KAPITAŁ LUDZKI
NARODOWA STRATEGIA SPÓJNOŚCI

Politechnika Wrocławska

UNIA EUROPEJSKA
EUROPEJSKI
FUNDUSZ SPOŁECZNY

ROZWÓJ POTENCJAŁU I OFERTY DYDAKTYCZNEJ POLITECHNIKI WROCŁAWSKIEJ

Wrocław University of Technology

Advanced Informatics and Control

Adam Janiak, Maciej Lichtenstein

ADVANCED ALGORITHMS IN COMBINATORIAL OPTIMIZATION

Wrocław 2011

Projekt współfinansowany ze środków Unii Europejskiej w ramach
Europejskiego Funduszu Społecznego

Wrocław University of Technology

Advanced Informatics and Control

Adam Janiak, Maciej Lichtenstein

ADVANCED ALGORITHMS IN COMBINATORIAL OPTIMIZATION

Wrocław 2011

Copyright © by Wrocław University of Technology
Wrocław 2011

Reviewer: Andrzej Kasprzak

ISBN 978-83-62098-87-3

Published by PRINTPAP Łódź, www.printpap.pl

Contents

Glossary

ACO Ant colony optimization

AIS Artificial immune systems

APC Antigen presenting cell

CSA Clonal selection algorithm

DIVM Displaced inversion mutation

DM Displacement mutation

GA Genetic algorithm

GC Germinal center

GDA Great deluge algorithm

IM Insertion mutation

IVM Inversion mutation

KNAPSACK Knapsack problem

MHC Major histocompatibility complex

NK Natural killer cells

OBX Order-based crossover

PART Partition problem

PBX Position-based crossover

PMX Partially mapped crossover

RRT Record-to-record travel algorithm

SA Simulated annealing

SM Scramble mutation

TA Threshold accepting algorithm

TS Tabu search

TSP Traveling salesman problem

Chapter 1

Introduction

The process of optimization is the process of obtaining the *best*, if it is possible to measure and change what is *good* or *bad*. In practice, one wishes the *most* or *maximum* (e.g., salary, profit) or the *least* or *minimum* (e.g., expenses, energy). Therefore, the word *optimum* is taken to mean *maximum* or *minimum* depending on the circumstances; 'optimum' is a technical term which implies quantitative measurement and is a stronger word than *best* which is more appropriate for everyday use. Likewise, the word *optimize*, which means to achieve an optimum, is a stronger word than *improve*. Optimization theory is the branch of mathematics encompassing the quantitative study of optima and methods for finding them. Optimization practice, on the other hand, is the collection of techniques, methods, procedures, and algorithms that can be used to find the optima.

Optimization problems occur in most disciplines like engineering, physics, mathematics, economics, administration, commerce, social sciences, and even politics. Optimization problems abound in the various fields of engineering like electrical, mechanical, civil, chemical, and building engineering. Typical areas of application are modeling, characterization, and design of devices, circuits, and systems; design of tools, instruments, and equipment; design of structures and buildings; process control; approximation theory, curve fitting, solution of systems of equations; forecasting, production scheduling, quality control; maintenance and repair; inventory control, accounting, budgeting, etc. Some recent innovations rely almost entirely on optimization theory, for example, neural networks and adaptive systems.

Most real-life problems have several solutions and occasionally an in-

finite number of solutions may be possible. Assuming that the problem at hand admits more than one solution, optimization can be achieved by finding the best solution of the problem in terms of some performance criterion. If the problem admits only one solution, that is, only a unique set of parameter values is acceptable, then optimization cannot be applied.

This book is devoted to the *Nature-inspired* methods of solving hard combinatorial optimization problems. The scope of the book starts with the introduction to the optimization and, in particular, with the definition of combinatorial optimization problems (Chapter 2). Chapter 2 also deals with the elements of the Computation Complexity Theory, and summarizes the methods that can be applied to solve combinatorial optimization problems. Chapters 3-7 describe five Nature-inspired methods of problem solving. These methods are simulated annealing, tabu search, genetic algorithms, ant colony optimization, and artificial immune systems, respectively. Every method is described and an example is given. The book is concluded with the references to other books and papers that can be good start to for deeper understanding of the methods described on the forthcoming pages.

This book is not intended to be complete or precise. It is a textbook summarizing the facts about the subjects of the course "Advanced algorithms in combinatorial optimization" that is given on the Faculty of Electronics, Wrocław University of Technology, to the computer engineering students. All details of the subjects are given during the lectures of the mentioned above course, and this textbook is only a "helpful hand" for those that do not attend to the classes much often. On the other hand it can be helpful for the students of other fields related to the algorithmic issues of problems solving.

Chapter 2

Combinatorial optimization

2.1 Introduction

The general optimization problem can be viewed as a couple (S, f), where S is the set of problem feasible solutions, and $f : S \to \mathbb{R}$ is the objective function that assigns to each solution $s \in S$ a real (or integer) number, which evaluates the solution worth. The aim of the optimization problem can be stated as: find the element $s* \in S$ for which function f is maximized or minimized. Since minimization and maximization are very similar (the problem of maximization of function f is equivalent to the problem of minimization of function $-f$), we will focus our attention on minimization problems.

Depending on the properties of the set S and the function f we can obtain various classes of optimization problems, such as continuous optimization, discrete optimization, etc. One of this classes are *combinatorial optimization problems* that are the main topic of this book.

In any **combinatorial optimization problem the set S is finite, and variables that define solutions are discrete in nature.** The popularity of combinatorial optimization problems stems from the fact that in many real-world problems the objective function and constraints are of different nature (nonlinear, nonanalytic, black box, etc.) whereas the search space is finite.

Any (combinatorial) optimization problem can be stated as follows.

Definition 1 *Given the set S and the function f, find global minimum of function f.*

The definition of the global minimum is as follows.

Definition 2 *The global minimum to the problem* (S, f) *is an element* $s* \in S$ *such that* $\forall s \in S : f(s*) \leq f(s)$.

Since the set S in any combinatorial optimization problem has finite number of elements, there always exists a procedure for its solution. This procedure, called *complete enumeration* or *brute force method*, iterates through every solution $s \in S$ for which it calculates the function value $f(s)$, and then returns a solution or a set of solutions with smallest calculated function value. It is clear that this method returns the global minimum or the set of global minima. The time required by this procedure is linearly dependent of the number of elements in the set S and the time required to calculate the single value of function f. So if we can calculate the function value for every s in finite time, then the brute force method runs in finite time. On the other hand, time required for the brute force method may be unacceptable form practical point of view. We will discuss this issue in what follows.

2.2 Examples of combinatorial problems

There are many combinatorial problems that arose from real-life issues. In this section we present some of them (the most important ones). Not all problems stated here are optimization problems, but as we see later on, they are also combinatorial in nature, and have many relations to combinatorial optimization problems.

The problems of partition the given set into some numbers of disjoint subsets have many forms. We present in this section some of them. The first problem we are going to present is called just a *Partition Problem* and usually is denoted by PART. Its definition is as follows.

Definition 3 (PART) *There is a given the set* $N = \{1, ..., n\}$ *of* n *elements. Each element* $j \in N$ *has a value* x_j *which is non negative integer, such that all* x_j *sum up to an even value, i.e.,* $\sum_{j \in J} x_j = 2B$*, where* B *is an integer. Is there a subset* $X \subseteq N$ *such that* $\sum_{j \in X} = B$*?*

The problem seems quite easy to solve; let us consider the following example.

We are given 5 elements (set $N = \{1, 2, 3, 4, 5\}$) with values x_j, $j \in N$ summarized in the following table:

j	x_j
1	3
2	5
3	6
4	4
5	2

It is easy to calculate that in the above example the value of $B = \frac{1}{2}\sum_{j \in J} x_j = 10$, and thus we are looking for subset X for which sum of elements values equals 10. After quick look at the above table we can say that the answer to the question stated in the problem is "yes". The example subset X contains elements 3 and 4, i.e. $X = \{3, 4\}$ because $x_3 + x_4 = 6 + 4 = 10$. It is also easy to observe that the set $N \backslash X = \{1, 2, 5\}$ also gives the "yes" answer to the problem's question. That's why this problem is called the "Partition Problem". We are looking for partition of the set N into two subsets X and $N \backslash X$, with the same value of the sum of element's values.

At first look this problem is easy to solve. But will it be easy to partition the set of $n = 20$ elements? What about the set of $n = 1,000$ elements? Before we give an answer to these questions we will stop for a while and consider the problem's solution space.

The first question is "*what is the solution of the partition problem?*" And "*how many such solutions are possible?*" The answer to the first question is simple: the solution to the PART is the subset of the set N. The second question need some calculations, but it is known that there are exactly 2^n different subsets of the set of n elements.

Now, we can think about simple, complete enumeration algorithm that solve the PART. This algorithm can be stated as follows.

```
1: for each X ⊆ N do
2:    if ∑_{j∈X} = B then
3:       return "yes" answer
4:    end if
5: end for
6: return "no" answer
```

In the worst case, the algorithm have to check every subset X of the set N. Note that for $n = 20$ the number of such subsets equals $2^n = 2^{20} = 1,048,576$. That's over the 1 million solutions to check! Yes, but nowadays computers are fast! Assume that the computer can check each subset in $1\mu s$. Thus the entire algorithm will run a little longer than a 1 second.

And what about $n = 100$? Let's see... $2^n = 2^{100} = ...$

$$1,267,650,600,228,229,401,496,703,205,376$$

WOW! That is a really big number! And the computer will spend something like 1.26×10^{24} seconds, i.e., something like $40,000,000,000,000,000$ years to complete this task! Well no, our world will be gone much sooner.

One can say that the computers are getting faster very quickly, so this issue will not exist in near future. Well, consider one billion times faster computer. Will it help? Yes, the processing will take now "only" 40 million years ;). And finally consider the PART with $n = 1000$ elements ... so you see that the issue will not be solved by faster computers. Finally, we have to state that the PART is an example of decision problem. The decision problem is a problem that always has "yes" or "no" answer.

Another interesting problem can be defined as follows.

Definition 4 (KNAPSACK) *There is a given set $N = \{1, ..., n\}$ of n elements. Each element $j \in N$ has given its size $a_j \geq 0$ and its value $w_j \geq 0$. There is also given a value B which represents the knapsack capacity. The problem is to find the subset $X \subseteq N$ such that*

$$\sum_{j \in X} a_j \leq B, \tag{2.1}$$

and the value of

$$V = \sum_{j \in X} w_j$$

is maximal.

The problem is to find the subset of elements that fit into a given knapsack (the inequality (2.1)) with maximal total value. It is easy to notice that the solution space of this problem also contains 2^n elements (all the subsets of the set N) but not all solutions in this set are feasible, i.e., they do not satisfy inequality (2.1).

The next problem we are going to present is called the *Traveling Salesman Problem* (TSP for short) and has different solution space, than the problems two previously defined.

Definition 5 *TSP There is a given set $N = \{1, ..., n\}$ of n cities and the matrix $D = (d_{ij} : i \in N, j \in N, i \neq j)$ of distances between them, where $d_{ij} \geq 0$ is the distance from the city i to the city j. The problem is to find a tour that minimizes the total distance. The tour have to visit all the cities in N and starts and ends in the same city.*

The tour in the TSP can be represented by the *permutation* of the set N. Let $\pi = (\pi(1), ..., \pi(n))$ denotes the permutation of the elements of the set N, where $\pi(j)$ is the jth element of this permutation. For example if $\pi = (4, 2, 3, 1)$ then the route starts with city 4, then visits city 2, next city 3, next city 1 and finally returns to city 4. For such representation the TSP can be stated as follows.

Find permutation π such that the value of

$$F = \sum_{j=1}^{n-1} d_{\pi(j)\pi(j+1)} + d_{\pi(n)\pi(1)},$$

is minimized.

It is easy to notice that since the tour is in fact a cycle (starts and ends in the same city) then the first element of π can be fixed to an arbitrary city. Thus the problem solution space will contain all the permutations of the $n - 1$ elements, and there is $(n - 1)!$ such permutations. This factorial function $(n - 1)!$ increases very fast with the increase of the value of n. For example for $n = 10$ it has a value of 3,628,80, whereas for $n = 20$ it has a value of 121,645,100,408,832,000.

2.3 Introduction to Computational Complexity Theory

In previous section we stated that it is possible to find an algorithm for the solution of "virtually any" combinatorial optimization problem, however, the simplest brute force approach may (and usually is) impractical. The reason for that is that any algorithm requires the two kind of resources to execute: time and space. The time complexity of an algorithm is the number of steps required to solve a problem of size n, where n is the count of problem input data (e.g. number of elements in PART or KNAPSACK, number of cities in TSP, etc.).

2.3.1 Complexity of algorithms

The goal in the determination of the computational complexity of an algorithm is not to obtain its exact running time, but an asymptotic bound on the step count of its execution. The Landau notation (or *Big-O notation*) makes use of such asymptotic analysis. It is one of the most popular notations in the analysis of algorithms.

Definition 6 (Big-O notation) *An algorithm has a complexity* $t(n) = O(g(n))$ *if there exist positive constants* n_0 *and* C *such that* $t(n) \leq Cg(n), \forall n > n_0$.

In this case, the function $t(n)$ is upper bounded by the function $g(n)$. The Big-O notation can be used to compute the time or the space complexity of an algorithm. Some properties of the Big-O notation are summarized below.

Property 1 *If* $a(n) = O(t(n))$ *and* $b(n) = O(g(n))$ *then* $a(n)+b(n) = O(t(n)+g(n)) = O(\max\{t(n), g(n)\})$.

Property 2 *If* $a(n) = O(t(n))$ *and* $b(n) = O(g(n))$ *then* $a(n) \times b(n) = O(t(n) \times g(n))$.

Property 3 *If positive polynomial* $p(n)$ *is of degree* k *then* $p(n) = O(n^k)$.

Remind that positive polynomial $p(n)$ of degree k is a function:

$$p(n) = a_k n^k + a_{k-1} n^{k-1} + ... + a_2 n^2 + a_1 n + a_0,$$

where $a_j \geq 0$, $j = 1, ..., k-1$ and $a_k > 0$.

The complexity of algorithms classifies them into main two classes *polynomial-time algorithm* and *above-polynomial-time algorithms*.

Definition 7 (polynomial-time algorithm) *An algorithm is a polynomial-time algorithm if its complexity is* $O(p(n))$*, where* $p(n)$ *is a positive polynomial function of the problem size* n.

Definition 8 (above-polynomial-time algorithm) *An algorithm is an above-polynomial-time algorithm if its complexity cannot be bounded by any polynomial* $p(n)$.

Note, that for example an algorithm with time complexity $O(n \log n)$ is polynomial time algorithm because it can be bounded by $O(n^2)$ for any base of the logarithm.

A special case of the above-polynomial-time algorithms are the *exponential-time algorithms* which can be defined as follows.

Definition 9 (exponential-time algorithm) *An algorithm is an exponential-time algorithm if it is above-polynomial-time algorithm and its complexity is $O(a^n)$, where a is a real constant strictly greater than 1.*

From the practical point of view, either polynomial as well as above-polynomial time algorithm may be not efficient, taking into account that exponential-time algorithms are usually useless to solve moderate size of problem instances. For example brute force method introduced in previous section is impractical, but also an algorithm with the time complexity of $O(n^{100})$ usually cannot be applied in practice.

The significant part of the computational complexity theory deals not with the complexity of algorithms, but with the complexity of the problems.

2.3.2 Complexity of problems

The complexity of a problem is "in a sense" equivalent to the complexity of the best possible algorithm solving that problem. From the theoretical point of view, the problem is *tractable* (which means "easy to solve") if there exist a polynomial-time algorithm for its solution. On the other hand, the problem is *intractable* (difficult to solve) if such algorithm doesn't exist.

The computational complexity theory deals in its basis with languages (a strings over some alphabet) and Touring machines. However, all the results are applicable to the decision, as well as optimization problems.

To be more correct, the results of the computational complexity theory are applicable to decision problems, however, every optimization problem can be "converted" to its decision version. Such conversion is done in the following way.

If a optimization problem is defined as "minimize (or maximize) a function $f(s)$ over the set S", then its decision version of that problem is "given an integer F, is there a solution $s' \in S$ such that $f(s') \leq F$ (or $f(s') \geq F$ for maximization)?".

The main aspect of the complexity of problems is to categorize them into complexity classes. A complexity class is a set of all problems that can

be solved using a given amount of time or space. The are two main classes that categorize problems according to their time complexity: $\mathcal{P}$ and $\mathcal{NP}$.

Definition 10 (the class $\mathcal{P}$) *The given problem P belongs to the class $\mathcal{P}$ if there exists a polynomial-time algorithm for its solution.*

Definition 11 (the class $\mathcal{NP}$) *The given problem P belongs to the class $\mathcal{NP}$ if there exists a polynomial-time algorithm that verifies answer to the problem for a given (guessed) solution.*

It is quite obvious that $\mathcal{P} \subseteq \mathcal{NP}$. The question whether $\mathcal{P} = \mathcal{NP}$ is the most important open question of the whole theory (many research has been done in this matter and the question is still open, however, it seems that the correct answer is "no", i.e. $\mathcal{P} \neq \mathcal{NP}$).

To define the most important class in the whole theory we have to define an additional term – the *polynomial reduction*.

Definition 12 (the polynomial reduction) *The polynomial reduction of the decision problem P_1 into the decision problem P_2 (which will be denoted as $P_1 \propto P_2$) is a function t that express every datum in P_2 by the data of P_1 and satisfies the following conditions:*

- *the values of t can be calculated in polynomial time for every instance of P_1,*
- *the constructed instance of P_2 has the "yes" answer if and only if the source instance of P_1 also has "yes" answer.*

The construction of the polynomial reduction is not an easy task. For the details we refer the reader to the literature.

Finally we can define the most important classes of the theory of computational complexity.

Definition 13 (the class of $\mathcal{NP}$-complete problems) *The decision problem P is $\mathcal{NP}$-complete if $P \in \mathcal{NP}$ and for any other problem $Q \in \mathcal{NP}$, $Q \propto P$.*

From the definition of the polynomial reduction follows that if a polynomial-time algorithm exists to solve an $\mathcal{NP}$-complete problem, then all problems of class $\mathcal{NP}$ may be solved in polynomial time. This, however, seems to

be not the case, thus no polynomial-time algorithms seems to exist for any $\mathcal{NP}$-complete problem.

The $\mathcal{NP}$-completeness is defined for the decision problems. The similar term is defined for the optimization problems.

Definition 14 (the class of $\mathcal{NP}$-hard problems) *The optimization problem is $\mathcal{NP}$-hard if its decision version is $\mathcal{NP}$-complete.*

The relations between the complexity classes of the problems are depicted in Figure 2.1.

Note, that every problem defined in the previous section (PART, KNAPSACK, TSP) belongs to the class of $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problems.

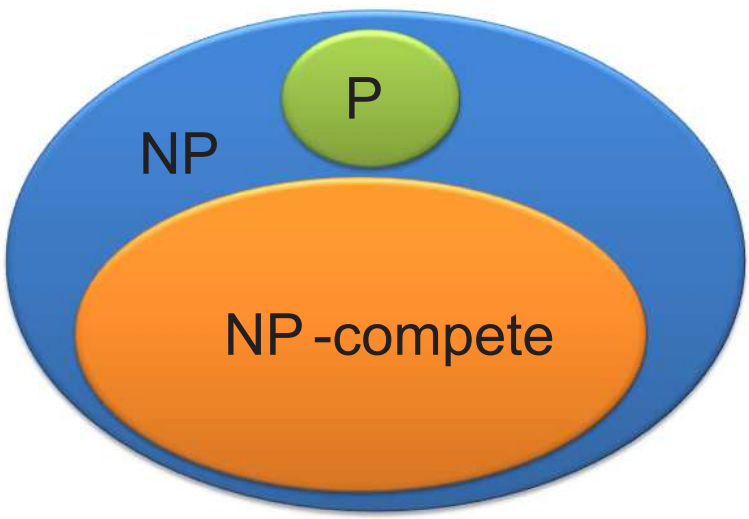

Figure 2.1: The relation of complexity classes of problems

Most of the real-world optimization problems are NP-hard for which provably efficient algorithms do not exist. They require exponential time (unless $\mathcal{P} = \mathcal{NP}$) to be solved to optimality.

2.4 Combinatorial optimization methods

Following the complexity of the problem, it may be solved by an exact method or an approximate method. Exact methods obtain optimal solutions and guarantee their optimality. For NP-complete problems, exact algorithms are non polynomial time algorithms. Approximate (or heuristic) methods may generate near optimal solutions in a reasonable time for practical use, but there is no guarantee of finding a global optimal solution.

2.4.1 Exact methods

In the class of exact methods one can find the following classical algorithms: *dynamic programming*, *branch and bound*, *constraint programming*, and $A*$ family of search algorithms developed in the artificial intelligence community. Those enumerative methods may be viewed as tree search algorithms. The search is carried out over the whole interesting search space, and the problem is solved by subdividing it into simpler problems.

Dynamic programming

Dynamic programming is based on the recursive division of a problem into simpler subproblems. This procedure is based on the Bellman's principle that says that "*the sub policy of an optimal policy is itself optimal*". This stage wise optimization method is the result of a sequence of partial decisions. The procedure avoids a total enumeration of the search space by pruning partial decision sequences that cannot lead to the optimal solution.

Branch and bound

The branch and bound algorithm and $A*$ are based on an implicit enumeration of all solutions of the considered optimization problem. The search space is explored by dynamically building a tree whose root node represents the problem being solved and its whole associated search space. The leaf nodes are the potential solutions and the internal nodes are subproblems of the total solution space. The pruning of the search tree is based on a bounding function that prunes subtrees that do not contain any optimal solution.

Constraint programming

Constraint programming is a language built around concepts of tree search and logical implications. Optimization problems in constraint programming are modeled by means of a set of variables linked by a set of constraints. The variables take their values on a finite domain of integers. The constraints may have mathematical or symbolic forms.

Exact methods can be applied to small instances of difficult problems. For NP-hard optimization problems the order of magnitude of the maximal size of instances that state-of-the-art exact methods can solve to optimality is up to 100. Moreover, to achieve this some of the exact algorithms have to implemented on large networks of workstations.

2.4.2 Approximation and heuristic algorithms

In the class of approximate methods, three subclasses of algorithms may be distinguished: *approximation algorithms*, *approximation schemes*, and *heuristic algorithms*. Unlike heuristics, which usually find "good" solutions in a reasonable time, approximation algorithms provide provable solution quality and provable run-time bounds.

Approximation algorithms

In approximation algorithms, there is a guarantee on the bound of the obtained solution from the global optimum. An ε-approximation algorithm generates an approximate solution s not less than a factor ε times the optimum solution $s*$.

Approximation schemes

There are two classes of approximation schemes (which are in fact families of algorithms), namely:

Polynomial-time approximation scheme (PTAS). An algorithm scheme is a PTAS if it is polynomial-time $(1+\epsilon)$-approximation algorithm for any fixed $\epsilon > 0$. Note that running time of the algorithm may depend exponentially on the value of ϵ ($1/\epsilon$ to be more precise).

Fully polynomial-time approximation scheme (FPTAS). An algorithm scheme is a FPTAS if it is polynomial-time $(1+\epsilon)$-approximation algorithm

for any fixed $\epsilon > 0$, and it is polynomial either in the problem size as well as in $1/\epsilon$.

Heuristics and metaheuristics

Heuristics find "good" solutions on large-size problem instances. They allow to obtain acceptable performance at acceptable costs in a wide range of problems. In general, heuristics do not have an approximation guarantee on the obtained solutions. They may be classified into two families: specific heuristics and metaheuristics. Specific heuristics are tailored and designed to solve a specific problem and/or instance. Metaheuristics are general-purpose algorithms that can be applied to solve almost any optimization problem. They may be viewed as upper level general methodologies that can be used as a guiding strategy in designing underlying heuristics to solve specific optimization problems. This book is mainly focused on the metaheuristic approaches.

Chapter 3

Simulated annealing and its extensions

3.1 Introduction

Simulated annealing (SA) is a random-search technique which exploits an analogy between the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and the search for a minimum in a more general system; it forms the basis of an optimization technique for combinatorial and other problems.

Simulated annealing was developed in 1980 to deal with highly nonlinear problems. SA approaches the global maximization problem similarly to using a bouncing ball that can bounce over mountains from valley to valley. It begins at a high *temperature* which enables the ball to make very high bounces, which enables it to bounce over any mountain to access any valley, given enough bounces. As the temperature declines the ball cannot bounce so high, and it can also settle to become trapped in relatively small ranges of valleys. A generating distribution generates possible valleys or states to be explored. An *acceptance distribution* is also defined, which depends on the difference between the function value of the present generated valley to be explored and the last saved lowest valley. The acceptance distribution decides probabilistically whether to stay in a new lower valley or to bounce out of it. All the generating and acceptance distributions depend on the temperature. It has been proved that by carefully controlling the rate of

cooling of the temperature, SA can find the global optimum. However, this requires infinite time.

3.2 Simulated annealing (SA)

Simulated annealing's major advantage over other methods is an ability to avoid becoming trapped in local minima. The algorithm employs a random search which not only accepts changes that decrease the objective function f (assuming a minimization problem), but also some changes that increase it. The latter are accepted with a probability:

$$P = e^{\frac{-\Delta}{T}},$$

where Δ is the increase of the objective function, and T is a control parameter, which by analogy with the original application is known as the system *temperature* irrespective of the objective function involved. The implementation of the basic SA algorithm is straightforward. Algorithm 1 shows its structure.

Algorithm 1 Basic scheme of simulated annealing

1: generate the initial solution s
2: set solutions $current = s$, and $best = s$,
3: set the initial temperature T
4: **while** stopping condition is not satisfied **do**
5: generate perturbed solution p on the basis of $current$ solution
6: calculate $\Delta = f(p) - f(current)$
7: **if** $\Delta \leq 0$ **then**
8: $current = p$
9: **else**
10: **if** $U[0,1] \leq e^{\frac{-\Delta}{T}}$ **then**
11: $current = p$
12: **end if**
13: **end if**
14: **if** $f(current) \leq f(best)$ **then**
15: set $best = current$
16: **end if**
17: Decrease the temperature T according to annealing scheme
18: **end while**

Simulated annealing can deal with highly nonlinear models, chaotic and noisy data and many constraints. It is a robust and general technique. Its main advantages over other local search methods are its flexibility and its ability to approach global optimality. The algorithm is quite versatile since it does not rely on any restrictive properties of the model. SA methods are easily *tuned*. For any reasonably difficult nonlinear or stochastic system, a given optimization algorithm can be tuned to enhance its performance and since it takes time and effort to become familiar with a given code, the ability to tune a given algorithm for use in more than one problem should be considered an important feature of an algorithm.

Since SA is a metaheuristic, a lot of choices are required to turn it into an actual algorithm. There is a clear tradeoff between the quality of the solutions and the time required to compute them. The tailoring work required to account for different classes of constraints and to fine-tune the parameters of the algorithm can be rather delicate. The precision of the numbers used in implementation is of SA can have a significant effect upon the quality of the outcome.

To turn the framework given in Algorithm 1 into an actual algorithm, the following elements have to be provided:

- an initial solution,
- a generator of random changes in solutions (i.e., how to perturb the current solution),
- an *annealing scheme* – an initial temperature and rules for lowering it as the search process progress.

In the following we will discuss these three elements.

3.2.1 Solutions representation and perturbations

When attempting to solve an optimization problem using the SA algorithm, the most obvious representation of the control variables is usually appropriate. However, the way in which new solutions are generated may need some thought. The solution generator should introduce small random changes, and allow all possible solutions to be reached.

The SA algorithm does not require or deduce derivative information, it merely needs to be supplied with an objective function for each trial solution it generates. Thus, the evaluation of the problem functions is essentially

a "black box" operation as far as the optimization algorithm is concerned. Obviously, in the interests of overall computational efficiency, it is important that the problem function evaluations should be performed efficiently, especially as in many applications these function evaluations are by far the most computationally intensive activity.

Some thought needs to be given to the handling of constraints when using the SA algorithm. In many cases the routine can simply be programmed to reject any proposed changes which result in constraint violation, so that a search of feasible space only is executed. However, there is one circumstance in which this approach cannot be followed: if the feasible space defined by the constraints is (suspected to be) disjoint, so that it is not possible to move between all feasible solutions without passing through infeasible space. In this case the problem should be transformed into an unconstrained one by constructing an augmented objective function incorporating any violated constraints as penalty functions.

Essentially, the initial solution should be selected randomly. Any other selection of initial solution (e.g. delivered by some heuristic) requires to adjust the annealing scheme discussed later on in some sophisticated manner. Usually, such approach doesn't improve the algorithms efficiency and effectiveness.

3.2.2 Annealing schemes

The annealing scheme determines the degree of uphill movement permitted during the search and is thus critical to the algorithm's performance. The principle underlying the choice of a suitable annealing schedule is easily stated: the initial temperature should be high enough to *melt* the system completely and should be reduced towards its *freezing point* as the search progresses. Choosing an annealing schedule for practical purposes is something of an art.

The standard implementation of the SA algorithm is one in which homogeneous Markov chains of finite length are generated at decreasing temperatures. The following parameters should therefore be specified:

- an initial temperature,
- a final temperature or a stopping conditions,
- a length for the Markov chains, and
- a rule for decrementing the temperature.

Initial temperature

A suitable initial temperature is one that results in an acceptance probability of value close to 1. In other words, there is an almost 100% chance that a change which increases the objective function will be accepted. The value of initial temperature will clearly depend on the objective function and, hence, be problem-specific. It can be estimated by conducting an initial search in which all increases are accepted (i.e., the fixed number of iterations of simulated annealing in which all perturbed solutions are unconditionally accepted) and calculating the maximum objective increase observed δf. Then, the initial temperature T_0 is given by:

$$T_0 = \frac{-\delta f}{\ln(p)},$$

where p is a probability close to 1 (e.g. 0.8–0.9).

Final temperature and stopping conditions

In some simple implementations of the SA algorithm the final temperature is determined by fixing the number of temperature values to be used, or the total number of solutions to be generated (total number of iterations). Alternatively, the search can be halted when it ceases to make progress. Lack of progress can be defined in a number of ways, but a useful basic definition is no improvement (i.e. no new best solution) being found in an entire Markov chain at one temperature.

Length of Markov chains

An obvious choice for L, the length of the Markov chain, is a value that depends on the size of the problem. Alternatively it can be argued that a minimum number of transitions t_{min} should be accepted at each temperature. However, as temperature approaches 0, transitions are accepted with decreasing probability so the number of trials required to achieve t_{min} acceptances approaches 1 . Thus, in practice, an algorithm in which each Markov chain is terminated after L transitions or t_{min} acceptances, whichever comes first, is a suitable compromise.

Decreasing the temperature

In the SA algorithm, the temperature is decreased gradually such that

$$T_i > 0, \forall i,$$

and

$$\lim_{i \to \infty} T_i = 0,$$

where i denotes the iteration of an algorithm.

There is always a compromise between the quality of the obtained solutions and the speed of the cooling scheme. If the temperature is decreased slowly, better solutions are obtained but with a more significant computation time. The temperature T can be updated in different ways:

Linear In the trivial linear scheme, the temperature T is updated as follows: $T = T - \alpha$, where α is a specified constant value. Hence, we have

$$T_i = T_0 - i\alpha,$$

where T_i represents the temperature at iteration i.

Geometric In the geometric scheme, the temperature is updated using the formula

$$T = \alpha T,$$

where $\alpha \in [0, 1]$. It is the most popular cooling function.

Logarithmic The temperature at iteration i is calculated using the following formulae:

$$T_i = \frac{T_0}{\log(i)}.$$

This scheme is too slow to be applied in practice but has been proven to have the property of convergence to a global optimum.

Modified logarithmic The main trade-off in a cooling scheme is the use of a large number of iterations at a few temperatures or a small number of iterations at many temperatures. Modified logarithmic scheme such as

$$T_i = \frac{T_{i-1}}{1 + \alpha T_{i-1}}$$

may be used where α is some constant parameter. Only one iteration is allowed at each temperature in this very slow decreasing function.

Random number generation

A significant component of an SA code is the random number generator, which is used both for generating random changers in the control variables and for the (temperature dependent) increase acceptance test. It is important, particularly when tackling large scale problems requiring thousands of iterations, that the random number generator used have good spectral properties.

3.3 An example of simulated annealing algorithm

The implementation of simulated annealing is actually quite simple in any high-level programming language. We'll describe three of the functions that make up the simulated annealing implementation, the main simulated annealing algorithm, perturbing a tour, and decreasing the temperature.

3.3.1 Solution perturbation

Given a solution, we can create an adjacent solution using the function that randomly select two cities in the tour, and swap them. An additional loop is required to ensure that we have selected two unique random points (so that we don't swap a single city with itself). Once selected, the two cites are swapped and the function is complete.

3.4 Temperature change

The temperature schedule is a factor in the probability for accepting a worse solution. In this example, we will use a geometric decay for the temperature:

$$T = \alpha T.$$

In this case, we use an alpha of 0.999. The temperature decay using this equation is shown in Figure 3.1.

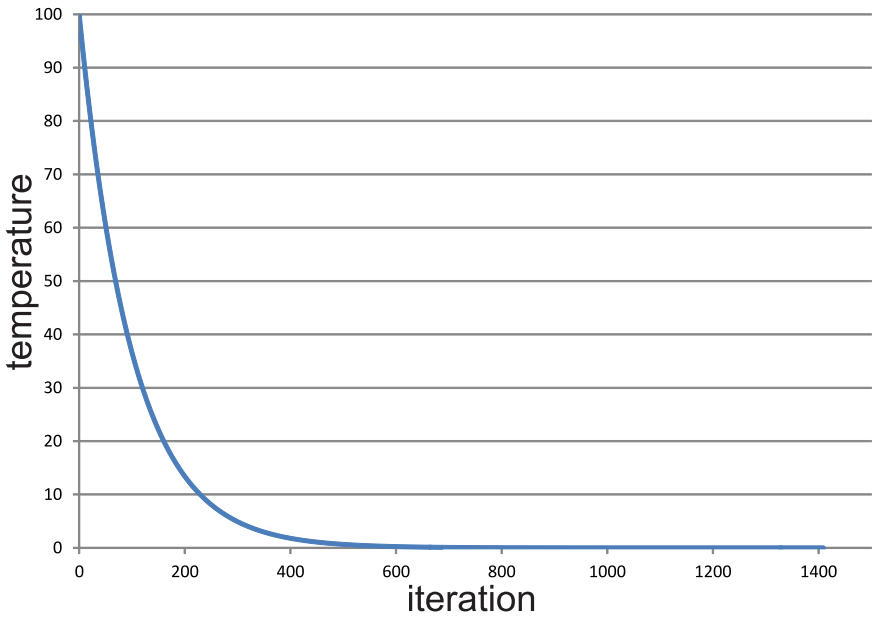

Figure 3.1: Temperature change during the search process

3.5 Main loop of the algorithm

The algorithm loops around the temperature, constantly reducing until it reaches a value near zero. The initial solution has been initialized prior to this function with a randomly generated tour. We take the current solution and perturb it (randomly alter it) for a number of iterations (the length of the Markov chain). If the new solution is better, we accept it by copying it into the current solution. If the new solution is worse, then we accept it with a probability defined by earlier. The worse the new solution and the lower the temperature, the less likely we are to accept the new solution. When the Markov chain reaches its length, the temperature is reduced and the process continues. When the algorithm completes, we receive the city tour.

3.6 Sample run of the algorithm

The relative fitness of the solution over a run is shown in Figure 3.2 This graph shows the length of the tour during the decrease in temperature. Note at the left-hand side of the graph that the relative fitness is very erratic. This is due to the high temperature accepting a number of poorer solutions. As the temperature decreases (moving to the right of the graph), poorer solutions are not accepted as readily. At the left-hand side of the graph, the algorithm permits exploration of the state space, where at the right-hand of the graph, the solution is fine-tuned.

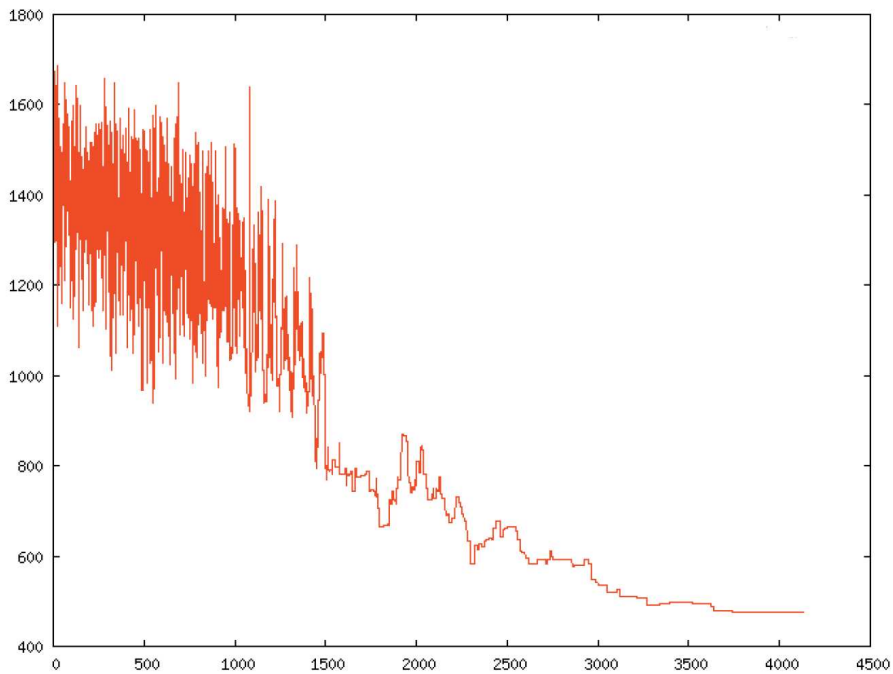

Figure 3.2: Best tour length found during the search process

3.7 Modifications of simulated annealing

Other similar methods of simulated annealing have been proposed in the literature, such as threshold accepting, great deluge algorithm, record-to-record travel, and demon algorithms The main objective in the design of those simulated-annealing-inspired algorithms is to speed up the search of the SA algorithm without sacrificing the quality of solutions.

3.7.1 Threshold accepting (TA)

Threshold accepting (TA) may be viewed as the deterministic variant of simulated annealing. TA escapes from local optima by accepting solutions that are not worse than the current solution by more than a given threshold V, i.e., if Δ is less than a given threshold V then a new perturbed solution is accepted. If $\Delta > V$ then current solution remains unchanged. The threshold value V is decreased during the search process, which is similar to annealing scheme in classical SA method.

The threshold V is updated according to an annealing schedule. It must be set as a deterministic non-increasing step function in the number of iterations i. The threshold decreases at each iteration and then reaches the value of 0 after a given number of iterations.

TA is a fast algorithm compared to SA because the generation of random number and exponential functions consume a significant amount of computational time. The literature reports some performance improvements compared to the simulated annealing algorithm in solving combinatorial optimization problems such as the traveling salesman problem.

3.7.2 Record-to-record travel (RRT)

This algorithm is also a deterministic optimization algorithm inspired from simulated annealing. The algorithm accepts a non-improving neighbor solution with an objective value less than the $record$ minus a deviation D. $record$ represents the best objective value of the visited solutions during the search, i.e., $record = best$ in the scheme presented in Algorithm 1. The bound decreases with time as the objective value $record$ of the best found solution improves.

The record-to-record travel algorithm has the advantage to be dependent on only one parameter, the deviation D value. A small value for the deviation

will produce poor results within a reduced search time. If the deviation is high, better results are produced after an important computational time.

3.7.3 Great deluge algorithm (GDA)

The great deluge algorithm was proposed by Dueck in 1993. The main difference with the SA algorithm is the deterministic acceptance function of neighboring solutions. The inspiration of the GDA algorithm comes from the analogy to the direction a hill climber would take in a great deluge to keep his feet dry. Finding the global optimum of an optimization problem may be seen as finding the highest point in a landscape. As it rains incessantly without end, the level of the water increases. The algorithm never makes a move beyond the water level. It will explore the uncovered area of the landscape to reach the global optimum.

A generated neighbor solution is accepted if the absolute value of the objective function is less than the current boundary value, named $waterlevel$. The initial value of the $waterlevel$ is equal to the initial objective function. The level parameter in GDA operates somewhat like the temperature in SA. During the search, the value of the level is decreased monotonically. The decrement is a parameter of the algorithm.

The great deluge algorithm needs the tuning of only one parameter, the rs value that represents the rain speed. The quality of the obtained results and the search time will depend only on this parameter. If the value of the rs parameter is high, the algorithm will be fast but will produce results of poor quality. Otherwise, if the rs value is small, the algorithm will generate relatively better results within a higher computational time. An example of a rule that can be used to define the value of the rs parameter may be the following: a value smaller than 1% of the average gap between the quality of the current solution and the $waterlevel$.

3.7.4 Demon algorithm (DA)

The demon algorithm is another simulated annealing-based algorithm that uses computationally simpler acceptance functions.

The acceptance function is based on the energy value of the demon. The demon energy is initialized with a given value D. A non-improving solution is accepted if the demon has more energy than the decrease of the objective value. When a DA algorithm accepts a solution of increased objective value, the change value of the objective is added to the demon's

energy. In the same manner, when a DA algorithm accepts an improving solution, the decrease of the objective value is debited from the demon.

The acceptance function of demon algorithms is computationally simpler than in SA. It requires a comparison and a subtraction, whereas in SA it requires an exponential function and a generation of a random number. Moreover, the demon values vary dynamically in the sense that the energy D depends on the visited solutions during the search, whereas in SA and TA the temperature (or threshold) is not dynamically reduced. Indeed, the energy absorbed and released by the demon depends mainly on the accepted solutions.

3.8 Conclusions

As with genetic algorithms, discussed later on, a major advantage of SA is its flexibility and robustness as a global search method. It is a "weak method" which does not use problem-specific information and makes relatively few assumptions about the problem being solved. It can deal with highly non-linear problems and "almost any" functions. Simulated annealing is a very powerful and important tool in a variety of disciplines. A disadvantage is that the SA methods are computation-intensive. Faster variants of simulated annealing exist, but they are not easy to code and therefore they are not widely used.

Chapter 4

Tabu search (TS)

4.1 Introduction

Tabu search (TS) can be viewed as beginning in the same way as ordinary local or neighborhood search, proceeding iteratively from one point (solution) to another until a chosen termination criterion is satisfied. Each solution, say s, has an associated neighborhood $N(s)$, and each solution $s\prime \in N(s)$ can be reached from s by an operation called a *move*.

TS can be contrasted with a simple descent method where the goal is to minimize some function $f(s)$, where $s \in S$. Such a method only permits moves to neighbor solutions that improve the current objective function value and ends when no improving solutions can be found. The final s obtained by a descent method is called a local optimum, since it is at least as good as or better than all solutions in its neighborhood. The evident shortcoming of a descent method is that such a local optimum in most cases will not be a global optimum, i.e., it usually will not minimize $f(s)$ over all $s \in S$.

TS behaves like a simple descent algorithm, but it accepts non-improving solutions to escape from local optima when all neighbors are non-improving solutions. Usually, the whole neighborhood is explored in a deterministic manner, whereas in SA a random neighbor is selected. As in local search, when a better neighbor is found, it replaces the current solution. When a local optima is reached, the search carries on by selecting a candidate worse than the current solution. The best solution in the neighborhood is

selected as the new current solution even if it is not improving the current solution. Tabu search may be viewed as a dynamic transformation of the neighborhood. This policy may generate cycles – that is, previous visited solutions could be selected again. To avoid cycles, TS discards the neighbors that have been previously visited. It memorizes the recent search trajectory. Tabu search manages a memory of the solutions or moves recently applied, which is called the tabu list. This tabu list constitutes the short-term memory. At each iteration of TS, the short-term memory is updated. Storing all visited solutions is time and space consuming. Indeed, we have to check at each iteration if a generated solution does not belong to the list of all visited solutions. The tabu list usually contains a constant number of tabu moves. Usually, the attributes of the moves are stored in the tabu list.

By introducing the concept of solution features or move features in the tabu list, one may lose some information about the search memory. We can reject solutions that have not yet been generated. If a move is "good", but it is tabu, do we still reject it? The tabu list may be too restrictive – a non-generated solution may be forbidden. Yet for some conditions, called aspiration criteria, tabu solutions may be accepted. The admissible neighbor solutions are those that are non-tabu or hold the aspiration criteria.

The framework of Tabu Search consists of several steps which are described below and depicted in Algorithm 2.

Algorithm 2 Basic scheme of tabu search

1: generate the initial solution s
2: set solutions $current = s$, and $best = s$
3: create tabu list TL, and add s to TL
4: **while** stopping condition is not satisfied **do**
5: Select best, non-tabu solution s' from $N(current)$
6: Check aspiration criterion
7: **if** $f(s') < f(current)$ **then**
8: set $current = s'$
9: **end if**
10: **if** $f(current) < f(best)$ **then**
11: set $best = current$
12: **end if**
13: update TL with $current$
14: **end while**

The tabu list (short-term memory) and the aspiration criterion are the

basic mechanisms in tabu search method. Commonly used additional mechanisms consist of long-term memory, and medium-term memory. In the following we describe the roles of these three kinds of memory in the search process.

4.2 Short-term memory

The role of the short-term memory is to store the recent history of the search to prevent cycling. The naive straightforward representation consists in recording all visited solutions during the search. This representation ensures the lack of cycles but is seldom used as it produces a high complexity of data storage and computational time. For instance, checking the presence of all neighbor solutions in the tabu list will be prohibitive. The first improvement to reduce the complexity of the algorithm is to limit the size of the tabu list. If the tabu list contains the last k visited solutions, tabu search prevents a cycle of size at most k. Using hash codes may also reduce the complexity of the algorithms manipulating the list of visited solutions. In general, attributes of the solutions or moves are used. This representation induces less important data storage and computational time but skips some information on the history of the search. For instance, the absence of cycles is not ensured. The most popular way to represent the tabu list is to record the move attributes. The tabu list will be composed of the reverse moves that are forbidden. This scheme is directly related to the neighborhood structure being used to solve the problem. If the move m is applied to the solution s to generate the solution s' then the move m' that translates solution s' back into s is stored in the list. This move is forbidden for a given number of iterations, named the *tabu tenure* of the move. If the tabu list contains the last k moves, tabu search will not guarantee to prevent a cycle of size at most k.

The size of the tabu list is a critical parameter that has a great impact on the performance of the tabu search algorithm. At each iteration, the last move is added to the tabu list, whereas the oldest move is removed from the list. The smaller is the value of the tabu list, the more significant is the probability of cycling. Larger values of the tabu list will provide many restrictions and encourage the diversification of the search as many moves are forbidden. A compromise that depends on the landscape structure of the problem and its associated instances must be found.

4.3 Long-term memory

Long-term memory has been introduced in tabu search to encourage the diversification of the search. The role of the long-term memory is to force the search in non explored regions of the search space. The main representation used for the long-term memory is the frequency memory. As in the recency memory, the components associated with a solution have to be defined first. The frequency memory will memorize for each component the number of times the component is present in all visited solutions. The diversification process can be applied periodically or after a given number of iterations without improvement.

As for the intensification, the diversification of the search is not always useful. It depends on the landscape structure of the target optimization problem. For instance, if the landscape is a "massif central" where all good solutions are localized in the same region of the search space within a small distance, diversifying the search to other regions of the search space is useless. The search time assigned to the diversification and the intensification components of TS must be carefully tuned depending on the characteristics of the landscape structure associated with the problem.

4.4 Medium-term memory

The role of the intensification is to exploit the information of the best found solutions (elite solutions) to guide the search in promising regions of the search space. This information is stored in a medium-term memory. The idea consists in extracting the (common) features of the elite solutions and then intensifying the search around solutions sharing those features. A popular approach consists in restarting the search with the best solution obtained and then fixing in this solution the most promising components extracted from the elite solutions. The main representation used for the medium-term memory is the recency memory. First, the components associated with a solution have to be defined; this is a problem specific task. The recency memory will memorize for each component the number of successive iterations the component is present in the visited solutions. „It is common to start the intensification process after a given period or a certain number of iterations without improvement.

4.5 Example algorithm

Tabu Search is a heuristic that, if used effectively, can promise an efficient near-optimal solution to the TSP. The basic steps of the algorithm applied to the TSP are presented below.

4.5.1 Solution representation

AA feasible solution is represented by a sequence of cities where cities appear in the order they are visited and each city appears only once. The first and the last visited cities are fixed to 1. The starting city is not specified in the solution representation and is always assumed to be the city 1.

4.5.2 Initial solution

A good feasible, yet not-optimal, solution to the TSP can be found quickly using a greedy approach. Starting with the first city in the tour, find the nearest city. Each time find the nearest unvisited city from the current city until all the cities are visited.

4.5.3 Neighborhood

A neighborhood of a given solution is defined as any other solution that is obtained by a pair wise exchange of any two cities in the solution (swap move). This always guarantees that any neighborhood of a feasible solution is always a feasible solution (i.e., does not form any sub-tour). If we fix city 1 as the start and the end city for a problem of n cities, there are $O(n^2)$ such neighbors of a given solution. At each iteration, the neighbor with the best objective value (minimum distance) is selected.

4.5.4 Tabu list

To prevent the process from cycling in a small set of solutions, some attribute of recently visited solutions is stored in a Tabu List, which prevents their occurrence for a limited period. For TSP problem, the attribute used is a pair of cities that have been exchanged recently. A Tabu structure stores the number of iterations for which a given pair of nodes is prohibited from exchange.

4.5.5 Aspiration criterion

Tabu list may sometimes be too powerful: it may prohibit attractive moves, even when there is no danger of cycling, or they may lead to an overall stagnation of the searching process. It may, therefore, become necessary to revoke tabus at times. In TSP it may be done by allowing a move, even if it is tabu, if it results in a solution with an objective value better than that of the current best-known solution.

4.5.6 Diversification

Quite often the process may get trapped in a space of local optimum. To allow the process to search other parts of the solution space (to look for the global optimum), it is required to diversify the search process, driving it into new regions. This is done using *frequency based memory*. The frequency information is used to penalize non-improving moves by assigning a larger penalty (frequency count adjusted by a suitable factor) to swaps with greater frequency counts. This diversifying influence is allowed to operate only on occasions when no improving moves exist. Additionally, if there is no improvement in the solution for a pre-defined number of iterations, frequency information can be used for a pair wise exchange of cities that have been explored for the least number of times in the search space, thus driving the search process to areas that are largely unexplored so far.

4.5.7 Halting

The algorithm terminates if a pre-specified number of iterations is reached.

4.6 Conclusiuons

TS has been successfully applied to many optimization problems. Compared to simulated annealing, various search components of TS are problem specific and must be defined. The search space in TS is much larger than in local search and simulated annealing. The degree of freedom in designing the different ingredients of TS is important. The representation associated with the tabu list, the medium-term memory, and the long-term memory must be designed according to the characteristics of the optimization problem at hand. This is not a straightforward task for some optimization problems.

Moreover, TS may be very sensitive to some parameters such as the size of the tabu list.

Chapter 5

Genetic algorithms

"Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning." – Richard Cook

5.1 Introduction

Reading the above Cook's sentence, the first question that arises is "*Why the Universe is winning?*". The answer is quite simple: "*Because the Universe uses genetic algorithm.*"

GAs are a special subset of metahueristics, which use a form of biological mimicry which emulates the process of natural selection.

"*Three billion years of evolution can not be wrong. It is the most powerful algorithm there is.*"

This quotation from Goldberg sums up the aim of Genetic Algorithms; to model nature, and harness their proven ability to refine solutions, or animals, to a very efficient form. They are a form of metahueristic search, used to find solutions to difficult problems, possibly even NP-hard, where it is not feasible to enumerate all possibilities in order to find the best solution.

A Genetic Algorithm (GA) is a type of metahueristic algorithm, designed to operate on optimization problems. Optimization problems typically de-

mand that a certain variable be either minimized or maximized, while remaining legal within some set of constraints. These problems are often extremely large in their nature, usually to the point of NP-hardness, which effectively means that finding the exact or optimum solution is infeasibly difficult. To enumerate every possible solution and evaluate them to determine which is the optimum would take an inordinate amount of time.

GAs work by creating a selection of possible problems, called the population, and breeding them with each other. This alone would not refine the solutions in any way, so the process of natural selection is used to *kill off* the section of the least efficient solutions in each generation. Each solution is evaluated by a *fitness method* which uses some problem-specific algorithm to calculate the *goodness* of that solution. In a simple model of natural selection the best solutions from each generation are used to create the next one, and the worst are deleted.

To create the next generation, a combination of mutation and crossover is used. Crossover takes some qualities from two solutions and creates another - this models breeding and procreation. Mutation is also used, as in evolution, randomly to change these solutions in the hope that a desirable feature is introduced. Mutation is also important in order to avoid local optima, which are areas of the search space that appear to be optimum, but are actually just isolated by neighboring solutions of a less desirable nature.

5.2 Natural selection and mutation in Nature

Within nature, members of a population are born, procreate, and die. Procreation creates offspring which are a combination of the two parents, with occasional mutation also operating on the genes. This mutation does not necessarily have to be obvious or large. The mutation of a single gene can have little or no effect, but equally may have large repercussions - entirely dependent on its role within the body. It is often the case that combinations of genes affect a certain characteristic so that the alteration of a gene may have no obvious effect, but actually subtly alter many characteristics.

Mutation can occur within any cell in the body, and usually occurs during replication. There are mechanisms which reduce the amount of mutation that is allowed to occur, but they are not infallible. There are two types of cell in living creatures; somatic and germline. Germline cells produce sperm and eggs, and all other cells are somatic. Therefore if the mutation occurs in the somatic cells, then this mutation will die with the cell, but if it

occurs in the germline cells then it will be passed onto offspring - provided the organism is not detrimentally affected to the point of not surviving to procreation.

These mutations can be beneficial or harmful, and can provide the animal with an advantage over the other members of the species, or cause it to be less capable of survival than others. The mutations are more than likely to be detrimental than beneficial, as "*there are more ways of being dead than being alive*"[1], i.e., within the vast space of possible gene sequences, there are few that represent living and surviving organisms, and an almost limitless amount of pools of non-living amino acids.

For example, an increase in the capability to detect certain smells may make the animal a better hunter, or enable it to detect predators better, and in either case it would provide the animal with an advantage over other members of that species. This would mean that it would be more likely to survive to adulthood, and to procreate, spreading its genes. An animal with a detrimental mutation however, such as a reduced sense of smell, would be more likely to succumb to starvation or attack from predators before procreation could occur. This is natural selection, and is a natural feedback process which causes '*good*' genes to spread, and takes '*bad*' genes out of the pool. It is this interplay between entirely random mutation, and non random selection that makes up the process of evolution, causing species to adapt to their environment - not by intent but by default. It is a process that takes an almost unimaginable length of time to occur. There is little doubt that usually feedback mechanisms operate to regulate the size of populations

5.3 Evolution as a paradigm for problem solving

The powerful refinement and improvement abilities of natural selection can be harnessed to solve combinatorial optimization problems using a computer.

By creating a model of an environment, where the organisms become potential solutions to the problem, and genes become variables modeling that solution, we can recreate natural selection to '*breed*' solutions that increase in fitness with each generation. We can simulate all processes of evolution; procreation can be modeled by combining two or more solutions in certain ways, mutation can be modeled using random number genera-

[1] E. Dawkins "*The Blind Watchmaker*"

tors, natural selection and death can be modeled using a fitness evaluation method, and selecting which solutions will '*survive*' to the next generation.

In this way we can explore the search space, refining our solutions, and avoiding local optimums by including random mutation — some of which will be detrimental and will not survive to procreation, and some which will be beneficial and will steer the solutions towards unexplored space.

Work conducted in the 1950's and 1960's in cellular automata started the idea of using GAs to solve problems inherent in engineering, wherever they constituted optimization problems.

In the 1980s research into GAs started, and an international conference for the field was founded. As early as 1995 there were several successful examples of GA optimization being used in industry including Texas Instruments designing chips to minimize size but maintain functionality. Critical designs such as the engine plans leading to the development of the Boeing 777 engine by General Electric were also developed using GAs.

US West uses GAs to design fiber-optic cable networks, cutting design times from two months to two days, and saving US West $1 million to $10 million on each network design. Genetic Algorithm derived designs have now even been used in satellites by NASA, with the development of an aerial being taken completely out of engineers hands. The orbit of those same satellites is now even determined with the use of a Genetic Algorithm.

GAs and genetic programming algorithms, which use GA type evaluation and mutation to write functional code, are by no means a '*silver bullet*'. There are several reasons for which they are not suited to all problems, and these are examined here. Sometimes GAs or genetic programming provide solutions that are so complex or convoluted that no human programmer could decipher what is being performed within. Because they do not follow any logical examination of the problem, as a human designer would, they may find an extremely counter-intuitive way to achieve a certain task. No design of any kind is actually performed in order to find a solution, so apparent logic in the result is not guaranteed.

The methods that GAs use to design systems are not necessarily logical so the finished code, no matter how effective, may be all but indecipherable to the human user. This means that sometimes full testing is not possible, and code that appears to work completely cannot be proven to work in all cases. Goldberg talks of the difference between conceptual machines and material machines, i.e. an algorithm and a vehicle engine respectively. One is fully testable, but the other is not necessarily so.

Although this is a normal problem with testing; not all cases can be

tested, but if the code is readable then a talented tester can devise test cases that will likely trip the system up which is not possible with highly complex code. This creates ethical problems with implementing GA derived designs in mission critical or real time applications. Bearing in mind that a failure in air traffic control, life support hardware etc could be fatal, or that failure in a financial institution could be disastrous in other ways. GA is also used to develop actual mechanical devices. Goldberg tells an amusing story about an airline passenger worrying about the design and its testing. If you were to imagine a plane as a GA, and the passenger as a GA user, then you could imagine the stress that the thought of a failure would cause.

One interesting problem that arises from the use of Genetic Algorithms is the moral or ethical implications arising from a design that is not of human derivation. If there is a fundamental flaw in a human design, and this leads to a failure involving financial loss or human injury then the blame is apportioned to the engineer responsible for the negligent design. However, if a GA designs a system that leads to a failure (possible due to unforeseen emergent behavior as a result of massive complexity) then it is difficult to find the root cause of this issue.

5.4 General scheme of a genetic algorithm

The way genetic algorithms work is essentially mimicking evolution. First, you figure out a way of *encoding* any potential solution to your problem as a "*digital*" chromosome. Then, you create a start population of random chromosomes (each one representing a different candidate solution) and evolve them over time by "*breeding*" the fittest individuals and adding a little mutation here and there. With a bit of luck, over many generations, the genetic algorithm will converge upon a solution. Genetic algorithms do not guarantee a solution, nor do they guarantee to find the best solution, but if utilized the correct way, you will generally be able to code a genetic algorithm that will perform well. The best thing about genetic algorithms is that you do not need to know how to solve a problem; you only need to know how to encode it in a way the genetic algorithm mechanism can utilize.

Typically, the chromosomes are encoded as a series of binary bits. At the start of a run, you create a population of chromosomes, and each chromosome has its bits set at random. The length of the chromosome is usually fixed for the entire population. The important thing is that each chromo-

some is encoded in a such way that the string of bits may be *decoded* to represent a solution to the problem at hand. It may be a very poor solution, or it may be a perfect solution, but every single chromosome represents a possible solution. Usually the starting population is *terrible* in terms of the solutions the chromosomes of this population encode. Anyway, an initial population of random chromosomes is created (let's say one hundred of them for this example), and then you execute Algorithm 3.

Algorithm 3 Basic scheme of genetic algorithm

1: Generate initial population of n chromosomes
2: **while** solution is not found **do**
3: Test each chromosome to see how good it is at solving the problem and assign a fitness score accordingly
4: Select two members from the current population. The probability of being selected is proportional to the chromosome's fitness the higher the fitness, the better the probability of being selected.
5: Dependent on the *Crossover Rate*, *crossover* the chosen chromosomes at a randomly chosen point
6: Step through the chosen chromosome's bits and *mutate* dependent on the *Mutation Rate*.
7: Repeat steps 4, 5, and 6 until a new population of n members has been created.
8: **end while**

Each loop through the algorithm is called a *generation* (steps 3 through 7). We call the entire loop an *epoch* and will be referring to it as such in the remaining text. In the following we will discuss each step of the presented scheme of GA. It will become clear that each step is indeed an algorithm in its own right, and that there are numerous choices of strategy for each. A GA is simply an abstraction of a subset of algorithms. It should also be noted that the huge variation of approaches possible for each of the several components of a GA means that there is a vast number of programs that fall under this catch-all title. A taxonomy of different approaches to the GA idea would be an extremely complex tree.

5.4.1 Population size

Population size is the term used for the number of solutions held by the GA. The population size is a critical variable, which presents a trade-off

between the computational power of each generational iteration, and the computational intensity of each iteration. A small number of solutions allows a greater number of generations to iterate in a given time, because each generation will be less computationally intensive.

A large population size will provide more variety in each generation, which means greater likelihood of a benefcial solution, but the trade-off is that each generation will take longer to compute. However, within a relatively small amount of generations the solutions will begin to converge on a similar solution. The amount of variation on average will decrease as the generation number increases, although the rate will vary on the magnitude of the mutation. Therefore it may be beneficial to choose a smaller value for the population size.

Small variations, if beneficial, will propagate through a small population just as they will through a larger one. Although a large population size means that there is more mutation per generation, the faster computation of a smaller population size would be capable of allowing a similarly large amount of mutation over a small number of solutions within the same time period. The only constraint on the population size is that the initial population needs to be at least 2 solutions large, as with any smaller amount no crossover operations would be possible, and it would be impossible to derive any more beneficial solutions in this way.

5.4.2 Population initialization

The initialization of the first generation population could conceivably alter the speed with which the program finds a solution that satisfies the halting algorithm. The distance of the solution's fitness from optimum would affect the number of generations needed to satisfy the halting algorithm.

In the extreme case it is possible that one of the initial solutions actually satisfies the halting algorithm, in which case the program would complete before any crossover or mutation was necessary.

With an entirely homogenous population, the algorithm would rely on mutation in order to provide variance in early generations, as crossover between similar solutions achieves little. By creating a varied initial population it is conceivable that good solutions would be found quicker, and the algorithm would complete faster.

The exact method of initialization that would provide the optimum start is unknown, as is the magnitude of the positive effect possible. It is highly likely that a varied population would be more beneficial than a homogenous one

to some extent however, and any attempt to vary the genes would provide a small advantage at the very least.

There are several possible techniques for initializing populations, but the most efficient method would vary not only from problem to problem, but also from instance to instance within each problem.

Pre-defined variance

One possible approach would be to have designed, pre-defined variance in the solutions. These could be identical for each run of the algorithm, or designed for each instance of the problem. It may be possible to tailor. In this way, possible "good" traits could be engineered into the algorithm. For example, in the scenario presented in Table 5.1, a GA is searching for a solution to a problem, where the optimum value is 8. The fitness algorithm simply calculates the integer value of the binary string which makes up the solution representation, and returns the inverted difference between that value and the optimum.

Table 5.1: Example scenario for pre-defined variance population initialization

binary string chromosome	integer value of the string	fitness value
0000	0	-8
0011	3	-5
0110	6	-2
1001	9	-1
1100	12	-4
1111	15	-7

If it was possible to estimate the approximate region of the optimal solution, then it would be possible to create the initial solutions within this range. Within a few generations, any initial population would congregate around the optimal solution. However, by approximating the optimum with the initialization, this process can be speeded up. The amount of effort required to tailor a good range of solutions around an approximation would not be beneficial however, as the saving made in computational time would be minimal in comparison.

It is conceivable that a system could be developed where a user inputs an approximate solution, and the program randomly generates the initial

population with slight derivations from this. This would reduce the amount of time necessary to set up the algorithm, but retain most of the benefit of this approach.

Randomly generated solutions and evenly spaced solution

A very simple method of initializing solutions would be to randomly generate them. As a GA would have code to create random variation in the mutation method, it would be a trivial matter to adapt this to fully populate the initial designs.

Another method would be to take the search space, and the number of solutions in the population, and to calculate an even spread throughout.

5.4.3 Fitness evaluation

The fitness evaluation section of the GA determines which solutions are removed, and which progress to the next generation. It uses an algorithm to calculate how good a certain solution is. Once each generation has been created, the fitness of each solution is evaluated and ranked.

Fitness evaluation mimics the environment in which members of a species would live. The higher the fitness value, the more likely a solution is to *survive* to the next generation, and the lower it is the more likely it is that the solution will *die off*. Therefore the fitness evaluation mimics anything dangerous in a real world environment - predators, disease, famine, etc.

The fitness evaluation is extremely problem specific, and must be designed very carefully. It can be a simple case of maximizing one value, or there may be many complex and interacting values to take into account. These must be analyzed, weighted and summed within the fitness evaluation code, and condensed to one value with which that solution may be judged.

5.4.4 Selection

Roulette wheel selection is a method of choosing members from the population of chromosomes in a way that is proportional to their fitness for example, the fitter the chromosome, the greater probability it has of being selected. It does not guarantee that the fittest member goes through to the next generation, merely that it has a very good probability of doing so. It works like this: Imagine that the population's total fitness score is represented by a pie chart, or roulette wheel (see Figure 5.1). Now, you assign a

slice of the wheel to each member of the population. The size of the slice is proportional to that chromosome's fitness score the fitter the member is, the bigger the slice of pie it gets. Now, to choose a chromosome, all you have to do is spin the wheel, toss in the ball, and grab the chromosome that the ball stops on.

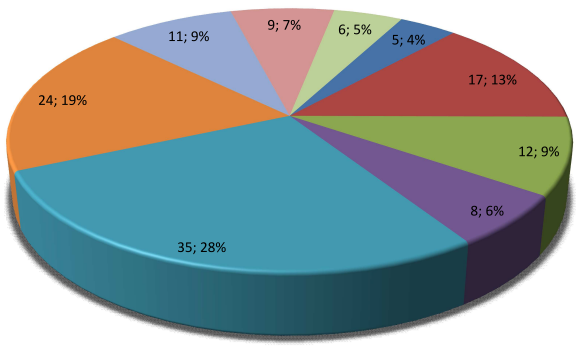

Figure 5.1: Example roulette wheel of nine chromosomes (labels denote the fitness value and percentage of the wheel, respectively)

5.4.5 Crossover operations

Crossover is one of the two main methods used in GAs in order to alter solutions. It is important to use both, as one maintains desirable traits while combining solutions, and the other introduces random noise in the hope of creating emergent properties that add value.

Crossover is the operation that allows solutions to combine. If mutation was used alone then it would essentially be a random solution generator, but would still work. Crossover allows the retention of desirable traits, without the need to keep entire solutions.

Mutation alone is not mathematically guaranteed to work, although it is infinitesimally unlikely that it would not at some point reach optimality or close to it. Crossover is the method by which a GA can take value from previous solutions, and build the next generation of solutions in a semi-intelligent manner.

The method of crossover requires careful thought, and its success is highly dependent on the problem to which it is applied. The encoding that is used to represent a single solution needs to be paired to crossover. For example, if a solution consists of three variables, all 4 bits long, then our total solution occupies 12 bits. The simplest crossover approach would be to bisect two solutions, and create two new solutions by combining the start of one and the end of another, and vice versa. If this approach was taken though, the efficiency of the program would be greatly reduced. Any time that a good value for the middle variable was found there is a good chance that it would be overwritten by the next generation. However, if the crossover was designed with the solution in mind, then the solutions could be trisected and recombined to preserver the value of individual variables. This would be more likely to arrive at a solution in a reasonable length of time.

Single point crossover

Single point crossover is the simplest form of crossover operation. It simply takes the bits of a pair of solutions, and bisects them at the same arbitrary point, the tail or head sections are then exchanged. (See Figure 5.2) In this way the pair of parent solutions can create a pair of children that share certain aspects of both solutions. If the two sections that make up the new child both contain features that are beneficial then a successful evolution has occurred, and a solution that exceeds the fitness of previous solutions has been created.

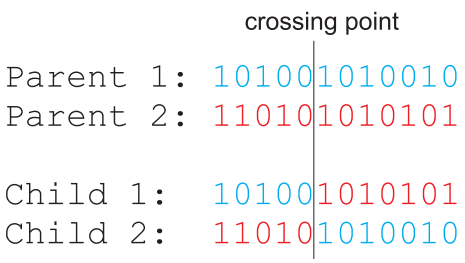

Figure 5.2: Example of the single point crossover operation applied to binary chromosomes

Double point crossover

With double point crossover the strategy is similar to single point, but the transferred section of bits does not have to include the tail or head of a solution. (See Figure 5.3) This enables greater flexibility in the sections altered, and also provides all genes with an equal chance of exchange, whereas in the single point strategy any point chosen is guaranteed to swap the end gene, and will favor genes towards the edge of the solution.

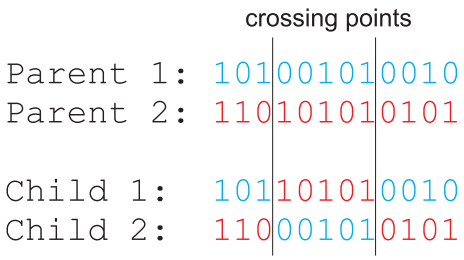

Figure 5.3: Example of the double point crossover operation applied to binary chromosomes

Cut and splice crossover

Cut and splice is ax technique that does not maintain solution length, which means that it is not acceptable for all GAs. In both double and single point crossover the length of the solution is kept, and therefore genes can have specific properties. For example, in a GA that designed possible engine parts each gene could represent a quality of that part such as height or thickness. In single and double point crossover these values would be swapped around, but maintain their context and meaning, but in a strategy which varies solution length this is not possible. Instead, cut and splice is more fitting to problems where the solution strings represent a solution to some singular problem to optimize, for example in certain combinatorial optimization problems.

Uniform crossover

In uniform crossover the bits from each parent are swapped, depending upon a probability. In half uniform crossover the number of differing bits between the two parents is calculated, and this number is divided by two. The resulting number is the number of non-matching bits that will be exchanged.

5.4.6 Mutation operations

With a straightforward mutation-only algorithm it would be possible to reach an optimal solution, but this entirely random approach would not yield any significant increase over any other brute-force approach. Because of this it would not classify as a GA, or even a metaheuristic.

To fully equip a GA with the ability to find a solution as close as possible to optimality within a given time it is desirable to use a combination of crossover and mutation operations. With solely crossover operations there is a distinct possibility that the algorithm would work towards a local optimum and remain there, as promising solutions are crossed with others a local optimum would quickly propagate through the solutions in use, and stall the algorithm. With a mutation operation involved as well, random solutions are thrown into contention throughout the cycle of finding a solution, and this may eventually enable an algorithm to branch out from underneath a local optimum, in order to pursue other avenues.

Mutation is an essential part of any successful genetic algorithm. Without mutation an algorithm would simply combine the fittest solutions. This would mean that the generations would quickly converge into an amalgam of good solutions, and cease to improve. Mutation can be performed in a number of ways, and each will be appropriate for different problems.

In fact, mutation alone is capable of finding a solution, even the optimum solution, given sufficient time. For example, consider an algorithm with a population of only one solution, that only operates upon the solution with mutation. After each generation the solution is either better or worse than the previous one, and either fulfills the halting algorithm, or does not. It will eventually stumble upon a good enough solution. However, with a well designed GA we can reach a satisfactory result far quicker. It should be noted however that it can often be useful to create a simple algorithm such as this in order to get started in development of a more complex and intelligent system.

5.4.7 Halting

Halting is an important and difficult problem in GAs. Without some form of halting criteria, that is checked at every generation, the program would continue to run, even once significant gains in fitness where no longer being generated by new generations. There are many techniques that are used to halt GAs, and their appropriateness depends entirely on the application, but they fall into two main categories; those that are based on the runtime of the algorithm, and those that are based on the quality of the solutions.

Resource based halting

It is often the case that a GA can only be allocated a certain amount of a resource, specifically time or computing cycles, to complete. In real time critical systems it may be vital to arrive at a solution within a given period, and in this case it is the role of the algorithm to find the best solution possible in that time. Even in non-critical systems such as GPS route planners it is necessary to impose some time constraints to avoid annoying the user; it is unlikely that a user would be willing to wait hundreds of hours to find the optimal route when 5 seconds of calculation may find a route near to optimality.

In cases such as this the halting criteria are time or computing cycle based, rather than being associated with the fitness of the final solution. The algorithm will check, at the end of every generation cycle, to see if it has exceeded its allocated time, or if it is likely to in the next cycle, and the algorithm can then be halted. In extremely time critical systems this anticipation of possible time overruns can be used to avoid the algorithm exceeding its time constraints in between halting checks. The number of generations can also be the limiting factor, in a very loose time sensitive case, which although not accurate in terms of time constraints is very simple to implement.

Solution fitness based halting

If time is less of a constraint, then halting can be based on the fitness of the solutions. This is more desirable from a reliability point of view, as the output can be guaranteed to be of a certain quality, determined in the halting method. Complete enumeration and evaluation is still a desirable practice, and it does not mean that they are obsolete. The reason for this is that there

is no way to find optimality for a problem, without enumerating all possible solutions, which can be a very time intensive procedure. A large traveling salesman problem would demand that an enormous number of different solutions be enumerated and evaluated in order to find an optimal solution, where as a GA could be run for an arbitrary amount of time, and with each iteration reach a better solution.

To implement solution based halting, the algorithm must be provided information about what is an acceptable solution, in terms of the fitness evaluation method. In this way the algorithm can check at each generation, to see if the best solution of that generation exceeds the acceptability levels, and halt if it does. If not then the algorithm simply proceeds with the next iteration of the generation loop. This acceptability level can be set to a certain value, derived from the user, or it can be derived by some algorithm within the system, that ensures bounded in-optimality.

Progress based halting

Another method that can be used is to monitor the progress that the algorithm is making, in terms of numerical improvements correlating to the fitness of the best solution of each successive generation. This progress can be recorded and analyzed, in order to determine the most reasonable time to stop the algorithm. It is highly likely that the algorithm will go through a very short initial phase of confusion, especially if the initial population was randomly generated and extremely varied, and then go through a period of rapid improvement, before tailing off in a curve.

There will be exceptions to this curve, in that mutation will occasionally throw up a new solution that avoids a local optimum, and causes a period of noticeable growth again. However, there will come a time when the progress that the algorithm makes will be negligible. This can be used as a halting strategy in many ways, for example the algorithm could be instructed to halt if 3 successive generations did not improve the best solution by more than 1%.

5.5 An Example of the Genetic Algorithm

5.5.1 Basics notion and the traps we have to avoid

Now that you understand the basics of genetic algorithms, we will spend this chapter looking at a completely different way of encoding genetic algorithms that solves problems involving permutations. A good example of this is the famous problem called "Traveling Salesman Problem". This problem is usually abbreviated to the TSP, which saves a lot of typing ;).

One of the great things about tackling the TSP during your learning curve and the main reason we are devoting over a section to it – is that it is a fantastic way of witnessing how making changes to your code can affect the results. Often, when coding genetic algorithms, it is not easy to visualize what effect a different mutation or crossover operator has on your algorithm, or how a particular optimization technique performs, but the TSP provides you with great visual feedback, as you shall see. As you will discover when you start tinkering with your own genetic algorithms, finding a solution for over 15,000 cities is quite an achievement! You will be starting modestly, though. Twenty or so cities should be plenty for your first routing. Although, trying to get your algorithm to perform well on larger numbers of cities can get addictive!

At this point, it may be a good idea for you to make coffee, sit back, close your eyes, and spend a few minutes thinking about how you might tackle this problem...

As you may have realized, you can not take the same approach that you did in previous section to crossover operations. The main difference with the TSP is that solutions rely on permutations, and therefore, you have to make sure that all your chromosomes represent a valid permutation of the problem – a valid tour of all the cities. If you were to represent possible solutions using the binary encoding and crossover operator from the previous section, you would see how you would run into difficulties very quickly. Take the eight city example. You could encode each city as a 3-bit binary number, numbering the cities from 0 to 7. So, if you had two possible tours, you could encode them like this:

possible tour	binary encoded tour
3, 4, 0, 7, 2, 5, 1, 6	011 100 000 111 010 101 001 110
2, 5, 0, 3, 6, 1, 4, 7	010 101 000 011 110 001 100 111

Now, choose a crossover point (represented by a "|") after the fourth city, and see what offspring you get.

before crossover (parents)

no	binary encoded tour	decoded tour
1	011 100 000 111|010 101 001 110	3, 4, 0, 7, 2, 5, 1, 6
2	010 101 000 011|110 001 100 111	2, 5, 0, 3, 6, 1, 4, 7

after crossover (offspring)

no	binary encoded tour	decoded tour
1	011 100 000 111|110 001 100 111	3, 4, 0, 7, 6, 1, 4, 7
2	010 101 000 011|010 101 001 110	2, 5, 0, 3, 2, 5, 1, 6

You can see that there is a major problem! Both of the offspring have produced tours that contain duplicate cities, which of course means they are invalid.

To get this to work, you would have to code some hideous error-checking function to remove all the duplicates, which would probably lead to the destruction of any improvement gained up to that point in the tour. So, in the quest for a solution, a different type of crossover operator needs to be invented that spawns only valid offspring. Also, can you imagine what the previous mutation operator would do with this type of encoding? That is right, duplicate tours again. So, you also need to think about how you might implement a new type of mutation operator. Before you go any further though, does not binary encoding seem rather inelegant to you in the context of this problem? A better idea would be to use integers to represent each city. This, as you will see, will make life a lot easier all around. So, to clarify, the tour for parent one, shown in the preceding table, would be simply represented as a vector of integers `(3, 4, 0, 7, 2, 5, 1, 6)`.

You will save a lot of computer time if you do it this way, because you do not have to waste processor cycles decoding and encoding the solutions back and forth from binary notation.

As before, a crossover operator, a mutation operator, and a fitness function need to be defined. The most complex of these for the TSP is the crossover operator, because, as discussed earlier, a crossover function must provide valid offspring. So, we will wade in at the deep end and start with operator called PMX - *Partially Mapped Crossover*.

5.5.2 Partially-Mapped Crossover

There are many solutions that provide valid offspring for a permutation-encoded chromosome: Partially-Mapped Crossover, Order Crossover, Alternating-Position Crossover, Maximal-Preservation Crossover, Position-Based Crossover, Edge-Recombination Crossover, Subtour-Chunks Crossover, and Intersection Crossover to name just a few. In this section we will be discussing one of the most popular crossover types: Partially-Mapped Crossover, or PMX as it is more widely known. Later on, we will give descriptions of some of the alternatives, because it will be good practice for you to experiment with different operators to see what effect they may have on the efficiency of your genetic algorithm. But for now, let's just use PMX.

So, assuming the eight city problem has been encoded using integers, two possible parents may be:

```
Parent 1:  (2 , 5 , 0 , 3 , 6 , 1 , 4 , 7)
Parent 2:  (3 , 4 , 0 , 7 , 2 , 5 , 1 , 6)
```

To implement PMX, you must first choose two random crossover points let's say after 3rd and 6th city. So, the split is made at the "|"s, like so:

```
Parent 1:  (2 , 5 , 0 ,|3 , 6 , 1|, 4 , 7)
Parent 2:  (3 , 4 , 0 ,|7 , 2 , 5|, 1 , 6)
```

Then you look at the two center sections and make a note of the mapping between parents. In this example: 3 is mapped to 7, 6 is mapped to 2, and 1 is mapped to 5.

Now, iterate through each parent's genes and swap the genes wherever a gene is found that matches one of those listed. Step by step it goes like this:

```
Step 1 (here the children are just direct copies of their parents)
Child 1:  (2 , 5 , 0 , 3 , 6 , 1 , 4 , 7)
Child 2:  (3 , 4 , 0 , 7 , 2 , 5 , 1 , 6)

Step 2 (3 and 7)
Child 1:  (2 , 5 , 0 , 7 , 6 , 1 , 4 , 3)
Child 2:  (7 , 4 , 0 , 3 , 2 , 5 , 1 , 6)
```

```
Step 2 (6 and 2)
Child 1:  (6 , 5 , 0 , 7 , 2 , 1 , 4 , 3)
Child 2:  (7 , 4 , 0 , 3 , 6 , 5 , 1 , 2)

Step 3 (1 and 5)
Child 1:  (6 , 1 , 0 , 7 , 2 , 5 , 4 , 3)
Child 2:  (7 , 4 , 0 , 3 , 6 , 1 , 5 , 2)
```

And we have got it! The genes have been crossed over and you have ended up with valid permutations with no duplicates. This operator can be a little difficult to understand at first, so it may be worth your while to read over the description again. And then, when you think you've grasped the concept, try performing this crossover yourself with pen and paper. Make sure you understand it completely before you go on

5.5.3 The exchange mutation (EM)

After PMX, this operator is a pushover! Remember, you have to provide an operator that will always produce valid tours. The Exchange Mutation (EM) operator does this by choosing two genes in a chromosome and swapping them. For example, given the following chromosome:

```
(5 , 3 , 2 , 1 , 7 , 4 , 0 , 6)
```

The mutation function chooses two genes at random, for example 4 and 3, and swaps them:

```
(5 , 4 , 2 , 1 , 7 , 3 , 0 , 6)
```

which results in another valid permutation.

5.5.4 Deciding on a fitness function

A fitness function, which gives an increasing score the lower the tour length, is required. You could use the reciprocal of the tour length, but that does not really give much of a spread between the best and the worst chromosomes in the population. Therefore, when using fitness proportionate selection,

it is almost pot luck as to whether the fitter genomes will be selected. A better idea is to keep a record of the worst tour length in each generation and then iterate through the population again subtracting each genome's tour distance from the worst. This gives a little more spread, which will make the roulette wheel selection much more effective. It also effectively removes the worst chromosome from the population, because it will have a fitness score of zero and, therefore, will never get selected during the selection procedure.

5.5.5 Selection

Roulette wheel selection is going to be used again but this time with a difference. To help the genetic algorithm converge more quickly, in each epoch before the selection loop you are going to guarantee that n instances of the fittest genome from the previous generation will be copied unchanged into the new population. This means that the fittest genome will never be lost to random chance. This technique is most often referred to as elitism which will be discussed later on.

5.5.6 Alternative operators for the TSP

The first topic we are going to cover will be a discussion of those alternative mutation and crossover operators for the traveling salesman problem. Although none of them will improve the algorithm by a staggering amount, we feel that we should spend a little time going over the more common ones because it is interesting to see how many different ways there are of approaching the problem of retaining valid permutations. Also, some of them give very interesting and thought provoking results when you watch the TSP algorithm in progress. More importantly, though, it will teach you that for every problem, there can be a multitude of ways to code the operators. Again – and we know we keep saying this – please make sure you play around with different operators to see how they perform. You will learn a lot.

There have been many alternative mutation operators dreamed up by enthusiastic genetic algorithm researchers for the TSP. Here are descriptions of a few of the best

Scramble mutation (SM)

Choose two random points and "scramble" the cities located between them.

```
(0 , 1 , 2 , 3 , 4 , 5 , 6 , 7)
```

becomes

```
(0 , 1 , 2 , 5 , 6 , 3 , 4 , 7)
```

Displacement mutation (DM)

Select two random points, grab the chunk of chromosome between them, and then reinsert at a random position displaced from the original.

```
(0 , 1 , 2 , 3 , 4 , 5 , 6 , 7)
```

becomes

```
(0 , 3 , 4 , 5 , 1 , 2 , 6 , 7)
```

This is particularly interesting to watch because it helps the genetic algorithm converge to a short path very quickly, but then takes a while to actually go that few steps further to get to the solution.

Insertion mutation (IM)

This is a very effective mutation and is almost the same as the DM operator, except here only one gene is selected to be displaced and inserted back into the chromosome. In tests, this mutation operator has been shown to be consistently better than any of the alternatives mentioned here.

```
(0 , 1 , 2 , 3 , 4 .  5 , 6 , 7)
```

becomes

```
(0 , 1 , 3 , 4 , 5 , 2 , 6 , 7)
```

Inversion mutation (IVM)

This is a very simple mutation operator. Select two random points and reverse the cities between them.

```
(0 , 1 , 2 , 3 , 4 , 5 , 6 , 7)
```

becomes

```
(0 , 4 , 3 , 2 , 1 , 5 , 6 , 7)
```

Displaced Inversion Mutation (DIVM)

Select two random points, reverse the city order between the two points, and then displace them somewhere along the length of the original chromosome. This is similar to performing IVM and then DM using the same start and end points.

```
(0 , 1 , 2 , 3 , 4 , 5 , 6 , 7)
```
becomes
```
(0 , 6 , 5 , 4 , 1 , 2 , 3 , 7)
```

As with mutation operators, inventing crossover operators that spawn valid permutations has been a popular sport amongst genetic algorithm enthusiasts. Here are the descriptions of the better ones.

Order-Based Crossover (OBX)

To perform order-based crossover, several cities are chosen at random from one parent and then the order of those cities is imposed on the respective cities in the other parent. Let's take the example:

```
Parent 1:  (2 , 5 , 0 , 3 , 6 , 1 , 4 , 7)
Parent 2:  (3 , 4 , 0 , 7 , 2 , 5 , 1 , 6)
```

The underlined cities are the cities which have been chosen at random. Now, impose the order – 5, 0, then 1 – on the same cities in `Parent 2` to give `Offspring 1` like so:

```
Offspring 1:  (3 , 4 , 5 , 7 , 2 , 0 , 1 , 6)
```

City one stayed in the same place because it was already positioned in the correct order. Now the same sequence of actions is performed on the other parent. Using the same positions as the first:

```
Parent1:  (2 , 5 , 0 , 3 , 6 , 1 , 4 , 7)
Parent2:  (3 , 4 , 0 , 7 , 2 , 5 , 1 , 6)
```

`Parent 1` becomes:

```
Offspring 2:  (2 , 4 , 0 , 3 , 6 , 1 , 5 , 7)
```

Position-Based Crossover (PBX)

This is similar to Order-Based Crossover, but instead of imposing the order of the cities, this operator imposes the position. So, using the same example parents and random positions, here is how to do it.

```
Parent1:  (2 , 5 , 0 , 3 , 6 , 1 , 4 , 7)
Parent2:  (3 , 4 , 0 , 7 , 2 , 5 , 1 , 6)
```

First, move over the selected cities from `Parent 1` to `Offspring 1`, keeping them in the same position.

```
Offspring 1:  (* , 5 , 0 , * , * , 1 , * , *)
```

Now, iterate through `Parent 2`'s cities and fill in the blanks if that city number has not already appeared. In this example, filling in the blanks results in:

```
Offspring 1:  (3 , 5 , 0 , 4 , 7 , 1 , 2 , 6)
```

Get it? Let's run through the derivation of `Offspring 2`, just to be sure. First, copy over the selected cities into the same positions.

```
Offspring 2:  (* , 4 , 0 , * , * , 5 , * , *)
```

Now, fill in the blanks.

```
Offspring 2:  (2 , 4 , 0 , 3 , 6 , 5 , 1 , 7)
```

Now that you've seen how others have tackled the crossover operator, can you dream up one of your own? This is not an easy task, so congratulations if you can actually invent one! We hope running through a few of

the alternative operators for the traveling salesman problem has given you an indication of the scope you can have with genetic algorithm operators.

5.6 Extended mechanisms

For the remainder of this chapter, we are going to discuss various tools and techniques you can apply to just about any kind of genetic algorithm to improve its performance.

5.6.1 Elitism

As previously discussed, elitism is a way of guaranteeing that the fittest members of a population are retained for the next generation. To expand on this, it can be better to select n copies of the top m individuals of the population to be retained. It is often find that retaining about 2-5% of the population size gives good results.

It is worth to note that using elitism works well with just about every other technique described in this chapter.

5.6.2 Steady state selection

Steady state selection works a little like elitism, except that instead of choosing a small amount of the best individuals to go through to the new generation, steady state selection retains all but a few of the worst performers from the current population. The remainder are then selected using mutation and crossover in the usual way. Steady state selection can prove useful when tackling some problems, but most of the time it s inadvisable to use it.

5.6.3 Fitness proportionate selection

Selection techniques of this type choose offspring using methods which give individuals the better chance of being selected the better their fitness score. Another way of describing it is that each individual has an expected number of times it will be chosen to reproduce. This expected value equates to the individual's fitness divided by the average fitness of the entire population. So, if you have an individual with a fitness of 6 and the average fitness of

the overall population is 4, then the expected number of times the individual should be chosen is 1.5.

5.6.4 Tournament selection

To use tournament selection, n individuals are selected at random from the population, and then the fittest of these genomes is chosen to add to the new population. This process is repeated as many times as is required to create a new population of genomes. Any individuals selected are not removed from the population and therefore can be chosen any number of times.

This technique is very efficient to implement because it does not require any of the preprocessing or fitness scaling sometimes required for roulette wheel selection and other fitness proportionate techniques. Because of this, and because it is a darn good technique anyway, you should always try this method of selection with your own genetic algorithms. The only drawback is that tournament selection can lead to too quick convergence with some types of problems. There is an alternative description of this technique: A random number is generated between 0 and 1. If the random number is less than a pre-determined constant, for example T (a typical value would be 0.75), then the fittest individual is chosen to be a parent. If the random number is greater than T then the weaker individual is chosen. As before, this is repeated until a new population of the correct size has been spawned.

Chapter 6

Ant colony optimization (ACO)

6.1 The biological motivation

The inspiring source of ACO is the pheromone trail laying and following behavior of real ants which use pheromones as a communication medium. In analogy to the biological example, ACO is based on the indirect communication of a colony of simple agents, called (artificial) ants, mediated by (artificial) pheromone trails.

The whole idea of ACO is motivated by observing the behavior of real (living) ants in the laboratory. In many ant species, individual ants may deposit a pheromone (a particular chemical that ants can smell) on the ground while walking. By depositing pheromone they create a trail that is used, e.g., to mark the path from the nest to food sources and back. In fact, by sensing pheromone trails foragers can follow the path to food discovered by other ants. Also, they are capable of exploiting pheromone trails to choose the shortest among the available paths leading to the food. Deneubourg and his colleagues used a double bridge connecting a nest of ants and a food source to study pheromone trail laying and following behavior in controlled experimental conditions. They ran a number of experiments in which they varied the ratio between the length of the two branches of the bridge. The most interesting, for our purposes, of these experiments is the one in which one branch was longer than the other. In this experiment, at the start the

ants were left free to move between the nest and the food source and the percentage of ants that chose one or the other of the two branches was observed over time. The outcome was that, although in the initial phase random oscillations could occur, in most experiments all the ants ended up using the shorter branch.

This result can be explained as follows. When a trial starts there is no pheromone on the two branches. Hence, the ants do not have a preference and they select with the same probability either of the two branches. Therefore, it can be expected that, on average, half of the ants choose the short branch and the other half the long branch, although stochastic oscillations may occasionally favor one branch over the other. However, because one branch is shorter than the other, the ants choosing the short branch are the first to reach the food and to start their travel back to the nest. But then, when they make a decision between the short and the long branch, the higher level of pheromone on the short branch biases their decision in its favor. Therefore, pheromone starts to accumulate faster on the short branch which will eventually be used by the great majority of the ants.

It should be clear by now how real ants have inspired AS and later algorithms: the double bridge was substituted by a graph, and pheromone trails by artificial pheromone trails. Also, because we wanted artificial ants to solve problems more complicated than those solved by real ants, we gave artificial ants some extra capacities, like a memory (used to implement constraints and to allow the ants to retrace their path back to the nest without errors) and the capacity for depositing a quantity of pheromone proportional to the quality of the solution produced (a similar behavior is observed also in some real ants species in which the quantity of pheromone deposited while returning to the nest from a food source is proportional to the quality of the food source found).

6.2 The ACO algorithm

Algorithm 4 presents the template algorithm for ACO. First, the pheromone information is initialized. The algorithm is mainly composed of two iterated steps: solution construction and pheromone update. An optional *daemon actions* can also take place.

Algorithm 4 Basic scheme of Ant Colony Optimization algorithm

```
1: Initialize the pheromone trails
2: while stopping condition is not satisfied do
3:   for each ant do
4:     Construct a solution using pheromone trail
5:   end for
6:   Update the pheromone trails
7:   Execute optional daemon actions
8: end while
```

6.2.1 The artificial ants

Artificial ants used in ACO are stochastic solution construction procedures that probabilistically build a solution by iteratively adding solution components to partial solutions by taking into account:

- heuristic information on the problem instance being solved, if available, and
- (artificial) pheromone trails which change dynamically at run-time to reflect the agents' acquired search experience.

A stochastic component in ACO allows the ants to build a wide variety of different solutions and hence explore a much larger number of solutions than greedy heuristics. At the same time, the use of heuristic information, which is readily available for many problems, can guide the ants towards the most promising solutions.

Informally, the behavior of ants in an ACO algorithm can be summarized as follows. A colony of ants concurrently and asynchronously move through adjacent states of the problem by building paths on some graph G. They move by applying a stochastic local decision policy that makes use of pheromone trails and heuristic information. By moving, ants incrementally build solutions to the optimization problem. Once an ant has built a solution, or while the solution is being built, the ant evaluates the (partial) solution and deposits pheromone trails on the components or connections it used. This pheromone information will direct the search of future ants.

Besides ants' activity, an ACO algorithm includes two additional procedures: *pheromone trail update* and optional *daemon actions*.

6.2.2 Pheromone update and daemon actions

The update of the pheromone is carried out using the generated solutions. A global pheromone updating rule is applied in two phases:

- *Pheromone evaporation* is the process by means of which the pheromone deposited by previous ants decreases over time. From a practical point of view, pheromone evaporation is needed to avoid a too rapid convergence of the algorithm towards a suboptimal region. It implements a useful form of forgetting, favoring the exploration of new areas of the search space.

- *A reinforcement phase* where the pheromone trail is updated according to the generated solutions.

Three different strategies of reinforcement phase may be applied

- *Online step-by-step pheromone update:* The pheromone trail is updated by an ant at each step of the solution construction

- *Online delayed pheromone update:* The pheromone update is applied once an ant generates a complete solution. For instance, each ant will update the pheromone information with a value that is proportional to the quality of the solution found. The better the solution found, the more accumulated the pheromone.

- *Off-line pheromone update:* The pheromone trail update is applied once all ants generate a complete solution. This is the most popular approach.

Daemon actions can be used to implement centralized actions which cannot be performed by single ants. Examples are the activation of a local optimization procedure, or the collection of global information that can be used to decide whether it is useful or not to deposit additional pheromone to bias the search process from a non-local perspective. As a practical example, the daemon can observe the path found by each ant in the colony and choose to deposit extra pheromone on the components used by the ant that built the best solution. Pheromone updates performed by the daemon are called *off-line pheromone updates*.

6.3 An example of ACO

Let us consider the application of ACO algorithms to the TSP problem. Designing an ACO algorithm for the TSP needs the definition of the pheromone trails and the solution construction procedure.

6.3.1 Pheromone trails

A pheromone t_{ij} will be associated with each edge (i, j) of the graph G. The pheromone information can be represented by an $n \times n$ matrix T where each element t_{ij} of the matrix expresses the desirability to have the edge (i, j) in the tour. The pheromone matrix is generally initialized by the same values. During the search, the pheromone will be updated to estimate the utility of any edge of the graph.

6.3.2 Solution construction

Each ant will construct a tour in a stochastic way. Given an initial arbitrary city i, an ant will select the next city j with the probability

$$P_{ij} = \frac{t_{ij}}{\sum_{k \in M} t_{ik}}, \text{ for each } j \in M,$$

where M is the set of all not yet visited cities. The ants use a randomly selected initial city in the construction phase.

The additional problem-dependent heuristic is defined by considering the values $q_{ij} = \frac{1}{d_{ij}}$, where d_{ij} represents the distance between the cities i and j. The higher the heuristic value q_{ij}, the shorter the distance d_{ij} between cities i and j. Computing the decision transition probabilities, P_{ij} is performed as follows:

$$P_{ij} = \frac{t_{ij}^{\alpha} q_{ij}^{\beta}}{\sum_{k \in M} t_{ik}^{\alpha} q_{ik}^{\beta}}, \text{ for each } j \in M,$$

where α and β are controlling parameters. If $\alpha = 0$, the proposed ACO algorithm will be similar to a stochastic greedy algorithm in which the closest cities are more likely selected. If $\beta = 0$, only the pheromone trails will guide the search. In this case, a rapid emergence of *stagnation* may occur where all ants will construct the same suboptimal tour. Hence, a good trade-off must be found in using those two kinds of information.

Then, the pheromone update procedure has to be specified. For instance, each ant will increment the pheromone associated with the selected edges in a manner that is proportional to the quality of the obtained tour:

$$t_{ij} = t_{ij} + \delta,$$

where $\delta = 1/f$, and f is the length of the tour computed by an ant. Then, good tours will emerge as the result of the cooperation between ants through the pheromone trails.

6.3.3 Pheromone evaporation

The classical evaporation procedure is applied for the pheromone trails. For each edge, its pheromone t_{ij} will evaporate as follows:

$$t_{ij} = (1 - \varepsilon)t_{ij},$$

where $\varepsilon \in [0, 1]$ represents the reduction rate of the pheromone.

Initializing the numerous parameters of ACO algorithms is critical. Some sensitive parameters such as α and β can be adjusted in a dynamic or an adaptive manner to deal with the classical trade-off between intensification and diversification during the search. The optimal values for the parameters α and β are very sensitive to the target problem. The number of ants is not a critical parameter. Its value will mainly depend on the computational capacity of the experiments.

Chapter 7

Artificial Immune Systems (AIS)

7.1 Introduction

Immunology is a relatively new science. Its origin is addressed to Edward Jenner, who discovered, over 200 years ago (in 1796) that the *vaccinia*, or *cowpox*, induced protection against human smallpox, a frequently lethal disease. Jenner called his process *vaccination*, an expression that still describes the inoculation of healthy individuals with weakened, or attenuated samples of agents that cause diseases, aiming at obtaining protection against these diseases. When Jenner introduced the vaccination, nothing was known about the ethnological agent of immunology. Nowadays, humans still do not know exactly how we fight the pathogens, but much progress has been made in this matter. This progress lead us to the concept of *artificial immune systems (AIS)*.

Recently, attention has been drawn to mimic biological immune systems for the development of novel optimization algorithms. Indeed, the immune system is highly robust, adaptive, inherently parallel, and self-organized. It has powerful learning and memory capabilities and presents an evolutionary type of response to infectious foreign elements. An AIS may be seen as an adaptive system that is inspired by theoretical immunology and observed immune processes.

The biological processes that are simulated to design AIS algorithms

include *pattern recognition, clonal selection for B-cells, negative selection of T cells, affinity maturation, danger theory,* and *immune network theory.* In this chapter we will focus on *clonal selection algorithm,* which is highly related to the genetic algorithms.

It should be noted that as Artificial Immune Systems is still a young and evolving field, there is not yet a fixed algorithm template.

7.2 Natural immune system

The main purpose of the immune system is to keep the organism free from pathogens that are unfriendly foreign microorganisms, cells, or molecules. The immune system is a complex set of cells, molecules, and organs that represents an identification mechanism capable of perceiving and combating a dysfunction from our own cells (*infectious self*), such as tumors and cancerous cells, and the action of exogenous microorganisms (*infectious non-self*). In the surface of pathogens like viruses, fungi, bacteria, and parasites, there are antigens (Ag) that simulate the immune responses. Antigens represent substances such as toxins or enzymes in the microorganisms that the immune system considers foreign. There exist two types of immunities: the *innate immune system* and the *adaptive immune system.*

The innate immune system plays a crucial role in the initiation and regulation of immune responses. The innate system is called so because the body is born with the ability to recognize a microbe and destroy it immediately. It protects our body from nonspecific pathogens.

The adaptive immune system completes the innate one and removes the specific pathogens that persist to it. The adaptive immunity is made essentially by lymphocytes, a certain type of white blood cells that has two types of cells: the *B cells* and the *T cells.* These cells are responsible for recognizing and destroying any antigen.

7.2.1 The cells of immune system

The immune system is composed of a great variety of cells that are originated in the bone marrow, where plenty of them mature. From the bone marrow, they migrate to patrolling tissues, circulating in the blood and lymphatic vessels. Some of them are responsible for the general defense, whereas others are "trained" to combat highly specific pathogens. For an

efficient functioning, a continuous cooperation among the agents (cells) is necessary.

Lymphocytes are small leukocytes that possess a major responsibility in the immune system. There are two main types of lymphocytes: B lymphocyte (or B cell), which, upon activation, differentiate into *plasmocyte* (or plasma cells) capable of secreting antibodies; and T lymphocyte (or T cell). Most of the lymphocytes is formed by small resting cells, which only exhibit functional activities after some kind of interaction with the respective antigens, necessary for proliferation an specific activation. The B and T lymphocytes express, on their surfaces, receptors highly specific for a given antigenic determinant. The B cell receptor is a form of the antibody molecule bound to the membrane, and which will be secreted after the cell is appropriately activated.

B cells and antibodies

The main functions of the B cells include the production and secretion of antibodies (Ab) as a response to exogenous proteins like bacteria, viruses and tumor cells. Each B cell is programmed to produce a specific antibody. The antibodies are specific proteins that recognize and bind to another particular protein. The production and binding of antibodies is usually a way of signaling other cells to kill, ingest or remove the bound substance. As the antibody molecule represents one of the most important recognition devices of the immune system.

T cells

The T cells are called so because they mature within the thymus. Their function include the regulation of other cells' actions and directly attack the host infected cells. The T lymphocytes can be subdivided into three major subclasses: T helper cells (Th), cytotoxic (killer) T cells and suppressor T cells. The T helper cells, or Th cells for short, are essential to the activation of the B cells, other T cells, macrophages and natural killer (NK) cells. They are also known as CD4 or T4 cells. The killer T cells, or cytotoxic T cells, are capable of eliminating microbial invaders, viruses or cancerous cells. Once activated and bound to their ligands, they inject noxious chemicals into the other cells, perforating their surface membrane and causing their destruction. The suppressor T lymphocytes are vital for the maintenance of the

immune response. They are sometimes called CD8 cells, and inhibit the action of other immune cells. Without their activity, immunity would certainly loose control resulting in allergic reactions and autoimmune diseases. The T cells work, primarily, by secreting substances, known as cytokines or, more specifically, lymphokines and their relatives, the monokines produced by monocytes and macrophages. These substances constitute powerful chemical messengers. The lymphokines promote cellular growth, activation and regulation. In addition, lymphokines can also kill target cells and stimulate macrophages.

Natural killer cells

The natural killer cells (NK) constitute another kind of lethal lymphocytes. Like the killer T cells, they contain granules filled with powerful chemicals. They are designated natural killers because, unlike the killer T cells, they do not need to recognize a specific antigen before they start acting. They attack mainly tumors and protect against a great variety of infectious microbes. These cells also contribute to the immune regulation, secreting large amounts of lymphokines.

7.2.2 How it all works?

As discussed previously, our body is protected by a diverse army of cells and molecules that work in concert with each other, where the ultimate target of all immune responses is an antigen (Ag), which is usually a foreign molecule from a bacterium or other invader.

Specialized antigen presenting cells (APCs), such as macrophages, roam the body, ingesting and digesting the antigens they find and fragmenting them into antigenic peptides. Pieces of these peptides are joined to major histocompatibility complex (MHC) molecules and are displayed on the surface of the cell. Other white blood cells, called T cells or T lymphocytes, have receptor molecules that enable each of them to recognize a different peptide-MHC combination. T cells activated by that recognition divide and secrete lymphokines, or chemical signals, that mobilize other components of the immune system. The B lymphocytes, which also have receptor molecules of a single specificity on their surface, respond to those signals. Unlike the receptors of T cells, however, those of B cells can recognize parts of antigens free in solution, without MHC molecules. When activated, the

B cells divide and differentiate into plasma cells that secrete antibody proteins, which are soluble forms of their receptors. By binding to the antigens they find, antibodies can neutralize them or precipitate their destruction by complement enzymes or by scavenging cells. Some T and B cells become memory cells that persist in the circulation and boost the immune system's readiness to eliminate the same antigen if it presents itself in the future. Because the genes for antibodies in B cells frequently suffer mutation and editing, the antibody response improves after repeated immunizations, this phenomenon is called affinity maturation.

Through the recognition and distinction of specific molecular patterns, the antibodies play a central role in the immune system. Antigens are diverse in structure, forcing the antibody repertoire to be large. The genetic information necessary to code for this exceedingly large number of different, but related, proteins is stored in the genome of a germline cell and transmitted through generations.

7.3 The clonal selection principle

The clonal selection principle is the algorithm used by the immune system to describe the basic features of an immune response to an antigenic stimulus. It establishes the idea that only those cells that recognize the antigens proliferate, thus being selected against those which do not. Clonal selection operates on both T cells and B cells. The immune response occurs inside the lymph nodes and the clonal expansion of the lymphocytes occurs within the germinal centers (GCs), in the follicular region of the white pulp, which is rich in antigen presenting cells. When an animal is exposed to an antigen, some subpopulation of its bone marrow's derived cells (B lymphocytes) respond by producing antibodies. Each cell secretes only one kind of antibody, which is relatively specific for the antigen. By binding to these immunoglobulin receptors, with a second signal from accessory cells, such as the T-helper cell, an antigen stimulates the B cell to proliferate (divide) and mature into terminal (non-dividing) antibody secreting cells, called plasma cells. While plasma cells are the most active antibody secretors, large B lymphocytes, which divide rapidly, also secrete Ab, albeit at a lower rate. While B cells secrete Ab, T cells do not secrete antibodies, but play a central role in the regulation of the B cell response and are preeminent in cell mediated immune responses. Lymphocytes, in addition to proliferating or differentiating into plasma cells, can differentiate into long-lived B memory

cells. Memory cells circulate through the blood, lymph and tissues, probably not manufacturing antibodies, but when exposed to a second antigenic stimulus commence differentiating into large lymphocytes capable of producing high affinity antibody, preselected for the specific antigen that had stimulated the primary response. The main features of the clonal selection theory are:

- the new cells are copies of their parents (clone) subjected to a mutation mechanism with high rates (somatic hypermutation),
- elimination of newly differentiated lymphocytes carrying self-reactive receptors,
- proliferation and differentiation on contact of mature cells with antigens;
- the persistence of forbidden clones, resistant to early elimination by self-antigens, as the basis of autoimmune diseases.

The analogy with natural selection should be obvious, the fittest clones being the ones that recognize antigen best or, more precisely, the ones that are triggered best. For this algorithm to work, the receptor population or repertoire, has to be diverse enough to recognize any foreign shape. A mammalian immune system contains an heterogeneous repertoire of approximately 10^{12} lymphocytes in human, and a resting (unstimulated) B cell may display around 10^5–10^7 identical antibody-like receptors. The repertoire is believed to be complete, which means that it can recognize any shape.

7.3.1 Hypermutation

In a T cell dependent immune response, the repertoire of antigen-activated B cells is diversified basically by two mechanisms: hypermutation and receptor editing. Only high-affinity variants are selected into the pool of memory cells. This maturation process takes place in a special micro environment called germinal center (GC).

Antibodies present in a memory response have, on average, a higher affinity than those of the early primary response. This phenomenon, which is restricted to T-cell dependent responses, is referred to as the maturation of the immune response. This maturation requires that the antigen binding sites of the antibody molecules in the matured response be structurally different from those present in the primary response. Three different kinds of

mutational events have been observed in the antibody V-region: *point mutations*, *short deletions*, and *non-reciprocal exchange of sequence following gene conversion* (repertoire shift).

Random changes are introduced into the variable region genes and occasionally one such change will lead to an increase in the affinity of the antibody. It is these higher-affinity variants which are then selected to enter the pool of memory cells. Not only the repertoire is diversified through a hypermutation mechanism but, in addition, mechanisms must exist such that rare B cells with high affinity mutant receptors can be selected to dominate the response. Due to the random nature of the somatic mutation process, a large proportion of mutating genes become non-functional or develop harmful anti-self specificities. Those cells with low affinity receptors, or the self-reactive cells, must be efficiently eliminated (or become anergic) so that they do not significantly contribute to the pool of memory cells. How B cells with compromised antigen binding abilities are eliminated is not fully understood. Apoptosis in the germinal centers is likely. Apoptosis is a subtle cell death process, often equated with programmed cell death.

The analysis of the development of the antibody repertoire expressed on B cells in the germinal center has clearly demonstrated the key role that these structures play in the maturation of the immune response. Both processes are of vital importance for the maturation — hypermutation of the variable region and selection of higher-affinity variants. The increase in antibody affinity from the primary to the secondary response, and so on, shows that maturation of the immune response is a continuous process.

A hypermutation mechanism is quite rapid. On average one mutation per cell division will be introduced. A rapid accumulation of mutations is necessary for a fast maturation of the immune response, but the majority of the changes will lead to poorer or non-functional antibodies. If a cell that has just picked up a useful mutation continues to be mutated at the same rate during the next immune responses, then the accumulation of deleterious changes may cause the loss of the advantageous mutation. Thus, a short burst of somatic hypermutation, followed by a breathing space to allow for selection and clonal expansion, may form the basis of the maturation process. The selection mechanism may provide a means by which the regulation of the hypermutation process is made dependent on receptor affinity. Cells with low affinity receptors may be further mutated and, as a rule, die through apoptosis. In cells with high-affinity antibody receptors however, hypermutation may be inactivated.

7.3.2 The clonal selection vs. genetic algorithms

The clonal selection functioning of the immune system reveals it to be a remarkable microcosm of Charles Darwin's law of evolution, with the three major principles of repertoire diversity, variation and natural selection, each playing an essential role. Repertoire diversity is evident in that the immune system produces far more antibodies than will be effectively used in binding with an antigen. In fact, it appears that the majority of antibodies produced do not play any active role whatsoever in the immune response. As previously discussed, natural variation is provided by the variable gene regions responsible for the production of highly diverse population of antibodies, and selection occurs, such that only antibodies able to successfully bind with an antigen will reproduce and maintained as memory cells. The similarity between adaptive biological evolution and the production of antibodies is even more striking when one considers that the two central processes involved in the production of antibodies, genetic recombination and mutation, are the same ones responsible for the biological evolution of sexually reproducing species. The recombination and editing of immunoglobulin genes underlies the large diversity of the antibody population, and the mutation of these genes serves as a fine-tuning mechanism (see Section 4). In sexually reproducing species, the same two processes are involved in providing the variations on which natural selection can work to fit the organism to the environment (Holland, 1995). Thus, cumulative blind variation and natural selection, which over many millions of years resulted in the emergence of mammalian species, remain crucial in the day-by-day ceaseless battle to survival of these species. It should also be noted that recombination of immunoglobulin genes involved in the production of antibodies differs somewhat from the recombination of parental genes in sexual reproduction. In the former, nucleotides can be inserted and deleted at random from recombined immunoglobulin gene segments and the latter involves the crossing-over of parental genetic material, generating an offspring that is a genetic mixture of the chromosomes of its parents. Whereas adaptive biological evolution proceeds by cumulative natural selection among organisms, research on the immune system has now provided the first clear evidence that ontogenetic adaptive change can be achieved by cumulative blind variation and selection within organisms. The natural selection can be seen to act on the immune system at two levels. First, on multiplying lymphocytes for selection of higher affinity clones for reaction with pathogenic microbes. Second, on multiplying people for selection of the germ-line genes that are

most able to provide maximal defense against infectious diseases coupled with minimal risk of autoimmune disease.

7.4 An example of the clonal selection algorithm (CSA)

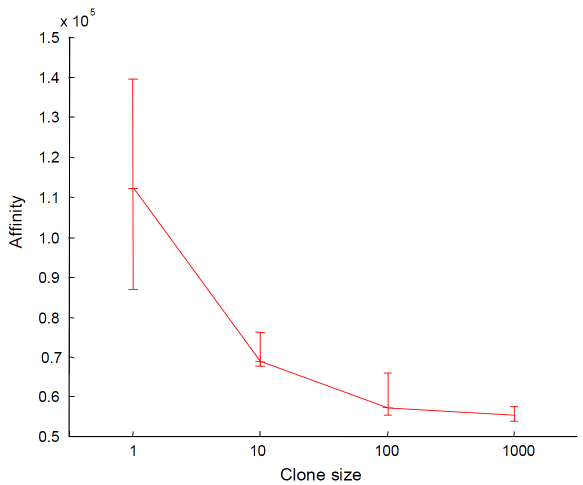

Figure 7.1: Trade-off between the speed of the maturation of the population and the clone size, for the TSP

The clonal selection principle is used by the immune system to describe the basic features of an immune response to an antigenic stimulus. It establishes the idea that only those cells that recognize the antigens proliferate, thus being selected against those which do not. The selected cells are subject to an affinity maturation process, which improves their affinity to the selective antigens. In this section, we present a the clonal selection algorithm, which takes into account the affinity maturation of the immune response. The algorithm is capable of solving TSP problem. The main immune aspects taken into account in the CSA are as follows:

- maintenance of the memory cells functionally disconnected from the

repertoire,

- selection and cloning of the most stimulated individuals,
- death of non-stimulated cells,
- affinity maturation and re-selection of the higher affinity clones,
- generation and maintenance of diversity, and
- hypermutation proportional to the cell affinity.

The algorithm works as follows depicted in Algorithm 5.

Algorithm 5 Basic scheme of CSA algorithm

1: Generate a set of candidate solutions, composed of the subset of memory cells added to the remaining population.
2: **while** solution is not found **do**
3: Determine the m best individuals of the population, based on an affinity measure.
4: Clone (reproduce) these m best individuals of the population, giving rise to a temporary population of clones. The clone size is an increasing function of the affinity measure of the antigen.
5: Submit the population of clones to a hypermutation scheme, where the hypermutation is proportional to the affinity of the antibody. A maturated antibody population is generated.
6: Re-select the improved individuals from matured antibody population to compose the memory set. Some members of the population can be replaced by other improved members of matured antibody population.
7: Replace low affinity antibodies of the population, maintaining its diversity.
8: **end while**

The algorithm was allowed to run 20 generations, with a population of size $M = 10$. In the Figure 7.1 we present the trade-off between the speed of the repertoire maturation and the clone size, for the TSP problem. The maximum, minimum and mean values, taken over ten runs are presented. We can notice that, the larger the clone size, i.e., the size of the intermediate population of clones, the faster the reach of local optima.

Figures 7.2–7.7 presents how the immune algorithm evolves the best solution for a population of 300 individuals, with a rate of 20% of newcomers. In this case, low affinity individuals are allowed to enter the repertoire after each 20 generations. This scheduling is supposed to leave a breathing space to allow for the achievement of local optima, followed by the replacement of the poorer ones.

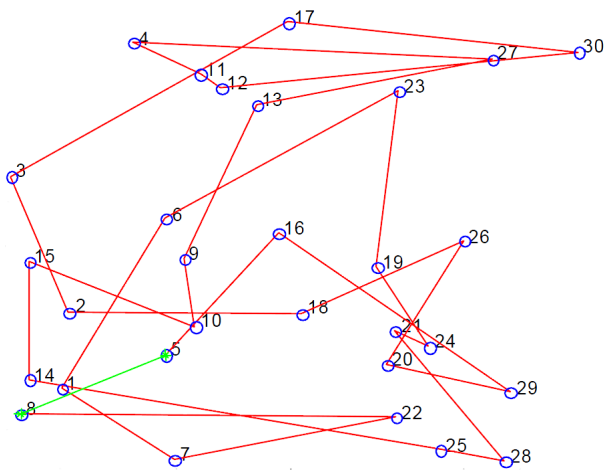

Figure 7.2: Solution obtained at 10th generation of the CSA for exemplary TSP instance

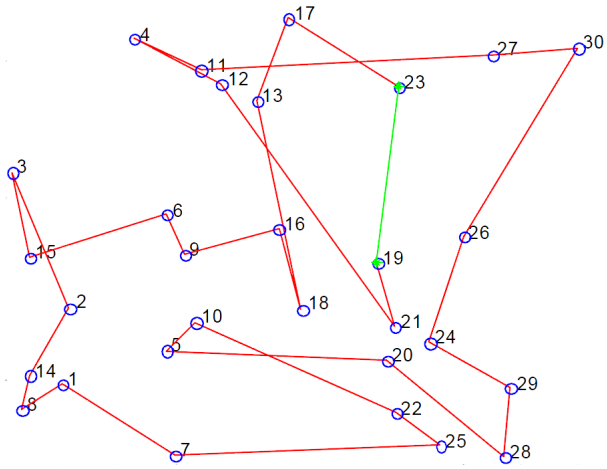

Figure 7.3: Solution obtained at 50th generation of the CSA for exemplary TSP instance)

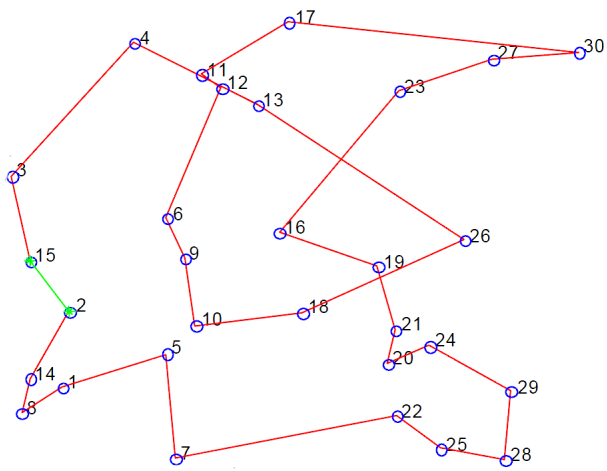

Figure 7.4: Solution obtained at 100th generation of the CSA for exemplary TSP instance)

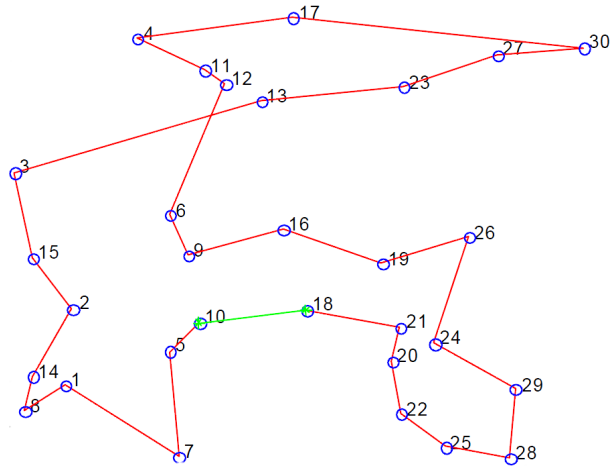

Figure 7.5: Solution obtained at 200th generation of the CSA for exemplary TSP instance)

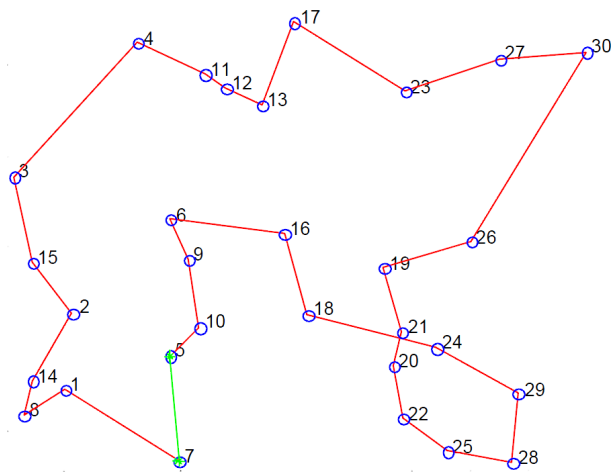

Figure 7.6: Solution obtained at 250th generation of the CSA for exemplary TSP instance)

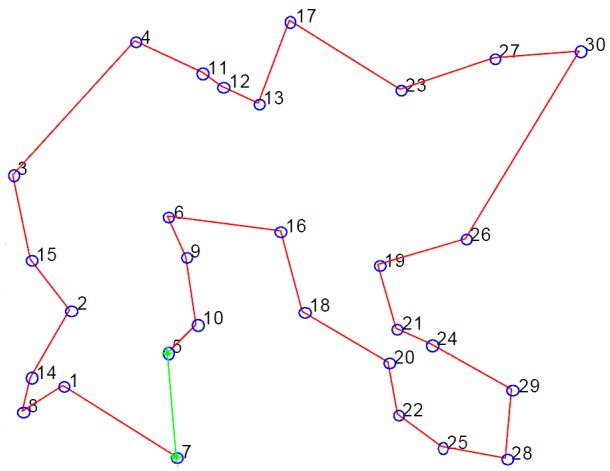

Figure 7.7: Solution obtained at 300th generation of the CSA for exemplary TSP instance)

Chapter 8

Further reading

Anyone who wants to improve oneself skills in the topics discussed in this book should consider visiting library for good literature. In the following text we will propose some books and papers that deal with particular topics considered within this book. First of all, the best (in our opinion) book that deals with Nature-inspired methods of problem is [Tal09]. Some results can be also found in [AL97] and [dC06].

Regarding the optimization itself, we may refer the reader to the books [Bel57], [Ber04], [APP06], and [Ata99]. More advanced topics can be found in [Mie99], [Ehr05], [Apt03], and [DV99].

There are not so many books that deal with the Computational Complexity Theory. Historically the first results are in [Kar72]. One of the first books is [GJ79], however, it may be difficult to obtain it. For quite clear and complete presentation we refer the reader to the book [Weg05]. The most complete, however, somehow difficult to read (at least for us) is the book [Pap94]

The simulated annealing algorithm was introduced in [KGV83]. Some good texts regarding this topic can found in [AK89], [AL87], and [Aze92]. Similarly, the tabu search method was introduced in [Glo89] and [Glo90]. Good books that deal with tabu search are: [Gen02], [Glo96], and [Vos93].

The genetic algorithms are based on the theory of evolution introduced by Charles Darwin in [Dar59], whereas algorithms were introduced by Holland in [Hol75], and extended by his student in [Gol89]. There are large number of publications regarding GA method. We can advise the following book to read: [BFM00], [CVL02]. More popular-science approach can be

found in [Daw86].

Topics considered in this book regarding Ant Colony Optimization are more precisely described in [DB05], and [DS01]. Advanced topics in this matter are covered by [BM08] and [BDT99].

Finally, the good books regarding Artificial Immune Systems are [Das99] and [dCT99]. The background on the CSA algorithm is precisely presented in [Bur59].

Bibliography

[AK89] E. H. L. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*. Wiley, 1989.

[AL87] E. H. L. Aarts and R. J. M.Van Laarhoven. *Simulated Annealing: Theory and Applications*. Reidel, Dordrecht„ 1987.

[AL97] E. H. L. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization*. Wiley, 1997.

[APP06] G. Appa, L. Pitsoulis, and H. P.Williams. *Handbook on Modeling for Discrete Optimization*. Springer, 2006.

[Apt03] K. Apt. *Principles of Constraint Programming*. Cambridge University Press, 2003.

[Ata99] M. J. Atallah. *Handbook of Algorithms and Theory of Computing*. CRC Press, 1999.

[Aze92] R. Azencott. *Simulated Annealing: Parallelization Techniques*. Wiley, New York, 1992.

[BDT99] E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence: From Natural to Artificial Systems*. Oxford University Press, 1999.

[Bel57] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[Ber04] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, 2004.

[BFM00] T. Back, D. B. Fogel, and T. Michalewicz. *Evolutionary Computation: Basic Algorithms and Operators*. Institute of Physics Publishing, 2000.

[BM08] C. Blum and D. Merkle. *Swarm Intelligence: Introduction and Applications*. Springer, 2008.

[Bur59] F. M. Burnet. *The Clonal Selection Theory of Acquired Immunity*. University Press, Cambridge, 1959.

[CVL02] C. A. C. Coello, D. A. Van Veldhuizen, and G. B. Lamont. *Evolutionary Algorithms for Solving Multi-Objective Optimization Problems*. Kluwer Academic Publishers, 2002.

[Dar59] C. Darwin. *On the Origin of Species by Means of Natural Selection*. John Murray, London, 1859.

[Das99] D. Dasgupta. *Artificial Immune Systems and Their Applications*. Springer, 1999.

[Daw86] R. Dawkins. *The Blind Watchmaker*. Longman, 1986.

[DB05] M. Dorigo and C. Blum. Ant colony optimization theory: A survey. *Theoretical Computer Science*, 344:243–278, 2005.

[dC06] L. N. de Castro. *Fundamentals of Natural Computing*. Chapman & Hall, 2006.

[dCT99] L. N. de Castro and J. I. Timmis. *Artificial Immune Systems: A New Computational Intelligence Approach*. Springer, 1999.

[DS01] M. Dorigo and T. Stutzle. The ant colony optimization metaheuristic: Algorithms, applications and advances. *Handbook of Metaheuristics*, pages 251–285, 2001.

[DV99] A. Dean and D. Voss. *Design and Analysis of Algorithms*. Springer, 1999.

[Ehr05] M. Ehrgott. *Multicriteria Optimization*. Springer, 2005.

[Gen02] M. Gendreau. *An introduction to tabu search. In Handbook of Metaheuristics*. Kluwer, 2002.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[Glo89] F. Glover. Tabu search. Part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[Glo90] F. Glover. Tabu search. Part II. *ORSA Journal on Computing*, 2:4–32, 1990.

[Glo96] F. Glover. *Tabu search and adaptive memory programming. In Advances, Applications and Challenges*. Kluwer, 1996.

[Gol89] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[Hol75] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[Kar72] R.M. Karp. Reducibility among combinatorial problems. *in: R.E. Miler, J.W. Ratcher (eds), Complexity of Computer Computations, Plenum Press, NY*, pages 85–103, 1972.

[KGV83] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimisation by simulated annealing. *Science*, 220:671–680, 1983.

[Mie99] K. Miettinen. *Nonlinear Multiobjective Optimization*. Kluwer, 1999.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison Wesley, 1994.

[Tal09] El-Ghazali Talbi. *Metaheuristics. From desing to implementation*. A John Wiley and Sons, Inc., 2009.

[Vos93] S. Voss. *Tabu search: Applications and prospects. In Network Optimization Problems*. World Scientific, USA, 1993.

[Weg05] Ingo Wegener. *Complexity Theory. Exploring the Limits of Efficient Algorithms*. Springer, 2005.