen und diese vergleichend gegenüber den herkömmlichen Verfahren abgegrenzt.

Die neu entwickelten Verfahren werden an Modellproblemen getestet, welche im zweiten Teil der Arbeit vorgestellt werden. Verwendet wurden sowohl einfache theoretische Modelle wie *Frustrierte Periodische Sequenzen* als auch praktisch relevante Probleme wie das der RNA Sekundärstrukturen. Die verschiedenen Modellprobleme werden bezüglich ihrer Eigenschaften und Schwierigkeitsgrade untersucht und miteinander verglichen, um die Effizienz der verwendeten Optimierungsverfahren abschätzen zu können.

Der dritte Teil der Arbeit präsentiert wichtige Ergebnisse der im Rahmen dieser Arbeit durchgeführten umfangreichen numerischen Simulationen. Es wird demonstriert, wie sensitiv die Optimierungsergebnisse von den verwendeten Parametern der Algorithmen (wie z.B. Ensemblegröße, Temperatur oder Mutationsrate) abhängen und das ein relativ scharf umrissenes evolutionäres Fenster der Parameter existiert, innerhalb dessen die Optimierungsresultate deutlich besser sind. Eine im Rahmen dieser Arbeit entwickelte adaptive Parametersteuerung wird an den im zweiten Teil vorgestellten Modellproblemen getestet und gezeigt, daß es möglich ist, den Optimierungsprozeß automatisch innerhalb des evolutionären Fensters zu halten.

Der letzte Teil gibt Einblick in die im Rahmen dieser Arbeit verwendete Computer-Software und das vom Autor entwickelte Programmpaket. Es wird hervorgehoben, daß die in `C++` objektorientiert und modular geschriebene Software leicht an andere Optimierungsaufgaben angepaßt werden kann und dank graphischer Benutzeroberfläche auch einfach zu bedienen ist.

EVOLUTIONARY ALGORITHMS AND OPTIMIZATION



Axel Reimann

Author: Axel Reimann, 2001

Cover : Ribonucleic Acid, Structure of loop E from E. Coli 5s Rrna
ORGANISM SCIENTIFIC: Escheria coli
C. C. Correll, B. Freeborn, P. B. Moore and T. A. Steitz
30th Sep. 1997, PDB Code: 354D
visualized using Cn3D

Dedicated to my parents and friends.

Contents

1 Introduction **9**

2 Learning from Nature **11**

2.1 The Theoretical Framework 11

2.1.1 The Concept of Fitness Landscapes 12

2.1.2 Properties of Fitness Landscapes 14

The Density of States 15

The Autocorrelation Function 17

2.1.3 Stochastic Modeling of Basic Evolutionary Strategies . . 20

The Darwin Strategy 21

The Boltzmann Strategy 23

The Mixed Boltzmann-Darwin Strategy 25

2.1.4 Other Stochastic Optimization Strategies 27

Simulated Annealing 27

Genetic Algorithms 28

3 Model Problems **31**

3.1 Correlated Random Landscapes 31

3.2 Frustrated Periodic Sequences 34

3.3 The LABS Problem 35

3.4 The RNA and NK Model Compared 36

3.4.1 The NK Model 36
3.4.2 RNA Secondary Structures 38

4 Optimizing the Search Process **43**
4.1 Exact Stochastic Simulations 43
4.1.1 The Direct Method 44
4.1.2 The First Reaction Method 45
4.1.3 The Next Reaction Method 45
4.2 The Evolutionary Window 46
4.2.1 Comparing Fitness Landscapes 46
4.2.2 Exploring Parameter Windows 50
1. Constant Temperature 52
2. Variable Temperature 54
4.3 Mastering Intrinsic Search Parameters 58
4.3.1 Ensemble Size Adaptation 58
4.3.2 Temperature Adaptation 59
4.3.3 Mutation Rate Adaptation 61
First Approach: The Ensemble Variability 64
Second Approach: The Relative Ensemble Dispersion . 66
Third Approach: The Ensemble Entropy 68
4.4 An Adaptive Evolutionary Algorithm 73

5 Software **77**
5.1 Newly Developed Software 77
5.1.1 Optimization Programs 77
The User Interface 78
The Workflow 78
5.1.2 The SimRNA Mutation Operator 80

5.1.3 The SimRNA Source Code 82
The RNA-Strand Class 82
The Main Loop 108
5.1.4 MPI_generate 116
5.2 Open Source Software 129
5.2.1 The Message Passing Interface MPI 129
5.2.2 The Vienna RNA Package 129
5.2.3 Free Visualization Software 129
5.2.4 Free External Libraries 131

A Polio Virus Type 1 Subsequence **133**

B Glossary **137**

C Acknowledgment **165**

List of Figures

2.1 Simple Fitness Landscape 12
2.2 Complex Fitness Landscape 13
2.3 Density of States 16
2.4 Landscapes with Different Correlation Length 18
2.5 Autocorrelation 19
2.6 Discrete Representation of a Fitness Landscape 22
2.7 Mixed Strategy 26
2.8 Crossover Operator 29

3.1 Sequence Evaluation Scheme 34
3.2 Purines 38
3.3 Pyrimidines 39
3.4 Primary Structure 39
3.5 Secondary Structure 40
3.6 Pseudo Loop 41

4.1 Autocorrelation Functions for Different Problems 49
4.2 Small Search Space 50
4.3 Vast Search Space 51
4.4 Parameter Sweep on Testmodels 53
4.5 Evolutionary Window (Frustrated Periodic Sequences) 55
4.6 Evolutionary Window (Low Autocorrelation Binary Strings) . . 56

4.7 Evolutionary Window (RNA Secondary Structure) 57
4.8 Error Threshold (Scheme) 62
4.9 Ensemble Variability 65
4.10 Relative Ensemble Dispersion 66
4.11 LABS Problem: Comparison of Different Sensors 67
4.12 FPS Problem: Ensemble Entropy 70
4.13 FPS Problem: Entropy Sensor 71
4.14 RNA Problem: Entropy Sensor 71
4.15 Ensemble Histograms 72
4.16 Adaptation Results for RNA Problem 75

5.1 SimRNA User Interface 79
5.2 Block Diagram: Optimization Programs 80
5.3 Implementation Scheme of the SimRNA Mutation Operator . . 81

A.1 Best Folding: $L = 100, ACV01148, 5' - cloverleaf$ 134
A.2 Suboptimal Foldings: $L = 100, ACV01148, 5' - cloverleaf$ . . . 135

Symbols

A_{ij}	mutation matrix A
β	inverse temperature
d_{rel}	relative ensemble dispersion
E	energy
F	fitness
γ	selection probability
H	Hamiltonian
$\hat{H}_{ens}$	ensemble entropy
k	Boltzmann constant
L	problem size; sequence length
m	tournament size
N	number of observations, ensemble size
$n(F)$	density of states
$\mathcal{O}$	order symbol
p	probability density
P	probability
r	correlation length
R_k	lag k autocorrelation coefficient
$R(t)$	evolution rate

S	entropy
σ	standard deviation
σ_a^2	autocovariance
σ^2	variance
t	time
T	temperature
v	ensemble variability
U	potential energy
V	potential
W	statistical weight
$\bar{x}$	mean value of x
x_i	occupation number of state i
Z	partition function

Chapter 1

Introduction

This work is a theoretical approach to a practical problem: optimization. Everyday life is full of tasks related to optimization. Wherever resources, like energy, space, food supply etc., are limited, the question of efficiency and, thus, the need for optimization arises.

In biology this issue becomes literally a matter of life or death. Any living being not optimally adapted to its surroundings will most likely vanish over time due to natural selection [1, 2]. The adaptation problem becomes even more intricate considering that environmental parameters are not static, but instead change over time. Since short term changes might also happen within the lifespan of an individual it is obvious that adaptation or optimization is an ongoing process that in itself needs to be efficient with respect to time and energy consumption.

In the paragraph above adaptation and optimization could essentially be used interchangeably, underlining the close relationship between the two processes. Adaptation can be perceived as the optimization of one or more items under several given constraints. In engineering it is an often encountered problem that the optimization of one crucial parameter directly or indirectly influences other parameters in a sometimes unpredictable way. Optimization here means finding compromises to reach contradictory goals, e.g. gas mileage versus engine power or stability of a construction versus its weight. The situation can easily

get out of hand when the number of parameters and constraints surpasses a certain threshold. Even though engineers have learned by experience to circumvent or tackle many well behaved problems, some others can no longer be successfully approached with conventional methods. What can one learn from nature? It seems that biology has come up with some exceedingly well-working remedies to solve dynamic multi-parameter optimization problems that could hardly be solved analytically in any given reasonable time span. In order to take advantage of evolutionary strategies however, one has to understand first of all how they work and why they perform as well as they indeed do. Secondly, those strategies need to be modeled mathematically to be of any benefit in engineering. Last but not least, any given algorithm needs to be tuned with regards to its efficiency.

This work describes theoretical models for different 'standard' evolutionary algorithms known as e.g. *Metropolis Algorithm* [3], *Simulated Annealing* [4, 5, 6, 7, 8, 9] or *Boltzmann strategy* [10, 11] and *Evolutionary Algorithms* [12, 13, 14]. It furthermore investigates the power of mixed strategies combining ideas from both physics and biology, like the *Boltzmann-Darwin Strategy*. The investigated algorithms will be applied to different test problems in computer simulations, and their respective results will be analyzed with respect to time consumption, result quality and search parameter dependence. The test problems include optimization of artificial strings (Frustrated Periodic Sequences and Low Autocorrelation Binary Strings [*LABS*]), as well as RNA folding problems (RNA secondary structure). It will be shown that the chosen optimization parameters crucially influence the optimization result. For all investigated problems only a small *evolutionary window* of parameters leads to an efficient search process. The introduction of a new nonlinear numerical sensor allows to improve the investigated algorithms by automatically adapting their intrinsic parameters to the evolutionary window.

Chapter 2

Learning from Nature

2.1 The Theoretical Framework

Conventional problem-solving strategies follow a strict algorithm. It is the deterministic nature of these algorithms that embodies both the advantages and disadvantages. A classical deterministic algorithm, by definition, solves a given problem in a finite number of steps. Many problems are, however, *NP* or *NP complete* problems,[1] and the necessary computation time t to solve the problem, for example often grows exponentially with the problem size L, that is, the problem is said to be of order $\mathcal{O}\big(\exp[L]\big)$.

If a problem is not exactly deterministically solvable in polynomial time, it might however still be possible to *approximate* it in polynomial time. An elegant way to circumvent deterministic limitations is to introduce stochastic elements to problem solving methods. Evolutionary algorithms, inspired by physics and biology, do just that. It takes some insight to understand how exactly stochastic can help to solve problems.

[1] for an exhaustive reference cf. **'A compendium of NP optimization problems'** at:
http://www.nada.kth.se/~viggo/problemlist/compendium.html

Natural evolution is undoubtedly driven by at least two dominating forces: *mutation* and *selection*. The following paragraphs investigate how these processes can be modeled mathematically and how randomness helps by coming into play.

2.1.1 The Concept of Fitness Landscapes

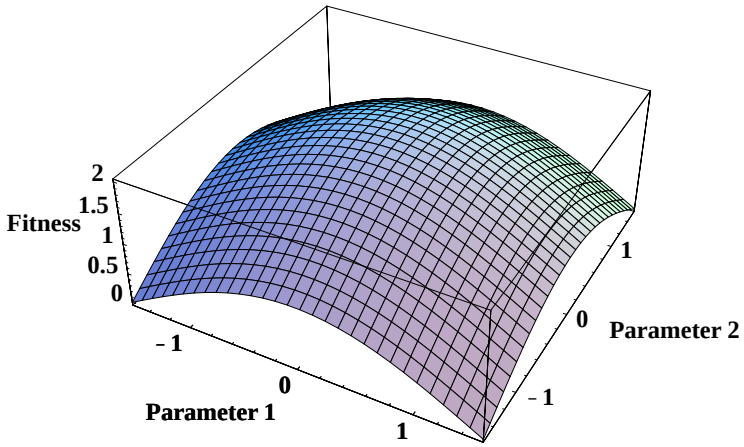


Figure 2.1: Simple imaginary two-dimensional fitness landscape (continuous)

A simple engineering problem might depend, for example, on n parameters x_n. By assigning these parameters to the axes of a simple diagram, one can plot all solutions to the problem for all given parameter combinations for low dimensional problems. The n-dimensional space spanned by the n parameters is simply called parameter space. Figure 2.1 shows a three dimensional plot for an imaginary two-dimensional problem. The single peaked plane stretching into the z-direction represents the set of solutions to the respective parameter combinations: $\{(x_1, x_2)\} \rightarrow \{F(x_1, x_2)\}$. The different solutions have a different fitness with respect to the posed problem; hence, it is legitimate to also speak of a fitness landscape.

The problem with finding an optimal parameter combination or equivalently with finding the best fitness values can now easily be illustrated as the search for

the top of the hill in Figure 2.1. If the underlying analytic relation were known, then it would be possible to use the rich toolbox of classical algorithms implementing well-known analytical solution techniques. If, on the other hand, analytic solutions are impossible to find, and the number of parameters (parameter combinations) runs out of bounds, simple trial and error methods will, likewise, no longer suffice.

A simple alternative approach to find the maximum (or optimum respectively) is known as the *method of steepest descent*, the *gradient strategy*, or more descriptively, *hill climbing*. Starting somewhere in the parameter space, one follows the inclination (gradient) by varying the parameters until the optimum is found. This method works well for simple fitness landscapes such as the one seen in Figure 2.1.

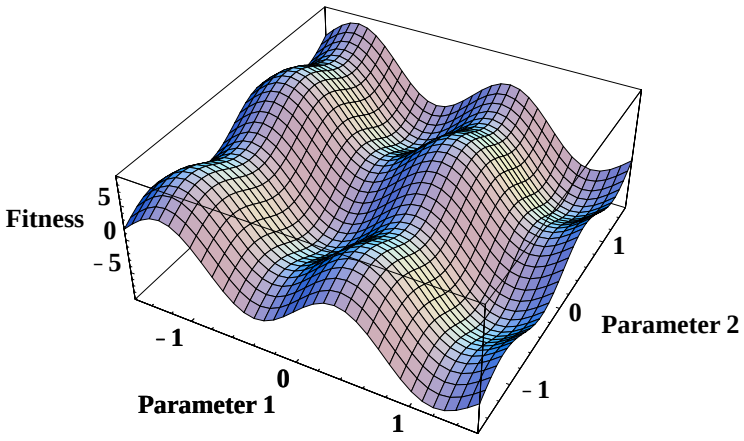


Figure 2.2: More complex imaginary fitness landscape (continuous) with several local minima and maxima

As soon as the underlying dynamics becomes more complex and the fitness landscape becomes more rugged, this method is probably doomed to fail. The search process will ultimately end in a local optimum, which is not necessarily the global optimum. Figure 2.2 illustrates such a fitness landscape. A way of working around this would be to simultaneously start several search processes beginning with different starting points in parameter space. The search process

can be imagined as being carried out by an uncoupled seeker ensemble. Another ansatz is to also allow downhill movements under certain circumstances. While dead ends in the search process can be circumvented this way, the search speed is degraded. In order to efficiently search for the global optimum it might become necessary to drop inefficient seekers or adjust the probability of downhill movements. A number of different search strategies have been developed with these ideas in mind. A few of them will be introduced in section 2.1.3.

It is important to know that even though the fitness landscape completely determines the structure of the optimization problem, it is *not* true that, in reverse, the optimization problem uniquely defines the fitness landscape [15]. Scanning along the fitness plane, one successively encounters the fitness values for neighboring parameter settings. There is no immediate information, however, about how the neighborhood is defined in parameter space. In other words, it is the set of allowed steps in parameter space that defines the respective neighborhood structure and, in turn, generates a fitness landscape as just one of many possible representations of the problem.

Therefore, choosing a proper set of allowed steps in parameter space can influence the solvability of an optimization problem in the same way that choosing a proper coordinate system influences the solvability of any problem in physics.

2.1.2 Properties of Fitness Landscapes

The fitness landscapes illustrated so far have been continuous. In order to be numerically tractable, however, fitness landscapes that are not inherently discrete need to be suitably sampled (Figure 2.6 shows an example of a discrete fitness landscape representation). Keeping this in mind, the following paragraphs do not explicitly distinguish between continuous and discrete fitness landscapes.

As can be derived from Figure 2.1 and Figure 2.2 already, fitness landscapes can have very different shapes. The typical features of fitness landscapes (ruggedness, number of peaks etc.) represent the inherent difficulty of the corresponding optimization problem. Efficient search algorithms, therefore, need to have an idea regarding the kind of landscape upon which they are working. While for smooth landscapes gradient-based optimization methods with only a few seekers perform best, they are almost useless in rugged landscapes. Because the complete fitness landscape is usually unknown[1], some sort of numerical measure describing the landscape is necessary to guide an optimization algorithm. Two candidates, the *density of states* and the *autocorrelation function*, will be introduced here.

The Density of States

The density of states $n(E)$ is an important tool in physics to characterize thermodynamical systems. It describes how often a certain energy value E is realized in a size N system, meaning how likely it is to encounter a particular energy realization in this system.

It is easy to adopt this idea for optimization purposes, as it is straightforward to consider fitness values F instead of energy levels. The definition of the density of states describing the frequency of particular fitness values in the entire fitness landscape then becomes [15]:

$$n(F) = \frac{\mathrm{d}N}{\mathrm{d}F}. \tag{2.1}$$

The probability to find a certain fitness realization therefore is:

$$P(F) = n(F)p(x(F)) \tag{2.2}$$

[1] Otherwise, the optimization problem were solved already.

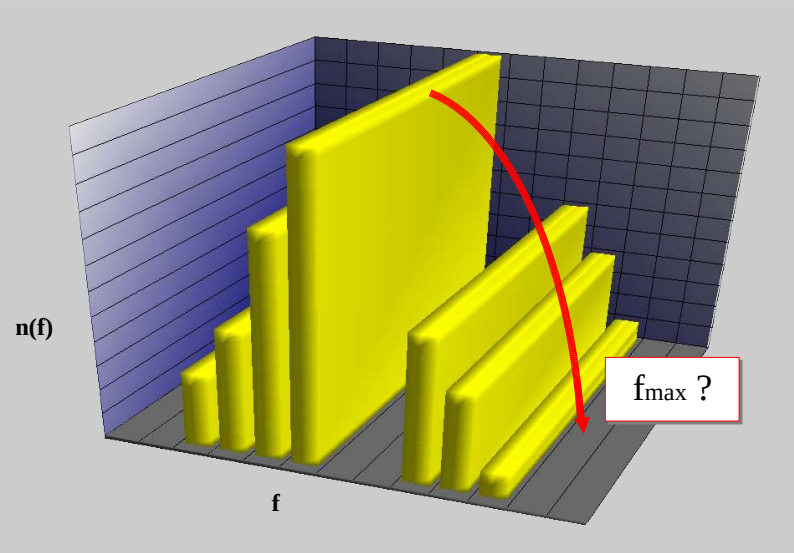


Figure 2.3: Partial knowledge of the density of states may help guessing the quality of the optimal solution and approximating the necessary effort required by means of extrapolation.

where $p(x)$ is the conditional probability density function. The probability density $p(F)$ is, of course, normalized and simply states that it is certain that the system is in only one particular state at any given moment:

$$\int_{-\infty}^{\infty} p(x_1 \ldots x_n)\, \mathrm{d}x_1 \ldots \mathrm{d}x_n \stackrel{!}{=} 1. \tag{2.3}$$

Since the complete fitness landscape has to be considered unknown, $n(F)$ (or $P(F)$ respectively) is also an unknown function. It is possible, however, to construct a picture of the density of states in steps *while* the optimization is in progress.[1] This procedure reflects the growing knowledge of the optimization problem and can, thus, also be expressed by using a measure taken from infor-

[1] This can easily be done by generating a histogram with respect to found fitness values and normalizing the outcome according to eq. (2.3).

mation theory, the entropy $S(f)$:

$$S(\bar{F}) = k \ln\big(n(\bar{F})\Delta F\big). \tag{2.4}$$

Using the definition of a statistical weight[1]: $W(\bar{F}) = n(\bar{F})\Delta F$, the last equation can be written in short as:

$$S(\bar{F}) = k \ln W(\bar{F}). \tag{2.5}$$

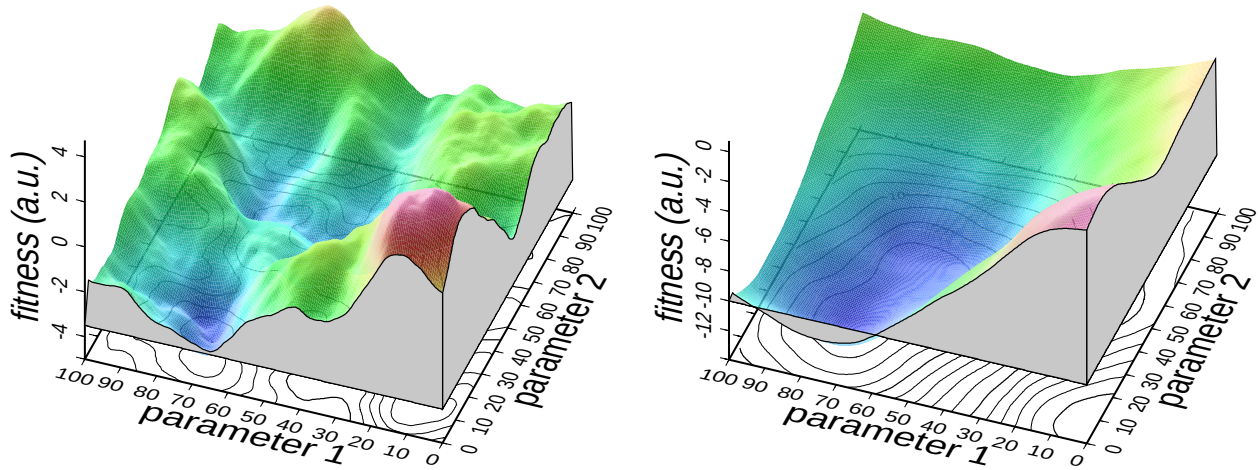
The entropy S can represent the currently missing knowledge about the investigated problem within a single number. The minimal value $S = 0$ is reached for a completely unveiled landscape. Even the partial knowledge of the density function gives valuable information about the system. For example, it enables the prediction of the optimization result and thereby provides some guidelines for the necessary computation time that still has to be invested [15]. Figure 2.3 gives an impression of the procedure. As all predictions based on extrapolation, the outcome has to be taken cum grano salis.

The Autocorrelation Function

In order to understand the autocorrelation function, one first has to have an understanding of the terms *autocovariance* and *variance*. The first term, autocovariance, literally means "how something varies with itself" [16]. It is the average of the deviation of a function from its mean value $\bar{x}$ at point x_t joint by the corresponding deviation at a lagged point x_{t+k} (cf. Figure 2.5). So the autocovariance σ_a^2 can be written as:

$$\sigma_a^2 = \frac{1}{N+1} \sum_{t=1}^{N-k} (x_t - \bar{x})(x_{t+k} - \bar{x}) \tag{2.6}$$

[1] The statistical weight denotes the number of realizations of a certain fitness level.



(a) correlation length: $r = 10$

(b) correlation length: $r = 50$

Figure 2.4: Two fitness landscapes with different correlation length r. The landscape in subfigure (a) has a relatively short correlation length while the subfigure (b) in contrast shows a highly correlated landscape.

The autocovariance can be normalized and made dimensionless to have a useful means of comparing different functions. This is achieved by a standardization with the variance σ^2 which essentially reflects the fluctuation of a function around its mean value:

$$\sigma^2 = \frac{1}{N+1} \sum_{t=1}^{N} (x_t - \bar{x})^2 \tag{2.7}$$

The resulting fraction of autocovariance and variance for a given lag k is the so-called autocorrelation coefficient R_k:

$$R_k = \frac{\sigma_a^2}{\sigma^2} = \frac{\sum_{t=1}^{N-k} (x_t - \bar{x})(x_{t+k} - \bar{x})}{\sum_{t=1}^{N} (x_t - \bar{x})^2} \tag{2.8}$$

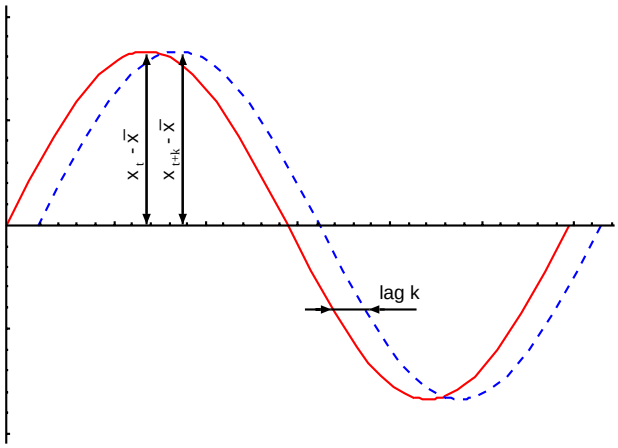


Figure 2.5: Graph and lagged graph of a function. For simplicity, the mean $\bar{x}$ is set to zero.

The entire series of autocorrelation coefficients constitutes the autocorrelation function. Since the autocorrelation coefficients can vary from -1 to $+1$ the autocorrelation function (correlogram) is confined to the same interval: $\{-1, 1\}$. As can immediately be seen from eq. (2.8) the correlogram is able to reflect linear dependencies only.

Nevertheless, autocorrelation provides a useful means of categorizing fitness landscapes. The most interesting value is the correlation length, which measures in generic units (i.e. Hamming distance) in how many steps the autocorrelation function has decreased from 1 to the value $1/e$ (which is roughly 0.37). Examples of different autocorrelation functions can be found in section 4.2.1, Figure 4.1. To give an impression of fitness landscapes with different correlation length compare Fig 2.4(a) and Fig 2.4(b). While the highly correlated landscape in Figure 2.4(b) has one pronounced valley and smooth inclinations, the shortly correlated landscape in Figure 2.4(a) shows numerous peaks and troughs within a generally rough surface. Please note that the parameter intervals $\{0, 100\}$ are, of course, the same for both landscapes.

2.1.3 Stochastic Modeling of Basic Evolutionary Strategies

Taking a look at natural evolutionary processes and adaptation, several strategies can be observed [17, 18, 19, 20]. These strategies include changes in genotype (mutations), changes in phenotype, selection processes, learning, and knowledge transfer (communication). It would require far too much computational power to try to mimic all of these processes for optimization purposes. A more promising ansatz for numerical evolutionary optimization algorithms is to place one or more virtual seekers, each representing one possible parameter combination, onto the fitness landscape in question and restrict the strategy to fundamental processes:

1. First and foremost, every seeker has to have a sophisticated concept of *how* to move about the search space. A movement in the search space is equivalent to a change in parameter space (cf. Figure 2.4). The new parameter combination represents a new potential solution to the problem with a fitness level that is usually different. These movements in search space (parameter changes) will henceforth be called mutations. This is inspired by the fact that in biology mutations also potentially change the fitness of an individual.[1]

2. Secondly, if a seeker ensemble is used instead of a single seeker, there has to be a way to drop inefficient candidates. The process of canceling seekers (and optionally replacing them with better ones) will, again in analogy to biology, be called selection. The selection process constitutes a basic seeker coupling or seeker communication.

3. The search strategy needs to be adaptive to ensure efficiency while the seekers zero in to global optima. Seeker agility that is too high can cause the

[1] The close relation between mutation in optimization and mutation in biology becomes visible in Genetic Algorithms where a mutation operator alters one or more bits of a string (a virtual gene) at a time (cf. page 29).

ensemble to spread unnecessarily in the late optimization phase. This adaptation can be achieved by techniques introduced later on as *annealing* or *mutation rate adaptation*.

The introduced evolutionary strategies differ by their realization of the basic processes given in the enumeration above. A relatively simple strategy is the Darwin strategy:

The Darwin Strategy

The ingredients for the Darwin Strategy are:

- mutation processes
- self reproduction of superior species showing best fitness

It is relatively easy to mathematically model this behavior [21]: The problem is defined as the search for a maximum on a potential V_i representing the fitness landscape, or search space, respectively. The index i denotes the fact that the potential that is probably continuous, is reduced to an integer set with s states ($i = 1, \ldots, s$) in order to be numerically tractable. The number s can still grow extremely large, however.

Thus, the parameter/fitness landscape as shown in Figure 2.2 gets translated onto a state/potential landscape as sketched in Figure 2.6. Modeling the seeker population as the occupation number x_i of state i, it becomes possible to describe mutations as transitions from state j to state i and arrange the transition rates in matrix form A_{ij}. This leads directly to the following balance equation:

$$\partial_t x_i = \sum_{j=1}^{s} \Big(A_{ij} x_j - A_{ji} x_i \Big). \tag{2.9}$$

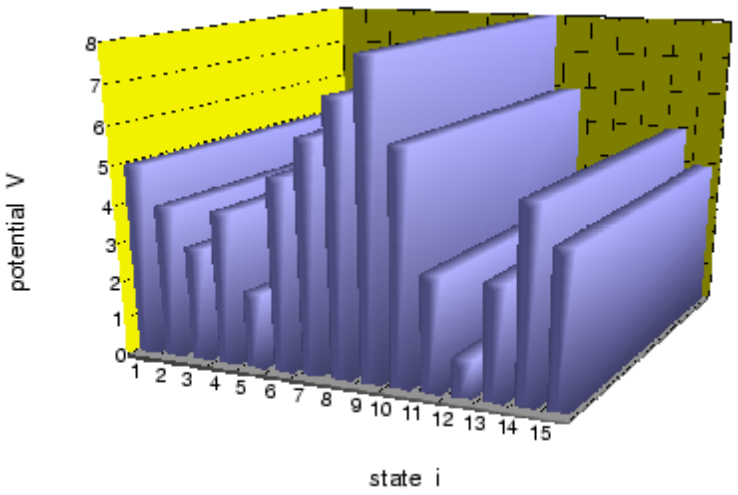


Figure 2.6: Discrete representation of a continuous fitness landscape as shown for example in Figure 2.2.

In the interest of simplicity, the number of seekers x_0 can be kept constant throughout the search process:

$$\sum_{i=1}^{s} x_i(t) = x_0 = \text{const.} \tag{2.10}$$

Adding the fitness-dependent self-reproduction yields a FISHER-EIGEN equation describing the problem-solving dynamics [17]:

$$\partial_t x_i = (\langle U \rangle - U_i)\, x_i + \sum_{j=1}^{s} \left(A_{ij} x_j - A_{ji} x_i \right). \tag{2.11}$$

By assuming symmetrical mutation rates $A_{ij} = A^0_{ij}$, with A^0_{ij} therefore being a symmetrical matrix, one can solve eq. 2.11 with the ansatz:

$$x_i(t) = \exp\left[-\int_0^t \langle U_{t'} \rangle \, \mathrm{d}t' \right] y_i(t) \tag{2.12}$$

leading to

$$\partial_t y_i(t) = -\sum_{j=1}^{s} H^D_{ij}\, y_i(t). \tag{2.13}$$

The HEISENBERG matrix H^D is defined as

$$H_{ij}^D := -A_{ij}^0 + \delta_{ij}\left(\sum_{k=1}^{s} A_{ki}^0 - V_i\right) \tag{2.14}$$

The solution may now be expressed in terms of the eigenvalues ε_n and eigenfunctions y_n of the eigenvalue problem as [17, 18]:

$$\sum_{j=1}^{s} H_{ij}^D\, y_j^n = \varepsilon_n^D\, y_i^n; \qquad n = 1 \ldots s \tag{2.15}$$

$$y_i(t) = \sum_{n=1}^{s} \exp\left[-\varepsilon_n^D\, t\right] a_i^n\, y_i^n \tag{2.16}$$

The time dependent-occupation numbers are as follows:

$$x_i(t) = \frac{y_i(t)}{\sum\limits_{j=1}^{s} y_j(t)} \tag{2.17}$$

This strategy has a highly erratic search path, since motion along gradients is not explicitly modeled. By implementing the latter feature, one arrives at the so-called Boltzmann Strategy.

The Boltzmann Strategy

This fundamental strategy describes processes corresponding to the second law of thermodynamics. It is also known as the *Metropolis Algorithm* [3]. It combines the following two basic elements:

- motion along gradients to reach steepest ascent/descent of thermodynamic functions
- stochastic processes including thermal and hydrodynamic fluctuations leading to random changes in order to avoid locking in local maxima or minima respectively

A theoretical model can be constructed analogously to the Darwin strategy by considering a set of states $i = 1, 2, \ldots, s$. Again, each state is characterized by a potential energy $U_i = -V_i$ and a relative frequency in the seeker ensemble population $x_i(t)$ at time t. The simplest model of a Boltzmann Strategy searching for minima of U_i is described by the following master equation:

$$\partial_t x_i(t) = \sum_{j=1}^{s} \Big(A_{ij}\, x_j(t) - A_{ji}\, x_i(t) \Big) \tag{2.18}$$

with the following transition rates:

$$A_{ij} = A^0_{ij} \begin{cases} 1 & \text{if } \Delta U < 0, \\ \exp[-\beta\, \Delta U] & \text{if } \Delta U \geq 0. \end{cases} \tag{2.19}$$

A process searching for maxima of U_i can be implemented by simply changing the sign of the ΔU conditions in eq. 2.19. The parameter β, known from thermodynamics to typically be $\beta = 1/kT$, has the meaning of a reciprocal temperature. The Boltzmann constant k can be set to 1 without altering the character of the search strategy. Now, how can the equations 2.18 and 2.19 actually be portrayed?

While the Darwin strategy allows the seeker ensemble to wander about indifferently (leading to a symmetric transition matrix A^0_{ij}) unless they are terminated by selection processes, the Boltzmann strategy takes energy changes ΔU into account. Mutation steps leading to improvements (uphill for maximization and downhill for minimization) are always accepted, whereas degradations are exponentially weighted with respect to the threshold's height. The idea, obviously, is to take the best characteristics from simple gradient search methods (fast search and easy implementation) while avoiding their pitfalls (trapping in local optima). The exponential weight (Boltzmann factor) assures that drastic degradations are rarely ever accepted.

This construction as a whole causes the distribution of seekers to assume the form of the well-known Boltzmann distribution [15]:

$$x_i = \frac{1}{Z} \exp\left[-U_i/T\right] \tag{2.20}$$

$$Z = \sum_{i=1}^{s} \exp\left[-U_i/T\right] \tag{2.21}$$

that is centered around the maxima (or minima respectively) of the fitness landscape as time goes to infinity. Therefore, the master equation (2.18) indeed describes an optimizing process.

The parameter F in the equations above denotes a problem-dependent fitness based upon the energy U_i and the search direction (maximization/minimization). The dimensionless normalization factor Z is the partition function.

The Mixed Boltzmann-Darwin Strategy

It is intuitively clear that the gradient-guided search of the Boltzmann Strategy is very effective for smooth fitness landscapes, while the Darwin strategy shows its strength in shortly-correlated, rugged landscapes, where its ability to tunnel high fitness barriers is advantageous. Numerical experiments show that a search strategy combining the basic ingredients of both the Darwin and the Boltzmann strategy easily surpasses both pure search algorithms (cf. section 4.2). Going back to equations (2.11, 2.18), it is straightforward to write down the master equation for the Boltzmann-Darwin dynamics. The equation contains the selection term, the mutation term, and the Boltzmann factor (hidden inside the mutation matrix):

$$\frac{\mathrm{d}}{\mathrm{d}t} x_t(t) = \gamma \underbrace{\Big(\langle U \rangle - U_i\Big)\, x_i(t)}_{\text{selection term}} + m \underbrace{\sum_{j=1}^{s} \Big(A_{ij}\, x_j(t) - A_{ji}\, x_i(t)\Big)}_{\text{mutation term}} \tag{2.22}$$

The mutation matrix A_{ij} is defined according to eq. (2.19). The new factor γ denotes the selection strength, whereas the factor m denotes the mutation rate. Since, numerically, one can only execute one step at a time, both are related via:

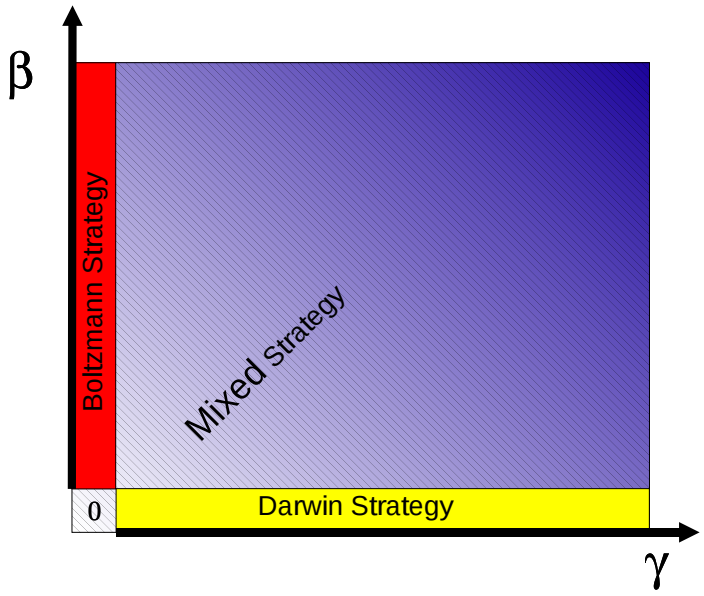


Figure 2.7: Parameter space for the different search strategies; Boltzmann Strategy: $\beta \neq 0$, $\gamma \equiv 0$; Darwin Strategy: $\beta \equiv 0$, $\gamma \neq 0$

$m+\gamma \stackrel{!}{=} 1$. It is easy to see now that the pure Boltzmann Strategy is contained in eq. (2.22) for $\gamma = 0$, while the pure Darwin strategy is obtained by setting $\gamma = 1$ and $\beta \to 0$.

So far, the selection is restricted to fitness proportional survival. In order to also allow nonlinear selection functions, eq. (2.22) needs to be written in a somewhat more general form [22]:

$$\frac{\mathrm{d}}{\mathrm{d}t}x_t(t) = \gamma\, f(\Delta U)\, x_j(t)\, x_i(t) + m \sum_{j=1}^{s} \Big(A_{ij}\, x_j(t) - A_{ji}\, x_i(t) \Big) \tag{2.23}$$

Now it is possible to introduce a selection such as

$$f(\Delta U) = \text{const} - \Theta(\Delta U). \tag{2.24}$$

This is used for all numerical simulations in this work (cf. chapter 4).

Here, Θ describes a step function which switches from **0** (all values less than 0) to **1** (all values greater than 0).

Very efficient and, therefore, used in the numerical simulations is the so-called tournament selection, which works as follows:

1. In a selection step randomly pick m seekers from the ensemble.
2. Rank the obtained m seekers according to their fitness.[1]
3. Replace the worst seeker with the best of the m candidates.

Obviously, the strategy now requires at least an $N > m$ seeker ensemble which is then globally coupled via selection processes. The tournament size m is a free parameter. Since the worst of the m seekers is dropped in a selection step, by increasing m one indirectly also increases the selection strength. A typical tournament size chosen for numerical simulations is $m = 4$.

2.1.4 Other Stochastic Optimization Strategies

Simulated Annealing

In 1983, KIRKPATRICK and co-workers introduced a new optimization strategy that was inspired by thermodynamics [4]. Simulated Annealing basically extends the Metropolis algorithm (cf. eq. (2.18)) by making the temperature a variable in the search process.

While high temperatures are beneficial in the early optimization phase (they allow for widespread seeker ensembles), it makes the search inefficient in zeroing in on the fitness optima. The idea, therefore, is to cool down the temperature along the search path to enable the ensemble to finally focus.

[1] Efficiency demands that instead of a complete ranking which is at least of order $\mathcal{O}(L \log[L])$ the best and the worst seeker must be found only. The latter is an $\mathcal{O}(L)$ problem.

The crucial point using simulated annealing is the actual annealing schedule. Several *ad hoc* schedules have since been proposed[1], but they are hard to motivate in theory. It was, however, possible to partially deduce optimal annealing schedules analytically for special problems (spin glass [5]; Ising model [24]). In 1993 ANDRESEN proposed an annealing schedule that suggested a constant thermodynamic annealing speed that adapted itself to the optimization problem [25, 26, 27]. His basic physical idea was to minimize the cumulative entropy production for the cooling process. The resulting schedule contained the constant annealing speed v_c as a free parameter and the relaxation coefficient ε as well as the heat capacity C as problem dependent values:

$$\frac{\mathrm{d}T}{\mathrm{d}t} = \frac{v_c\, T}{\varepsilon\, \sqrt{C}} \tag{2.25}$$

The last equation can be written equivalently as [26, 28]:

$$v_c = \frac{\langle U \rangle - U_{eq}(T)}{\sigma} \tag{2.26}$$

with $U_{eq}(T)$ being the internal energy the system would have if it were in equilibrium with its surroundings at temperature T. In eq. (2.25), $C(T)$ and $\varepsilon(T)$ are estimated based on the entire past history of the annealing [25]. This makes numerical simulations using ANDRESEN's schedule somewhat tedious.

Genetic Algorithms

Genetic algorithms that are outside the scope of this work appeared first in the 1970's and, in a way, pioneered evolutionary algorithms.[2] It was mainly the works of HOLLAND [19, 29, 30], GOLDBERG [31, 32], DE JONG [33, 34, 35] et. al. that laid the theoretical foundation.

Essentially, the difference between evolutionary algorithms and genetic algorithms is the representation of search space elements. Genetic algorithms, or

[1] Among them: linear cooling, exponential cooling, fast simulated annealing [23] etc.

[2] The first Proceedings of the International Conference on Genetic Algorithms did not appear before 1985.

GAs for short, restrict themselves to a bit-string representation of data structures reflecting some sort of chromosome representation.

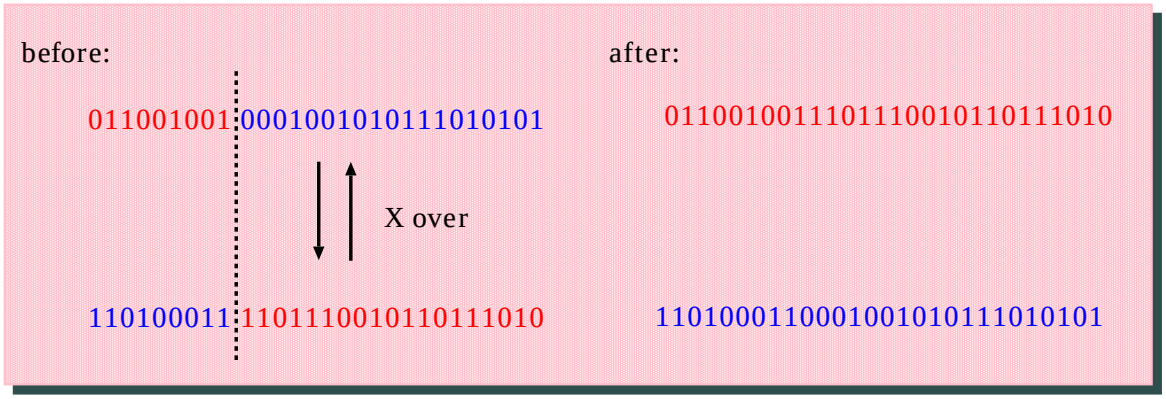


Figure 2.8: One possible realization of a crossover operator working on a bit string. First, a common crossover point for two candidate strings is randomly chosen. In a second step the tails of both strings are exchanged.

All operators, such as mutation, are therefore binary operators like *insertion*, *deletion*, *bit inversion*, or *string reversion*. This artificial restriction makes it easy to introduce a crossover operator[1] to the search dynamics. This operator, as seen in Figure 2.8, is able to efficiently exchange building blocks between different seekers. This is the starting point for the 'schema theorem'[2], which investigates *why* genetic algorithms are actually able to optimize.[3] It is evident, however, that problems that cannot be split into the form of building blocks will not benefit from crossover operations. At this point evolutionary algorithms are more appropriate tools to tackle the optimization problem.

[1] Some authors prefer to typeset crossover as *Xover*
[2] For a detailed introduction refer to [12].
[3] For a different explanation, cf. [36, 37].

Chapter 3

Model Problems

3.1 Correlated Random Landscapes

In a working paper, STEINBERG [38] proposed an approach to generate n-dimensional random landscapes with a predefined correlation length r, which will be briefly introduced here. Correlated random fitness landscapes generated as described below offer a nice set of features to test the effectivity of evolutionary algorithms: A typical landscape has numerous local maxima and minima, a known correlation length and a given number of dimensions. Figure 2.4 shows examples of such landscapes for two dimensions.
To construct the landscape the energy $U(x)$, the mean value $\langle U(x)\rangle$, and the correlation function are predefined.

$$\langle U(\vec{x})\rangle = 0 \tag{3.1}$$

$$\langle U(\vec{x})\,U(\vec{x'})\rangle = K(|\vec{x}-\vec{x'}|) \tag{3.2}$$

Decomposing the fitness landscape to uncorrelated Gaussian random numbers yields:

$$U(\vec{x}) = \int_{-\infty}^{\infty} d\vec{x'}\, h(\vec{x},\vec{x'})\, \xi(\vec{x'}) \tag{3.3}$$

$$\langle \xi(\vec{x})\, \xi(\vec{x'}) \rangle = \delta(\vec{x}-\vec{x'}) \tag{3.4}$$

To determine the yet unknown function $h(\vec{x})$ one can combine eq. 3.2 and eq. 3.3.

$$\langle U(\vec{x})U(\vec{x'}) \rangle = \int_{-\infty}^{\infty} d\vec{x''} h(\vec{x},\vec{x''})h(\vec{x'},\vec{x''}) \tag{3.5}$$

Introducing the Fourier spectrum of the correlation function

$$S_{UU} = \int_{-\infty}^{\infty} d\vec{x'}\, \langle U(\vec{x})\, U(\vec{x}+\vec{x'}) \rangle\, e^{i\vec{k}\vec{x'}} \tag{3.6}$$

and returning to eq. 3.3 yields

$$S_{UU} = |H(\vec{k})|^2 \tag{3.7}$$

with

$$|H(\vec{k})|^2 := \int_{-\infty}^{\infty} d\vec{x'} h(\vec{x'}) e^{i\vec{k}\vec{x'}} \int_{-\infty}^{\infty} d\vec{x''} h(\vec{x''}) e^{i\vec{k}\vec{x''}} \tag{3.8}$$

In general, eq. 3.7 becomes

$$S_{UU} = S_{\xi\xi} |H(\vec{k})|^2 \tag{3.9}$$

with $S_{\xi\xi}$ being the Fourier transform of the random number's correlation function. The last equation finally leads to:

$$h(x) = \int_{-\infty}^{\infty} d\vec{k} \sqrt{\frac{S_{UU}}{S_{\xi\xi}}}\, e^{-i\vec{k}\vec{x}} \tag{3.10}$$

It is then quite simple to get from the continuous to the discrete landscape. The example shown in Figure 2.4 was generated using the following iteration:

$$n_{0000} = \sqrt{\langle U_{00}\, U_{00}\rangle} \tag{3.11}$$

$$\begin{aligned} n_{isjt} = \frac{1}{n_{sstt}} \Big(& \langle U_{ij}\, U_{st}\rangle \\ & - \sum_{l=0}^{t-1} \sum_{k=0}^{s} n_{ikjl}\, n_{sktl} \\ & - \sum_{k=0}^{s-1} n_{ikjt}\, n_{sktt} \Big) \end{aligned} \tag{3.12}$$

$$U_{ij} = \sum_{k\le i} \sum_{l\le j} n_{ikjl}\, \xi_{kl} \tag{3.13}$$

It lies in the algorithm's iterative nature that generating already relatively small landscapes (100 steps in each direction) becomes quite computation intensive for $n = 3$ or more dimensions. Therefore, the software developed to generate these fitness landscapes was designed to benefit from multiprocessor machines (cf. section 5.1.4).

3.2 Frustrated Periodic Sequences

As the name suggests, Frustrated Periodic Sequences introduced by ENGEL and FEISTEL [17] are an example of frustrated fitness functions. The aim of the problem is to optimize two contradictory goals (alphabetic order versus periodicity). So the optimal solution has to be a compromise.
A sequence consists for example of λ letters:

$$\lambda \in \{A,B,C,D\}.$$

The fitness function $F(x)$ is defined as follows:
The function $\alpha(x)$ denotes the number of letters occurring in alphabetic order. (The sequence (D,A) is also considered to be alphabetical.)
The function $\pi(x)$ is defined as the number of letters occurring with period $p \neq \lambda$.

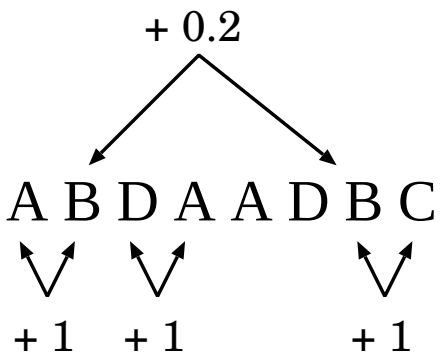


Figure 3.1: Frustrated Periodic Sequence evaluation scheme for a period $p = 5$ and $b = 0.2$.

Then, the fitness function is calculated as

$$F(x) = \alpha(x) + b\,\pi(x). \tag{3.14}$$

The free parameter b weighs between preferably alphabetic or periodic sequences. For $b = 0$ optimal sequences are purely alphabetic, while for $b \to \infty$ optimal sequences are purely periodic. Maximal frustration is reached if one chooses the

parameter λ to be [15]

$$b = \frac{1}{p}$$

Figure 3.1 demonstrates the evaluation of a sequence for $p = 5$ and $b = 0.2$. Frustrated Periodic Sequences form Gaussian landscapes with respect to the density of states. Their structure, however, is rather simple. In the case of maximal frustration, the best sequences are made of building blocks:

- alphabetic structure: $\underbrace{ABCDA}_{\text{block 1}}\,\underbrace{BCDAB}_{\text{block 2}}$
- periodic structure: $\underbrace{ABCDA}_{\text{block 1}}\,\underbrace{ABCDA}_{\text{block 2}}$

These building blocks induce a high degeneracy of optimal sequences and exponentially long correlations in the fitness landscape (cf. Figure 4.1), rendering the problem rather easy, despite its appearing complexity.

3.3 The LABS Problem

The LABS (low autocorrelation binary sequences) problem introduced in 1990 by GOLAY has been studied intensely [39, 40, 13]. It is undoubtedly a hard problem to solve. The optimization goal is to minimize the autocorrelation of a binary string S. The string S is composed of $+1$ and -1 bits:

$$S = \{s_1, s_2, \ldots, s_L\}; \quad s_i \in \{-1, +1\} \tag{3.15}$$

The autocorrelation coefficient R for distance k is given by:

$$R_k = \sum_{i=1}^{L-k} s_i s_{i+k}. \tag{3.16}$$

As mentioned above, the aim is to minimize the quadratic sum E of all autocorrelation coefficients:

$$E = \sum_{k=1}^{L-1} R_k^2 \tag{3.17}$$

or equivalently maximize the so called MERIT-factor F:

$$F = \frac{L^2}{2E} \tag{3.18}$$

For most (but not all) odd length sequences, the highest Merit factor is achieved by skew-symmetric configurations. Skew-symmetric sequences fulfil the relation

$$s_{\mu+i} = (-1)^i s_{\mu-i}; \quad \mu = \frac{L+1}{2} \tag{3.19}$$

and, therefore, have $R_k = 0$ for all odd k. Due to the $\{+1;-1\}$ symmetry, the optimal sequence is degenerated, but the optimization still resembles the search for the infamous needle in a haystack.

3.4 The RNA and NK Model Compared

3.4.1 The NK Model

The NK model is an abstract model introduced by KAUFFMAN [20] in the framework of population genetics. In its structure it is very similar to the well-studied spin glasses introduced by EDWARDS, ANDERSON [41], et. al. A spin glass is typically described as a two or three dimensional lattice carrying N coupled spins which can point either up or down. Hence, there are 2^N possible configurations with a total energy given by the Hamiltonian:

$$H = -\sum_{\substack{i,j \\ i \neq j}} J_{ij}(s_i \times s_j) \quad s_i, s_j = \pm 1 \tag{3.20}$$

where s_i and s_j are the orientations of the two spins. J_{ij} is the energy reflecting how strongly the two are coupled and, therefore, prefer to be in the more favorable relative orientation. Analogously, the NK model consists of N positions (gene loci) with two different possible states (alleles), 1 and 0. The parameter K stands for the average number of other loci which epistatically affect the fitness contribution of each locus. A possible third parameter describes how the K are distributed among the N. According to KAUFFMAN, it turns out that to a very large extent *only* N and K matter.

As the number of K increases the conflicting constraints lead to an increasingly more rugged, multi-peaked fitness landscape. Examining the landscape structure as a function of N and K shows two interesting extremes:

- $K = 0$ **:** This corresponds to a highly correlated, very smooth fitness landscape with a single peak. The difference in fitness between neighboring N is $1/N$, thus for large N the fitness of one-mutant neighbors is very similar.
- $K = N - 1$: This case corresponds to a fully random fitness landscape. Thus, the number of local fitness optima is extremely large and as the number of loci N increases, the local optima fall toward the mean fitness value of the fitness landscape.

The fitness landscape itself can be constructed as follows:

1. Assign to each locus i the K other loci which influence it.
2. For each of the possible 2^{K+1} combinations, assign for each locus i a fitness contribution w_i drawn at random from the interval $[0,1]$.
3. The fitness value of a given genotype is defined as the average of all contributions w_i:

$$W = \frac{1}{N}\sum_{i=1}^{N} w_i$$

3.4.2 RNA Secondary Structures

One particular optimization problem has gained increasing interest in physics and biology over the last couple of years: the stochastic folding kinetics of RNA[1] sequences into secondary structures [42, 43]. RNA sequences consist of bases that can be either purines (Figure 3.2) or pyrimidines (Figure 3.3).

While the bases Adenine and Guanine are the so called purines, the bases Thymine, Uracil, and Cytosine are pyrimidines. Thymine, however, is present in DNA[2] strands only, so a symbolic RNA sequence consists of the letters **A**(denine), **C**(ytosine), **G**(uanine), and **U**(racil): $\{A,C,G,U\}$.

A member of the purines can chemically bind to a member of the pyrimidines and vice versa. The result is what is know as a base pair. The most common ones are the Watson-Crick pairs ((G,C) and (A,U)) plus the 'twisted' pair (G,U). Thus, a plain RNA strand (primary structure; Fig 3.4) can curl up in the three dimensional space to form a secondary structure [3] (Figure 3.5a).

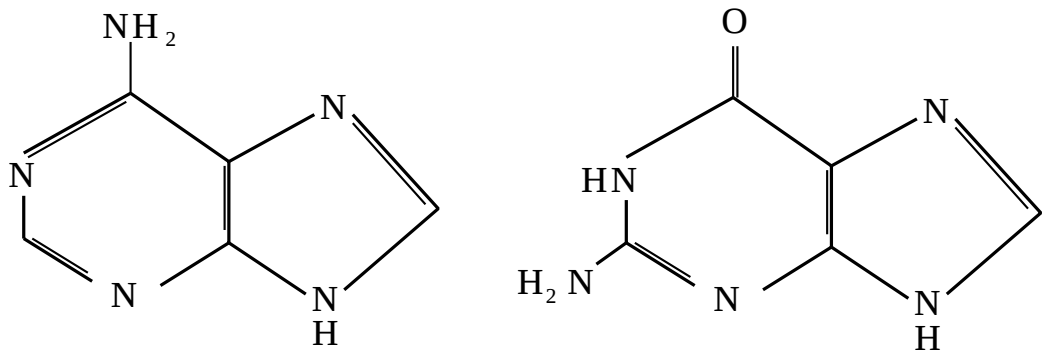


Figure 3.2: Purines: The bases Adenine and Guanine can be found as building blocks for RNA as well as DNA sequences.

[1] RNA: ribonucleic acid
[2] DNA: deoxyribonucleic acid
[3] online database: http://www.rcsb.org/pdb/

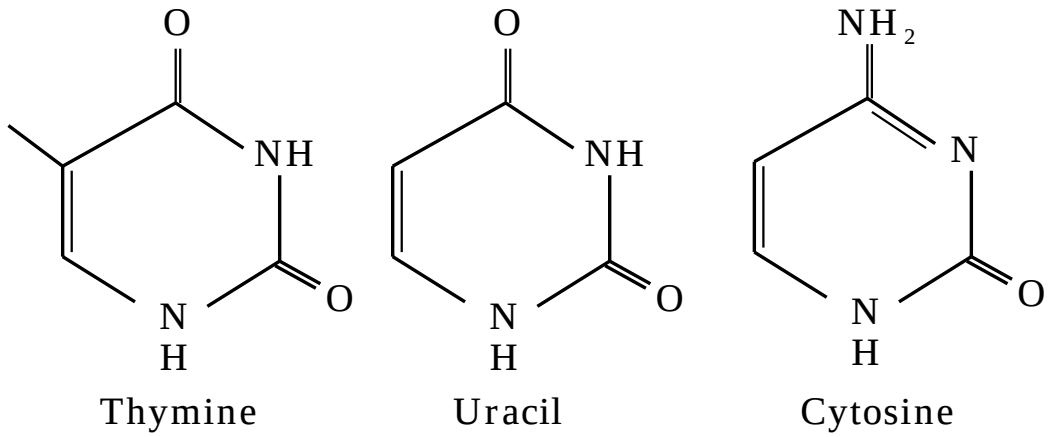


Figure 3.3: Pyrimidines: Uracil is found in RNA sequences only, while Thymine is specific to DNA sequences.

G-G-C-C-A-G-A-U-U-G-A-G-C-C-U-G-G-G-A-G-C-U-C-U-C-U-G-G-C-C

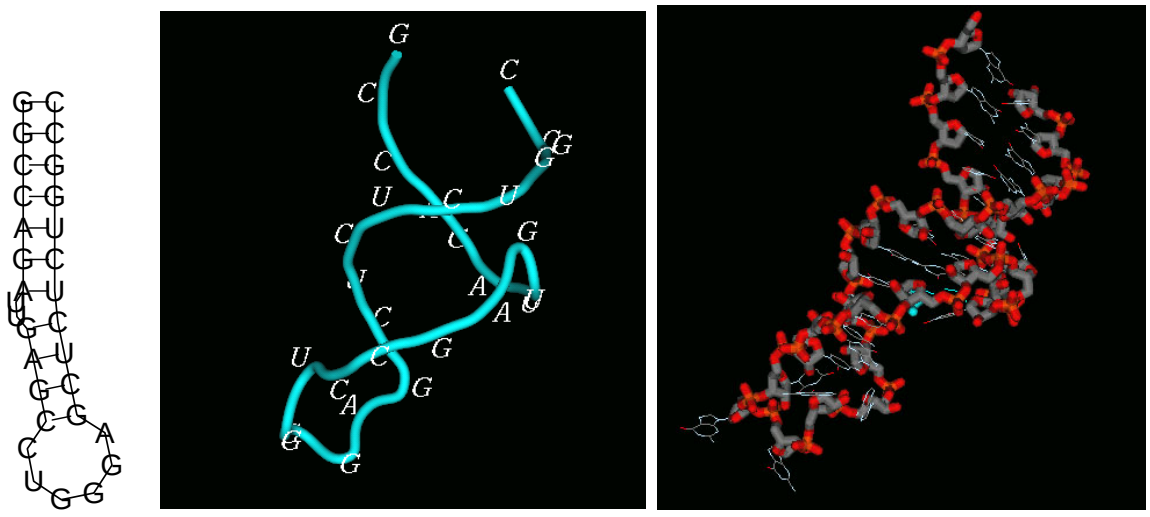
Figure 3.4: Primary structure of an RNA sequence with 30 bases. This RNA strand is the *HIV-2 Tar-Arganinamide Complex* which has the key **1AJU** in the online protein database.

The secondary structure, forming e.g. loops and 'hairpins', can fold into higher level structures like α-helices and β-sheets itself.The Figures 3.5(b) and 3.5(c) show such higher level structures of an RNA sequence.

It is not trivial to estimate the free energy of RNA secondary structures. Each base pair and each loop contributes a specific binding energy. In this work, the software *'Vienna RNA package'* Version 1.4 was used to numerically evaluate RNA sequences. This software package includes experimental data of binding energies and is freely available.[1]

In order to simplify matters somewhat, secondary structures can be written in a commonly used bracket notation: The positions of bases within an RNA strand

[1] http://www.tbi.univie.ac.at/∼ivo/RNA/



(a) (b) backbone (c) details

Figure 3.5: Secondary structure of the RNA strand shown in Figure 3.4. Subfigure (a) shows the bindings and secondary structure, subfigure (b) shows how the secondary structure curls up in a helix like structure and subfigure (c) gives a detailed picture of the bases.

are marked by dots, while base pairs are symbolized by closed parentheses. The sequence shown in Figure 3.4 with a secondary structure as displayed in Figure 3.5(a) can symbolically be written as:

$$(((((((..((((.....))))))))))$$

This binding structure together with the original sequence can be fed to the *RNAeval* routine contained in the Vienna package to obtain the free energy of the given secondary structure . All parentheses are assumed to be properly stacked, since this software is not yet able to take pseudo loops into account. Pseudo loops are higher order structures that occur whenever two bindings intersect, as shown in Figure 3.6. The optimization problem considered here is the search for a secondary structure, which minimizes the free energy of the RNA sequence.[1] Since even for short sequences the search space is enormous and the fitness landscape

[1]The minimal free energy conformation is often – but not always – the biologically occurring structure.

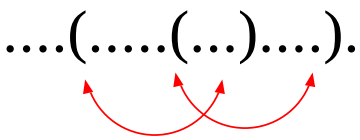


Figure 3.6: Intersecting bindings generate a pseudo loop.

is almost uncorrelated (cf. Figure 4.1), this optimization problem is particularly hard to solve.

Additionally, numerical simulations show that most initial folding steps increase the free energy compared to an unfolded sequence, since a single binding naturally forms a simple loop. Hence, it is a crucial point to design the numerical mutation operator used in evolutionary algorithms to allow for multiple bindings and dissections.

Sometimes simpler combinatorial models like the NK-model are used to mimic problems like RNA folding. It is therefore helpful to have a comparison of both problems [44]. The following table lists both the advantages and disadvantages of either model.

	NK-Model	RNA secondary structure
Advantages	• relatively simple model • analytically treatable • all values known • easy to implement numerically	• practically relevant • numerical implementation freely available • relevant values partially known
Drawbacks	• comparable to RNA models for limited parameter set • mostly poor correspondence to RNA data	• computationally intensive • pseudo knots not yet treatable • several unknown parameters • energy functional disputed

Chapter 4

Optimizing the Search Process

4.1 Exact Stochastic Simulations

A serious problem that has not yet been discussed is the fact that master equations, such as eq. (2.22), contain probabilities as variables. While it is still possible to write down the complete set of coupled differential equations for a system with very few possible states, the method becomes not feasible for large systems.

A possible way to generate valid trajectories according to the master equation is to choose the transitions and transition times for *a single trajectory* according to the correct probability distribution. This idea was proposed independently by FEISTEL [17, 45] and GILLESPIE. The latter suggested two different algorithms for numerical simulations [46, 47], which can be proven to be mathematically equivalent [46]. The *Direct Method* explicitly calculates which transition occurs next and when. The second one, the *First Reaction Method*, calculates a time τ at which the particular transition occurs for each transition A_{ij}, chooses the one with smallest τ, and executes it at time τ. Both algorithms will be briefly described in the following subsections.

4.1.1 The Direct Method

As stated above, the direct method follows the two questions:

- Which transition $j \rightarrow i$ occurs next?
- At what time τ does it occur?

The probability density $p_{ij}(\tau)$ that the next occurring transition is $j \rightarrow i$ at time τ is:

$$p_{ij}(\tau) = p_{ij} \exp\left[-\tau \sum_s p_{sj}\right] d\tau \tag{4.1}$$

The probability distribution P_{ij} for all transitions $j \rightarrow i$ can now easily be calculated as:

$$P_{ij} = p_{ij} \int_0^\infty \exp\left[-\tau \sum_s p_{sj}\right] d\tau = \frac{p_{ij}}{\sum_s p_{sj}} \tag{4.2}$$

The time distribution can be determined as well:

$$p(\tau)d\tau = \left(\sum_s p_{sj}\right) \exp\left(-\tau \sum_s p_{sj}\right) d\tau \tag{4.3}$$

The knowledge of both distributions can now be used to set up the following algorithm:

Direct Method Algorithm

1. Initialize seeker ensemble; set $t = 0$.
2. Calculate p_{sj} for all s.
3. Choose transition according to eq. (4.2).
4. Choose τ according to eq. (4.3).
5. Execute transition, set $t = t + \tau$ and go to step 2.

4.1.2 The First Reaction Method

Instead of directly calculating the probability distributions for both transition and time, one can equivalently calculate a putative time τ_i for each transition and then execute the one which would occur first. This is the *First Reaction Method* which has the advantage that it requires the generation of only one random number instead of two for each transition.

First Reaction Method

1. Initialize seeker ensemble; set $t = 0$.
2. Calculate p_{sj} for all s.
3. Calculate all putative times τ_i according to eq. (4.3).
4. Set $\tau = \min_i \tau_i$.
5. Choose transition with time τ.
6. Execute transition, set $t = t + \tau$ and go to step 2.

4.1.3 The Next Reaction Method

The *Next Reaction Method*, proposed by GIBSON and BRUCK [50], is an advancement of the algorithms introduced above. While these scale linearly with the number of transitions r, the *Next Reaction Method* performs $\mathcal{O}(\log(r))$ in a worst case scenario. The main ideas used according to GIBSON et. al., are:

1. Store all transition times τ_i.
2. Be extremely sensitive in recalculating the transition probabilities.
3. Re-use transition times τ_i where appropriate.
4. Switch from relative time (time between reactions) to absolute time.
5. Use efficient data structures to store transitions as well as transition times.

To realize the second and last points, the authors rely on dependency graphs and priority queues for numerical efficiency. The high effort quickly pays off when comparing simulation times.

The simulations carried out in this work implemented a variant of GILLESPIE's Direct Method, since the calculations of the extensive investigative ensemble statistics far outweighed everything else.

4.2 The Evolutionary Window

As discussed in subsection 2.1.3 (p. 25) mixed evolutionary strategies provide the highest flexibility for optimization tools, in terms of tuning measures. This benefit is paid for by the introduction of numerous free parameters such as ensemble size N, temperature $T = 1/\beta$, and selection pressure γ (cf. eq. (2.22)). This section investigates the influence of all these inherent search parameters on the optimization outcome using some model problems introduced in chapter 3.

4.2.1 Comparing Fitness Landscapes

In order to understand the results of numerical simulations, one has to have an impression of the underlying fitness landscape. As laid out in section 2.1.2, it is

helpful to either determine the density of states or the autocorrelation function. Here, an easy method to obtain the latter one will be introduced.[1]

A simple approach is to take a sample of the fitness landscape and calculate the whole spectrum of autocorrelation coefficients according to eq. (2.8). To reduce sampling effects, it is necessary to average the autocorrelation function over many different samples afterwards.

As already discussed in section 2.1.1 it is the mutation operator that generates a representation of the fitness landscape by determining the set of allowed moves in parameter space. Using the idea introduced above, the easiest approach is to simply start a search process with a single seeker at a randomly chosen position to get a sample of the fitness landscape, then calculate the autocorrelation function and iterate the procedure many times to have an averaged result.

For an infinite temperature, the seeker's path resembles what is known as a *random walk* across the landscape. It might however be easier to visualize the movement as a random flight where the temperature value symbolizes an altitude.[2] It is shown in Figure 4.1 how temperature-dependent the obtained autocorrelation function indeed is. For the models investigated, the correlation length decreases with increasing temperature for maximization problems (Figure 4.1 top and center) and vice versa for minimization problems (Figure 4.1 bottom).

This is easy to understand when referring to the picture used above. The higher the seeker's altitude is, the more structures will come into its scope and will decrease the correlation length. At high temperatures the RNA folding landscape becomes almost uncorrelated (correlation length $r \approx 1.7$ at temperature $T = 10$).

[1] Two different ways to investigate the density of states is described in [15].

[2] This picture is appropriate for a minimization problem only; for a maximization the inverse temperature β would correspond to the imaginary altitude.

Sticking to interesting temperature regions (cf. section 4.2.2), the optimization problems can most assuredly be ranked according to their difficulty level from easiest to most difficult as follows:

1. Frustrated Periodic Sequences (exponentially long correlation)
2. LABS problem (short correlation length)
3. RNA folding problem (almost uncorrelated)

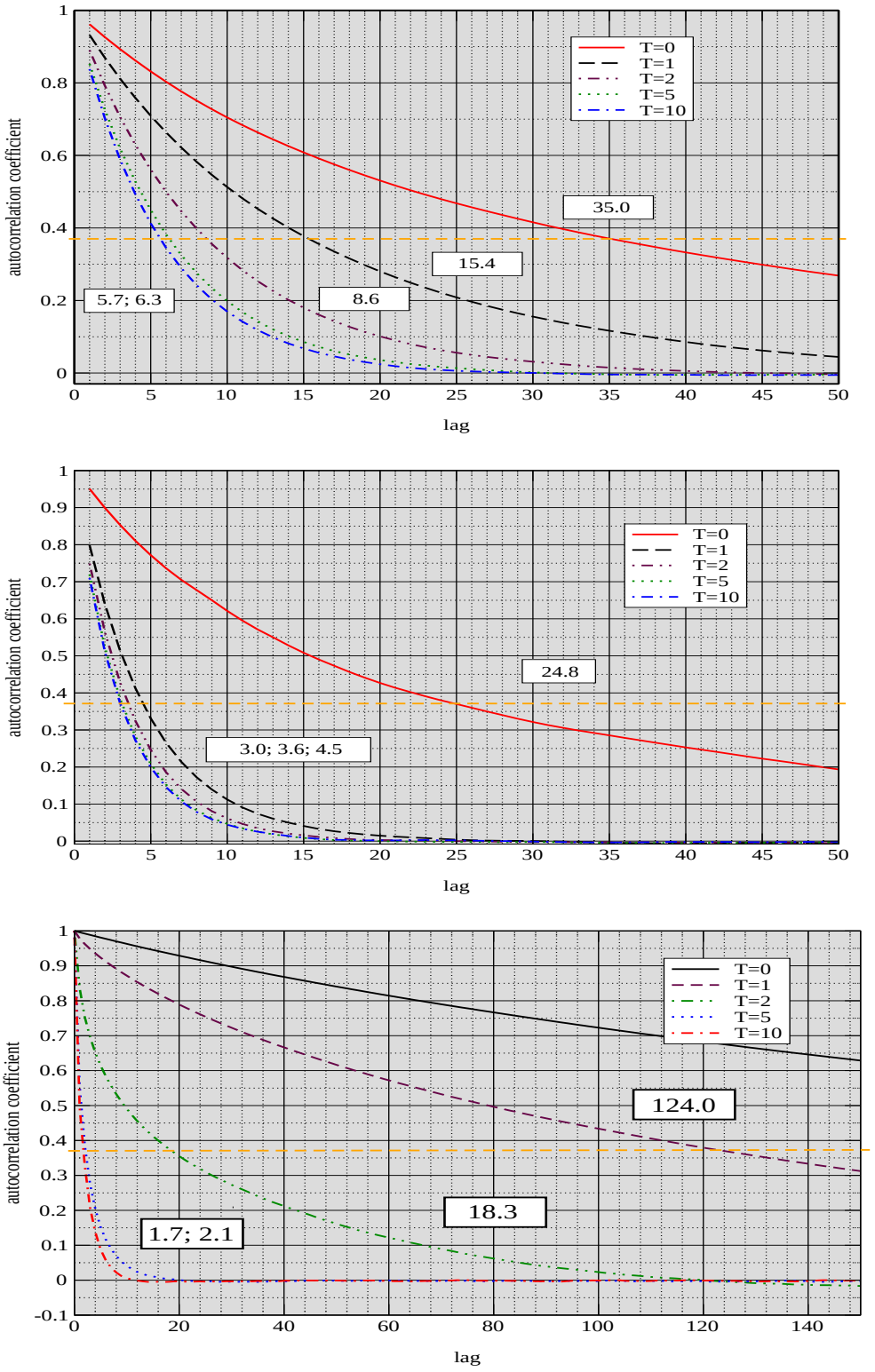


Figure 4.1: Temperature dependence of the autocorrelation function for length $L = 15$ Engel sequences (top), an $L = 32$ LABS problem (middle) and an $L = 100$ RNA folding problem (bottom, Polio virus Type 1, AC V01148; 5'-cloverleaf, cf. Appendix A). For each temperature, the respective correlation length is denoted.

4.2.2 Exploring Parameter Windows

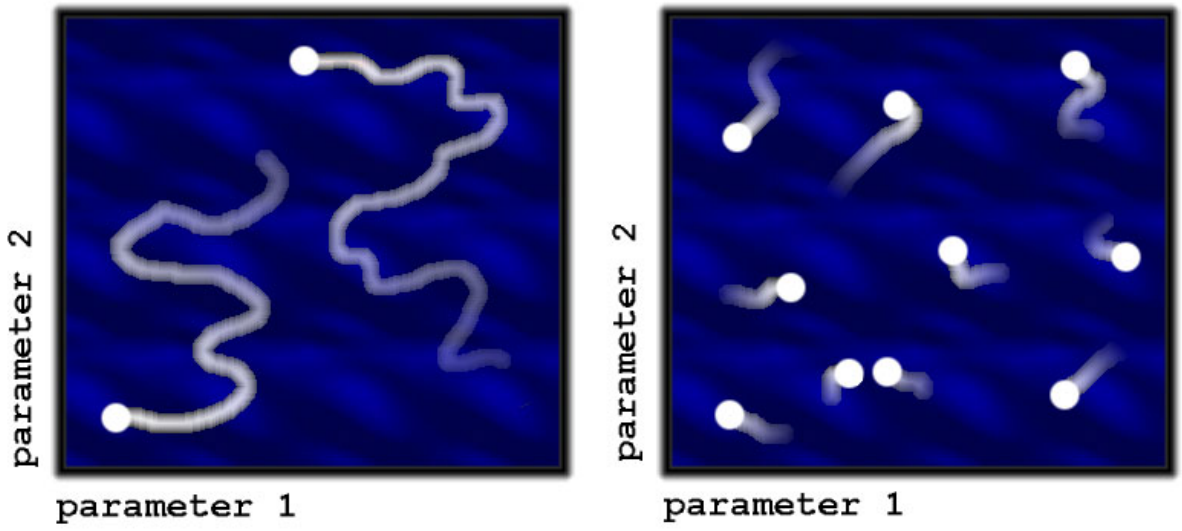


Figure 4.2: Computation time shared among the seeker ensemble implies that smaller ensembles (left figure) can explore longer optimization paths per seeker in the search space than bigger ensembles can (right figure).

Having three model problems of different difficulty level at hand, it is possible to numerically investigate the generic influence of the search parameters (ensemble size N, temperature T and mutation rate P_{mut}) on the optimization result.

All numerical simulations were carried out in such a way that a given absolute computation time was shared among all seekers of the ensemble. Thus, small ensembles allowed for longer search paths per seeker. In the limit of either infinite computation time or a small search space, there should be no notable influence of the ensemble size on the search result (granted, that the fitness landscape is ergodic[1]). For random initial conditions the entire search space can be equally well covered, as seen in Figure 4.2.

[1]If this is not the case, i.e. if some points in the search space are unreachable, the mutation operator is obviously ill-designed.

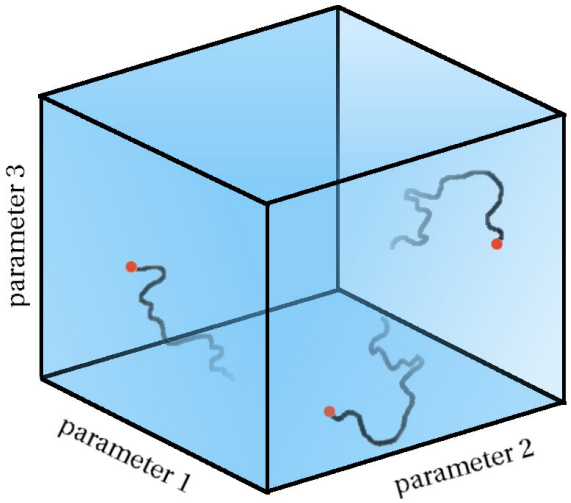


Figure 4.3: Despite long search paths, small seeker ensembles can not efficiently cover high-dimensional search spaces.

In the case of common optimization problems, the computation time is usually quite limited. As shown in Figure 4.3, the size of the seeker ensemble now makes a big difference indeed. Even an ensemble having only a couple of seekers cannot efficiently cover the search space, despite having longer search paths. The situation becomes even worse if the search space is high-dimensional. Clearly, bigger ensembles can be spread across the fitness landscape more easily. The ensemble size is, however, limited by the computation time, as seen in Figure 4.2. Too many seekers turn the search strategy into pure guessing with a simulation time per seeker diminishing to zero.

Summarizing the last paragraphs, it is now possible to make a few projections on the generic influence of the ensemble size for realistic optimization conditions (i.e. vast search space and limited computation time):

Uncoupled seeker ensemble: The volume of the search space obviously increases exponentially with the number of dimensions. At first glance, a linear

change in the ensemble size is therefore neglectable for uncoupled seekers.[1] Since the computation time is shared among the seekers however, one can expect a decreasing optimization result with increasing ensemble size.

Coupled seeker ensemble: Once the seekers form a coupled ensemble the initial conditions (initial distribution in search space) become crucially important. For small ensemble sizes seeker communication provides no advantages. On the other hand, ensembles that are too large are handicapped by insufficient computation time. One can, therefore, expect a pronounced optimum with respect to the ensemble size for coupled seekers, unless the fitness landscape is trivial.[2]

With these expectations in mind, it is now necessary to have a look at some numerical simulations and either verify or disregard the above conclusions.

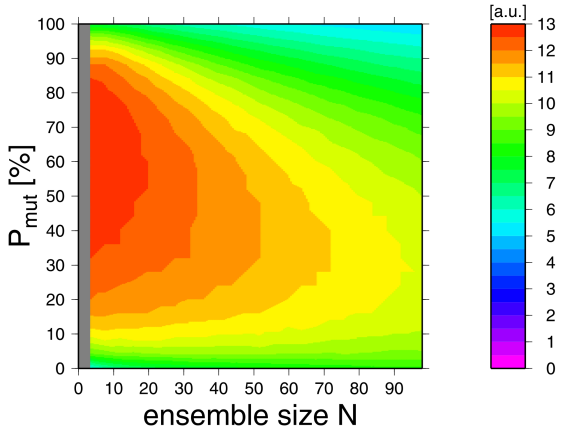
1. Constant Temperature

Figure 4.4 shows a summary for an exhaustive parameter sweep on all three test models. A mutation rate of $P_{mut} \equiv 100\%$ indicates absent selection steps and, therefore, represents an uncoupled seeker ensemble. Notably in this case an increasing ensemble size causes a decreasing optimization result as expected, regardless of the test problem. It is also immediately visible that the best results can be achieved only for a relatively small parameter window. This distinct window, called an *evolutionary window* from now on, always encloses mutation rates of $0\% < P_{mut} < 100\%$ and ensemble sizes with $N > 1$ seekers. A pure Boltzmann strategy ($P_{mut} \equiv 100\%$) turns out to be less effective than the Darwin

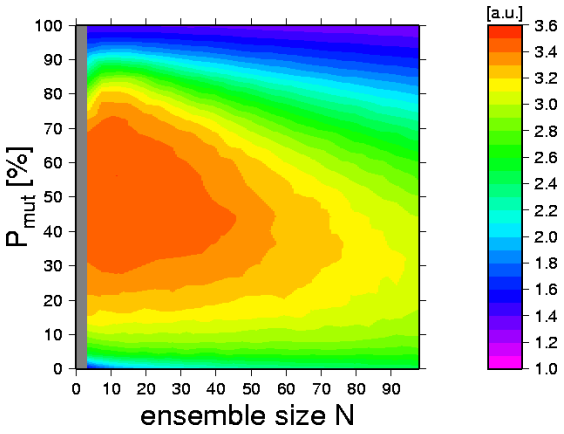
[1] The seekers are uncoupled if e.g. selection is missing. Thus, there is no communication between the individual seekers of the ensemble.

[2] For trivial landscapes communication does not have any benefits and the optimal (degenerated) ensemble consists of 1 seeker only.

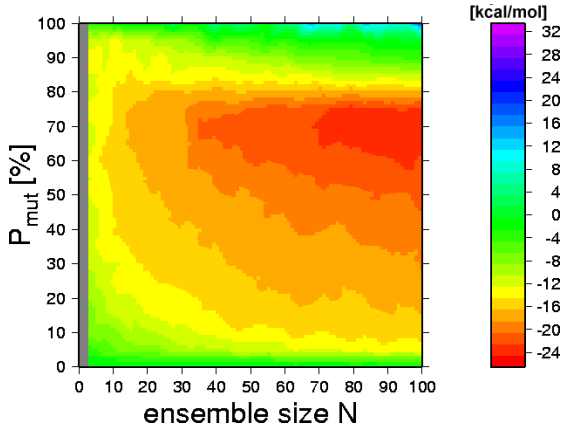
strategy and the mixed strategies, because the Boltzmann strategy cannot cover the evolutionary window (cf. Figure 2.7 on page 26).



(a) Frustrated Periodic Sequences



(b) LABS problem



(c) RNA secondary structures

temperature	T=1
calculation time	t=500
repetitions	R=1000
String Length:	
Frust. Per. Seq.	L=15
LABS Problem	L=32
RNA Second. Str.	L=100

(d) Search Parameters

Figure 4.4: Optimization results for an exhaustive parameter sweep on all three test models show a distinct parameter window (red area) with significantly better optimization results. The RNA sequence used in subfigure 4.4(c) (which displays the free energy instead of the fitness so that best results are again indicated by red colors) is the sequence of the first 100 base pairs of Polio virus type 1 Mahoney *AC V*01148 (cf. Appendix A). For all three models, random initial conditions were used.

Besides these common properties, Figure 4.4 also reveals some interesting differences between the test problems used. For any chosen mutation rate, frustrated periodic sequences do not benefit from an ensemble based optimization. In other words, small seeker numbers are the best choice. This indicates that, as stated above, that the fitness landscape is rather trivial. It is very unlikely that seekers get stuck in local optima along their respective search paths.

In contrast, the evolutionary window shows a pronounced maximum at ensemble sizes of $N \approx 10$ seekers for the LABS problem. Considering the short correlation length of the fitness landscape (cf. Figure 4.1), this is another hint that the optimization of *Low Autocorrelated Binary Sequences* is rather difficult.

Looking at Figure 4.4(c), one must keep in mind that in the case of RNA secondary structures, one is looking for the *minimal* free energy. The color scale was therefore inverted to assure that best results are again displayed in red.
The vast search space[1] and an almost uncorrelated landscape dramatically shift the evolutionary window, which is clearly marked again, so that optimal seeker ensembles contain some $N \approx 100$ seekers.

2. Variable Temperature

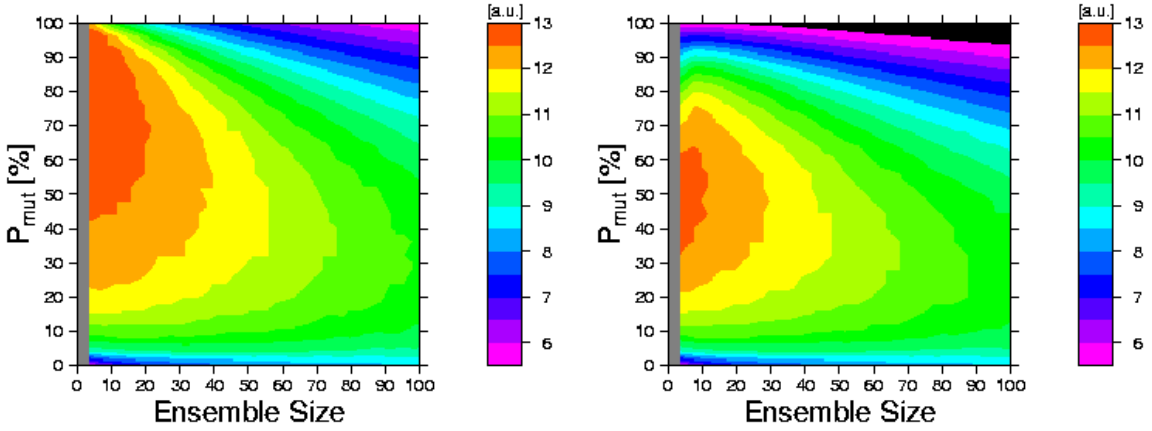
So far, the temperature was kept constant at $T = 1$ for all simulations. Since the various fitness landscapes' autocorrelation function has turned out to be very temperature dependent, the evolutionary window is also expected to show a dependence on temperature.

The results of the first problem investigated, Frustrated Periodic Sequences, is shown in Figure 4.5. The color scales are identical for all four subfigures; fitness values below $F = 5.5$ are displayed in black. Comparing the subfigures, the following statements can be made:

[1] Considering 4 bases and 3 possible base pairings for length L strings, ...

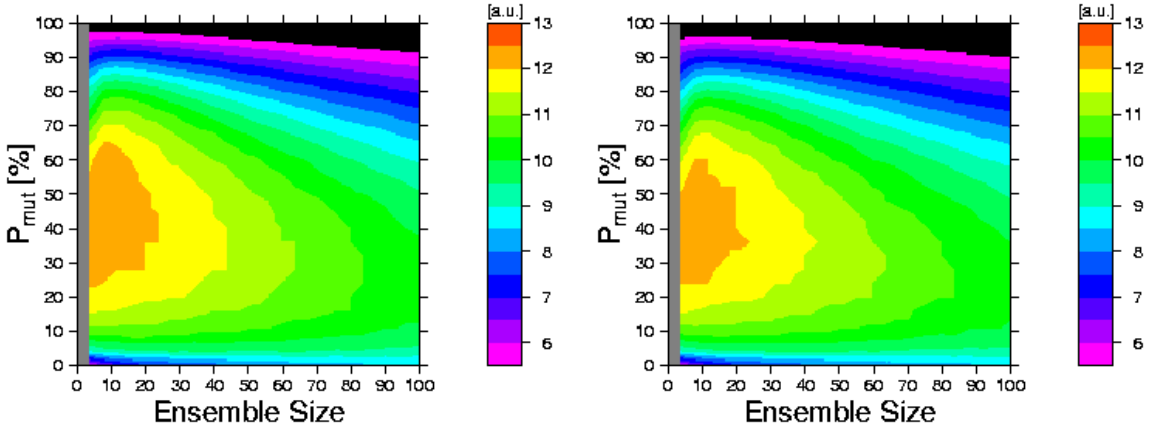
1. As can be seen, an increasing temperature shifts the evolutionary window towards lower mutation rates.

2. Furthermore, the evolutionary window shrinks quickly as the temperature rises.



(a) temperature $T = 0$

(b) temperature $T = 2$

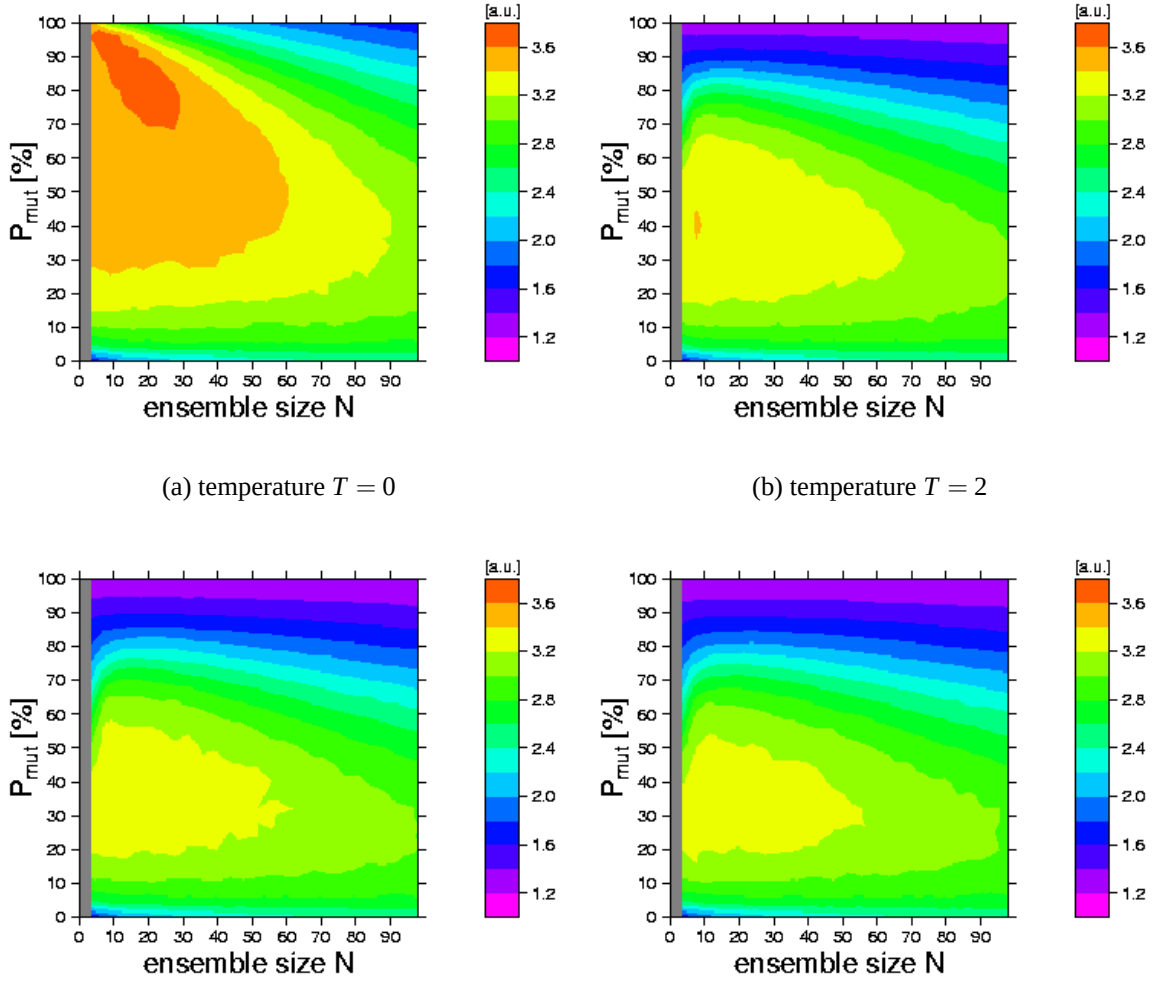


(c) temperature $T = 4$

(d) temperature $T = 6$

Figure 4.5: Mean ensemble fitness: Temperature dependence of the evolutionary window displayed for Frustrated Periodic Sequences. Sequence length $L = 15$; computation time $t = 500$; averaged over 1000 runs; Fitness values below $F = 5.5$ are shown in black.

The first finding is evidence to a shifted error threshold [51, 52] caused by an increased acceptance of missteps with increased temperature. The second finding is closely linked to the first one and could already be anticipated. The long autocorrelation of the fitness landscape and the fact that just a few seekers suffice to explore the fitness landscape (without trapping in local optima) suggest a trivial optimization problem.



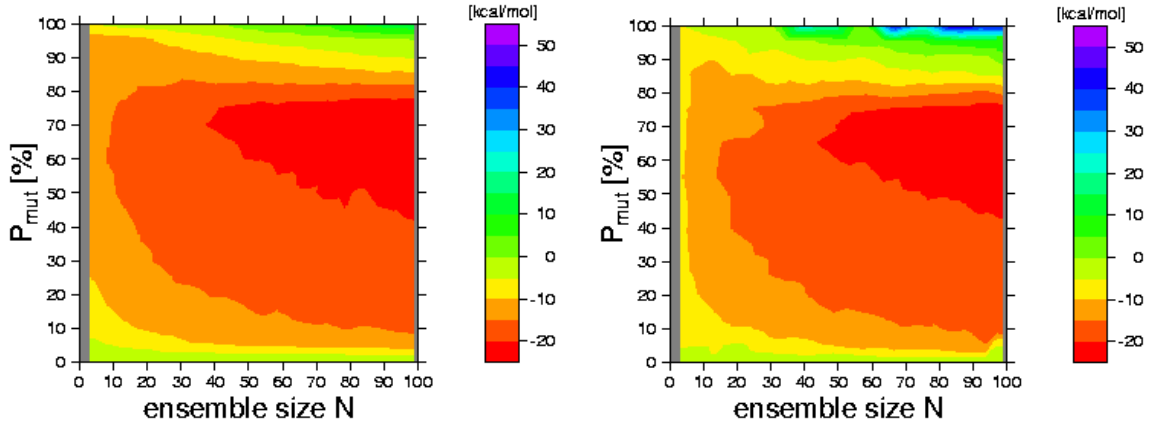
(a) temperature $T = 0$

(b) temperature $T = 2$

(c) temperature $T = 4$

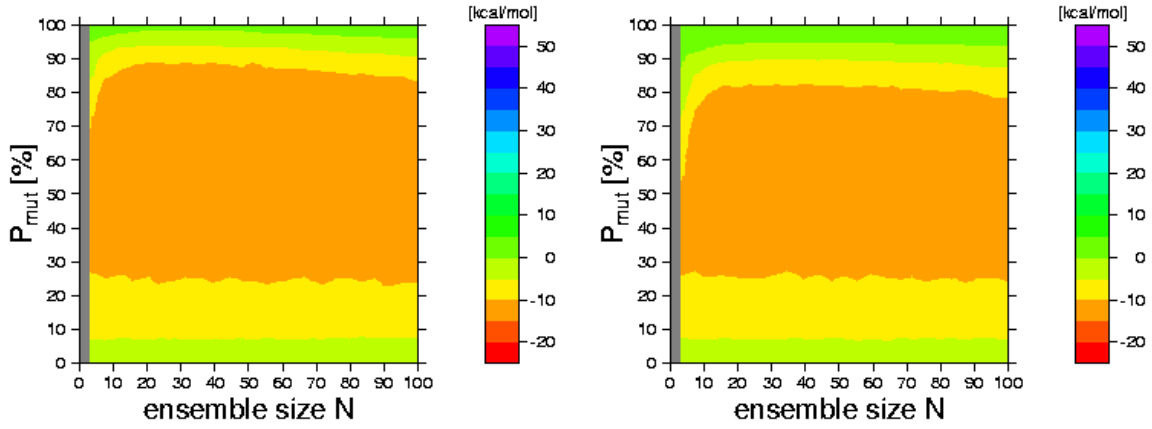
(d) temperature $T = 6$

Figure 4.6: Mean ensemble fitness: Temperature dependence of the evolutionary window displayed for Low Autocorrelation Binary Strings. Sequence length $L = 32$; computation time $t = 500$; averaged over 1000 runs



(a) temperature $T = 0$

(b) temperature $T = 2$



(c) temperature $T = 4$

(d) temperature $T = 6$

Figure 4.7: Mean free energy: Temperature dependence of the evolutionary window displayed for RNA secondary structure optimization. Sequence length $L = 100$; computation time $t = 500$; averaged over 100 runs

Since there is, therefore, no need to accept steps with lower fitness (as higher temperatures permit), the evolutionary window is expected to shrink. The same behavior as seen for Frustrated Periodic Strings can be observed for Low Autocorrelated Binary Strings. The sharply limited evolutionary window at $\{T = 0,\ 5 \leq N \leq 20,\ 68\% \leq P_{mut} \leq 98\%\}$ shrinks and shifts towards lower mutation rates as the temperature increases. Since the LABS problem is non-trivial,

optimal ensemble sizes are about $N \approx 10$ in contrast to the Frustrated Periodic Sequences.

The RNA secondary structure optimization is somewhat special, as can be seen in Figure 4.7. The evolutionary window does not get shifted noticeably with increasing temperature, but rather disappears above a certain threshold. For temperatures $T \geq 4$, the optimization result is almost independent of the ensemble size N.

4.3 Mastering Intrinsic Search Parameters

The mixed evolutionary algorithms introduced, including the pure strategies as special cases, basically have three intrinsic search parameters: the ensemble size N, the temperature T, and the mutation rate P_{mut}. As demonstrated in the section above, all these parameters must be carefully adjusted in order to ensure an efficient optimization process.

A user-friendly algorithm should be enabled to automatically adapt all its intrinsic parameters. Since the optimal parameter window, the evolutionary window, is three-dimensional, three cross-dependent adaptation strategies have to be developed. As a first step one could try to adapt each parameter individually.

4.3.1 Ensemble Size Adaptation

Very few attempts can be found in literature dealing with the adaptation of seeker ensemble sizes. There are also no new contributions developed in the scope of this work. The main obstacle is the difficult analysis involved in modelling evolutionary algorithms. There is basically only one model problem, binary strings or so called Bitstrings[1], that is analytically solvable in the linear case.

[1] A well known implementation of this model is also known as the ONEMAX-Problem.

If one does not want to rely on *ad hoc* assumptions, the problem to fix the ensemble size can be approached by introducing a meta-optimization-algorithm. The idea is to start the search process with differently sized, competing subpopulations [53]. During an evaluation interval, each subpopulation may demonstrate its performance. Afterwards, the different populations are rated and accordingly adapted. This is the so-called migration interval.

The advantage of a meta algorithm (i.e. to have a tool to adjust an intrinsic search parameter) faces a few disadvantages:

1. The meta algorithm unavoidably binds scarcely available computational resources.

2. The meta algorithm introduces a set of additional intrinsic parameters such as the *number of subpopulations*, a *quality criterion* to rate the subpopulations, the length of the *evaluation interval*, the length of the *migration interval*, and a *gain criterion* for the ensemble size adaptation.

4.3.2 Temperature Adaptation

In contrast to ensemble size adaptation, temperature control techniques have been thoroughly investigated [4, 5, 6, 7, 8, 9, 23, 54]. The simplest forms of annealing schedules are fixed functions like linear or exponential cooling. More sophisticated variants are sensitive to the underlying fitness landscape.

A good example of a theoretically motivated annealing schedule (the one introduced by ANDRESEN) is discussed in section 2.1.4 on page 27. In the mentioned schedule, the temperature is controled according to:

$$\frac{\mathrm{d}\,T}{\mathrm{d}\,t} = \frac{v_c\,T}{\varepsilon\,\sqrt{C}}; \quad v_c = \frac{\langle U\rangle - U_{eq}(T)}{\sigma} \tag{4.4}$$

The heat capacity $C(T)$ and the relaxation coefficient ε can be estimated by recording the complete history of the annealing process. The latter is a require-

ment that makes working with ANDRESEN's schedule resource-hungry and the implementation unnecessarily demanding.

It is possible, however, to simplify the procedure and avoid the necessary maintenance of history records. As a first step, one can assume the relaxation coefficient to be constant. For the second step, the heat capacity needs to be substituted by a more easily accessible quantity:
The heat capacity is defined as

$$C = \frac{\partial \langle H \rangle}{\partial T}. \tag{4.5}$$

The expectation value of the Hamilton operator can be expressed as:

$$\langle H \rangle = \frac{\int H e^{-\frac{H}{T}} \mathrm{d}\Gamma}{\int e^{-\frac{H}{T}} \mathrm{d}\Gamma} =: \frac{u}{v}. \tag{4.6}$$

Using the substitutions u and v for numerator and denominator and keeping in mind that

$$u' = \frac{\partial}{\partial T} \int H e^{-\frac{H}{T}} \mathrm{d}\Gamma = \frac{1}{T^2} \int H^2 e^{-\frac{H}{T}} \mathrm{d}\Gamma \tag{4.7}$$

$$v' = \frac{\partial}{\partial T} \int e^{-\frac{H}{T}} \mathrm{d}\Gamma = \frac{1}{T^2} \int H e^{-\frac{H}{T}} \mathrm{d}\Gamma, \tag{4.8}$$

$$\tag{4.9}$$

equation (4.5) can be written as:

$$\frac{\partial \langle H \rangle}{\partial T} = \frac{\int e^{-\frac{H}{T}} \mathrm{d}\Gamma \int H^2 e^{-\frac{H}{T}} \mathrm{d}\Gamma - \left(\int H e^{-\frac{H}{T}} \mathrm{d}\Gamma \right)^2}{T^2 \left(\int e^{-\frac{H}{T}} \mathrm{d}\Gamma \right)^2} \tag{4.10}$$

A simplification of the last equation yields:

$$C = \frac{\partial \langle H \rangle}{\partial T} = \frac{1}{T^2} \left(\langle H^2 \rangle - \langle H \rangle^2 \right) = \frac{1}{T^2} \langle (H - \bar{H})^2 \rangle. \tag{4.11}$$

The last equation states that the heat capacity can be expressed via the variation of the Hamiltonian. Using the relation:

$$\langle (H - \bar{H})^2 \rangle = \sigma_H^2 \tag{4.12}$$

where σ_H denotes the standard deviation of the Hamiltonian, one finally gains a simplified annealing schedule:

$$\boxed{\frac{\mathrm{d}\,T}{\mathrm{d}\,t} = \frac{v_c\,T^2}{\sigma_H}.} \tag{4.13}$$

This schedule was successfully used in numerical simulations [55]. A closely related schedule, the so-called Standard Deviation Schedule (SDS), was proposed by MAHNIG and MÜHLENBEIN [10] and also successfully implemented in the context of this work [22]. The SDS controls the temperature according to:

$$\frac{\mathrm{d}\,\beta}{\mathrm{d}\,t} = \frac{v_c}{\sigma_F}. \tag{4.14}$$

4.3.3 Mutation Rate Adaptation

Thinking about the role of the mutation rate, a few ideas immediately come to mind. Since the evolutionary algorithms introduced basically implement selection and mutation processes only[1], it is clearly the mutation driving the optimization process. Selection, on the other hand, operates on already existing solutions only. Introducing an evolution rate $R(t)$ as the average change of the ensemble fitness [10]:

$$R(t) = \frac{\mathrm{d}\,\langle F\rangle}{\mathrm{d}\,t} \tag{4.15}$$

or

$$R(t) = \langle F(t+1)\rangle - \langle F(t)\rangle \tag{4.16}$$

respectively, from the master equation (2.22) follows:

$$R(t) = \gamma\left(\langle F^2\rangle - \langle F\rangle^2\right) + m\sum_{ij} A_{ij}\,(\Delta F)\,y_i. \tag{4.17}$$

[1] In contrast to typical Genetic Algorithms there is no crossover operator involved here.

In the special case of absent mutation $(m = 0 \rightarrow \gamma = 1)$, the last equation reads:

$$R(t) = \sigma_F^2 \geq 0. \tag{4.18}$$

At least gained optimization results are not lost. New solutions are found only by chance due to a widespread ensemble. Therefore, one can conclude that the mutation rate should be as high as possible ($P_{mut} \rightarrow 100\%$) in order to analyze the search space at a quick pace. On the other hand, however, this cannot be the whole truth. As all numerical simulations show (cf. section 4.2.2), the evolutionary window ends well below $P_{mut} = 100\%$. The idea, borrowed from nature, to introduce selection steps is an important part of mixed strategies to ensure an efficient search process by dropping inefficient seekers.

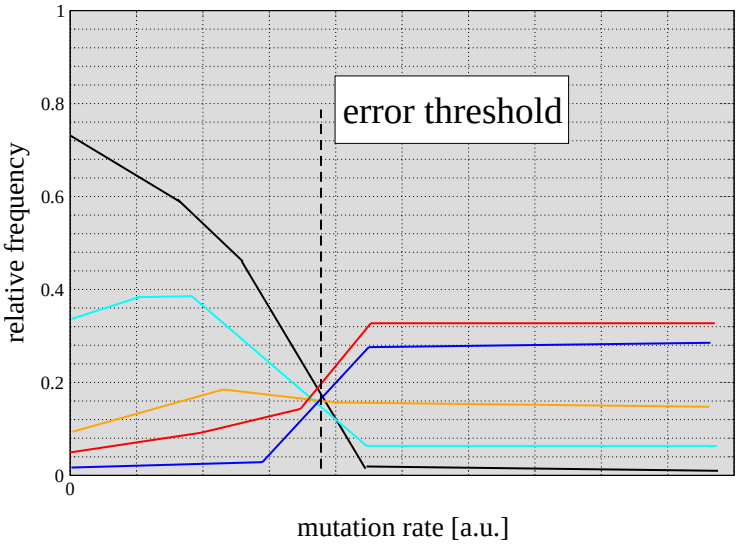


Figure 4.8: Beyond the error threshold, the different fitness values are distributed randomly and independently of the mutation rate. The figure sketches the phase transition as observed in numerical experiments.

A detailed analysis reveals that, raising the mutation rate, the transition from an efficient to an inefficient search happens quickly at a certain threshold. This transition, known as the *error threshold* [51, 52], marks the critical mutation rate, beyond which solutions obtained by evolutionary processes are destroyed

more frequently than selection can reproduce them. Many attempts have been undertaken to analytically predict this threshold [56, 57]. Most trials came up with empirical data and collected evidence that the error threshold and optimal mutation rates are indeed correlated. Only for Genetic Algorithms was it possible to find an analytic expression for a restricted number of fitness landscapes.[1] For infinite and asexually reproducing populations, the critical value was found to be [58, 59]:

$$P_{mut}^{crit} = \frac{\ln(\sigma)}{\xi}. \tag{4.19}$$

The value ξ here denotes the chromosome length used to encode the problem. A series expansion allows an approximate prediction for finite size N ensembles:

$$P_{mut}^{crit}(N) = \frac{\ln(\sigma)}{\xi} - \frac{2\sqrt{\sigma-1}}{\xi\sqrt{N}} + \frac{2\ln\sigma\sqrt{\sigma-1}}{\xi^2\sqrt{N}}\ldots \tag{4.20}$$

The estimators given by eq. (4.19) and eq. (4.20) were, as mentioned above, derived for Genetic Algorithms and asexual reproduction. Taking sexual reproduction into account, the critical threshold is typically lower [58].

Since the results of NOWAK, SCHUSTER, OCHOA, et. al. cannot be simply transferred to be used for evolutionary algorithms, this work proposes a hands-on method. As sketched in Figure 4.8, the critical mutation rate is imprinted in the ensemble's fitness distribution. It should therefore be possible to somehow numerically detect the onset of the phase transition. To this end, an easily accessible sensor is necessary. More concrete, the sensor has to fulfil the following requirements:

1. It needs to be sensitive for the error threshold.
2. For efficiency reasons, it must be numerically easy to aquire.
3. Ideally, it has to be ensemble size and temperature independent.

[1] Namely the bitstring model, the Royal Road -, and the Royal Staircase fitness function were investigated.

4. Preferably, the sensor can be applied to any optimization problem without change.

One can think of uncountable variants of statistical measures, including linear and non-linear terms, all of which have to be tested against the needs stated above. A few investigated examples will be introduced and compared in the following subsections.

First Approach: The Ensemble Variability

In case of absent selection, the chance that all seekers of the ensemble are different is very high. In case of absent mutation, on the other hand, the ensemble quickly focuses so that nearly all seekers are identical. As a first attempt, one might therefore define a numerical sensor, the ensemble variability v, as the number of different seekers N_{diff} normalized by the ensemble size N:

$$v = \frac{N_{diff}}{N} \tag{4.21}$$

Since fitness values can be degenerated, the variability is actually twofold: It is possible to define the variability with respect to either phenotype (v_{fit}: two seekers are counted identical if they have the same fitness) or genotype (v_{gen}: two seekers are counted identical only if they represent the same point in the fitness landscape, even though they might have the same fitness). In a highly degenerated landscape (plateau structure) the latter has a significantly higher sensitivity [21]. As numerical experiments confirm, the ensemble variability fulfils at least the first two requirements: it is sensitive towards the error threshold [55] as seen in Figure 4.9, and it is easy to calculate.

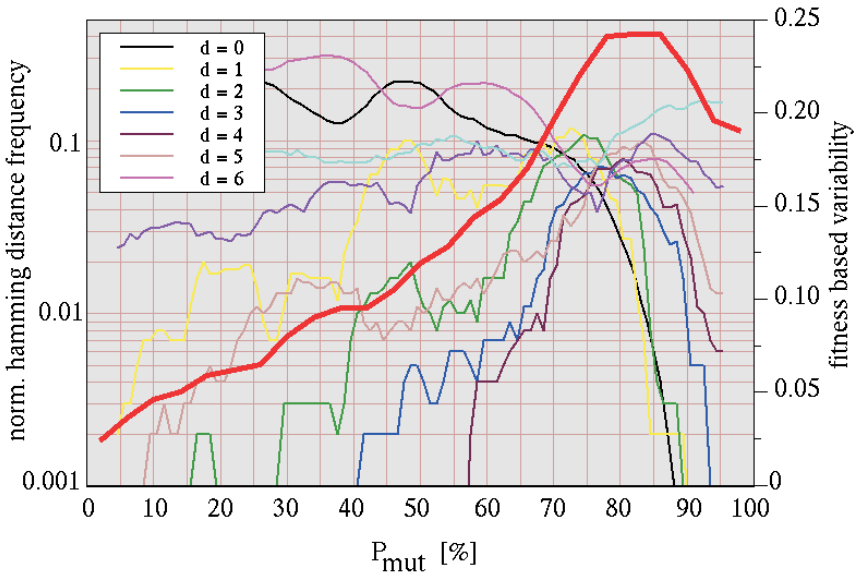


Figure 4.9: Mixed Evolutionary Algorithm; 4 seeker tournament selection; Frustrated Periodic Sequence Model; length $L = 15$, ensemble size $N = 20$, temperature $T = 1$, periodicity bonus $b = 1$, time $t = 10^4$ – The solid red line marks the fitness based ensemble variability which nicely redraws the phase transition at $P_{mut} \approx 75\%$.

Even though it is possible to design successful adaptation techniques using this sensor [55] this approach has a couple of drawbacks that must not be overlooked. The range of possible values v is restricted to: $1/N \leq v \leq 1$. This introduces a strong bias for small ensembles $N \leq 10$.

Measuring the variability for *optimal* mutation rates, one observes a standard-deviation-like dependence with respect to the ensemble size:

$$v_{opt} \simeq \frac{1}{\sqrt{N}}. \tag{4.22}$$

The problems discussed are a strong motivation to look out for a better alternative showing less parameter dependencies while being just as sensitive. In the following step a more sophisticated sensor based on ensemble statistics will be introduced.

Second Approach: The Relative Ensemble Dispersion

Instead of counting different seekers to get a notion about about the ensemble distribution, one can also refer to off-the-shelf tools from statistics. It is a very simple and straightforward way to calculate the ensemble's mean fitness $\langle F \rangle$ and standard deviation σ_F. Combining both terms yields the relative ensemble dispersion:

$$d_{rel} := \frac{\langle F \rangle}{\sigma_F} \tag{4.23}$$

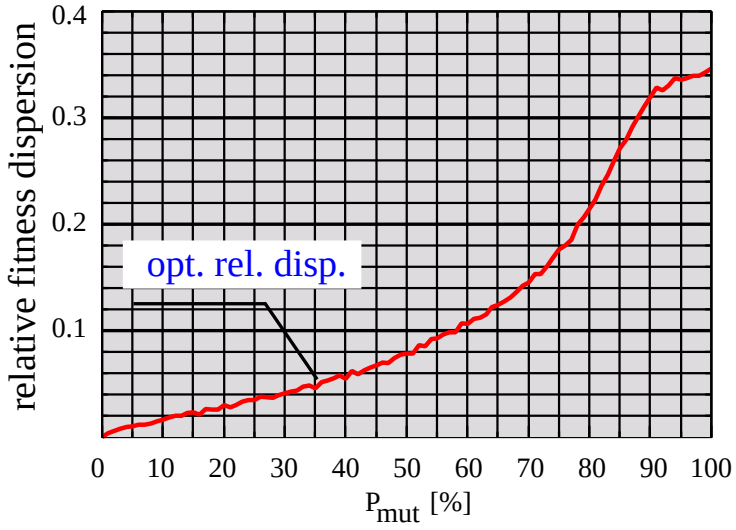
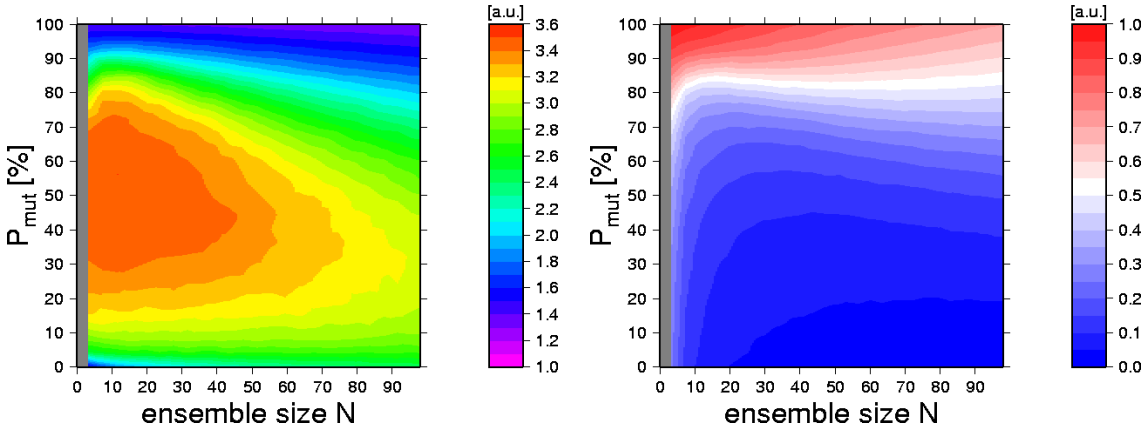


Figure 4.10: LABS problem of length $L = 32$: relative dispersion in dependence of the mutation rate m; simulation time $t = 500$; temperature $T = 1$; 4 seeker tournament selection; averaged over 1000 runs

Regardless of the mean fitness, the standard deviation can take any value including zero. This implies that the relative ensemble dispersion as defined above is not normalized.

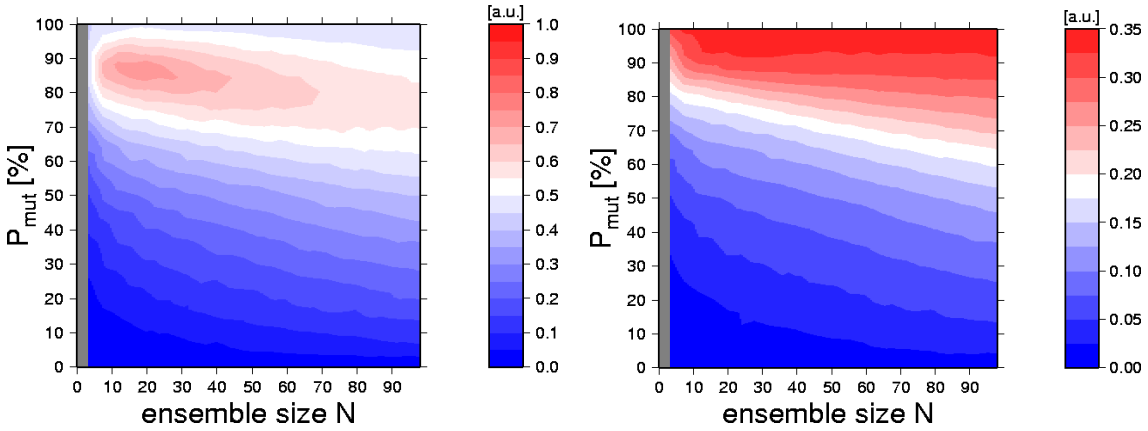
Just as the ensemble variability, the dispersion sensitively reflects the mutation rate's influence as displayed in Figure 4.10. It is a suitable numerical sensor since it is able to detect the areas of different optimization quality. As can be seen in Figure 4.11, the latter is ensemble-size-independent, making it a better sensor

than the ensemble variability. The figure also clearly shows that the standard deviation by itself cannot uniquely relate mutation rate and resulting fitness.



(a) mean fitness $\langle F \rangle$

(b) ensemble variability v



(c) ensemble dispersion d

(d) rel. ensemble dispersion d_{rel}

Figure 4.11: LABS problem of length $L = 32$; temperature $T = 1$; time $t = 500$; averaged over 1000 runs – Comparison of different numerical sensors: Subfigure (b) clearly shows the ensemble size dependence of the ensemble variability. The ensemble dispersion by itself ambiguously relates mutation rate and dispersion, as shown in subfigure (c). The relative ensemble dispersion (subfigure (d)) eliminates the ensemble-size-dependence while being sensitive towards areas of different fitness, as seen in subfigure (a).

While the relative dispersion surpasses the ensemble variability measured in terms of the necessities for a sensor formulated on page 63, it also shows weak spots [60]. The results of numerical simulations listed in Table 4.1 indicate that for temperatures $T > 0$, the temperature dependence could, at first glance, be neglected. It also shows, however, that the idea of an optimal relative fitness dispersion is crucially dependent on the optimization problem.

Temp.	Frustr. Period. Sequ.	LABS Problem	RNA second. struct.
0	< 0.001	< 0.005	< 0.050
1	0.020 ± 0.010	0.06 ± 0.02	0.15 ± 0.10
2	0.025 ± 0.010	0.05 ± 0.02	0.15 ± 0.10
4	0.035 ± 0.010	0.04 ± 0.02	0.35 ± 0.10
6	0.030 ± 0.010	0.04 ± 0.02	0.30 ± 0.10
8	0.025 ± 0.010	0.04 ± 0.02	0.36 ± 0.10

Table 4.1: Optimal relative fitness dispersion for different model problems at different temperatures. Tolerance values are due to averaging and graphical evaluation. Frustrated Periodic Sequence length: $L = 15$ and periodicity bonus $b = 0.2$; LABS length: $L = 32$; RNA sequence length: $L = 100$.

In a third approach, a nonlinear numerical sensor will be introduced that does not have any of the shortcomings seen before, but still provides all of the benefits. It is the only numerical estimator found in context of this work that satisfies all four demands formulated above.

Third Approach: The Ensemble Entropy

While the relative ensemble dispersion is already quite useful it nevertheless remains a linear measure and shows its limitations comparing different test models.

This last approach to design a numerical sensor borrows ideas from information theory. The crucial point is that an evenly scattered ensemble (high dispersion) represents the least amount of knowledge regarding its whereabouts, while an ensemble focused in a single point (highly ordered state), on the other

hand, represents a maximum amount of knowledge. The information-theoretical measure for (missing) knowledge is the so-called entropy:

$$H = -\sum_i P_i \ln P_i \tag{4.24}$$

So there already exists a non-linear measure to express the ensemble distribution (as explained in the previous approach) in a different way. It only needs to be translated to suit the needs. The occupation probabilities P_i will be substituted by relative occupation numbers. The latter can be easily obtained by generating an ensemble histogram at any given time. Since, in the beginning of the search process, there is nothing known about the respective fitness landscapes, it does not make much sense to operate with predefined bins generating the histogram. Instead, the (likely unequally spaced) bins are generated dynamically using the fitness values the respective seekers have assumed at any given moment.
The normalized ensemble entropy can thus be defined as [22]:

$$\hat{H}_{ens} = \sum_{i=1}^{N} \frac{x_i}{N} \log_N \frac{x_i}{N}. \tag{4.25}$$

Figure 4.12 demonstrates the sensitivity of this new sensor as an example of Frustrated Periodic Sequences. The highest gradient is just where the evolutionary window happens to be (in terms of the mutation rate) providing a very high sensitivity as demanded (cf. Figure 4.13). It is interesting to note that the ensemble entropy, like the ensemble dispersion, has ambiguous parameter intervals where a functional relation between entropy and mutation rate is missing. The problematic interval is beyond the error threshold as displayed for RNA sequences in Figure 4.14.

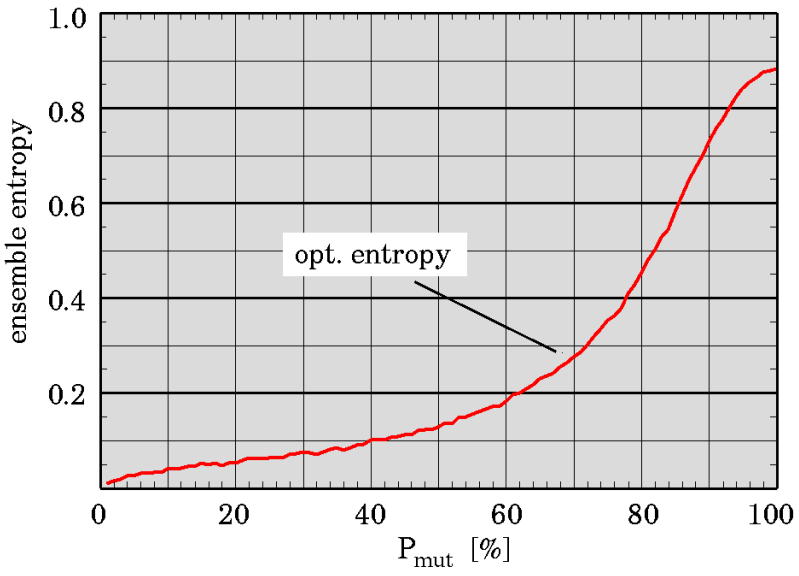
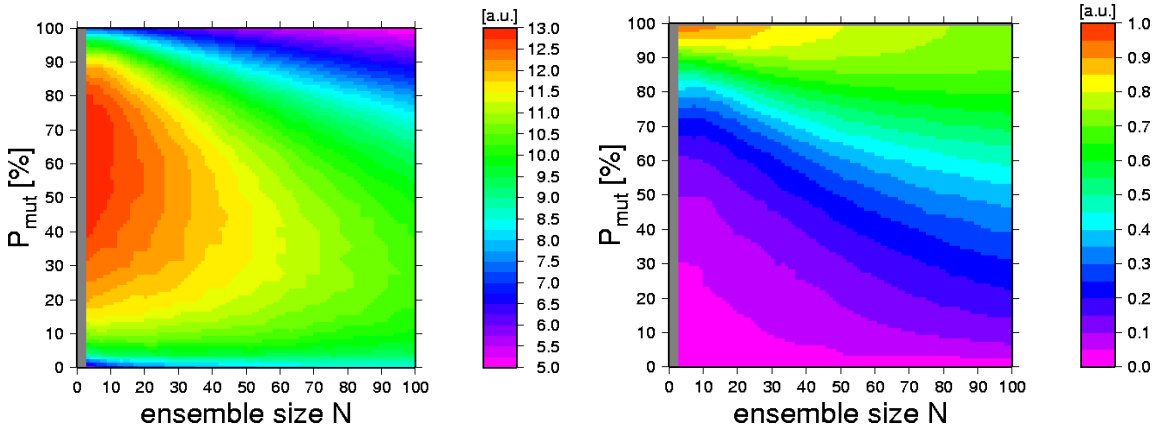


Figure 4.12: Frustrated Periodic Sequence length $L = 15$; ensemble entropy in dependence of the mutation rate m; simulation time $t = 500$, temperature $T = 1$; ensemble size $N = 20$; averaged over 1000 runs.

On the positive side, the optimal ensemble entropy denoting the evolutionary window is dependent neither on temperature nor on the optimization problem:

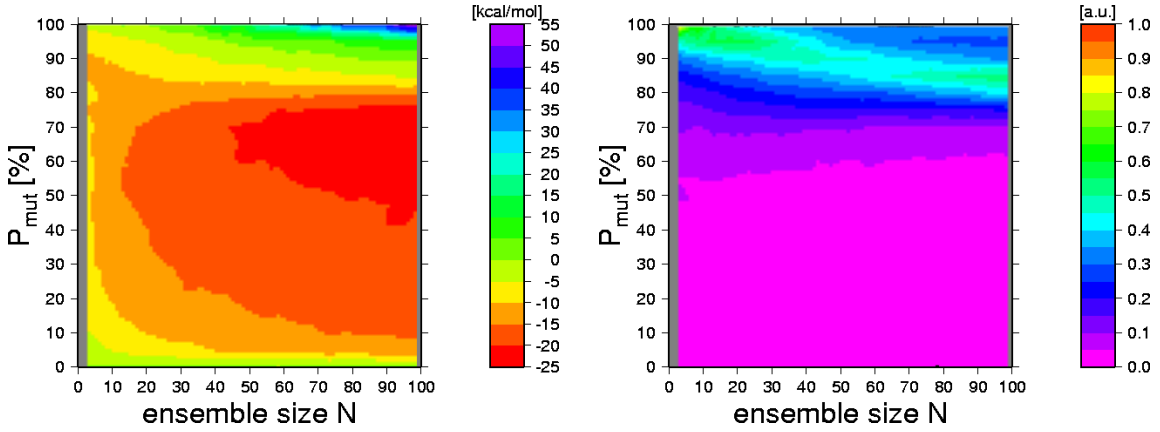
Temp.	Frustr. Period. Sequ.	LABS Problem	RNA second. struct.
0	0.15 ± 0.05	0.12 ± 0.05	0.12 ± 0.05
1	0.20 ± 0.05	0.10 ± 0.05	0.12 ± 0.05
2	0.14 ± 0.05	0.13 ± 0.05	0.15 ± 0.05
3	0.14 ± 0.05	0.10 ± 0.05	0.12 ± 0.05
$\vdots$	$\vdots$	$\vdots$	$\vdots$
10	0.15 ± 0.05	0.12 ± 0.05	0.16 ± 0.05

Table 4.2: Optimal ensemble entropy $\hat{H}_{ens}^{opt}$ for different model problems at different temperatures. Tolerance values are due to averaging and graphical evaluation. Frustrated Periodic Sequence length: $L = 15$ and periodicity bonus $b = 0.2$; LABS length: $L = 32$; RNA sequence length: $L = 100$.



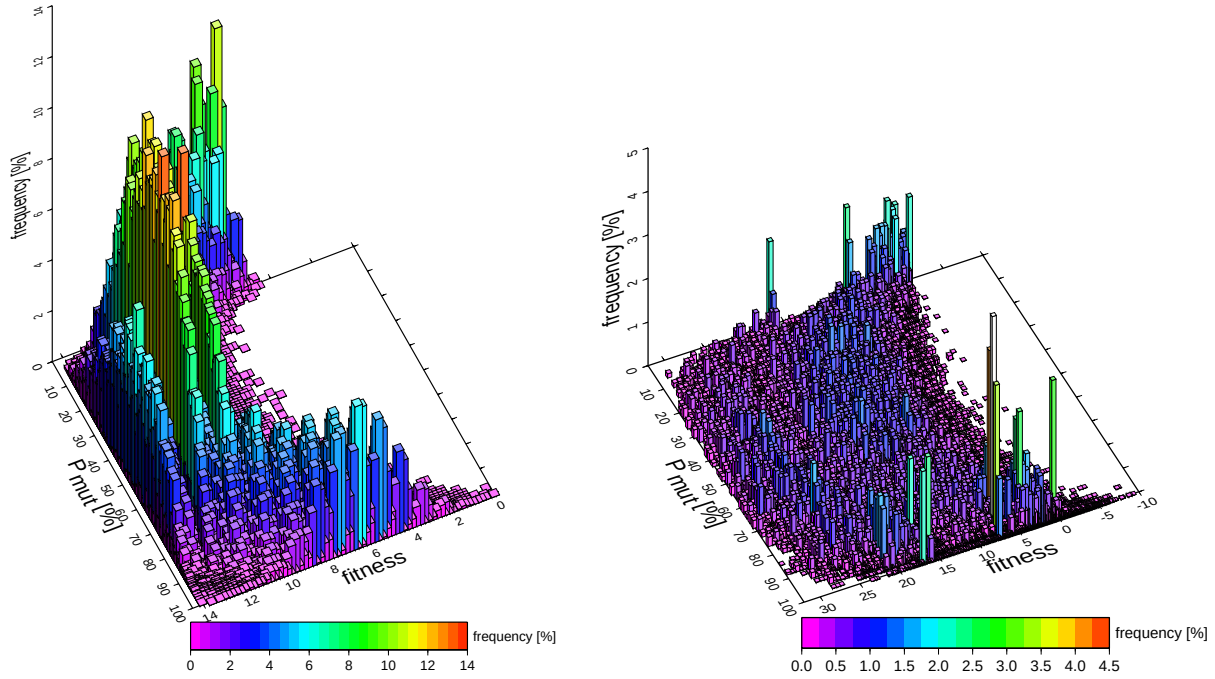
(a) mean fitness $\langle F \rangle$ (b) ensemble entropy $\hat{H}_{ens}$

Figure 4.13: Frustrated Periodic Sequence of length $L = 15$; periodicity bonus $b = 0.2$ – The entropy measure nicely redraws the areas of different fitness values independent of the ensemble size and may thus serve as a numerical sensor. The temperature was kept constant at $T = 1$; random initial sequences were used; the simulation time was $t = 500$; the results were averaged over 1000 runs. The best fitness values are obtained for an entropy around 0.20.



(a) mean free energy $\langle F \rangle$ (b) ensemble entropy $\hat{H}_{ens}$

Figure 4.14: RNA sequence of length $L = 100$; random initial conditions; temperature kept constant at $T = 1$; simulation time $t = 500$; the results are averaged over 1000 runs – The best fitness values are obtained for an entropy around $0.15 \ldots 0.40$



(a) Frustrated Periodic Sequence $L = 15; b = 0.2$

(b) RNA sequence $L = 100$

Figure 4.15: Ensemble histograms for two model problems used to calculate the entropy $\hat{H}_{ens}$. In order to be comparable, the mean free energy $\langle U \rangle$ was used to define a fitness as $F = -\langle U \rangle$ in subfigure (b).

It is enlightening to have a look at the ensemble histograms actually used to calculate the ensemble entropy. For Frustrated Periodic Sequences (Figure 4.15(a)) the error threshold is immediately visible. Beyond $P_{mut} \approx 55\%$ the ensemble distribution rapidly spreads out and loses focus.

The situation is very different considering the secondary structures of RNA sequences (Figure 4.15(b)). It is hardly possible to visualize some sort of threshold. It is even more amazing that the numerical procedure determining an optimal ensemble entropy still points towards the evolutionary window.

Summarizing, the ensemble entropy is well-suited to serve as a sensor for the evolutionary window. It is a sensitive and easily calculated measure, and it is not only independent of other intrinsic search parameters, but also independent of the optimization problem investigated. In the next chapter, an auto-adaptive evolutionary algorithm based on the entropy sensor will be introduced.

4.4 An Adaptive Evolutionary Algorithm

The material gathered in the last sections enables the construction of an adaptive evolutionary algorithm able to control its intrinsic search parameters with the exception of the ensemble size N. (The difficulties regarding ensemble sizing were discussed in section 4.3.1 on page 58.)

It seems reasonable to start out with randomly distributed seekers. The temperature should be set infinitely high ($\beta \to 0$), thus allowing all mutation steps regardless of their benefits. Also, the mutation rate should be set to its maximum (i.e. $m = 1$). These initial settings allow maximal flexibility and prevent a premature ensemble convergence in fitness space.

It is also intuitively clear that an adaptation towards a fixed mutation / selection ratio cannot be optimal for all given simulation times. For clearly insufficient computation time, for example, the best strategy is to guess solutions.

That corresponds to a setting with $m(t) \equiv 1$. It can be shown, however, that these concerns are negligible for a wide range of granted computation times [22].

Starting from the initial settings, the ensemble statistics quickly yields enough information to turn on adaptation for mutation rate and temperature, as introduced above. The complete recipe now looks like this:

Adaptive Evolutionary Algorithm

1. Start optimization with high temperature and disabled selection.
2. Beginning shortly thereafter, increase and control the mutation rate to keep the ensemble entropy at the optimum $\hat{H}_{ens}^{opt}$.
3. Follow the annealing schedule to adapt the temperature parameter.

The steps 2 and 3 can be carried out simultaneously [22]. The results, that can be achieved using the adaptation above, are absolutely comparable to those obtained by manually adjusting the intrinsic search parameters towards the evolutionary window. An example is shown in Figure 4.16 using the RNA sequence model. The fact that the best solution found in a single run ($F = 8$) is much better than the ensemble average ($F = 3.6 \pm 0.5$) indicates that the provided computation time for this optimization was not yet sufficient by far. Nevertheless, the adaptation was successful since even exhaustive parameter scans (manual parameter settings) could not achieve significantly better results.

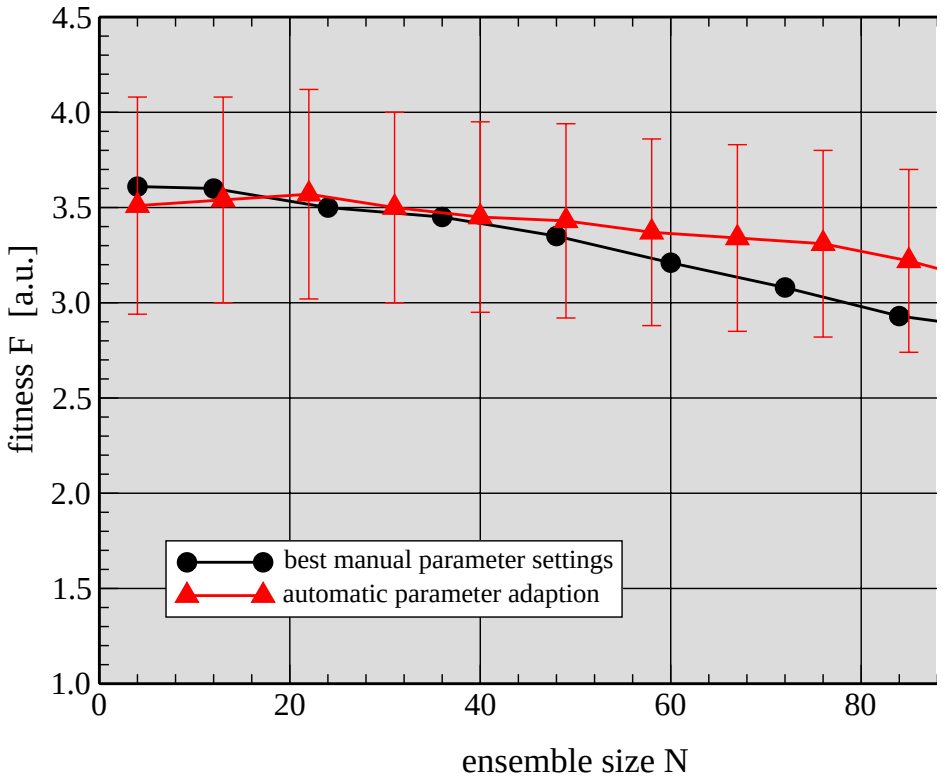


Figure 4.16: Expectation value for the ensemble's best seeker $\langle F_{max} \rangle$; LABS problem of length $L = 32$; comparison between exhaustive parameter scan and automatic parameter adaptation with initial conditions $m = 1, T = 10^3$, computation time $t = 500$, averaged over 1000 runs. The absolutely best solution found in the simulation is the string $S =$ 01010100000111111011011001110011 with fitness $F = 8$.

Chapter 5

Software

5.1 Newly Developed Software

5.1.1 Optimization Programs

Substantial effort has been invested in the development of a new optimization program suite. This suite namely consists of the the twin programs `SimLabs`, `SimEngel`, and `SimRNA` designed by the author to apply different evolutionary algorithms to the *LABS* problem (cf. section 3.2), the Frustrated Periodic Sequence problems (cf. section 3.3), and the RNA secondary structure optimization (cf. section 3.4).

These programs are written in `C++`, as opposed to, for example `C` or `Fortran` for the following reasons:

- **abstraction:** `C++` allows the definition of abstract data types, thus greatly reducing source code size and error proneness of the programs [61, 62, 63].
- **compiler availability:** Almost any computer platform offers highly developed `C++` compilers with sophisticated optimization routines.
- **flexibility:** The object-oriented and modular approach makes it easy to maintain and extend the program.

The User Interface

The developed optimization programs feature a complete command line interface as well as a graphical user interface (GUI). The command line interface offers a short description of all parameters if it encounters the option `-help`. If the option `-nox` is given all output is directed to *stdout* and *stderr* exclusively. The program then runs as a single thread. If the option `-nox` is missing the command line is parsed first, so the GUI comes up with its default values adjusted to the given command line parameters. When the 'Start' button is hit, the program spawns a new thread for the calculations, which is separated from the GUI thread, making it easy to update the GUI in parallel to the calculations. While the calculation is running the user is informed about the progress via the progress bar; all interactions regarding parameter changes are inhibited. Figure 5.1 shows the user interface for `SimRNA` program.

The Workflow

All three developed optimization programs share an identical workflow template as sketched in Figure 5.2. All problem specific details (seeker layout, mutation operator implementation etc.) are encapsulated in a separate seeker class.

Starting with an initialization sequence, the program enters a loop structure working through the requested number of repetitions, the externally set mutation rates, and ensemble sizes – and finally enters an inner cycle. The inner cycle represents the actual optimization process starting at time $t_0 = 0$ and running until the final time is reached. Within this time interval, only either mutation or selection steps are executed at a time (depending on the set mutation rate and the chosen optimization strategy), and necessary statistical calculations are carried out as explained in section 4.3.

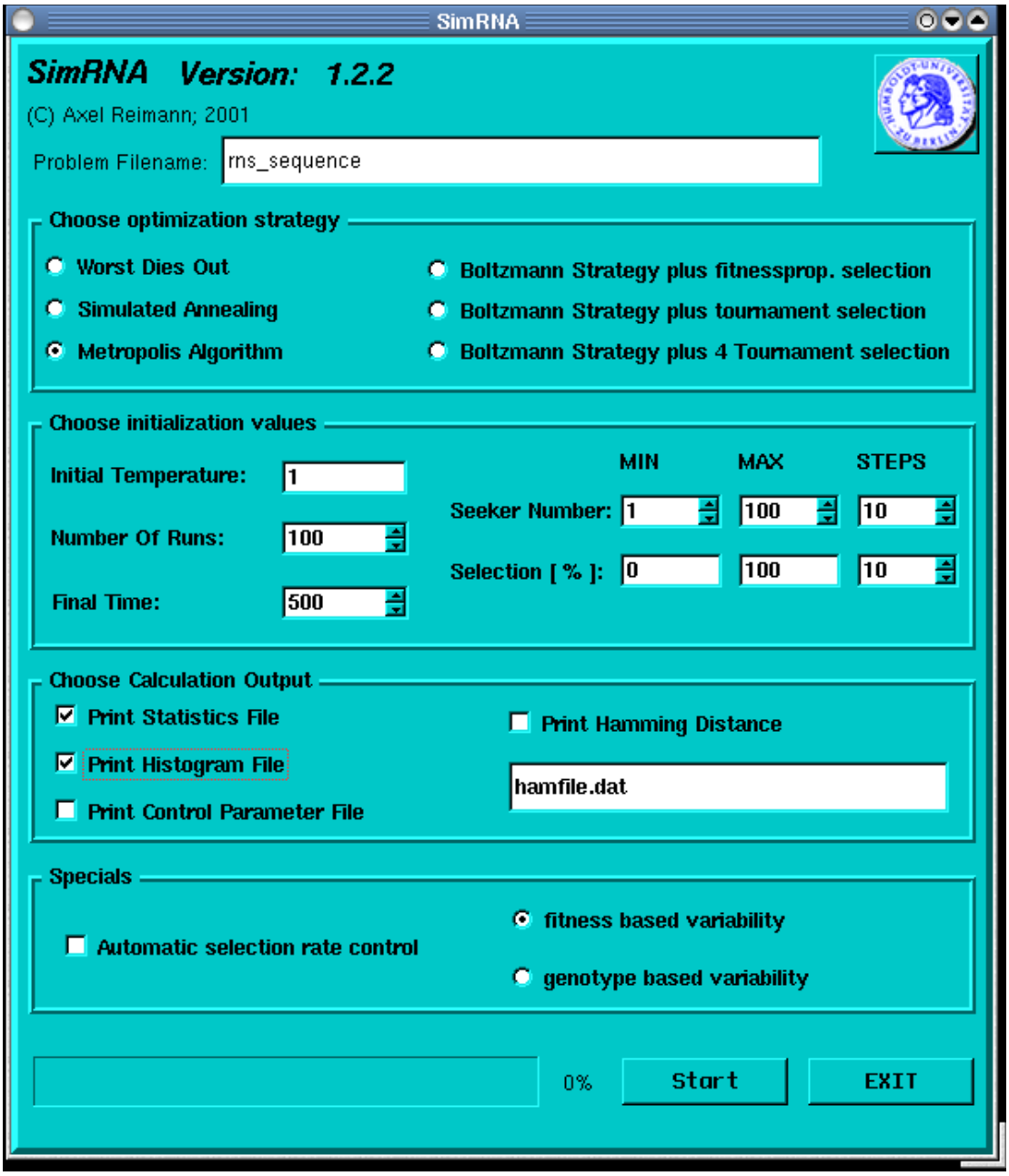


Figure 5.1: Graphical User Interface (GUI) of the `SimRNA` optimization program. The GUI uses the graphical routines of the Qt Toolkit (cf. sec. 5.2.4). The user interface became necessary, when the number of command line parameters grew too large. It allows strategy selection, the setting of all parameters as well as the number of repetitions, and enabling the ensemble statistics of interest.

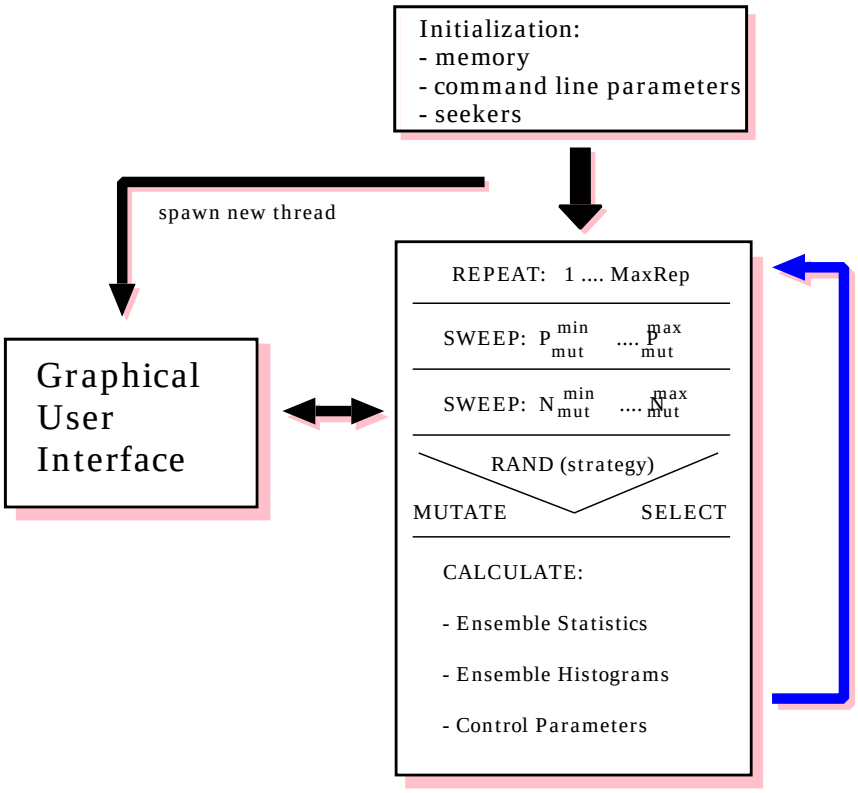


Figure 5.2: Workflow template which is common to all optimization programs developed in this work. The implementation in C++ allows to encapsulate **all** problem specific details in a separate seeker class. This guarantees that the software is easily adaptated to different optimization problems.

5.1.2 The SimRNA Mutation Operator

For almost all problems investigated here, the implemented mutation operator had a rather simple structure. The exception to the rule is the mutation operator designed for RNA secondary structure optimization. In this special case, an efficient operator has to fulfil the following minimal requirements:

1. Carry out only permitted bindings that yield valid pairs.
2. Avoid bindings that generate pseudo-loops (cf. section 3.4.2, p. 38).
3. Avoid search operations to find free binding locations.

The latter is a requirement ensuring that the necessary computation time does not grow order $\mathcal{O}(\log L)$ with the RNA strand length, but is ideally of order $\mathcal{O}(1)$ instead. The implementation of detailed house-keeping of free complementary spots, separatly done for each base $\{A,C,G,U\}$ via lookup-tables (bind operation) and reverse lookup-tables (resolve operation), has lead to a mutation operator meeting all of the requirements above. Figure 5.3 shows the final layout used in the numerical simulations.

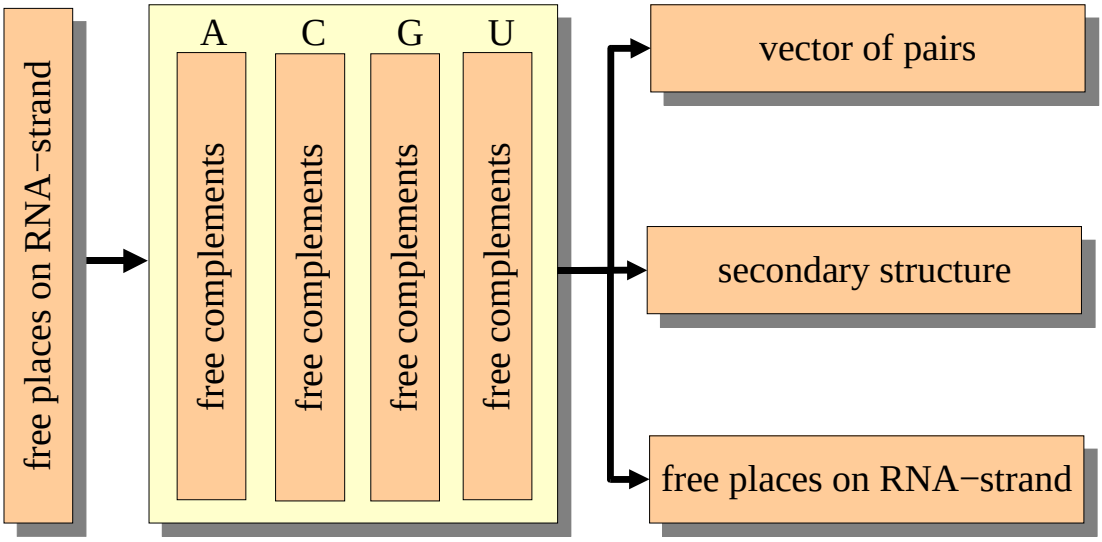


Figure 5.3: Implementation scheme of the SimRNA mutation operator. A pair-connect operator first looks up a free (unconnected) position on the RNA-strand, and then a free complementary position is looked up. If the connection of both positions does not result in a pseudo-loop, the pair-table, the complement tables, and the table of free (unconnected) strand positions are updated. The resulting secondary structure is stored in the corresponding vector. Disconnect-operations, in reverse, work on the complementary lookup-tables not shown in this figure.

The main components are the following vectors:

free_places This vector contains only unbound positions of the RNA-strand.

a,c,g,u - complements These four vectors contain the complementary bases for each base respectively. At position $pos1$, the position $pos2$ of a pair $(pos1,\ pos2)$ is stored.

pairs This vector is used to keep track of bound pair positions. It has the length of the RNA strand.

structure This vector contains the current secondary structure in bracket notation.

As mentioned above, for all these containers reverse lookup-tables had to be implemented in order to support fast resolve operations.

5.1.3 The SimRNA Source Code

This subsection does not list the complete source-code of the SimRNA program, but rather the small fraction of modules necessary to trace the steps of the various implemented evolutionary algorithms.

The RNA-Strand Class

The RNA-strand class encapsulates the problem-specific parts of the algorithms. It is defined (in the header file) as follows:

```
/* class definitions written for rns sequence
   simulations,
   cf. sim_rns.cc
   Axel Reimann (2001)

   Version: 0.1
*/

#ifndef _rns_string_h
#define _rns_string_h

#include <cstdlib>
#include <stdlib.h>
#include <ctype.h>
```

```
#include <iostream.h>
#include <cstring>
#include <string>
#include <vector>
#include "fold_vars.h"
#include "fold.h"

// energy evaluation using ViennaRNA package:

class rns_string {
  friend
    int hamming(const rns_string&, const rns_string&);
 public:
  // constructors & destructor
  rns_string(const std::string&, const std::string&);
  rns_string();
  ~rns_string();
  // member functions
  const char*       content();
  const char*       folding();
  int               mutate();
  void              evaluate();
  rns_string& operator=(const rns_string&);
  // elements
  unsigned int      length;
  double            value;
 private:
  int               a_index, c_index, g_index, u_index;
  __inline__ void   set_content(char*);
  __inline__ void   set_structure(char*);
  int               bind(void);
  int               dissolve(void);
  int               zip(int);
  int               unzip(void);
  void              connect(int, int);
  void              disconnect(int, int);
  int               try_pairing(int, int);
```

```
52    int                 check_pseudo_knots(int, int);
      std::string         string;
54    std::string         structure;
      std::vector<int> a_complements;
56    std::vector<int> c_complements;
      std::vector<int> g_complements;
58    std::vector<int> u_complements;
      std::vector<int> a_lookup;
60    std::vector<int> c_lookup;
      std::vector<int> g_lookup;
62    std::vector<int> u_lookup;
      std::vector<int> free_places;
64    std::vector<int> free_lookup;
      std::vector<int> pairs;
66    std::vector<int> pairs_lookup;
      std::vector<int> pairs_lookup_lookup;
68  };

70  #endif
```

The functionality of this class, as well as various contructors and the destructor, is encoded in its body:

```
#include "compare.h"
#include "rns_string.h"

#define DUMMY_VALUE 0

rns_string::~rns_string() {
}

rns_string::rns_string() {
  string     = "";
  structure  = "";
  value      = DUMMY_VALUE;
  a_complements.clear();
  c_complements.clear();
  g_complements.clear();
  u_complements.clear();
  a_lookup.clear();
  c_lookup.clear();
  g_lookup.clear();
  u_lookup.clear();
  a_index=0;
  c_index=0;
  g_index=0;
  u_index=0;
  free_places.clear();
  free_lookup.clear();
  pairs.clear();
  pairs_lookup.clear();
  pairs_lookup_lookup.clear();
};

rns_string::rns_string(const std::string& s,
                       const std::string& bindings)
{
```

```
  unsigned int i;
36
  if(s.length()!=bindings.length())
38  return;
  length     = s.length();
40  value      = DUMMY_VALUE;
  a_index    = 0;
42  c_index    = 0;
  g_index    = 0;
44  u_index    = 0;
  pairs.clear();
46  pairs_lookup.clear();
  // initialize structure and pairs
48  for(i=0; i<length; i++){
    string+=toupper(s[i]);
50  structure+=bindings[i];
    free_places.push_back(i);
52  pairs_lookup_lookup.push_back(-1);
    pairs.push_back(-1);
54  // initialize complement tables and lookup tables
    free_lookup.push_back(i);
56  }
  for(i=0; i<2*length;i++){
58  a_lookup.push_back(-1);
    c_lookup.push_back(-1);
60  g_lookup.push_back(-1);
    u_lookup.push_back(-1);
62  }
  for(i=0; i<length; i++){
64  switch(string[i]) {
    case 'A':
66    u_complements.push_back((int) i);
      u_lookup[(int) i] = u_complements.size()-1;
68    break;
    case 'C':
70    g_complements.push_back((int) i);
      g_lookup[(int) i] = g_complements.size()-1;
```

```
72          break;
        case 'G':
74          c_complements.push_back((int) i);
            u_complements.push_back((int) i);
76          c_lookup[(int) i] = c_complements.size()-1;
            u_lookup[(int) i] = u_complements.size()-1;
78          break;
        case 'U':
80          a_complements.push_back((int) i);
            g_complements.push_back((int) i);
82          a_lookup[(int) i] = a_complements.size()-1;
            g_lookup[(int) i] = g_complements.size()-1;
84          break;
        default:
86          cerr << "Unknown nucleotide in RNA sequence!\n";
            exit(1);
88      }
    }
90  }

92  const char* rns_string::content() {
    return(string.c_str());
94  }

96  const char* rns_string::folding() {
    return(structure.c_str());
98  }

100 void rns_string::set_content(char* Str) {
    this->string=Str;
102 }

104 void rns_string::set_structure(char* Str){
    this->string=Str;
106 }

108 rns_string& rns_string::operator=(const rns_string& Str) {
```

```
  if(this==&Str)
110   return *this;
  length          = Str.length;
112 value           = Str.value;
  free_places     = Str.free_places;
114 string          = Str.string;
  structure       = Str.structure;
116 pairs           = Str.pairs;
  pairs_lookup    = Str.pairs_lookup;
118 pairs_lookup_lookup = Str.pairs_lookup_lookup;
  a_complements   = Str.a_complements;
120 c_complements   = Str.c_complements;
  g_complements   = Str.g_complements;
122 u_complements   = Str.u_complements;
  a_lookup        = Str.a_lookup;
124 c_lookup        = Str.c_lookup;
  g_lookup        = Str.g_lookup;
126 u_lookup        = Str.u_lookup;
  free_lookup     = Str.free_lookup;
128 return *this;
}
130
int rns_string::mutate() {
132 enum            {bind,bind2,dissolve,pull_tight,pull_tight2,pull_up};
  int              mutation_operator, return_value=-1;
134
  //  mutation operators:
136 //  - bind           ..........  -> .(......)...
  //  - dissolve       .(.......)..  -> ..........
138 //  - pull tight     .(.......)..  -> .(((...)))..
  //  - pull up        .(((...)))..  -> .(.......)..
140
  // pick mutation operation
142 mutation_operator = (int) (6.0*rand()/(RAND_MAX+1.0));
  switch(mutation_operator){
144   //-----------------------------------------------------------------
    // if mutation operator fails, try complementary operation instead
```

```
146     case bind:
        case bind2:
148       return_value = rns_string::bind();
          if(return_value!=-1)
150         break;
        case dissolve:
152       return_value = rns_string::dissolve();
          if(return_value!=-1)
154         break;
          else
156         return_value = rns_string::bind();
        break;
158     case pull_tight:
        case pull_tight2:
160       return_value = rns_string::bind();
          if(return_value != -1){
162         return_value = rns_string::zip(return_value);
            break;
164       }
          else
166         return_value = rns_string::unzip();
        break;
168     case pull_up:
          return_value = rns_string::unzip();
170       if(return_value!=-1)
            break;
172       else
            return_value = rns_string::zip((int) 0);
174       break;
        default:
176       cerr << "Unknown mutation operator!\nBug in rns_string.cc...\n";
          exit(1);
178     }
        return return_value;
180   }

182   void rns_string::evaluate() {
```

```
  value = (double)
184   energy_of_struct((char *) string.c_str(),
                       (char *) structure.c_str());
186 return;
}
188
int rns_string::bind(void){
190 int              pos1,pos2,free,index,index2,pseudo_knots;

192 // find free places
  free = free_places.size();
194 if(free < 2)
    return -1;
196 // pick first candidate
  index = (int) (1.0*free*rand()/(RAND_MAX+1.0));
198 pos1 = free_places[index];
  // pick second candidate
200 switch(string[pos1]){
  case 'A':
202   free   = a_complements.size();
    if(free > 0) {
204     index2    = (int) (1.0*free*rand()/(RAND_MAX+1.0));
#ifdef PROG_DEBUG
206     if(index2 >= (int) a_complements.size()|| index2 < 0){
        cerr << "DEBUG: a_complements index2 out of bounds!\n";
208     cerr << index2 << endl;
        exit(1);
210     }
#endif
212     pos2      = a_complements[index2];
      if(abs(pos2 - pos1) < 3)
214     return -1;
      pseudo_knots = check_pseudo_knots(pos1,pos2);
216   if(pseudo_knots > 0)
        return -1;
218   a_index   = pos1;
      u_index   = pos2;
```

```
220       }
          else return -1;
222       break;
        case 'C':
          free    = c_complements.size();
          if(free > 0) {
226         index2    = (int) (1.0*free*rand()/(RAND_MAX+1.0));
#ifdef PROG_DEBUG
228         if(index2 >= (int) c_complements.size() || index2 < 0){
              cerr << "DEBUG: c_complements index2 out of bounds!\n";
230           cerr << index2 << endl;
              exit(1);
232         }
#endif
234         pos2      = c_complements[index2];
            if(abs(pos2 - pos1) < 3)
236           return -1;
            pseudo_knots = check_pseudo_knots(pos1,pos2);
238         if(pseudo_knots > 0)
              return -1;
240         c_index   = pos1;
            g_index   = pos2;
242       }
          else return -1;
244       break;
        case 'G':
246       free    = g_complements.size();
          if(free > 0) {
248         index2    = (int) (1.0*free*rand()/(RAND_MAX+1.0));
#ifdef PROG_DEBUG
250         if(index2 >= (int) g_complements.size() || index2 < 0){
              cerr << "DEBUG: g_complements index2 out of bounds!\n";
252           cerr << index2 << endl;
              exit(1);
254         }
#endif
256         pos2      = g_complements[index2];
```

```
      if(abs(pos2 - pos1) < 3)
        return -1;
      pseudo_knots = check_pseudo_knots(pos1,pos2);
      if(pseudo_knots > 0)
        return -1;
      g_index    = pos1;
      if(string[pos2]=='C')
        c_index    = pos2;
      else
        u_index    = pos2;
    }
    else
      return -1;
    break;
  case 'U':
    free    = u_complements.size();
    if(free > 0) {
      index2     = (int) (1.0*free*rand()/(RAND_MAX+1.0));
#ifdef PROG_DEBUG
      if(index2 >= (int) u_complements.size() || index2 < 0){
        cerr << "DEBUG: u_complements index2 out of bounds!\n";
        cerr << index2 << endl;
        exit(1);
      }
#endif
      pos2       = u_complements[index2];
      if(abs(pos2 - pos1) < 3)
        return -1;
      pseudo_knots = check_pseudo_knots(pos1,pos2);
      if(pseudo_knots > 0)
        return -1;
      u_index    = pos1;
      if(string[pos2]=='A')
        a_index    = pos2;
      else
        g_index    = pos2;
    }
```

```
294       else
            return -1;
296       break;
        default:
          cerr << "Illegal character: "
               << string[pos1]
300            << " in RNS string! Bug in rns_string.cc!?\n";
          exit(1);
302     }
        rns_string::connect(pos1,pos2);
304     return pos1;
    }
306
    int rns_string::dissolve(void){
308     int              pos1,index, bound;

310     // find bound pair
        bound = pairs_lookup.size();
312     if(bound < 1)
          return -1;
314     index  = (int) (1.0*bound*rand()/(RAND_MAX+1.0));
    #ifdef PROG_DEBUG
316     if(index>=(int) pairs_lookup.size()){
          cerr << "rand() index out of bounds in rns_string::dissolve!\n";
318       exit(1);
        }
320 #endif
        pos1   = pairs_lookup[index];
322 #ifdef PROG_DEBUG
        int pos2   = pairs[pos1];
324     if(pos2 < 0 || pos2 > (int) length) {
          cerr << "Bug detected in dissolve operator in rns_string.cc!\n";
326       cerr << "Pair management derailed.\n";
          exit(1);
328     }
    #endif
330     rns_string::disconnect(pos1,index);
```

```
  return 1;
}

int rns_string::zip(int pos1){
  int          pos2, pos1_backup, pos2_backup;
  int          bound, valid_pair=1, return_value=0;

  // sanity check
  bound = pairs_lookup.size();
  if(bound < 1)
    return -1;
  pos2 = pairs[pos1];
  pos1_backup = pos1;
  pos2_backup = pos2;
  // check inwards direction for
  // possible pairs: AU, CU, CG
  while(valid_pair && (pos2 - pos1 > 5)){
    valid_pair = 0;
    pos1 ++;
    pos2 --;
    valid_pair = rns_string::try_pairing(pos1, pos2);
    if(valid_pair)
      rns_string::connect(pos1,pos2);
  }
  // check outbound direction for
  // possible pairs: AU, CU, CG
  pos1 = pos1_backup;
  pos2 = pos2_backup;
  valid_pair = 1;
  while(valid_pair &&
        (pos2 - pos1 > 5) &&
        (pos1 > 0) &&
        (pos2 < (int)(length -1))){
    valid_pair = 0;
    pos1 --;
    pos2 ++;
    valid_pair = rns_string::try_pairing(pos1, pos2);
```

```
368     if(valid_pair)
          rns_string::connect(pos1,pos2);
370   }
    return return_value;
372 }

374 int rns_string::unzip(void){
    int         index, pos1, pos1_backup;
376   int         bound, return_value=0;
    static int valid_pair=1;
378
    // find bound pair
380   bound = pairs_lookup.size();
    if(bound < 1)
382     return -1;
    // check possible coordinates
384   index = (int) (1.0*bound*rand()/(RAND_MAX+1.0));
#ifdef PROG_DEBUG
386   if(index>=(int) pairs_lookup.size()){
      cerr << "rand() index out of bounds in rns_string::zip!\n";
388     exit(1);
    }
390 #endif
    pos1    = pairs_lookup[index];
392 #ifdef PROG_DEBUG
    int pos2  = pairs[pos1];
394   if(pos1 > pos2){
      cout << "DEBUG: Bug  detected in rns_string::unzip!\n"
396         << "Pair management derailed.\n";
      exit(1);
398   }
#endif
400   pos1_backup = pos1;
    index       = pairs_lookup_lookup[pos1];
402   rns_string::disconnect(pos1, index);
    while(valid_pair){
404     pos1 ++;
```

```
    valid_pair=0;
406 if(structure[pos1] != '.' &&
       structure[pos1] != ')'){
408   valid_pair=1;
      return_value++;
410   index = pairs_lookup_lookup[pos1];
#ifdef PROG_DEBUG
412   if(index < 0 || index > (int)(pairs_lookup.size()-1)){
        cerr << "DEBUG: pairs_lookup index " << index
414          << " out of range in rns_string::unzip!\n";
        exit(1);
416   }
#endif
418   // disconnect
      rns_string::disconnect(pos1, index);
420 }
  }
422 valid_pair=1;
  while(valid_pair && pos1_backup>0){
424   pos1_backup--;
    valid_pair=0;
426 if(structure[pos1_backup] != '.' &&
       structure[pos1_backup] != ')')
428   {
      valid_pair=1;
430   return_value++;
      index = pairs_lookup_lookup[pos1_backup];
432 #ifdef PROG_DEBUG
      if(index < 0 || index > (int)(pairs_lookup.size()-1)){
434     cerr << "DEBUG: pairs_lookup index " << index
             << " out of range in rns_string::unzip!\n";
436   exit(1);
      }
438 #endif
      // disconnect
440   rns_string::disconnect(pos1_backup, index);
    }
```

```
442     }
        return return_value;
444 }

446 void rns_string::connect(int pos1, int pos2){

448     int             last,index,index2;
        std::string     base_pair;
450
        //-----------------------------------------------
452     // connect positions pos1 and pos2 and
        // prevent multiple bindings of same position
454     //-----------------------------------------------
        // base pair        regards bases
456     // AU, UA           A,G,U
        // CG, GC           C,G,U
458     // GU, UG           A,C,G,U
        //-----------------------------------------------
460
        base_pair =  string[pos1];
462     base_pair += string[pos2];

464     if(base_pair[0]=='A' ||
           (base_pair[0]=='U' && base_pair[1]=='A')){
466       // A
          index                 = a_lookup[u_index];
468       index2                = a_complements.back();
#ifdef PROG_DEBUG
470       if(index >=(int) a_complements.size() || index < 0){
            cerr << "DEBUG: u_index out of range\n";
472         cerr << index << endl;
            exit(1);
474       }
#endif
476       a_complements[index] = index2;
          a_complements.pop_back();
478       a_lookup[index2]     = index;
```

```
    // G
480 index                = g_lookup[u_index];
    index2               = g_complements.back();
482 #ifdef PROG_DEBUG
    if(index >= (int) g_complements.size() || index < 0){
484   cerr << "DEBUG: u_index out of range\n";
      cerr << index << endl;
486   exit(1);
    }
488 #endif
    g_complements[index] = index2;
490 g_complements.pop_back();
    g_lookup[index2]     = index;
492 // U
    index                = u_lookup[a_index];
494 index2               = u_complements.back();
    #ifdef PROG_DEBUG
496 if(index >= (int) u_complements.size() || index < 0){
      cerr << "DEBUG: a_index out of range\n";
498   cerr << index << endl;
      exit(1);
500 }
    #endif
502 u_complements[index] = index2;
    u_complements.pop_back();
504 u_lookup[index2]     = index;
  }
506 if(base_pair[0]=='C' ||
     (base_pair[0]=='G' && base_pair[1]=='C')){
508 // C
    index                = c_lookup[g_index];
510 index2               = c_complements.back();
    #ifdef PROG_DEBUG
512 if(index >= (int) c_complements.size() || index < 0){
      cerr << "DEBUG: g_index out of range\n";
514   cerr << index << endl;
      exit(1);
```

```
516     }
#endif
518     c_complements[index] = index2;
        c_complements.pop_back();
520     c_lookup[index2]      = index;
        // G
522     index                 = g_lookup[c_index];
        index2                = g_complements.back();
524 #ifdef PROG_DEBUG
        if(index >= (int) g_complements.size() || index < 0){
526       cerr << "DEBUG: c_index out of range\n";
          cerr << index << endl;
528       exit(1);
        }
530 #endif
        g_complements[index] = index2;
532     g_complements.pop_back();
        g_lookup[index2]     = index;
534     // U
        index                = u_lookup[g_index];
536     index2               = u_complements.back();
#ifdef PROG_DEBUG
538     if(index >= (int) u_complements.size() || index < 0){
          cerr << "DEBUG: g_index out of range\n";
540       cerr << index << endl;
          exit(1);
542     }
#endif
544     u_complements[index] = index2;
        u_complements.pop_back();
546     u_lookup[index2]     = index;
      }
548   if((base_pair[0]=='G' && base_pair[1]=='U') ||
         (base_pair[0]=='U' && base_pair[1]=='G')){
550     // A
        index                = a_lookup[u_index];
552     index2               = a_complements.back();
```

```
#ifdef PROG_DEBUG
554     if(index >= (int) a_complements.size() || index < 0){
          cerr << "DEBUG: u_index out of range\n";
556       cerr << index << endl;
          exit(1);
558     }
#endif
560     a_complements[index] = index2;
        a_complements.pop_back();
562     a_lookup[index2]     = index;
        // C
564     index                = c_lookup[g_index];
        index2               = c_complements.back();
566 #ifdef PROG_DEBUG
        if(index >= (int) c_complements.size() || index < 0){
568       cerr << "DEBUG: g_index out of range\n";
          cerr << index << endl;
570       exit(1);
        }
572 #endif
        c_complements[index] = index2;
574     c_complements.pop_back();
        c_lookup[index2]     = index;
576     // G
        index                = g_lookup[u_index];
578     index2               = g_complements.back();
#ifdef PROG_DEBUG
580     if(index >= (int) g_complements.size() || index < 0){
          cerr << "DEBUG: u_index out of range\n";
582       cerr << index << endl;
          exit(1);
584     }
#endif
586     g_complements[index] = index2;
        g_complements.pop_back();
588     g_lookup[index2]     = index;
          // U
```

```
590     index                 = u_lookup[g_index];
        index2                  = u_complements.back();
592 #ifdef PROG_DEBUG
        if(index >= (int) u_complements.size() || index < 0){
594       cerr << "DEBUG: g_index out of range\n";
          cerr << index << endl;
596       exit(1);
        }
598 #endif
        u_complements[index] = index2;
600     u_complements.pop_back();
        u_lookup[index2]     = index;
602   }

604   // bind positions 1 and 2
      if(pos1 < pos2){
606     structure[pos1] = '(';
        structure[pos2] = ')';
608   }
      else {
610     structure[pos1] = ')';
        structure[pos2] = '(';
612   }
      pairs[pos1] = pos2;
614   pairs[pos2] = pos1;
      if(pos1 < pos2){
616     pairs_lookup.push_back(pos1);
        pairs_lookup_lookup[pos1] = pairs_lookup.size()-1;
618   }
      else{
620     pairs_lookup.push_back(pos2);
        pairs_lookup_lookup[pos2] = pairs_lookup.size()-1;
622   }
  #ifdef PROG_DEBUG
624   // consistency check
      for(unsigned i=0;i < pairs_lookup.size(); i++){
626     if(pairs[pairs_lookup[i]] == -1){
```

```
      cerr << "DEBUG: bug detected in rns_string::connect!\n"
628        << "pairs_lookup table inconsistent\n";
      exit(1);
630   }
  }
632 #endif
  index  = free_lookup[pos1];
634   index2 = free_lookup[pos2];
  last   = free_places.size()-1;
636   if(index  == last){
    free_places.pop_back();
638     last = free_places.back();
    free_places[index2] = last;
640     free_places.pop_back();
    free_lookup[last]   = index2;
642     return;
    }
644   if(index2 == last){
    free_places.pop_back();
646     last = free_places.back();
    free_places[index] = last;
648     free_places.pop_back();
    free_lookup[last]  = index;
650     return;
  }
652   last = free_places.back();
  free_places[index2] = last;
654   free_places.pop_back();
  free_lookup[last]   = index2;
656   last = free_places.back();
  free_places[index]  = last;
658   free_places.pop_back();
  free_lookup[last]   = index;
660   return;
}
662
void rns_string::disconnect(int pos1, int pairs_index){
```

```
664   int last, pos2;

666 #ifdef PROG_DEBUG
    if((pos1 < 0) || (pos1 >= (int) length)){
668   cerr << "pos1 : " << pos1
         << " out of range in rns_string::disconnect!\n";
670   exit(1);
    }
672 #endif
    pos2 = pairs[pos1];
674 #ifdef PROG_DEBUG
    if((pos2 < 0) || (pos2 >= (int) length)){
676   cerr << "pos2 : " << pos2
         << " out of range in rns_string::disconnect!\n";
678   exit(1);
    }
680 #endif

682 // update tables and lookup tables
    switch(string[pos1]) {
684 case 'A':
      u_complements.push_back(pos1);
686   u_lookup[pos1] = u_complements.size()-1;
      break;
688 case 'C':
      g_complements.push_back(pos1);
690   g_lookup[pos1] = g_complements.size()-1;
      break;
692 case 'G':
      c_complements.push_back(pos1);
694   u_complements.push_back(pos1);
      c_lookup[pos1] = c_complements.size()-1;
696   u_lookup[pos1] = u_complements.size()-1;
      break;
698 case 'U':
      a_complements.push_back(pos1);
700   g_complements.push_back(pos1);
```

```
    a_lookup[pos1] = a_complements.size()-1;
    g_lookup[pos1] = g_complements.size()-1;
    break;
  default:
    cerr << "Unknown nucleotide " << string[pos1]
         << " in RNA sequence!\n"
         << "Bug in rns_string::disconnect function!\n\n";
    exit(1);
  }
  switch(string[pos2]) {
  case 'A':
    u_complements.push_back(pos2);
    u_lookup[pos2] = u_complements.size()-1;
    break;
  case 'C':
    g_complements.push_back(pos2);
    g_lookup[pos2] = g_complements.size()-1;
    break;
  case 'G':
    c_complements.push_back(pos2);
    u_complements.push_back(pos2);
    c_lookup[pos2] = c_complements.size()-1;
    u_lookup[pos2] = u_complements.size()-1;
    break;
  case 'U':
    a_complements.push_back(pos2);
    g_complements.push_back(pos2);
    a_lookup[pos2] = a_complements.size()-1;
    g_lookup[pos2] = g_complements.size()-1;
    break;
  default:
    cerr << "Unknown nucleotide in RNA sequence!\n"
         << "Bug in rns_string::disconnect!\n\n";
    exit(1);
  }
  free_places.push_back(pos1);
  free_lookup[pos1]  = free_places.size()-1;
```

```
738   free_places.push_back(pos2);
      free_lookup[pos2] = free_places.size()-1;
740 #ifdef PROG_DEBUG
      if(structure[pos1] == '.' || structure[pairs[pos1]] == '.'){
742     cerr << "DEBUG: bug detected in rns_string::disconnect!\n"
           << "Pair table unbalanced.\n";
744     exit(1);
      }
746 #endif
      pairs[pos1]          = -1;
748   pairs[pos2]          = -1;
      last                 = pairs_lookup.back();
750   pairs_lookup[pairs_index] = last;
      pairs_lookup_lookup[last] = pairs_index;
752 #ifdef PROG_DEBUG
      pairs_lookup[(int) pairs_lookup.size()-1] = -1;
754   pairs_lookup_lookup[pos1] = -1;
    #endif
756   pairs_lookup.pop_back();
    #ifdef PROG_DEBUG
758   // consistency check
      for(unsigned i=0;i < pairs_lookup.size(); i++){
760     if(pairs[pairs_lookup[i]] == -1){
        cerr << "DEBUG: bug detected in rns_string::connect!\n"
762          << "pairs_lookup table inconsistent\n";
        exit(1);
764     }
      }
766 #endif
      // dissolve binding
768   structure[pos1]  = '.';
      structure[pos2]  = '.';
770   return;
    }
772
    int rns_string::try_pairing(int pos1, int pos2){
774   int return_value=0;
```

```
  std::string base_pair;
776
  // possible pairs: AU, GU, GC
778 if(abs(pos2 - pos1) <= 3)
    return return_value;
780 base_pair  = string[pos1];
  base_pair += string[pos2];
782 if(structure[pos1]=='.' && structure[pos2]=='.'){
    if((base_pair[0]=='A') && (base_pair[1]=='U')){
784   a_index = pos1;
      u_index = pos2;
786   return_value++;
    }
788 if((base_pair[0]=='U') && (base_pair[1]=='A')){
      u_index = pos1;
790   a_index = pos2;
      return_value++;
792 }
    if((base_pair[0]=='C') && (base_pair[1]=='G')){
794   c_index = pos1;
      g_index = pos2;
796   return_value++;
    }
798 if((base_pair[0]=='G') && (base_pair[1]=='C')){
      g_index = pos1;
800   c_index = pos2;
      return_value++;
802 }
    if((base_pair[0]=='G') && (base_pair[1]=='U')){
804   g_index = pos1;
      u_index = pos2;
806   return_value++;
    }
808 if((base_pair[0]=='U') && (base_pair[1]=='G')){
      u_index = pos1;
810   g_index = pos2;
      return_value++;
```

```
812       }
    }
814   return return_value;
}
816
int rns_string::check_pseudo_knots(int pos1,int pos2){
818   int  pos1_backup, pos2_backup, braces, i;
    char first='.',last='.';
820
    // check for pseudo knots
822   braces = 0;
    if(pos1 < pos2){
824     pos1_backup = pos1+1;
      pos2_backup = pos2-1;
826   }
    else {
828     pos1_backup = pos2+1;
      pos2_backup = pos1-1;
830       }
    for(i=pos1_backup;i<=pos2_backup;i++){
832     if(first=='.')
        first = structure[i];
834     else
        if(structure[i]!='.')
836         last = structure[i];
      if(structure[i]=='(')
838       braces++;

840     else
        if(structure[i]==')')
842         braces--;
    }
844   if(braces!=0 || first == ')' || last == '(')
      return 1;
846   return 0;
}
```

The Main Loop

The main loop of the different algorithms is contained in the control-module. This module basically realizes the workflow as described in Figure 5.2. The relevant part of the source code is listed below:

```
#include <iostream.h>
#include <time.h>
#include <ctype.h>
#include <math.h>
#include <fstream.h>
#include "global_defs.h"
#include "mutex_guard.h"

#define INFINITY 10000

int main_loop(void) {
  static   double      w, g, old_percent, delta_psel, ctrl_psel;
  static   double      temp_backup, psel_backup, max_time_backup;
  static   double      init_time;
  static   int         counter, cycles, delta_n;
  static   int         reaction_window=1, cycle_count=0;
  double               variability=0, difference=0, tau=0;
  std::string          rns_file_name;
  ifstream             rns_file;

  // standard initialization
  if(gui)
    Mutex_Guard main_loop_thread;
  // read RNA sequence from file
  rns_file_name  = problem_name + ".dat";
  rns_file.open (rns_file_name.c_str(), ios::in);
  if(!rns_file){
    cerr << rns_file_name
         << " couldn't be opened to read RNS sequence!\n";
    exit(FALSE);
```

```
  }
32  getline(rns_file, initstring);
  rns_file.close();
34  for(w=0;w<initstring.length();w++)
    initstruct+='.';
36  // continue initialization
  max_time_backup=max_time;
38  hash = new rank[n_max];
  allocated++;
40  if((rnd_generator=gsl_rng_alloc(rnd_generator_type))==NULL) {
    cerr << "Random number generator initialization failed!\n";
42    exit(FALSE);
  }
44  if(n_max==n_min)
    delta_n = 1;
46  else
    delta_n     = (int) ceil(n_max-n_min)/n_steps;
48  if(psel_max==psel_min)
    delta_psel = 1;
50  else
    delta_psel = (psel_max-psel_min)/psel_steps;
52  if(strategy!=tournament  &&
     strategy!=tournament4 &&
54     strategy!=fitness) {
    psel_max=psel_min;
56    delta_psel=1;
  }
58  if(strategy!=fitness) {
    seeker = new rns_string[n_max];   // reserve memory for seekers
60    allocated++;
  }
62  else {
    // memory for seekers + offspring
64    seeker = new rns_string[2*n_max];
    allocated++;
66  }
  // status 100% relates to:
```

```
68  g=(1+(psel_max-psel_min)/delta_psel) *
      (1+(n_max-n_min)/delta_n) * maxruns * max_time;
70  old_percent=0;
    // selection sweep
72  for(psel=psel_min; psel<=psel_max; psel+=delta_psel){
      // seeker number sweep
74    for(n=n_min;n<=n_max;n+=delta_n) {
        // new cycle initialization
76      init_call=TRUE;
        init_time = initstring.length()*n*2;
78      max_time+=init_time;
        statistics(&cycles);
80      //if(ham_print)
        //        ham_stat(&cycles);
82      if(hist_print || automatic)
          make_histogram(cycles, ctrl_psel);
84      init_call=FALSE;
        //
86      temp_backup       = temp;
        psel_backup       = psel;
88      epsilon           = 1.0/n;
        variability_goal = 1.0/sqrt(n);
90      temp              = init_temp;
        if(psel!=0)
92        ctrl_psel = psel;
        else
94        ctrl_psel = 0.1;
        // print parameters used
96      if(verbose) {
          cerr << "\nProgram version   : "
98             << VERSION << endl;
          cerr << "opt. problem name : " << problem_name << endl;
100       cerr << "variability       : ";
          if(variability_type==fitness_oriented)
102         cerr << "fitness based\n";
          else
104         cerr << "genotype based\n";
```

```
    if(automatic)
106   cerr << "autotuning        : enabled\n";
    cerr << "search strategy   : ";
108 switch(strategy) {
    case worst:
110   cerr << "kill worst seeker\n";
      break;
112 case metropolis:
      cerr << "Metropolis algorithm\n";
114   break;
    case tournament:
116   cerr << "Boltzmann strategy + tournament selection\n";
      break;
118 case tournament4:
      cerr << "Boltzmann strategy + tournament 4 selection\n";
120   break;
    case fitness:
122   cerr << "Boltzmann strategy + "
         << "fitness proportional selection\n";
124   break;
    case annealing:
126   cerr << "simulated annealing\n";
      break;
128 }
    cerr << "number of seekers : " << n << endl
130      << "repetitions       : " << maxruns << endl
         << "init. temperature : " << temp << endl;
132 if(strategy!=worst &&
       strategy!=metropolis &&
134    strategy!=annealing)
      cerr << "selection prob.   : " << psel << endl;
136 if(goal)
      cerr << "goal value        : " << goal_value << endl;
138 cerr << "sequence length   : " << initstring.length()
         << endl
140      << "evaluations       : " << max_time << endl
         << "\n\n";
```

```
142       if(!hist_print)
            cerr << "suppressed histogram file output\n";
144       if(strategy!=annealing)
            cerr << "suppressed temperature file output\n";
146       if(!ctl_print)
            cerr << "suppressed control file output\n";
148       if(!stat_print)
            cerr << "suppressed statistics file output\n\n";
150     } // end: if verbose
        // **** run following code 'maxruns' times ****
152     for(cycles=0;cycles<maxruns;cycles++) { // cycle sweep
          run_time=0;
154       // generating seekers
          for(counter=0;counter<n;counter++) {
156         seeker[counter]=rns_string(initstring.c_str(),
                                       initstruct.c_str());
158         seeker[counter].evaluate();
          }
160       temp         = INFINITY;
          psel         = 0;
162       //-------------------------------------------------------------
          //      life cycle
164       //-------------------------------------------------------------
          while(run_time <= max_time) { // time sweep
166         if(run_time > init_time && temp == INFINITY){
              temp = temp_backup;
168           psel = psel_backup;

170         }
            // status report
172         if(gui) {
              pthread_testcancel();
174           w=(1+(psel-psel_min)/delta_psel) *
                (1+(n-n_min)/delta_n) * (cycles+1) * run_time;
176           percent_done=(int) rint(w*100.0/g);
              if(percent_done>old_percent) {
178             old_percent=percent_done;
```

```
        IPC_Handler->AsyncHandler();
180   }
    }
182 // calculate running ensemble statistics
    if(strategy == annealing || verbose2)
184   statistics(&cycles);
    if(hist_print || automatic)
186   variability=make_histogram(cycles, ctrl_psel);
    //-------- automatic ----------------------
188 if(run_time > init_time &&
     automatic && (cycle_count==reaction_window)) {
190   cycle_count=0;
    difference=variability-variability_goal;
192   if(epsilon+0.01 >= fabs(difference))
      difference=0;
194   if(difference > 0) {
      ctrl_psel*=exp(1 + difference/epsilon);
196   if(ctrl_psel > 100)
        ctrl_psel=100;
198   }
    else
200   if(difference < 0) {
        ctrl_psel/=exp(1 - difference/epsilon);
202     if(ctrl_psel < 0.1)
          ctrl_psel = 0.1;
204   }
  }
206 else
    cycle_count++;
208 // ----- end automatic --------------------

210 ////////////////////////////////////////////////
    //    search according to selected strategy    //
212 ////////////////////////////////////////////////

214 switch(strategy) {
    case annealing:
```

```
216             // calculate new temperature for next time step
                if(temp > dtemp)
218               temp-=dtemp;
                dtemp=1.0/stdv_fitness;
220             mutation();
                run_time++;
222             break;
              case worst:
224           case metropolis:
                mutation();
226             run_time++;
                break;
228           case fitness:
              case tournament:
230           case tournament4:
                tau=gsl_ran_exponential(rnd_generator, rnd_mu);
232             run_time+=1.0*rnd_mu*tau;
                if(100.0 * gsl_rng_uniform(rnd_generator) >= ctrl_psel)
234               mutation();
                else
236               selection(hash);
                break;
238           default:
                cerr << "Internal program error in main loop!\n"
240                   << "Unknown optimization strategy. Exiting now.\n\n";
                exit(FALSE);
242             break;

244             ////////////////////////////////////////////////
              } // end strategy
246         }   // end time sweep
            statistics(&cycles);
248         if(hist_print || ctl_print || automatic)
              make_histogram(cycles, ctrl_psel);
250         if(ham_print)
              ham_stat(&cycles);
252       }     // end cycle sweep
```

```
      max_time=max_time_backup;
254   // print final results
      statistics(&cycles);
256   if(hist_print || ctl_print || automatic)
        make_histogram(cycles, ctrl_psel);
258   if(ham_print)
        ham_stat(&cycles);
260  }      // end seeker number sweep
  }        // end selection probability sweep
262  cout << "Done.\n";
  status=undefined;
264  if(gui) {
    percent_done=100;
266  IPC_Handler->AsyncHandler();
  }
268  delete[] hash;
  allocated--;
270  delete[] seeker;
  allocated--;
272 #ifdef PROG_DEBUG
  if(allocated != 0)
274   cerr << "Program still holds " << allocated << " arrays!\n";
#endif
276  //  return TRUE;

278  return 0;
}
```

5.1.4 MPI_generate

This program was developed by the author to effectively generate correlated Gaussian random fitness landscapes in up to five dimensions (cf. section 3.1). It is written in C and refers to the *MPI* standard for message passing on multi processor machines. Acceleration is achieved due to a simple divide and conquer strategy, so the landscape generation speed scales nicely with the number of processors involved to generate it.

```
/*-------------------------------------------------------------------
 generate correlated, random landscapes in up to 5 dimensions
 utilizing multiple processors
 cf. Steinberg, M.:
 "Konstruktion von korrelierten, zufaelligen Landschaften"

 Copyright (C) 1999 A. Reimann
 Version : 0.5

 This program is free software; you can redistribute it and/or
 modify it under the terms of the GNU General Public License
 as published by the Free Software Foundation; either version 2
 of the License, or (at your option) any later version.

 This program is distributed in the hope that it will be useful,
 but WITHOUT ANY WARRANTY; without even the implied warranty of
 MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 GNU General Public License for more details.

 You should have received a copy of the GNU General Public License
 along with this program; if not, write to the Free Software
 Foundation, Inc., 59 Temple Place - Suite 330,
 Boston, MA 02111-1307, USA.
-------------------------------------------------------------------*/


/* include header */
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <mpi.h>

#define DEBUG

#ifndef PI
#define PI              3.141592653
#endif
#ifndef twoPI
#define twoPI           6.2831853072
#endif
#ifndef SQRT2
#define SQRT2           1.4142135624
#endif
#define SEED            SQRT2
#define MAXDIM          5
#define INIT_TAG        1
#define STATUS_TAG      2
#define SOLVED_TAG      3

/* define prototypes */
int                 get_opts(int argc, char **argv);
__inline__ float    fitness(int, int, float*);
void                initialize_rand(float*);
void                master(int, char**);
void                slave(int);
void                usage(void);

/* define global variables */
float       Gamma, factor1, factor3, factor4, factor5;
int         dimension, size, plane, volume, volume4, volume5;
static int  root=0;
char        *outfilename;
```

```
64 /*-------------------------------------------------------------------*/
   int main (int argc,char **argv) {
66   int myrank;

68   MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
70
     if(myrank==root) {
72     master(argc, argv);
       fprintf(stderr,"done.\n");
74   }
     else
76     slave(myrank);
     return (0);
78 }
   /*-------------------------------------------------------------------*/
80
   void master(int argc, char** argv) {
82   MPI_Status  status;
     MPI_Request request;
84   FILE        *filehandle;
     char        filename[100];
86   int         source, dest, proc_nr, running, remainder, flag;
     int         buffer, count, i, j, *coordinate, *percentage;
88   long int    rand_nr;
     float       rand_mem, *chi, *result;
90   ldiv_t      lfraction;
     div_t       fraction;
92
     get_opts(argc, argv);
94   strcpy(filename, outfilename);
      /* initialize random numbers */
96   rand_nr=(long int) pow(size, dimension);
     rand_mem=1.0*sizeof(float)*(rand_nr+2);
98   if(rand_mem/1048576>1)
       fprintf(stderr,"allocating %.2f MB of memory\n",
100            rand_mem/1048576);
```

```
  else
    fprintf(stderr,"allocating %.2f kB of memory\n",
            rand_mem/1024);
  chi=malloc(rand_mem);
  if(chi==NULL) {
    fprintf(stderr,"Insufficient memory!\n");
    MPI_Abort(MPI_COMM_WORLD, 1);
    exit (1);
  }
  fprintf(stderr,"initializing random number reservoir\n");
  initialize_rand(chi);
  fprintf(stderr,"initializing slave processes:\n");
  /* seed the slaves                          */
  /* send: dimension, size, Gamma and         */
  /*       random number reservoir            */
  MPI_Comm_size(MPI_COMM_WORLD, &proc_nr);
  for(dest=1; dest<proc_nr; dest++) {
    fprintf(stderr,"Nr. %i ",dest);
    MPI_Send(&dimension, 1, MPI_INT, dest, INIT_TAG, MPI_COMM_WORLD);
    fprintf(stderr,".");
    MPI_Send(&size, 1, MPI_INT, dest, INIT_TAG, MPI_COMM_WORLD);
    fprintf(stderr,".");
    MPI_Send(&Gamma, 1, MPI_FLOAT, dest, INIT_TAG, MPI_COMM_WORLD);
    fprintf(stderr,".");
    MPI_Send(&rand_nr, 1, MPI_LONG, dest, INIT_TAG, MPI_COMM_WORLD);
    fprintf(stderr,".");
    MPI_Send(chi, rand_nr, MPI_FLOAT, dest, INIT_TAG, MPI_COMM_WORLD);
    fprintf(stderr,".");
    fprintf(stderr," initialized\n");
  }
  /* check, whether fitness[] splits evenly */
  /* send fraction of result array          */
  lfraction=ldiv(rand_nr, (proc_nr-1));
  if(lfraction.quot<=1){
    fprintf(stderr, "Warning: Problem too small to be treated ");
    fprintf(stderr,"efficiently on %i Processors.\n", proc_nr);
  }
```

```
138   if(lfraction.rem==0)
        {
140       #ifdef DEBUG
          fprintf(stderr,"task splits nicely\n");
142       #endif
          for(dest=1; dest<proc_nr; dest++)
144         MPI_Send(&lfraction.quot, 1, MPI_LONG,
                    dest, INIT_TAG, MPI_COMM_WORLD);
146     }
      else
148     {
        /* prepare some intelligent partitioning */
150       #ifdef DEBUG
          fprintf(stderr,"task splits inconveniently\n");
152       #endif
          if((lfraction.quot+lfraction.rem)>=(proc_nr-1)) {
154         lfraction.quot++;
            lfraction.rem=rand_nr-((proc_nr-2)*lfraction.quot);
156       }
          else
158         lfraction.rem=lfraction.rem+lfraction.quot;
        /* send fractions */
160       #ifdef DEBUG
          fprintf(stderr,"sending %i times %li\n",
162               proc_nr-2, lfraction.quot);
          fprintf(stderr,"      + 1 time  %li numbers.\n",
164               lfraction.rem);
          #endif
166       for(dest=1; dest<(proc_nr-1); dest++)
            MPI_Send(&lfraction.quot, 1, MPI_LONG,
168                   dest, INIT_TAG, MPI_COMM_WORLD);
          MPI_Send(&lfraction.rem, 1, MPI_LONG,
170                proc_nr-1, INIT_TAG, MPI_COMM_WORLD);
        }
172   /* prepare status information output */
      percentage=malloc(sizeof(int)*proc_nr);
174   for(i=0;i<proc_nr;i++)
```

```
    percentage[i]=0;
fprintf(stderr,"percent processed:\n");
/* receive status information */
#ifdef DEBUG
    fprintf(stderr,"(reallocating %g bytes)\n",rand_mem);
#endif
result=realloc(chi, rand_mem+100);
if(result==NULL) {
    fprintf(stderr,"Insufficient memory!\n");
    MPI_Abort(MPI_COMM_WORLD, 5);
    exit (5);
}
running=proc_nr-1;
while(running) {
    MPI_Irecv(&buffer, 1, MPI_INT, MPI_ANY_SOURCE,
              MPI_ANY_TAG, MPI_COMM_WORLD, &request);
    do
        MPI_Test(&request, &flag, &status);
    while(!flag);
    source=status.MPI_SOURCE;
    switch(status.MPI_TAG) {
    case STATUS_TAG:
        percentage[source-1]=buffer;
         if(buffer!=0) {
          for(i=0;i<(proc_nr-1);i++)
            fprintf(stderr,"%i  ",percentage[i]);
          fprintf(stderr,"\n");
        }
        break;
    case SOLVED_TAG:
        MPI_Recv(&count, 1, MPI_INT, source,
                 SOLVED_TAG, MPI_COMM_WORLD, &status);
        MPI_Recv(result+buffer, count, MPI_FLOAT, source,
                 SOLVED_TAG, MPI_COMM_WORLD, &status);
        running--;
        break;
```

```
212       }
    }
    /* write final result to disk */
    fflush(NULL);
216    fprintf(stderr,"\nWriting results to %s\n",filename);
    filehandle=fopen(filename, "w");
218    if(filehandle==NULL) {
      fprintf(stderr,"Couldn't open file %s for writing!\n",filename);
220      MPI_Abort(MPI_COMM_WORLD, 2);
    }
222    coordinate=malloc(dimension*sizeof(int));
    /* calculate coordinates from index */
224    for(j=0; j<rand_nr; j++) {
      remainder=j;
226      for(i=(dimension-1); i>0; i--){
        fraction=div(remainder,pow(size,i));
228        coordinate[i]=fraction.quot;
        remainder=fraction.rem;
230      }
      coordinate[0]=remainder;
232      for(i=0; i<dimension; i++)
        fprintf(filehandle,"%i\t",coordinate[i]);
234      fprintf(filehandle,"%f\n",result[j]);
    }
236    MPI_Finalize();
    return;
238 }

240 void slave(int myrank) {
    static int   root=0;
242    long int     rand_nr;
    int          i, fraction, offset, percent, nr_slaves;
244    MPI_Status   status;
    MPI_Request  request;
246    float        *chi, *result, step, intervall=5.0;
    ldiv_t       lfraction;
248
```

```
  /* receive dimension, size and gamma value */
250 MPI_Recv(&dimension, 1, MPI_INT, root,
          INIT_TAG, MPI_COMM_WORLD, &status);
252 MPI_Recv(&size, 1, MPI_INT, root,
          INIT_TAG, MPI_COMM_WORLD, &status);
254 MPI_Recv(&Gamma, 1, MPI_FLOAT, root,
          INIT_TAG, MPI_COMM_WORLD, &status);
256 /* allocate memory for random number reservoir */
  MPI_Recv(&rand_nr, 1, MPI_LONG, root,
258       INIT_TAG, MPI_COMM_WORLD, &status);
  if(!(chi=malloc(sizeof(float)*rand_nr))) {
260   MPI_Abort(MPI_COMM_WORLD, 5);
    exit(5);
262 }
  /* receive random number reservoir */
264 MPI_Recv(chi, rand_nr, MPI_FLOAT, root,
          INIT_TAG, MPI_COMM_WORLD, &status);
266 /* allocate memory for fitness values */
  MPI_Recv(&fraction, 1, MPI_LONG, root,
268       INIT_TAG, MPI_COMM_WORLD, &status);
  if(!(result=malloc(sizeof(float)*fraction))) {
270   MPI_Abort(MPI_COMM_WORLD, 5);
    exit(5);
272 }
  /* precalculate constant factors */
274 MPI_Comm_size(MPI_COMM_WORLD, &nr_slaves);
  nr_slaves--; /* master doesn't count */
276 plane=size*size;
  volume=plane*size;
278 volume4=volume*size;
  volume5=volume4*size;
280 factor1=sqrt(Gamma/(2.0*PI*PI));
  factor3=SQRT2/pow(PI,3/2);
282 factor4=1/(16*pow(PI,4));
  factor5=sqrt(Gamma/(2*pow(PI,5)));
284 /* start actual work */
  percent=-intervall;
```

```
286   step=fraction/intervall;
      if(myrank==nr_slaves)
288     offset=rand_nr-fraction;
      else
290     offset=(myrank-1)*fraction;
      for (i=0;i<fraction;i++) {
292     result[i]=fitness(offset+i, size, chi);
        lfraction=ldiv(i,(long) ceil(step));
294     if(lfraction.rem==0) {
          if(percent>=0)
296         MPI_Wait(&request,&status);
          percent+=intervall;
298       MPI_Isend(&percent, 1, MPI_INT, root,
                    STATUS_TAG, MPI_COMM_WORLD, &request);
300     }
      }
302   /* submit results to master */
      MPI_Send(&offset, 1, MPI_INT, root,
304             SOLVED_TAG, MPI_COMM_WORLD);
      MPI_Send(&fraction, 1, MPI_INT, root,
306             SOLVED_TAG, MPI_COMM_WORLD);
      MPI_Send(result, fraction, MPI_FLOAT, root,
308             SOLVED_TAG, MPI_COMM_WORLD);
      /* done */
310   MPI_Finalize();
      return;
312 }

314

    /*--------------------------------------------------------------------*/
316

    int get_opts(int argc, char **argv){
318   int i;

320   if(argc<2)
        usage();
322   for (i=0; i<argc; i++) {
```

```
    if (!strncmp(argv[i],"-h",2))
324   usage();
    if (!strncmp(argv[i],"-s",2))
326   size=atoi(argv[i+1]);
    if (!strncmp(argv[i],"-g",2))
328   Gamma=atof(argv[i+1]);
    if (!strncmp(argv[i],"-d",2))
330   dimension=atoi(argv[i+1]);
    if (!strncmp(argv[i],"-f",2) && i<(argc-1))
332   outfilename=argv[i+1];
  }
334 if(dimension<=0 || dimension > MAXDIM) dimension=1;
  fprintf(stderr,"dimension: %i\n", dimension);
336 if (size<=0) size=10;
  fprintf(stderr,"size      : %i\n", size);
338 fprintf(stderr,"gamma     : %.2f\n", Gamma);
  fprintf(stderr,"outfile   : %s\n", outfilename);
340 if (Gamma*2 >= size) {
    fprintf(stderr,"\nWARNING: Gamma comparatively high!\n\n");
342 }
  return 1;
344 }

346 void initialize_rand(float* chi) {
  float         v1, v2, v3, radius;
348 unsigned int i;

350 srand(SEED);
  for (i=0; i<((unsigned int)(pow(size, dimension)-2)); i+=2) {
352   do {
      v1=2.0*rand()/(RAND_MAX+1.0)-1.0;
354   v2=2.0*rand()/(RAND_MAX+1.0)-1.0;
      radius=v1*v1+v2*v2;
356   }
    /* pick two numbers in unit cycle */
358 while (radius>=1.0 || radius == 0.0);
    v3=sqrt(-2.0*log(radius)/radius);
```

```
360     chi[i]=v1*v3;
        chi[i+1]=v2*v3;
362   }
      return;
364 }

366 float fitness(int index, int size, float* chi) {
      int i, x1, y1, z1, a1, b1, x2, y2, z2, a2, b2;
368   int modulus;
      float fitness=0, r;
370
      switch (dimension) {
372   case 1:
        for (i=0;i<size;i++) {
374       r=1.0*abs(index-i);
          if(r!=0)
376         fitness+=chi[i]*factor1/(exp(r/Gamma)*sqrt(r));
        }
378     break;
      case 2:
380     x1=index%size;
        y1=index/size;
382     for (i=0; i<plane; i++) {
          x2=i%size;
384       y2=i/size;
          r=sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
386       fitness+=chi[i]*exp(-r/Gamma)/twoPI;
        }
388     break;
      case 3:
390     z1=index/plane;
        modulus=index%plane;
392     y1=modulus/size;
        x1=modulus%size;
394     for (i=0; i<volume; i++) {
          z2=i/plane;
396       modulus=i%plane;
```

```
      y2=modulus/size;
398   x2=modulus%size;
      r=sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+(z2-z1)*(z2-z1));
400   if(r!=0) fitness+=chi[i]*factor3*(exp(-r/Gamma)/r);
    }
402 break;
  case 4:
404 a1=index/volume;
    modulus=index%volume;
406 z1=modulus/plane;
    modulus=modulus%plane;
408 y1=modulus/size;
    x1=modulus%size;
410 for (i=0; i<volume4; i++) {
      a2=i/volume;
412   modulus=i%volume;
      z2=modulus/plane;
414   modulus=modulus%plane;
      y2=modulus/size;
416   x2=modulus%size;
      r=sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+(z2-z1)*(z2-z1)+
418           (a2-a1)*(a2-a1));
      fitness+=chi[i]*factor4*exp(-2*r/Gamma);
420 }
    break;
422 case 5:
    b1=index/volume4;
424 modulus=index%volume4;
    a1=modulus/volume;
426 modulus=modulus%volume;
    z1=modulus/plane;
428 modulus=modulus%plane;
    y1=modulus/size;
430 x1=modulus%size;
    for (i=0; i<volume5; i++) {
432   b2=i/volume4;
      modulus=i%volume4;
```

```
434         a2=modulus/volume;
            modulus=modulus%volume;
436         z2=modulus/plane;
            modulus=modulus%plane;
438         y2=modulus/size;
            x2=modulus%size;
440         r=sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1)+(z2-z1)*(z2-z1)+
                    (a2-a1)*(a2-a1)+(b2-b1)*(b2-b1));
442         fitness+=chi[i]*factor5*exp(-2*r/Gamma);
          }
444     }
        return fitness;
446 }

448 void usage(void) {
        fprintf(stderr,"program requires MPI to be installed\n\n");
450     fprintf(stderr,"invocation: mpi_generate [-s #] [-g #] [-d #]");
        fprintf(stderr," [-f file]\n");
452     fprintf(stderr,"s:\tsize of parameter space\n");
        fprintf(stderr,"g:\tcorrelation length\n");
454     fprintf(stderr,"d:\tdimension of parameter space\n");
        fprintf(stderr,"f:\toutput file\n\n");
456     MPI_Abort(MPI_COMM_WORLD, 1);
        exit;
458 }
```

5.2 Open Source Software

The software developed for this work, of course, does not reinvent the wheel. The programs and libraries listed below where used for data analysis and as building blocks in the auhor's simulation software.

5.2.1 The Message Passing Interface MPI

Whenever it comes to programming parallel machines, the problem of data and task synchronization arises. Usually, the tasks running in parallel solve this problem by sending messages back and forth. MPI is a library specification for message-passing, proposed as a standard by a broadly based committee of vendors, implementors, and users.[1] It was designed for high performance on both massively parallel machines and on workstation clusters. Implementations include, among others, MPICH and LAM (Local Area Multicomputer).

5.2.2 The Vienna RNA Package

The core of the Vienna RNA Package is formed by a collection of routines for the prediction and comparison of RNA secondary structures. These routines can be accessed through stand-alone programs, such as RNAfold, RNAdistance etc., which should be sufficient for most users; but they are also made available by a software library. [2]

5.2.3 Free Visualization Software

All figures in this work were generated using free software covered by the GPL. The following programs where particularly helpfull:

[1] http://www-unix.mcs.anl.gov/mpi/

[2] http://www.tbi.univie.ac.at/∼ivo/RNA/RNAlib.html

GMT The Generic Mapping Tools were developed at the School of Ocean and Earth Science and Technology, Hawaii. GMT is a free, public-domain collection of ∼60 UNIX tools that allow users to manipulate (x,y) and (x,y,z) data sets (including filtering, trend fitting, gridding, projecting, etc.) and produce Encapsulated PostScript File (EPS) illustrations ranging from simple x-y plots through contour maps to artificially illuminated surfaces and 3-D perspective views in black and white, gray tone, hachure patterns, and 24-bit color. GMT supports 25 common map projections plus linear, log, and power scaling, and comes with support data such as coastlines, rivers, and political boundaries. It is available at
$http://www.soest.hawaii.edu/gmt/$.

Vis5D is a system for interactive visualization of large 5-D gridded data sets. One can make isosurfaces, contour line slices, colored slices, etc of data in a 3-D grid then rotate and animate the image in real time. There's also a feature for trajectory tracing, a way to make text anotations for publications, etc. Vis5D uses a binary format to store its data, making it necessary to convert ASCII input. Vis5D is available for download at
$http://www.sourceforge.net/projects/vis5d/$.

XMGrace Grace is a WYSIWYG 2D plotting tool for the X Window System and Motif. Grace runs on practically any version of Unix. Also, it has been successfully ported to VMS, OS/2, and Win9*/NT (some minor functionality may be missing, though).

Grace is a descendant of ACE/gr, also known as Xmgr. It is available at
$http://plasma-gate.weizmann.ac.il/Grace/$.

5.2.4 Free External Libraries

All statistical calculations rely on the **GNU Scientific Library** GSL which is available[1] under the GNU Public License GPL Version 2. This library is currently under heavy development but nonetheless offers a tremendous and reliable help for numerical computations. For example, it embodies carefully crafted routines to avoid numerical artifacts due to rounding errors or variable overflows. The library version used for this work is GSL V.:0.6.

The program SimLabs also links to the `C++` **Standard Template Library** STL to access the vector class . This ensures an abstract interface, data type safety, and inhibits buffer overflows as well as memory leaks.

The graphical user interface (GUI) was realized with help of the **Qt GUI toolkit** which is Copyright (C) 1994-2000 Trolltech AS. The toolkit was, however, brought under the GPL version 2 in the year 2000.

The programs SimEngel and SimLabs need the qt libraries version 2.0 or above. Since the interprocess communication (IPC) is done using Qt's signal/slot mechanism and the libraries before version 2.2 were not thread safe, both programs contain their own mutexes and schedule all X events through a pipe. This prevents timing dependent crashes when both the X Server and the program interfere by trying to access the same resources.

[1] available at ftp://alpha.gnu.org/gnu

Appendix A

Polio Virus Type 1 Subsequence

All simulations with respect to RNA secondary structures where carried out using a 100 base sequence of Polio virus Type 1 Mahoney (AC V01148, 5'-cloverleaf). The primary structure of this sequence is (spaces are inserted for readability):

CCCUU CCCUC AUAUU
UUGUC CGCAU GUUCC
CAUGG CGUUA UGGCC
UCAUG AUCGG CGGUG
CACCC GGAGA CCCCA
CCCAU GUUGG GGUCU
CGACA AAAUU

The optimal folding (i.e. the least free energy secondary structure) determined by the Vienna RNA package[1] (version 1.4) has a free energy of $F = -32.0$ kcal/mol. This differs from the result $F = -28.09$ kcal/mol found in a work by ROSÉ [15]. His work relies on an earlier version of the Vienna RNA package, however, which used a different energy functional. The secondary structure found by the package's recursive algorithm is shown in Figure A.1.

[1] http://www.tbi.univie.ac.at/∼ivo/RNA/

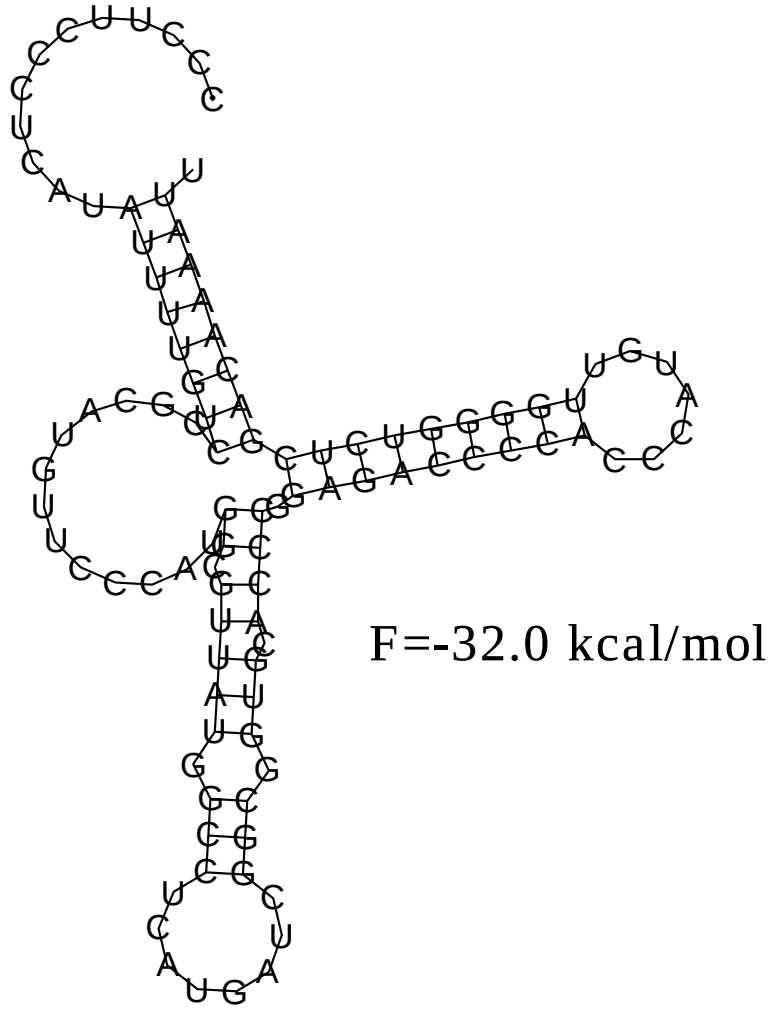


Figure A.1: Best secondary structure of the first $L = 100$ base pair sequence of Polio virus Type 1 (AC V01148, 5'-cloverleaf) found by the recursive algorithm included in the Vienna RNA package Version 1.4

In bracket notation, the secondary structure seen in Figure A.1 reads:

....((((((((..... ....((.(((((.(((..... ...))
).))) .)))) .(((((((((..... ..)))))))))))))))))).

This optimum is at least two-fold degenerated since the optimal folding found with the mixed evolutionary strategies introduced in this work has the same free energy but with a different secondary structure. In Figure A.2, this optimum as well as sub-optimal foldings found 'on the way' in the search process are shown.

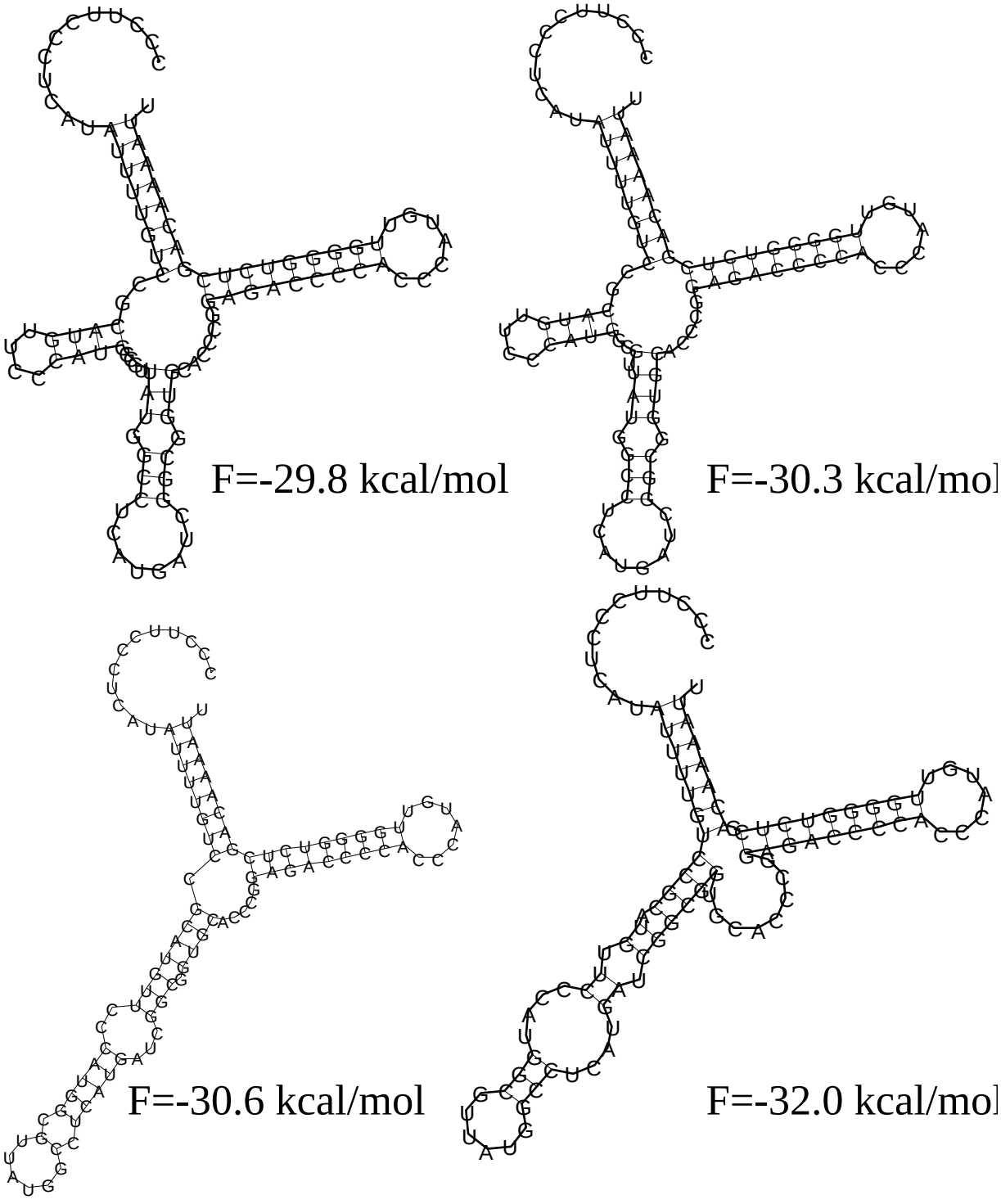


Figure A.2: Optimal and suboptimal secondary structures of the first part ($L = 100$) of Polio Virus Type 1 Mahoney (*AC V*01148) and their respective free energies found by the adaptive evolutionary algorithm [60] using some $N = 35$ seekers and a time limit of $t = 150.000$ steps.

Appendix B

Glossary

This glossary, which is not complete in any way, lists terms and explanations often encountered not only in this work, but also in related literature that is cited herein.

A

algorithm A complete, unambiguous procedure for solving a specified problem in a finite number of steps.

ASCII American Standard Code for Information Interchange; ASCII is the universal standard for the numerical codes computers use to represent all upper and lower-case letters, numbers, and punctuation.

autocorrelation The autocorrelation describes how a function varies with itself; i.e. it is a measure of self-similarity.

autocorrelation coefficient The autocorrelation coefficient R_k for a given lag k is confined to the interval $[-1,1]$ and calculated as follows:

$$R_k = \frac{\sum_{t=1}^{N-k} (x_t - \bar{x})(x_{t+k} - \bar{x})}{\sum_{t=1}^{N} (x_t - \bar{x})^2}$$

see also: *correlation coefficient*

autocorrelation function The autocorrelation function contains the entire series of autocorrelation coefficients.

B

Bernoulli trial A Bernoulli trial is an experiment with only two possible outcomes. The probability p of success and probability q of failure must satisfy $p+q=1$.

A *binomial random variable* counts the number of successes in n independent Bernoulli trials; a *geometric random variable* counts the number of independent trials until the first success.

bimodal distribution A relative frequency or probability distribution characterized by two peaks or humps rather than the more common single peak, which characterizes the normal distribution and most other standardized distributions.

binomial distribution A binomial random variable X is a discrete variable in the interval $[0,n]$ with the probability distribution:

$$P_k(X) = \begin{cases} \binom{n}{k} p^k q^{n-k}; & 0 \le k \le n \\ 0 \quad \text{otherwise} \end{cases}$$

It describes the number of successes $X = k$ for n independent trials in an experiment with only two possible outcomes p and q. The *mean* of X is np and the *variance* is $nqp = np(1 - p)$.

Box-Muller transformation The Box-Muller transformation allows the generation of Gaussian distributed random numbers y_1 and y_2, given two equally distributed random numbers x_1 and x_2:

$$y1 = \sqrt{-2\ln x_1}\cos(2\pi x_2)$$
$$y2 = \sqrt{-2\ln x_1}\sin(2\pi x_2)$$

The polar form of the Box-Muller transformation is both faster and more robust numerically. The algorithmic description of it is:

```
float x1, x2, w, y1, y2;

  do {
    x1 = 2.0 * ranf() - 1.0;
    x2 = 2.0 * ranf() - 1.0;
     w = x1 * x1 + x2 * x2;
  } while ( w >= 1.0 );

  w = sqrt( (-2.0 * log(w) )/w );
 y1 = x1 * w;
 y2 = x2 * w;
```

C

central limit theorem The average of a fixed random variable measured repeatedly and independently asymptotically becomes a *normal random variable* as the number of measurements increases.

Chi square random variable The probability distribution for the always non-negative random variable χ^2 is given by

$$f(x) = \frac{x^{\frac{\nu}{2}-1} e^{\frac{-x}{2}}}{2^{\frac{\nu}{2}} \Gamma(\nu/2)}$$

The variable represents the sum of a fixed number of squares of standard normal random variables; the number of terms in the sum is its degrees of freedom ν.

combinations The number of combinations C_k^n is the number of ways of choosing k objects out of a group of n objects, where two choices are considered to be the same if they contain the same k objects.

$$C_k^n = \binom{n}{k} := \frac{n!}{k!(n-k)!}$$

conditional probability The conditional probability is the probability $P(x_2|x_1)$ that an event x_2 will occur provided that an event x_1 has occured.

$$P(x_2|x_1) = \frac{P(x_2 \wedge x_1)}{P(x_1)}$$

correlation coefficient The correlation coefficient r is a measure normalized to the interval $[-1,1]$ describing the *covariance* of two variables X, Y.

$$r = \frac{Cov(X,Y)}{\sqrt{Var(X)Var(Y)}}$$

see also: *autocorrelation coefficient*

covariance The covariance measures whether two variables X and Y vary in the same way.

$$Cov(X,Y) = \langle XY \rangle - \langle X \rangle \langle Y \rangle$$

D

density of states The density of states describes how often a certain state is realized in a particular system.

distribution *see bimodal distribution, binomial distribution, Gamma distribution, Gaussian distribution, normal distribution, and lognormal distribution*

E

ergodic According to BOLTZMANN's hypothesis (1887), asystem trajectory reaches every point with $H = U$. This hypothesis could not be upheld mathematically [64] and in 1911 P. EHRENFEST and T. EHRENFEST formulated that *an ergodic system comes arbitrarily close to any point* $H = U$. [65]

exponential random variable The exponential random variable depending on a parameter α is determined by the following probability density function:

$$f(x) = \begin{cases} \frac{1}{\alpha} e^{-x/\alpha} & x > 0 \\ 0 \quad \text{otherwise} \end{cases}$$

F

fitness In order to commonly describe minimization and maximization problems it is convenient to introduce an abstract fitness which is always to be maximized and, therefore, defined as $F = V$ for a maximization and as $F = -V$ for a minimization problem.

fitness landscape The fitness landscape is a virtual landscape representing the search space. It is uniquely generated by the mutation operator.

frustrated problem An optimization problem is said to be frustrated if two or more contradictory goals are to be optimized

G

Gamma distribution The probability density function describing a gamma random variable depends on two parameters α and β. The distribution is skewed to the right and given by

$$f(x) = \frac{\beta^{\alpha}}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x}$$

Gamma function Generalized factorial function defined as

$$\Gamma(x) = \int_0^{\infty} t^{x-1} e^{-x} dx$$

Gaussian distribution A probability distribution that describes the behavior of many natural and man-made phenomena. The normal distribution is particularly useful because it can be described with a relatively simple equation and analyzed to reveal detailed characteristics of segments of the distribution.

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left[\frac{-(x-\bar{x})^2}{2\sigma^2}\right]$$

GPL GNU Public License, copyright license issued by the Free Software Foundation to protect free software

GMT Generic Mapping Tools; collection of software utilities for 2D and 3D data visualization

GSL GNU Scientific Library; scientific software library providing `C` and `C++` bindings; available under the terms of the GPL at *http://www.*

GUI Graphical User Interface; point and click interface for user/program interaction

H

hypergeometric distribution Given a population of size N, M objects of one type and $N-M$ objects of another type in a sample of n objects chosen without replacement, the number X of type M objects in the sample is hypergeometrically distributed. The mean of the hypergeometric distribution is nM/N. The probability distribution is given by

$$P_k(X) = \begin{cases} \frac{\binom{M}{k}\binom{N-M}{n-k}}{\binom{N}{n}} & 0 \leq k \leq \min(M,N) \\ 0 \quad \text{ortherwise} \end{cases}$$

I

IPC Inter Process Communication – implemented e.g. as System V IPC calls for messages, semaphores, and shared memory

K

kurtosis Kurtosis, a measure of how far the tails of the distribution of a variable x go, is defined as

$$\hat{k} = \frac{\langle (x-\overline{x})^4 \rangle}{\sigma^4}$$

M

Markov Process A stochastic process in which the future distribution of a variable depends only on the variable's current value or its n predecessors. Stock prices, for example, are widely assumed to follow a Markov process.

Metropolis algorithm The Metropolis algorithm is a stochastic optimization algorithm which, unlike gradient strategies, allows downhill steps with a certain probability.

mean The arithmetic mean of a set of N numbers can be calculated as:

$$\bar{x} = \frac{1}{N} \sum_{i=1}^{N} x_i$$

For a distribution $\rho(x)$ of numbers the mean value is defined as the expectation value of x or, in other words, the first moment $\langle X \rangle$ of the distribution:

$$\langle x \rangle = \langle X \rangle = \int \mathrm{d^n x \; x} \, \rho(\mathrm{x})$$

moment The $m-$th moment $\langle X^m \rangle$ of a distribution $\rho(x)$ is the expectation value of the monomial x^m:

$$\langle X^m \rangle = \int \mathrm{dx \; x^m} \, \rho(\mathrm{x})$$

Important moments of a distribution are, for example, the first moment (*mean value*) and a combination of first and second moment: *the variance*.

MPI The Message Passing Interface is a standard specification for message passing libraries (used in parrallel programs) defined by the MPI forum (a broadly based group of parallel computer vendors, library writers, and application specialists.)

multithreading see *thread*

mutation In the scope of this work the term ‘mutation’ describes a change of one ore more variables in parameter space which necessarily induces a move in search space.

mutation operator The mutation operator uniquely describes the set of allowed variable changes in parameter space. The definition of mutation steps generates a neighborhood structure in search space and thus uniquely defines the fitness landscape.

mutex locking variable to ensure exclusive access to shared resources in *multithreaded* programs, a simple form of a *semaphore*

N

normal distribution *see Gaussian distribution*

NP A problem is said to be NP (non deterministic polynomial) if it can not be solved by a deterministic algorithm in polynomial time with respect to the problem size.

NP complete A problem is said to be NP complete if it represents the worst case scenario of an NP problem. If an efficient (meaning polynomial) algorithm can be found for an NP complete problem, *all* NP problems of the same problem class can be solved efficiently. This is reflected in the still open question: $P \stackrel{?}{=} NP$.

O

OneMax Problem The OneMax Problem is in its simple, linear form the task to maximize the number of 1s in a bitstring. The solution is trivial and the problem is easy enough to be analytically solvable.

P

partition function The partition function Z, a dimensionless normalization factor, can be calculated as

$$Z = \int \exp[-\beta H].$$

The term H denotes the Hamilton operator.

PDF short for Portable Document Format, a file format developed by Adobe Systems. PDF captures formatting information from a variety of desktop publishing applications, making it possible to send formatted documents

and have them appear on the recipient's monitor or printer as they were intended. To view a file in PDF format, you need Adobe Acrobat Reader, a free application distributed by Adobe Systems.

Poisson distribution The Poisson distribution is the limit of the binomial distribution when the number of trials goes to infinity. Its variance and mean are both identical to α. The probability distribution is given by

$$P_k(X) = \begin{cases} \frac{\alpha^k e^{-\lambda}}{k!} & k \geq 0 \\ 0 \quad \text{otherwise} \end{cases}$$

postscript PostScript is a programming language optimized for printing graphics and text, a page description language. It was introduced by Adobe in 1985. The purpose of PostScript was to provide a convenient language in which to describe images in a device independent manner.

probability The probability P of an event X describing the likelihood of its occurrence was defined by LAPLACE as [66]

$$P(X) = \frac{\text{Number of elementary events favourable to X}}{\text{Number of all elementary events}}$$

probability density cf. random variable (continuous)

R

random variable (discrete) A random variable ξ is said to be discrete if it can take only finitely or countably many values ξ_i. The ξ_i must satisfy the normalization condition

$$\sum_i \xi_i \stackrel{!}{=} 1$$

random variable (continuous) A random variable ξ is said to be (absolutely) continuous when its distribution function can be represented as

$$P(\xi) = \int_{-\infty}^{\xi} p(t)\,\mathrm{d}t$$

The function $p(\xi)$ is called the probability density which must satisfy

$$\int_{-\infty}^{\infty} p(\xi)\,\mathrm{d}\xi \stackrel{!}{=} 1$$

Rastrigin's function This function is a multimodal function often used for testing purposes. Its global minimum $f(x) = 0$ is at $x_i = 0$. The function is defined as

$$f(x) = nA + \sum_{i=1}^{n} x_i^2 - A\cos(2\pi x_i)$$

The amplitude parameter is typically set to $A = 10$.

S

seeker A seeker actually represents a certain point in the fitness landscape and thus reflects a potential solution to the optimization problem.

selection The selection process replaces inferior seekers by better ones. The exact procedure differs depending on the optimization algorithm.

semaphore integer variable common to different processes or threads, for example, to assure exclusive access to shared resources

Simulated Annealing Simulated Annealing is an extended version of the Metropolis algorithm. During the optimization process, the temperature is lowered according to an annealing schedule.

skewness The skewness of a distribution (positive → right, negative → left) is given by

$$\frac{\langle (x-\bar{x})^3 \rangle}{\sigma^3}$$

spin glass theoretical model describing disordered magnetic materials as an n dimensional lattice of locally and globally coupled spins s; the Hamiltonian is

$$H = -\sum_{\substack{i,j \\ i \neq j}} J_{ij}(s_i \times s_j) \quad s_i, s_j = \pm 1$$

A spin glass is an example for a *frustrated problem*.

standard deviation standard deviation σ of a set of N numbers with mean $\bar{x}$:

$$\sigma = \sqrt{\frac{\sum_i (\bar{x} - x_i)^2}{N-1}}$$

statistical independence If two events x_i and x_j are mutually independent, their correlation is zero:

$$Cor(x_i, x_j) = Cov(x_i, x_j) \equiv 0$$

The inversion, however (if the correlation of two events is zero, they are statistically independent), is true for normally distributed events x only.

statistical weight The statistical weight in the scope of this work denotes the number of realizations of a certain fitness level in a discrete fitness landscape.

T

thread A program can be written to run several tasks in parallel as if they were separate programs. Such a program is said to be multithreaded, since every task constitutes a thread sharing common resources (memory, stack etc.) with all other threads of the program.

V

variance The variance $\sigma^2(x)$ of a distribution $\rho(x)$ is defined as:

$$\sigma^2(x) = \langle (X - \langle X \rangle)^2 \rangle = \langle X^2 \rangle - \langle X \rangle^2 = \int \mathrm{dx}\ (\mathrm{x}^2 - \langle \mathrm{X} \rangle^2)\ \rho(\mathrm{x}).$$

The square root of the variance is called *standard deviation*.

Bibliography

[1] C. Darwin, *On The Origin of Species by Means of Natural Selection, or the Preservation of Favoured Races in the Struggle for Life*. Harmondsworth: Penguin, 1968. 9

[2] R. A. Fisher, *The Genetical Theory of Natural Selection*. Oxford: Clarendon Press, 1930. 9

[3] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of state calculations by fast computing machines," *J. Chem. Phys.*, vol. 21, pp. 1087–1092, 1953. 10, 23

[4] S. Kirkpatrick, C. D. G. Jr., and M. P. Vecchi, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, pp. 671–680, 1983. 10, 27, 59

[5] G. S. Grest, C. M. Soukoulis, and K. Levin, "Cooling rate dependence for the spin-glass ground-state energy: Implications for optimization by simulated annealing," *Phys. Rev. Lett.*, vol. 56, no. 11, pp. 1148–1151, 1986. 10, 28, 59

[6] R. Diekmann, R. Lüling, and J. Simon, "Problem independent distributed simulated annealing and its applications," tech. rep., Department of Mathematics and Computer Science, University of Paderborn, Germany, 1993. 10, 59

[7] G. S. Stiles, “The effect of numerical precision upon simulated annealing,” *Phys. Lett. A*, vol. 185, 1994. 10, 59

[8] R. Desai and R. Patil, “Salo: Combining simulated annealing and local optimization for efficient global optimization,” in *FLAIRS-’96, Key West, FL*, pp. 233–237, 1996. 10, 59

[9] R. Frost and P. Heinemann, “Simulated annealing: A heuristic for parallel stochastic optimization,” in *PDPTA ’97*, 1997. 10, 59

[10] T. Mahnig and H. Muhlenbein, “A new adaptive boltzmann selection schedule sds,” in *Proceedings of the 2001 Congress on Evolutionary Computation CEC2001*, (COEX, World Trade Center, 159 Samseong-dong, Gangnam-gu, Seoul, Korea), pp. 183–190, IEEE Press, 27–30 May 2001. 10, 61

[11] T. Boseniuk, W. Ebeling, and A. Engel, “Boltzmann and Darwin strategies in complex optimization,” *Phys. Lett. A*, vol. 125, pp. 307–310, 1987. 10

[12] Z. Michalewicz, *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996. 10, 29

[13] B. Militzer, M. Zamparelli, and D. Beule, “Evolutionary search for low autocorrelated binary sequences,” *IEEE Trans. Evol. Comp.*, vol. 2, pp. 34–39, 1998. 10, 35

[14] H. Mühlenbein and T. Mahnig, “Mathematical analysis of evolutionary algorithms for optimization,” in *Proceedings of the Third Internatinal Symposium on Adaptive Systems*, (Havanna), pp. 166–185, 2001. 10

[15] H. Rosé, *Evolutionäre Strategien und Multitome Optimierung*. PhD thesis, Humboldt Universität Berlin, 1998. 14, 15, 17, 24, 35, 47, 133

[16] G. P. Williams, *Chaos Theory Tamed*. 1 Gunpowder Square, London EC4A 3DE: Taylor & Francis Ltd., 1997. 17

[17] W. Ebeling, A. Engel, and R. Feistel, *Physik der Evolutionsprozesse*. Akademie-Verlag, Berlin, 1990. 20, 22, 23, 34, 43

[18] R. Feistel and W. Ebeling, *Evolution of Complex Systems*. Kluwer Publ. Dordrecht, 1989. 20, 23

[19] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence*. Ann Arbor, MI: University of Michigan Press, 1975. 20, 28

[20] S. A. Kauffman, *The Origin of Order, Self-Organization and Selection in Evolution*. University of Pennsylvania and The Santa Fe Institute: Oxford University Press, 1993. 20, 36

[21] W. Ebeling and A. Reimann, *Biological Evolution and Statistical Physics*, ch. Evolutionary Strategies for Solving Optimization Problems. Springer, 2002. M. Lässig and A. Valleriani, eds. 21, 64

[22] A. Reimann and W. Ebeling, "Ensemble based control of evolutionary optimization algorithms," *Phys. Rev. E*, vol. 65, no. 046106, 2002. 26, 61, 69, 73, 74

[23] H. Szu and R. Hartley, "Fast simulated annealing," *Phys. Lett. A*, vol. 122, pp. 157–162, 1987. 28, 59

[24] G. T. Barkema and T. MacFarland, "Parallel simulation of the Ising model," *Phys. Rev. E*, vol. 50, pp. 1623–1628, August 1994. 28

[25] B. Andresen and J. M. Gordon, "Analytic constant thermodynamic speed-cooling strategies in Simulated Annealing," *Open Systems & Information Dynamics in Physical and Life Sciences*, vol. 2, pp. 1–12, April 1993. 28

[26] B. Andresen and J. M. Gordon, "Constant thermodynamic speed for minimizing entropy production in thermodynamic processes and simulated annealing," *Phys. Rev. E*, vol. 50, pp. 4346–4351, 1994. 28

[27] B. Andresen, "Parallel implementation of simulated annealing using an optimal adaptive annealing schedule," tech. rep., Physics Laboratory, University of Copenhagen, 1993. 28

[28] G. Ruppeiner *Nucl. Phys. B (Proc. Suppl.)*, vol. 5A, p. 116, 1988. 28

[29] J. H. Holland, "Royal Road functions," *Genetic Algorithm Digest*, vol. 7, August 1993. 28

[30] J. H. Holland and J. S. Reitman, "Cognitive systems based on adaptive algorithms," in *Pattern-Directed Inference Systems* (D. A. Waterman and F. Hayes-Roth, eds.), New York: Academic Press, 1978. 28

[31] D. E. Goldberg, *Genetic Algorithms in Search, Optimization and Machine Learning*. Reading, MA: Addison-Wesley, 1989. 28

[32] D. E. Goldberg and P. Segrest, "Finite Markov chain analysis of Genetic Algorithms," in *Proceedings of the Second International Conference on Genetic Algorithms* (J. J. Grefenstette, ed.), (Hillsdale, NJ), pp. 1–8, Lawrence Erlbaum Associates, 1987. 28

[33] K. A. D. Jong, "Adaptive system design: A genetic approach," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 10, no. 3, pp. 556–574, 1980. 28

[34] K. A. D. Jong, "On using genetic algorithms to search program spaces," in *Proceedings of the Second International Conference on Genetic Algorithms* (J. J. Grefenstette, ed.), (Hillsdale, NJ), pp. 210–216, Lawrence Erlbaum Associates, 1987. 28

[35] K. A. D. Jong, "Are Genetic Algorithms Optimisers?," in *Parallel Problem Solving from Nature* (R. Männer and B. Manderick, eds.), vol. 2, pp. 3–13, Sept. 1992. 28

[36] T. Mahnig and H. Mühlenbein, "Optimal mutation rate using Bayesian priors for estimation of distribution algorithms," in *Proceedings of the 1st Symposium on Stochastic Algorithms, Foundations and Applications*, Springer, 2001. 29

[37] H. Mühlenbein and T. Mahnig, "The Factorized Distribution Algorithm for additively decomposed functions," in *Proceedings of the 1999 Congress on Evolutionary Computation*, pp. 752–759, 1999. 29

[38] M. Steinberg, "Konstruktion von korrelierten, zufälligen Landschaften." Humboldt Universität Berlin. 31

[39] M. J. E. Golay and D. B. Harris, "A new search for skewsymmetric binary sequences with optimal merit factors," *Trans. Inf. Theory (USA)*, vol. 36, no. 5, pp. 1163–1166, 1990. 35

[40] S. Mertens, "Exhaustive search for low-autocorrelation binary sequences," *J. Phys. A*, vol. 29, pp. L473–L481, 1996. 35

[41] S. F. Edwards and P. W. Anderson, "Theory of spin glasses," *J. Phys. F*, vol. 5, p. 965, 1975. 36

[42] C. Flamm, W. Fontana, and I. L. Hofacker, "RNA folding at elementary step resolution," *RNA*, vol. 6, pp. 325–338, 2000. 38

[43] S. Wuchty, W. Fontana, I. L. Hofacker, and P. Schuster, "Complete suboptimal folding of RNA and the stability of secondary structures," *Biopolymers*, vol. 49, pp. 145–165, 1999. 38

[44] W. Fontana, P. F. Stadler, E. G. Bornberg-Bauer, T. Griesmacher, I. L. Hofacker, M. Tacker, P. Tarazona, E. D. Weinberger, and P. Schuster, "RNA folding and combinatory landscapes," *Phys. Rev. E*, vol. 47, pp. 2083–2099, March 1993. 41

[45] R. Feistel, *Anwendungen der Theorie stochastischer Systeme auf lineare und nichtlineare Probleme der Flüssigkeitsphysik*. PhD thesis, Universität Rostock, 1976. 43

[46] D. T. Gillespie, "A general method for numerically simulating the stochastic time evolution of coupled chemical reactions," *J. Comput. Phys.*, vol. 22, pp. 403–434, 1976. 43

[47] D. T. Gillespie, "Exact stochastic simulation of coupled chemical reactions," *J. Phys. Chem.*, vol. 81, pp. 2340–2361, 1977. 43

[48] D. T. Gillespie, "A rigorous derivation of the chemical master equation," *Physica A*, vol. 188, pp. 404–425, 1992.

[49] D. T. Gillespie, *Markov Processes: An Introduction for Physical Scientists*. Academic Press, 1992.

[50] M. A. Gibson and J. Bruck, "Efficient exact stochastic simulation of chemical systems with many species and many channels," *J. Phys. Chem. A*, vol. 104, no. 9, pp. 1876–1889, 2000. 45

[51] M. Eigen and P. Schuster, "The hypercycle," *Naturwiss.*, vol. 64, 65, pp. 541, 341, 1977, 1978. 56, 62

[52] M. Eigen, J. McCaskill, and P. Schuster, "Molecular quasi-species," *J.Phys. Chem*, vol. 92, pp. 6881–6891, 1988. 56, 62

[53] H. Mühlenbein and D. Schlierkamp-Voosen, "Adaptation of population sizes by competing subpopulations," in *International Conference on Evolutionary Computation*, (Nagoya, Japan), pp. 330–335, 1996. 59

[54] Salamon, Nulton, Harland, Pederson, Ruppeiner, and Liau, "Simulated annealing with constant thermodynamic speed," *Computer Physics Communications*, pp. 423–428, 1988. 59

[55] W. Ebeling, L. Molgedey, and A. Reimann, "Stochastic urn models of innovation and search dynamics," *Physica A*, vol. 287, pp. 599–612, 2000. 61, 64, 65

[56] M. Nowak and P. Schuster, "Error thresholds of replication in finite populations: Mutation frequencies and the onset of Muller's ratchet," *J. Theor. Biol.*, vol. 137, pp. 375–395, 1989. 63

[57] T. Bäck and M. Schutz, "Intelligent mutation rate control in canonical genetic algorithms," in *Proceedings of the 9th International Symposium, ISMIS*, (Zakopane (Poland)), pp. 158–167, Springer-Verlag, Berlin, June 1996. 63

[58] G. Ochoa, I. Harvey, and H. Buxton, "Error thresholds and their relation to optimal mutation rates," in *European Conference on Artificial Life*, pp. 54–63, 1999. 63

[59] G. Ochoa and I. Harvey, "Recombination and error thresholds in finite populations," in *Foundations of Genetic Algorithms 5* (W. Banzhaf and C. Reeves, eds.), pp. 245–264, San Francisco, CA: Morgan Kaufmann, 1999. 63

[60] W. Ebeling and A. Reimann, "Ensemble-based control of search dynamics with application to string optimization," *Z. Phys. Chem.*, vol. 216, no. 01, pp. 065–075, 2002. 68, 135

[61] B. Stroustrup, *The C++ Programming Language*. Reading: Addison-Wesley, 3rd ed., 1997. 77

[62] S. B. Lippman, *The C++ Primer*. Reading: Addison-Wesley, 2nd ed., 1991. 77

[63] E. Gode, *ANSI C++ (kurz & gut)*. Köln: O'Reilly, 1998. ISBN 3-89721-205-6. 77

[64] T. Haar, *Elements of Statistical Mechanics*. New York: Rinehart, 1954. 141

[65] R. Becker, *Theorie der Wärme*. Berlin, Heidelberg: Springer, 3rd ed., 1985. 141

[66] I. N. Bronshtein and K. A. Semendyayev, *Handbook of Mathematics*. Springer, reprint of the third ed., 1998. 146

[67] T. Asselmeier, W. Ebeling, and H. Rosé, "Evolutionary strategies of optimization," *Phys. Rev. B*, vol. 56, pp. 1171–1180, 1997.

[68] T. Asselmeier, *Schrödinger-Operatoren und Evolutionäre Strategien*. PhD thesis, Humboldt Universität Berlin, 1997.

[69] M. Conrad and W. Ebeling, "M. V. Volkenstein, evolutionary thinking and the structure of fitness landscapes," *BioSystems*, vol. 27, pp. 125–128, 1992.

[70] P. E. F. Carter Jr, "The generation and application of random numbers."

[71] L. Peleti, "Quasispecies evolution in general mean-field landscapes," *Europhys. Lett.*, 2000.

[72] W. E. Hart, *Adaptive Global Optimization with Local Search*. PhD thesis, University of California, San Diego, 1994.

[73] C. O. Book, *Mathematical Optimization*. Computational Science Education Project, 1995.

[74] H. Mühlenbein and T. Mahnig, "FDA - A scalable evolutionary algorithm for the optimization of additively decomposed functions," *Evolutionary Computation*, vol. 7, pp. 353–376, 1999.

[75] M. Fekete, I. L. Hofacker, and P. F. Stadler, "Prediction of RNA base pairing probabilities using massively parallel computers," *J. Comput. Biol.*, vol. 7, pp. 171–182, 2000.

[76] C. Flamm, I. L. Hofacker, and P. F. Stadler, "RNA in silico: The computational biology of RNA secondary structures," *Adv. Complex Syst.*, vol. 2, pp. 65–90, 1999.

[77] I. L. Hofacker, P. Schuster, and P. F. Stadler, "Combinatorics of RNA secondary structures," *Discr. Appl. Math.*, vol. 88, pp. 207–237, 1998.

[78] M. Gen and R. Cheng, *Genetic Algorithms & Engineering Optimization*. New York, Chichester, Weinheim, Brisbane, Singapore, Toronto: Wiley-Interscience (John Wiley & Sons Inc.), 2000.

[79] H.-M. Voigt, W. Ebeling, I. Rechenberg, and H.-P. Schwefel, eds., *Parallel Problem Solving from Nature - PPSN IV*, Springer, Sept. 1996. International Conference on Evolutionary Computation - The 4th International Conference on Parallel Problem Solving from Nature, Berlin, Germany.

[80] V. Nissen, *Einführung in evolutionäre Algorithmen*. Braunschweig, Wiesbaden: Vieweg, 1997. (computational intelligence); ISBN 3-528-05499-9.

[81] E. Chattoe, "Just how (un)realistic are Evolutionary Algorithms as representations of social processes?," *Journal of Artificial Societies and Social Simulation*, vol. 1, no. 3, 1998.

[82] I. Rechenberg, *Evolutionsstrategie: Optimierung Technischer Systeme nach Prinzipien der Biologischen Evolution*. Stuttgart: Frommann-Holzboog, 1973.

[83] T. Bäck, *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms*. Oxford University Press, 1996. ISBN: 0 1950997 10.

Index

Symbols

α-helix 39
β-sheet 39
ε 28

A

adaptation 9, 73
Adenine 38
algorithm 11
analytic solution 13
Anderson 36
Andresen 28, 60
annealing schedule 59
annealing speed 28
autocorrelation
  coefficient 7, 18, 19, 47
  function 17, 19, 47
autocovariance 8, 17

B

base pair 38
Boltzmann 141
  constant 7, 24
  distribution 24
  factor 24, 25
  strategy 10, 23, 52
Boltzmann-Darwin Strategy 10
Bruck 45
building block 35

C

chromosome representation 29
communication 20
compromise 9
Copyright 131
correlation 31, 35
  length 7, 19
correlogram 19
crossover operator 29
Cytosine 38

D

Darwin Strategy 21, 24, 53
de Jong 28
density of states 7, 15
DNA 38

E

Edwards 36

Ehrenfest 141
eigenvalue 23
energy 7
Engel 34
Engel Sequence 34
ensemble entropy 7, 69
ensemble size 7
ensemble variability 8, 64
entropy 8, 17
error threshold 56, 62, 72
evolution rate 7, 61
evolutionary
  algorithm 10, 28
  window 46, 52
evolutionary window 70

F

Feistel 34, 43
Fisher-Eigen equation 22
fitness 7, 25
  function 34
    frustrated 34
  landscape . . 12, 31, 33, 35, 37, 46
    discrete 14, 33
    properties 14
Fourier
  spectrum 32
  transform 33
free energy 39, 40
frequency 15
Frustrated Periodic Sequence 10, 34, 77
frustration . . . . . *see* fitness function, frustrated

G

Gaussian
  landscape 35, 116
  random number 31
genetic
  algorithm 28
genotype 20, 37
Gibson 45
Gillespie 43, 46
Gillespie-Feistel algorithm 43
GMT 130
Golay 35
Goldberg 28
GPL 129, 131
gradient search 13, 24
GSL 131
Guanine 38
GUI 78, 131

H

hairpin 39
Hamilton operator 60

Hamiltonian 7, 36
Hamming distance 19
heat capacity 28, 59
Heisenberg matrix 23
hill climbing 13
Holland 28

I

inverse temperature 7
IPC 131
Ising model 28

K

Kauffman 36, 37
Kirkpatrick 27

L

LABS 10, 77
Laplace 146
loop 39

M

Mühlenbein 61
Mahnig 61
master equation 24
mean value 8
memory leak 131
Merit-factor 36
method
  direct 44
  first reaction 45
  next reaction 45
Metropolis Algorithm 10, 23, 27
MPI_generate 116
mutation 12, 20
  operator 80
  rate 22
mutex 131

N

NK model 36
non polynomial . . . *see* NP complete
Nowak 63
NP complete 11

O

occupation number 8
Ochoa 63
optimization 9, 11, 25

P

parameter 9
  space 13, 47
partition function 8, 25
phenotype 20
pipe 131
Polio Virus Type 1 133
population
  genetics 36
  size 58, 63

potential 8
potential energy 8
probability 7, 15
density 7, 16
pseudo loop 40
purines 38
pyrimidines 38

Q

Qt GUI toolkit 131

R

random 12
random walk 47
relative ensemble dispersion . . . 7, 66
relaxation coefficient 28, 59
RNA
folding kinetics 38
secondary structure . . . 10, 38, 80
sequence 38, 39

S

sample 47
Schuster 63
second law of thermodynamics . . . 23
selection 12, 20
probability 7, 26
signal 131
Sim_Engel 77
Sim_Labs 77
Sim_RNA 77, 80
Simulated Annealing 10, 21, 27
slot 131
source code 77
spectrum 47
spin glass 28, 36
standard deviation 8
Standard Deviation Schedule 61
steepest descent 13
Steinberg 31
step function 27
STL 131
stochastic 11
strategy
mixed 25
structure
primary 38
secondary 38, 40

T

temperature 8, 27
thermodynamical system 15
thermodynamics 27
thread 78
Thymine 38
time 8
tournament selection 27
twisted pair 38

U

Uracil 38

V

variance 8, 17
vector class 131
Vis5D 130

W

Watson-Crick 38
weight, statistical 8, 17

X

XMGrace 130

Appendix C

Acknowledgment

The author wishes to thank the **‘Deutsche Forschungsgemeinschaft’** who has financially supported this work in the framework of the special research field ‘Sonderforschungsbereich 555 – Komplexe Nichtlineare Prozesse’.
I am furthermore indebted to:

- **Prof. Dr. Werner Ebeling** for supervising this work and numerous fruitful discussions,
- **Dr. Lutz Molgedey** for his seemingly endless patience in repeatedly explaining gory mathematical details,
- **Dipl. BW Claudia Lehmann** for her moral and practical support in everyday life, and
- **Ruth Perkins** for proof-reading this document (in an intermediate state; so I had enough time to introduce new mistakes) and answering uncounted style and grammar-related questions.

Author's Publication List

1. W. Ebeling, L. Molgedey, and A. Reimann, "Stochastic urn models of innovation and search dynamics", *Physica A*, vol. 287, pp. 599–612, 2000.

2. W. Ebeling and A. Reimann, in *Biological Evolution and Statistical Physics*, ch. Evolutionary Strategies for Solving Optimization Problems. Springer, 2002. M. Lässig and A. Valleriani, eds.

3. W. Ebeling and A. Reimann, "Ensemble-based control of search dynamics with application to string optimization", *Z. Phys. Chem.*, vol. 216 (01), pp. 065–075, 2002.

4. A. Reimann and W. Ebeling, "Ensemble based control of evolutionary optimization algorithms", *Phys. Rev. E*, vol. 65 (046106), 2002.

Selbständigkeitserklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig angefertigt und keine weiteren als die gegebenen Hilfsmittel verwendet zu haben.

_____

Axel Reimann, Berlin den: