3 Natural Solvers

A. Introduction

Natural solvers are methods inspired by processes in Nature, that help us solving a large range of problems, or help us finding and exploiting parallelism that is inherently present in natural systems. Natural solvers are a very promising class of solver techniques, in the sense that they allow to preserve domain properties in the mapping from model to parallel virtual machine. In this way natural solvers may be able to preserve inherent parallelism that can be found in natural phenomena.

The idea that in many natural processes a certain amount of computing takes place and that we should try to mimic these computational processes on a computer turns out to be very effective. For instance, Nature is very good at optimization, i.e. finding optimal solutions in highly dimensional spaces. Examples are slow cooling into perfect crystals, protein folding, and evolution itself. These natural processes inspired the development of such algorithms as *simulated annealing* and *genetic algorithms*. These two algorithms will be covered in detail in the following sections. The process of evolution has inspired a complete new scientific field, that of *evolutionary computing*. Nature is also very good in exploiting inherent parallelism. The prototypical examples are particle systems, where all particles follow dynamics according to some interaction potential. It turns out that particle methods can be quite powerful models, e.g. in the context of N-body simulations in astronomy or molecular dynamics in physics and chemistry. Also particle methods with highly constrained dynamics, so-called Lattice Gas and Lattice Boltzmann methods provide exiting new models for hydrodynamics. These two models are not part of this lecture, but are covered in detail in the lecture "Complex System Simulation". Furthermore, Nature is superb in learning and adaptation. The processes in the brain, although we still hardly understand them, have inspired the field of Artificial Neural Networks.

It is clear that many natural solvers exist, and we have to make a choice to which ones will be covered here. We will concentrate on two natural solvers that are very suited for optimization problems. First, we will discuss Simulated Annealing, inspired by the process of slow cooling into perfect regular crystals. Next we will study Genetic Algorithms, inspired by evolution. In both cases we will also discuss in detail parallel versions of these solvers.

B. Optimization

Many problems in Science and Engineering are optimization problems. You are confronted with a model that may have many parameters, and you need to optimize in some sense the model by tuning the parameters to optimal values. For instance, going back to the example of the roundabout in Chapter 1, you might want to know the optimal schedule of traffic lights on the roundabout, such that mean throughput of traffic is maximized and that the maximum waiting time for any car is always smaller than a certain value. This is an example of an optimization problem with a constraint. Another well know example from the chip industry is the layout of components on in a chip such that the wiring is minimized and heating of parts is constrained to a certain maximum.

In a mathematical optimization is the minimization of a cost function. For the roundabout the cost function is the mean waiting time, and for the chip layout it is the length of the wiring. In some cases it is possible to give a closed form mathematical formula for the cost function, in some case one must perform simulations to obtain the cost function. In any case, the cost function will have many parameters. The space that is spanned all these parameters is called configuration space. This is a highly dimensional space, and the cost function gives a mapping from this highly dimensional space to the space of real numbers. In most problems the number of possible configurations (if we can count them at all) is so large that explicit testing of all possibilities is just impossible.

Typically, a cost functions will have many local minima and maxima, as drawn schematically in Figure 1. Our task is now to find the global minimum or maximum of the cost function. Given the fact that we have a high dimensional configuration space and that the cost function has many local minima and maxima you may understand that these optimization problems are far from trivial.

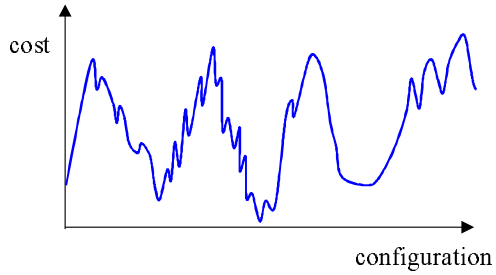


Figure 1: A typical behavior of a cost function in configuration space. Note that in reality configuration space is a highly dimensional space.

The canonical example of an optimization problem is an old time classic in the field of combinatorial optimization: the Travelling Salesman Problem (TSP). The TSP is easily stated, given N cities, and a $N \times N$ distance matrix where each entry ij gives the distance between city i and city j, find a *closed* tour such that each city is visited once and the total length of the tour is minimized. The TSP is easy to pose, but very hard to solve. It is well known that TSP is an NP-Hard problem. The total number of possible tours is $(N\text{-}1)!$, so indeed a huge configuration space. If we for instance would have $N = 33$ (the number of cities in a 1962 TSP context by Proctor and Gamble) the total number of possible tours is already 2.6×10^{35}, which is huge.

Solving a TSP with just a small number of cities is trivial. For a $N = 5$ TSP (with 24 possible tours), as drawn in Figure 2 the solution is easy. However, if the tour becomes larger the solution becomes more involved. For instance, consider a $N = 72$ TSP (with 1.7×10^{98} possible tours), see Figure 3. We show two possible solutions, quite different from each other, with a cost that is almost equal (and for this TSP quite low). However, did we find the global minimum? Is the left tour in Figure 3 the solution to the TSP? Did we get stuck in a local minimum or did we really find the global minimum? These are quite difficult questions.

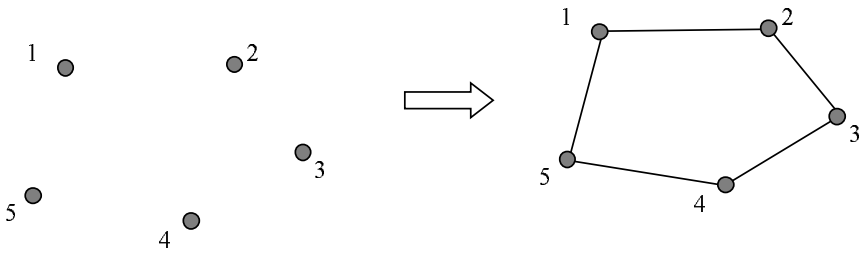


Figure 2: An example of a N *= 24 TSP with it's solution.*

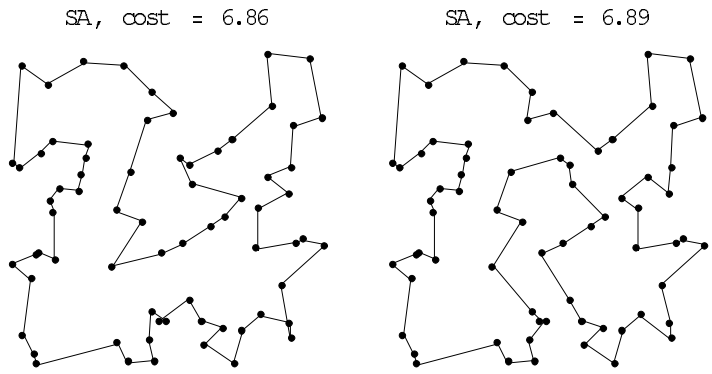


Figure 3: Two possible solutions to a 72 cities TSP. The solution to the left has a cost (i.e. length) of 6.86, the one on the right has a cost of 6.89.

Many highly specialized and tailored combinatorial optimization algorithms have been designed to tackle the TSP problem. These heuristic algorithms allow the solution of extremely large TSP. For instance, in 1998 Applegate, Bixby, and Chvatal, using a highly specialized branch and bound algorithm, were able to solve a $N = 13506$ TSP (connecting cities in the USA). The number of possible tours in this case is 1.1×10^{49932}. The solution is shown in Figure 4. For more information consult the TSP homepage (http://www.math.princeton.edu/tsp/index.html). By the time of writing the largest TSP solved ever was a $N = 15112$ TSP connecting cities in Germany. We will not delve in the specialised algorithms available for TSP, but use TSP as an example on the route to a powerful natural solver, i.e. Simulated Annealing.

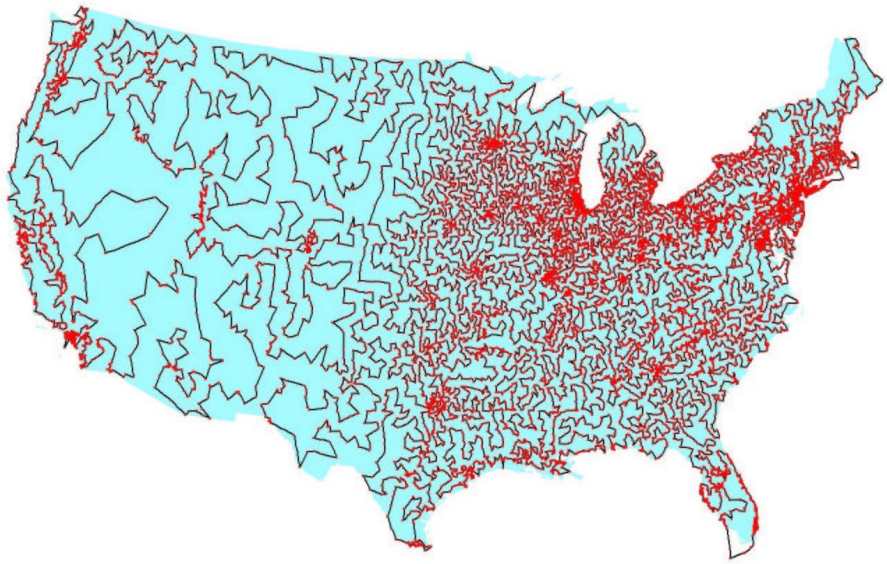


Figure 4: The solution, by Applegate, Bixby, and Chvatal, of a N = *13509 TSP.*

A tour in TSP is coded by a vector π. In the tour, city i is connected to city π_i, where the subscript i means the i-th element of π. The tour in Figure 5 has

$$\pi = \begin{pmatrix} 4 \\ 5 \\ 2 \\ 3 \\ 1 \end{pmatrix} \quad or \quad \begin{pmatrix} 5 \\ 3 \\ 4 \\ 1 \\ 2 \end{pmatrix}.$$

The cost C is easily calculated from π and the distance matrix d_{ij}, through

$$C = \sum_{i-1}^{N} d_{i,\pi_i} \ .$$

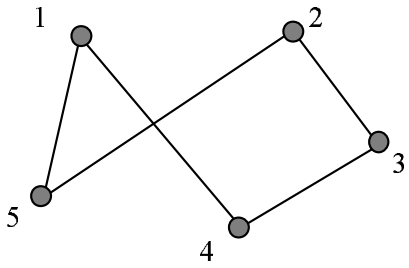


Figure 5: A possible tour in a N = 5 TSP.

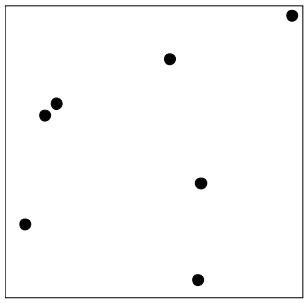


Figure 6: A 7 cities TSP, note the two cities very close together.

Let us now take a small TSP that still allows us to test all possible tours. We take a TSP with 7 randomly placed cities, see Figure 6. We have calculated the cost of all 6! = 720 tours, and have drawn this in Figure 7.

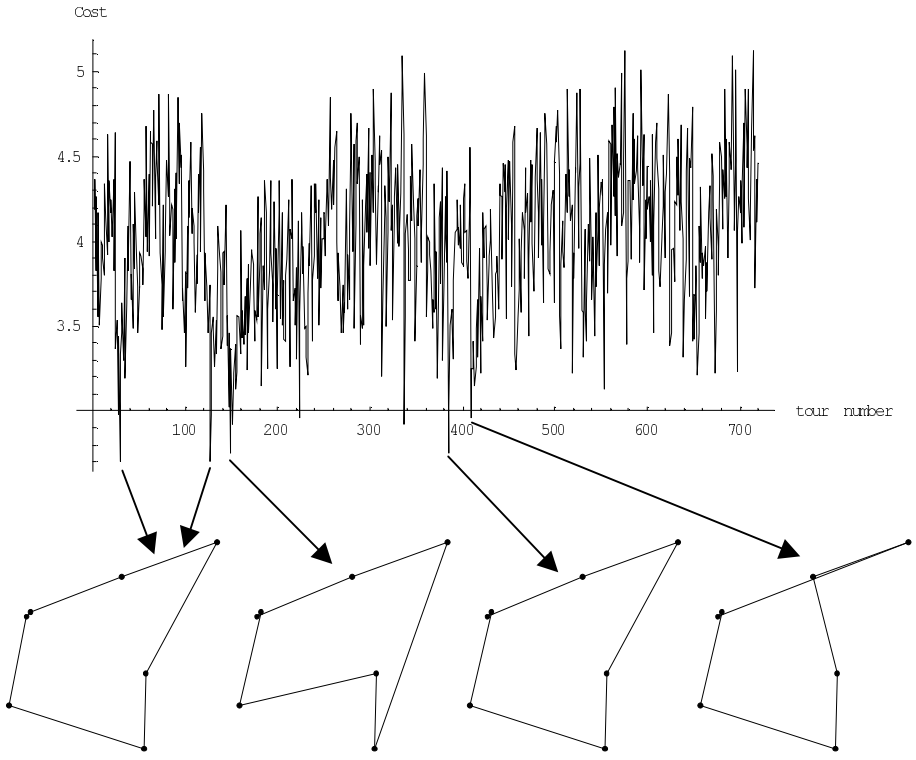


Figure 7: The cost function for all possible tours in the TSP of Figure 6. Also, for a number of points in the graph the associated tours are shown. The tour on the left corresponds to the two global minima (the shortest tour in clockwise and anti-clockwise direction). The other tours correspond to deep local minima in the cost function.

The cost function has two global minima (the shortest route in the clockwise and anti-clockwise direction). This is the most left tour in Figure 7. The other tours correspond to deep local minima of the cost function. For instance, the third tour (counting from left to right in Figure 7) has only interchanged the order in which the two cities very close together are visited. This tour is a very good candidate as the solution of the TSP, but

because we could calculate the cost all possible tours we know that an even better tour exists. For realistic optimization problems this is not possible.

Which method could we use to solve the optimization problem? The first and very naïve idea would be to just try all solutions and then take the best one. From the example of the TSP we already know that for discrete combinatorial optimization problems this is impossible because of the shear amount of possible solutions to the optimization problem. For continuous optimization problems trying all possibilities also is just impossible. A next, more clever idea would be to try an *iterative improvement* algorithm. In short, this works as follows. First we must have a way to calculate the cost function of configurations. Next, we need a procedure that generates a new configuration by applying a small perturbation to a given configuration. With these ingredients the iterative improvement procedure is shown in Algorithm 1.

```
start with a configuration
iterate
   generate next configuration in neighborhood
   if cost is lower, replace old configuration with new
   configuration
```

Algorithm 1: Iterative improvement algorithm.

The way to do this in TSP will be shown below. In continuous optimization problems the iterative improvement is implemented using a steepest descent algorithm. The idea is that you perturb the current configuration by calculating the gradient of the cost function in the current configuration, and that you then take a small step in configuration space in the direction of the gradient (or in the opposite direction, depending on what you are looking for, a maximum of minimum in the cost function). This algorithm is also know as *hill climbing*, for obvious reasons. The mean problem is that this works fine for smooth functions that have a single minimum or maximum. If the cost function has many local minima and maxima, then the outcome of the steepest descent algorithm depends on the initial starting condition. The probability that you get stuck in a local minimum is very high (for highly dimensional configuration spaces it is fair to say the probability that you get stuck in a local minimum or maximum is 1). This is shown in Figure 8.

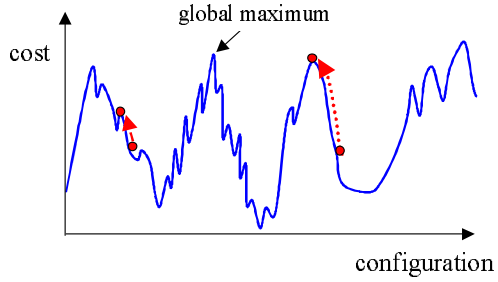


Figure 8: Steepest descent in searching for a maximum of a cost function. Two possible initial configurations are shown, that both get stuck in a local maximum which is not the global maximum.

To conclude, the disadvantages of the *greedy* iterative improvement is that it terminates in a local minimum and that the obtained local minimum depends on the initial configuration. Moreover, in general, it is not possible to give an upper bound for computation time. Possible solutions could be to

- execute with a large number of initial configurations;
- to use information gained from previous runs or introduce more complex generation mechanism, allowing to jump out of local minimum;
- accept, in a limited way, new configurations that correspond to an increase in the cost function.

This first solution will not work, it will still have the problems associated with the standard greedy iterative improvement. The other two solutions may give an interesting

route towards general purpose and robust solvers for optimization problems. The second solution, in a way, is how Genetic Algorithms operate. They will be discussed in section **Error! Reference source not found.**. The third solution is the main mechanism in Simulated Annealing, and this is the first natural solver that we will discuss.

C. Simulated Annealing

C.1. The Inspiration from Nature

In solid-state physics and other branches of science or e.g. in the semiconductor industry it is very important to get perfect crystals. That is, to get a piece of material were al the atoms are arranged in a perfect lattice. However, many of these crystal have defects, atoms are missing, or the crystal surface is not smooth but is roughened by kinks and steps, etc. There is a very nice experimental technique, called *annealing*, that allows for the creation of near perfect crystals. The idea is to heat a substance to a temperature just below the melting temperature, and next to let the substance cool very slowly to a final equilibrium situation, the perfect crystal.

The crystal can choose from many configurations of the atoms, and the perfect crystal corresponds to the minimal energy configuration. In that sense annealing is an "optimization calculation" with the perfect crystal the looked for solution. This calculation however can go wrong. If the crystal is not heated enough, or cooled to fast, it may get stuck in meta-stable states, corresponding to a local minimum in the energy landscape. This would result in e.g. amorphous silicon or glassy structures.

When the crystal is heated we put lots of energy in the system, the atoms start to vibrate wildly, and the system can explore many different configurations of the atoms. Even if the energy of a certain configuration is very high, it can still get there because we put thermal energy into the crystal. When we slowly decrease the temperature, the crystal will slowly settle in a low energy configuration. However, when this is a local energy minimum, it still may have enough thermal energy to jump out of this local minimum, and keep exploring configuration space to look for more favorable configurations.

From Statistical Mechanics we know that in thermal equilibrium, the probability that the crystal is in a state with energy E is given by the Boltzmann Distribution, with a probability density function

$$P(\mathrm{E} = E) = \frac{1}{Z(T)} \exp\left(-\frac{E}{kT}\right), \qquad [1]$$

where T is the temperature, k is the Boltzmann constant, and $Z(T)$ is a normalization function that normalizes the probability density function in the proper way. In Statistical Mechanics this function is known as the partition function. The $\exp(-E/kT)$ is the Boltzmann factor. At high temperature, the probability for any configuration is comparable, the complete space of all configurations (called the *phase space* in Statistical Mechanics) is accessible to the system. At very low temperatures, only the state with minimal energy has a finite probability. The Boltzmann distribution will be the main vehicle for our simulated annealing algorithm. We will use it to sample configurations from it.

C.2. From Nature to a Natural Solver

How would annealing translate to solving optimization problems? Table 1 shows the correspondence between both domains. The correspondence is clear. Our next task is to find a way to mimic annealing and develop a Simulated Annealing procedure. The first step is to define a Boltzmann distribution for the configurations. That is possible by the second correspondence in Table 1, resulting in

$$P(\text{configuration} = i) = \frac{1}{Z(c)} \exp\left(-\frac{C(i)}{c}\right). \qquad [2]$$

The energy is replaced by the cost function, and the temperature factor kT is replaced by a control variable c that slowly decreases to zero.

Statistical Mechanics		**Optimization**
configuration of the crystal	$\Rightarrow$	a possible configuration (e.g. a tour in the TSP)
energy of a configuration	$\Rightarrow$	cost of a configuration
annealing to a minimal energy	$\Rightarrow$	*simulated annealing* to a configuration with minimal cost, i.e. to the solution of the optimization problem

Table 1: The correspondence between the domain of Statistical Mechanics and optimization.

To better understand the Boltzmann distribution in Equation [2] we return to the example of the 7 cities TSP of the previous section. Because we have the cost function for this TSP for all configurations (see Figure 7) we can calculate the Boltzmann distribution as a function of the control variable, see Figure 9. For a very high "temperature" (c = 10) all configuration have more or less the same probability. If the "temperature" is decreased we start to see peaks with preferred configurations (i.e. configurations with smaller cost). Finally, for a very low "temperature" (c = 0.01) we find only two configurations with a finite probability. These configuration correspond to the configurations with the lowest cost function (see Figure 7).

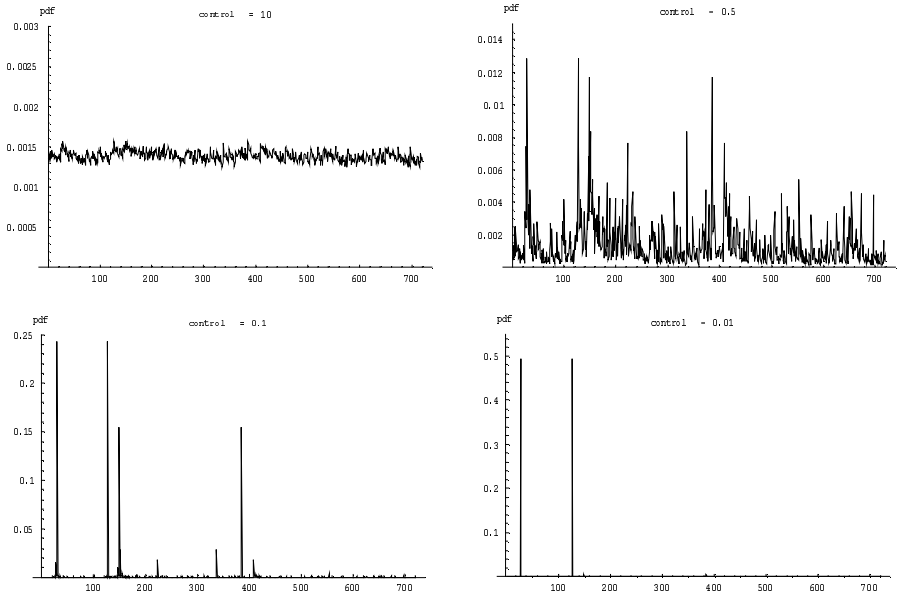


Figure 9: The Boltzmann distribution for the 7 cities TSP with cost function from Figure 7. The upper left panel has c = *10, the upper right panel has* c = *0.5, the lower left panel has* c = *0.1 and the lower right panel has* c = *0.01.*

We can now provide a first, very verbal version of simulated annealing, as shown in Algorithm 2. The second step in the simulated annealing algorithm gives rise to the question how one can sample configurations according to a given probability density function (i.e. the Boltzmann distribution). This is possible with the *Metropolis Algorithm* that will be introduced in the next section. The third step Algorithm 2 leads to the question of a *cooling schedule*, which will be further discuss in section C.4.

```
1. Start at a certain temperature (control variable)
2. Equilibrate the system and sample configurations according to
   the Boltzmann distribution
3. Slowly decrease the temperature, until the system settles
   into the, hopefully, global minimum.
```

Algorithm 2: Simulated Annealing, version 1.

C.3. The Metropolis Algorithm

Before we embark on this journey it is important to realize that here the metropolis algorithm will be introduced in a "hand waiving" manner. The lecture Stochastic Simulation [1] introduces these concept in a much more concise and detailed way. Here we just provide the most relevant aspects in order to understand the behavior of simulated annealing. If you are interested in technical details you are referred to [1] and references therein.

Real annealing of a crystal is a dynamical process, the system continuously changes its configuration, were the dynamics of the systems are based on underlying physical laws (in this case quantum mechanics). We would like to capture this dynamics in terms of transitions of one state to another. The probability that the physical system will be in a given state at a given time t_2 may be deduced from a knowledge of its state at any earlier time t_1, and does not depend on the history of the system before time t_1. Stochastic processes, which represent observations on physical systems satisfying this condition, are called Markov processes. A special kind of Markov process is a Markov chain. It may be defined as a stochastic process whose dynamics may be treated as a series of transitions between configurations of the process. The future development of the process, once it is in a given state, depends only on that state and not on how the process arrived in that state.

A Markov Chain is a sequence of trial configurations, where the outcome of each trial only depends on the outcome of the previous trial. With X_k a stochastic variable denoting the outcome of the k-th trial, we have the transition probability:

$$P_{ij}(k) = P[X_k = j \mid X_{k-1} = i]$$

With this transition matrix we can write

$$a_j(k) = \sum_{\substack{i \text{ from all} \\ \text{configurations}}} P_{ij}(k) a_i(k-1),$$

where $a_j(k) = P[X_k = j]$.

A Markov chain is called *inhomogeneous* if the associated transition probabilities depend on the trial number k. If the transition probabilities are independent of the trial number, the Markov chain is called *homogeneous*. A Markov chain with transition matrix P is *irreducible* or *ergodic*, if for each pair of solutions (i, j) there is a positive probability of reaching j from i in a finite number of trials. So given an arbitrary initial state every outcome in the state space is reached with probability > 0 for an infinitely long Markov chain that is irreducible. Finally, a Markov chain is called aperiodic if for any state i it is possible that after one trial the state remains i.

This may sound a bit abstract, but we need the concept of a Markov chain, i.e. a sequence of configurations, and the associated transition probabilities, i.e. the probability that the systems goes from one state to another, to introduce the Metropolis algorithm. An important property of a homogenous Markov chain is its stationary distribution, defined as

$$q_i = \lim_{k \to \infty} P[X_k = i \mid X_0 = j] \quad \forall j .$$

In words, independent of the initial configuration j the system will evolve, in the limit of infinite transitions, to a stationary distribution i. For our crystal in thermal equilibrium this would be the Boltzmann distribution. Without prove we state that the stationary distribution exists for ergodic aperiodic Markov chains.

The relation with simulated annealing is as follows. As will become clear, in simulated annealing we will also create Markov chains, whose stationary distributions depend on the control variable (the "temperature"). For certain specific transition matrices, which we will define below, on can proof that $\lim_{c\downarrow 0} \mathbf{q}(c) = \pi$ with

$$\pi_i = \begin{cases} \left|\Re_{opt}\right|^{-1} & \text{if } i \in \Re_{opt} \\ 0 & \text{otherwise} \end{cases},$$

where $\Re_{opt}$ is the set of optimal solutions and $|\Re_{opt}|$ is the number of elements in $\Re_{opt}$. [2] So, in simulated annealing we generate Markov chains, and in the limit of $c \to 0$ the stationary limit of that Markov chain will only have a finite probability for optimal configurations, and all other configurations will have a zero probability. We only need to observe the configurations that we obtained to find the wanted solution to our optimization problem. Again going back to the example of the 7 cities TSP, from Figure 7 we can conclude that $\Re_{opt} = \{29,127\}$, $\left|\Re_{opt}\right| = 2$, $\pi_{29} = \pi_{127} = 0.5$, and that π_i is zero for any other values of i.

Given the introduction of the idea of Markov chains we now can discuss the algorithm which has been introduced by Metropolis (see [3]). It is often very difficult or even impossible to generate random variables with an arbitrary distribution. Using the Metropolis algorithm one can generate a homogeneous Markov chain $\{X_n\}$ that steps through a state space, such that the points X_n are distributed according the required probability density function $p(x)$. One starts by defining a "random walk" with a transition probability Γ_{ij} being the probability to get from "state" X_i to X_j such that the distribution of the points (or states) X_i converges to $p(x)$. A sufficient condition for this is the so-called detailed balance condition:

$$p(X_i)\Gamma_{ij} = p(X_j)\Gamma_{ij}.$$

This equation tells us that the amount of transitions from state i to j has to be equal to the amount of transitions going from j to i. If this is true for all pairs of states the system can evolve to an equilibrium situation where on average all states are sampled according to the probability function.

The Metropolis algorithm is actually a random walk through phase-space. The asymptotic (i.e. after an infinite amount of steps) probability p_i, the chance of finding the system in state i, of this random walk is the desired probability function. But this is only true if certain conditions on the generation and acceptation of the proposed state are met. For this random walk we need to define a transition probability Γ_{ij} to go from state i to state j. The Γ_{ij} does not only include the acceptance of a new state but also the random generation of the new state :

$$\Gamma_{ij} = G_{ij}A_{ij} \qquad [3]$$

where G_{ij} is the chance of generating state j if the system is in state i. A_{ij} is the chance of accepting the new state j given that the system is in state i and is given by :

$$A_{ij} = \min\left(1, \frac{p(j)}{p(i)}\right) \qquad [4]$$

where $p(i)$ is the probability of the current state of the system, $p(j)$ is the probability of the proposed state and the acceptation rule A_{ij} makes sure that proposed states that are more important than the current are accepted and that less important states have a acceptance probability less than 1 (but larger than 0).

The Γ_{ij} transition probabilities have to be defined such that if a Markov chain is generated, the system will converge to the desired distribution (it should visit all possible states with a frequency according to their importance). To ensure this the following two conditions have to be satisfied.

(1) The simulated system should be ergodic i.e. all states should be accessible from all other states in a finite number of steps. If this were not so then there are states that do play a (important) role in the system but they can never be reached. A formal description for this (see [2]) is :

$$\forall i, j \exists n : 1 \le n < \infty \wedge (\Gamma^n)_{ij} > 0 ,$$

where n is the number of steps needed to go from i to j. $(\Gamma^n)_{ij}$ is the transition matrix to the power of n, i.e. it gives the probability of applying the transition function n times on state i to reach state j. This means that the function G_{ij} should be such that (in a finite number of steps) all states can be generated. Also the acceptance probability, A_{ij}, should not be zero (although it is allowed to be very small) since the product $((GA)^n)_{ij}$ is not allowed to be zero. If this all is true the Markov chain is said to be irreducible.

(2) The Markov chain must be aperiodic. This makes sure that the system can not get trapped in a series of states that are chosen periodically, since the system can not escape from such a loop. This means that starting from an arbitrarily chosen state i, the system is allowed to return to this state once in a while but there may not be any specific period between the returns. In mathematical terms it means that the greatest common divisor of all integers $n \ge 1$, such that $(\Gamma^n)_{ii} > 0$, is equal to 1.

The A_{ij} in Eq. 4 already satisfies the conditions for convergence (see [2]). If both conditions are satisfied the Markov chain will converge to the probability distribution.

Given this, one can generate X_{i+1} from X_i according to a probability density distribution $p(x)$ by the pseudo-code in Algorithm 3.

```
choose trial position X_t = X_i + δ_i with δ_i random over [-δ, δ]
calculate Γ_ij = min[1, p(X_t)/p(X_i)]
if Γ_ij = 1 then
   accept move and put X_i+1 = X_t
end if
if Γ_ij < 1 then
   generate random number R (uniform on [0, 1])
   if R ≤ Γ_ij then
      accept move and put X_i+1 = X_t
   else
      put X_i+1 = X_i
   end if
end if
```

Algorithm 3: Metropolis algorithm.

It is efficient to take δ such that roughly 50% of the trials are accepted. Furthermore, the walk can be started best at a value of X at which $p(X)$ has a maximum. It can be shown that this method guarantees that, for a large number of steps, all "states" are explored and equilibrium will be found according to Γ_{ij}.

In other words, we generate a chain of configurations X_n of the system of interest, such that the number of configurations in the "interval" $[X, X + dX]$ is proportional to $p(x)dx$. We do this as follows: Suppose that after some time the system has arrived in state X_n. We generate a trial configuration X_t from X_n by making a small change to X_n. (What is meant by "a small change" depends on the kind of system that is studied.) We then calculate the ratio:

$$p(X_t) / p(X_n) = r .$$

Dependent on the value of r the trial configuration is accepted or rejected. If $r \geq 1$ the trial is accepted. If $r < 1$ the trial is accepted with probability r. We do this by drawing a random uniform [0, 1] number ψ. If $r \leq \psi$ we accept X_t, else we reject it. If the trial configuration is accepted then we set $X_{n+1} = X_t$, else $X_{n+1} = X_n$. By repeating this Markov process we generate a chain of configurations $X_1, X_2, \ldots, X_N$. If N is large enough, every configuration will occur in the chain with a frequency that is proportional to the probability density $p(x)$.

In our specific case we deal with the Boltzmann distribution, see Eq. [1] and Eq. [2]. If we now combine the specific case of the Boltzmann distribution with the general Metropolis algorithm we find something very interesting. In general it will be very hard (impossible) to calculate the exact Boltzmann distribution, because we do not know the partition function $Z(T)$ or the normalization function $Z(c)$. However, because of the specific choice of the transition probability in the Metropolis algorithm we *do not need to know* the partition function. Just calculate Γ_{ij}:

$$\Gamma_{ij} = \min[1, \exp(-(E(x_j) - E(x_i)) / k_b T)] = \min[1, \exp(-\Delta E / k_b T)] ,$$

or in terms of cost functions and optimization

$$\Gamma_{ij} = \min[1, \exp(-(C(x_j) - C(x_i)) / c)] = \min[1, \exp(-\Delta C / c)] .$$

This means that in order to sample from the wanted Boltzmann distribution, using the Metropolis algorithm, we only need to generate a new configuration from the current one, and calculate the *difference* in energy (or cost). From that we can calculate the transition probability, which depends on the temperature (or control parameter). If the energy (or cost) of the new configuration is lower that the current one, we unconditionally accept the new configuration. If the energy of the new configuration is larger then the current one, we accept it with a probability equal to $\exp(-\Delta E/k_B T)$. Note that this last step, conditional acceptance of solutions with a higher step, was one of the possible solutions to the greedy iterative improvement algorithm. With this the Metropolis algorithm in terms of our optimization problems is given in Algorithm 4.

```
/* a Metropolis step */
Propose a new state j from current state i.
Calculate cost difference ΔC.
Calculate Aij = min[1,exp(- ΔC/c)]
if Aij = 1
   accept proposed state as next state in Markov chain
else
   take a uniformly distributed random number R between 0 and 1
   if Aij ≥ R
      accept proposed state as next state in Markov chain
   else
      reject proposed state, and take current state as next
      state in Markov chain
```

Algorithm 4: Metropolis Algorithm with Boltzmann distributions, applied to optimization.

As an example we will show how we could use the metropolis algorithm to sample tours in the 7 cities TSP example according to the probability distribution given by the Boltzmann distribution. Again, because the 7 cities example is so small we can calculate the exact distribution and compare to the result of the Metropolis algorithm.

To do this we need a method to generate a new tour from a given tour. We apply the so-called Lin 2-opt transition rule, were we randomly pick two cities in the current tour and reverse the order in which the cities between this pair of chosen cities are visited. For instance, in the example of Figure 10, we randomly pick cities 1 and 5, and then reverse the sub tour 1-4-3-2-5 to 1-2-3-4-5. This gives a neighborhood of $0.5N(N\text{-}1)$ tours, which is small compared to the total number of possible tours. In that sense the Lin 2-opt rule satisfies the condition of a "small" perturbation of the current configuration.

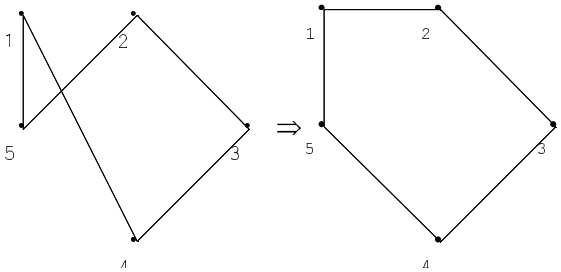


Figure 10: Example of the Lin 2-opt transtion rule for TSP.

We have performed the following experiment. Consider again the 7-cities TSP from Figure 6 and its associated cost function Figure 7. From this cost function we can now easily calculate the Boltzmann distribution for this TSP, as a function of the control parameter c, as is shown in Figure 9. We will now run the Metropolis algorithm and just measure the distribution of tours that is generated. We do this as follows. We start with a randomly chosen tour and generate a homogeneous Markov chain of length 20000 using the Metropolis algorithm (i.e. each tour is sampled on average 20000/710 = 28 times). Next we create a histogram of occurrences of tours in the Markov chain, and compare to the theoretically expected *pdf*, i.e. the Boltzmann distribution at a certain control parameter (or temperature). We have done this for four control parameter, c = 10, 0.5, 0.1, 0.01 respectively. The results are shown in Figure 11 to Figure 14. The pdf's as generated with the Metropolis algorithm agree very well with the theoretical ones. For a high control parameter, we expect a pdf that is almost independent of the specific tour. This is indeed observed in Figure 11. Here, the experimental results has a very large noise component, that would disappear is the length of the Markov chain would be increased. Also for lower values of the control parameter a very good agreement between the theoretical result and the one generated by the Metropolis algorithm is observed.

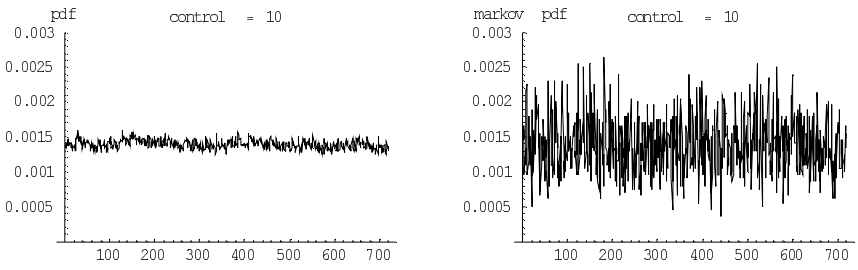


Figure 11: The theoretical pdf (left) and the pdf generated with the Metropolis algorithm for the Boltzmann distribution of all tours in the 7-cities TSP of Figure 6, for c = 10.

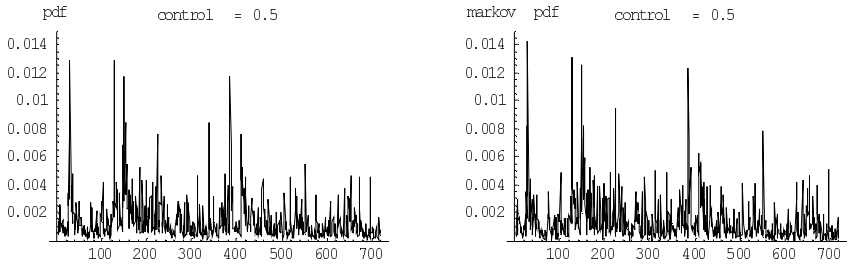


Figure 12: The theoretical pdf (left) and the pdf generated with the Metropolis algorithm for the Boltzmann distribution of all tours in the 7-cities TSP of Figure 6, for c = *0.5.*

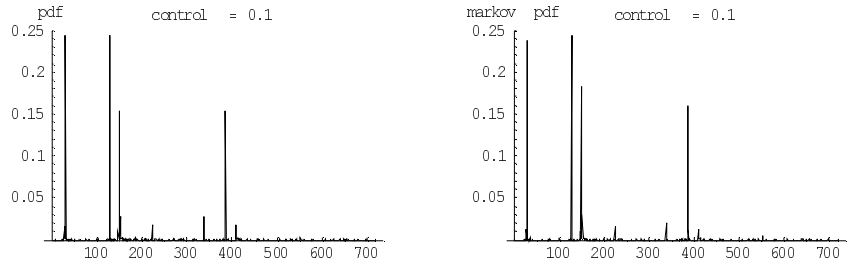


Figure 13: The theoretical pdf (left) and the pdf generated with the Metropolis algorithm for the Boltzmann distribution of all tours in the 7-cities TSP of Figure 6, for c = *0.1.*

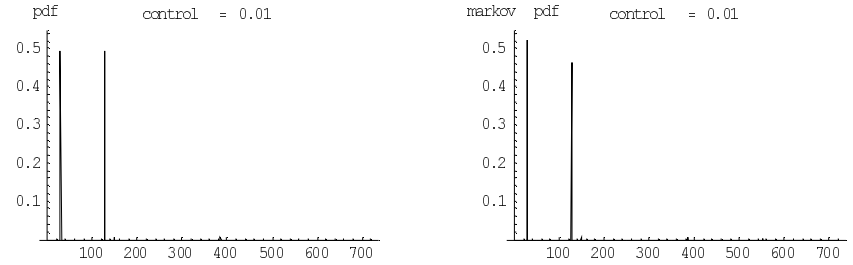


Figure 14: The theoretical pdf (left) and the pdf generated with the Metropolis algorithm for the Boltzmann distribution of all tours in the 7-cities TSP of Figure 6, for c = *0.01.*

C.4. The Simulated Annealing Algorithm

Simulated Annealing (SA) is an optimization algorithm, formulated by Kirkpatrick et. al. [4] in 1983, based on the annealing of solids. The SA algorithm is a stochastic optimization procedure. There are also non-stochastic optimization algorithms such as steepest descent where the path leading to the nearest minimum is taken. The consequence of this method is that is gets stuck in the nearest minimum, which does not need to be the global minimum of the function that is optimized. This was already discussed in section C.1. The stochastic nature of the SA method makes sure that the system does not get trapped in the nearest local minimum but is able to search the phase-space for deeper minima. In theory the method can guarantee that *the* global minimum of the function is found. The SA procedure searches for the state of the system with the lowest cost. If one has a problem that has to be maximized, it is possible to turn this into a minimization problem by putting a minus sign before the cost-function. The SA algorithm is applicable in many different areas. Examples are: image restoration [5], molecular optimization, [6, 7] load balancing, [8] wiring of chips and many more, see [2] and references therein.

Here we will first give the general idea of how the SA technique works. Analogous to physical annealing we start the system at a high temperature. With the Metropolis algorithm we let the system make a random walk through phase-space. During the random walk we slowly lower the temperature. The SA procedure is stopped when a

certain stop criterion is met. The result of this is that at a high temperature the costs of the states, given by the cost-function, is not important. The system can explore the phase-space virtually without constraints. If the temperature is lowered the cost of the cost-function begins to be more and more important and features of the function start to be visible in the probability distribution. Here the system will spend on average more time in states that have a low cost then in states with a higher cost. At this stage the system is reacting to the coarse features of the function. If the system is at a low temperature the details of the function to be optimized come within view. The system will spend most of the time in states that have a low cost, although it is still possible that a number of successive acceptation of cost increasing steps will bring the system into a new region. The process can be stopped if no lower cost has been found in a (large) number of steps.

There are two more or less distinct ways of doing SA. One works with homogenous Markov chains, the other works with an inhomogeneous Markov chain. For both methods a proof exists that the simulation will, in the end, converge to the optimal solution.

The homogenous Markov chain method

Simulated Annealing needs a method to let the system walk through the phase-space while the temperature is decreased. One way of doing this is to make a Markov chain at a series of (decreasing) temperatures while the temperature is kept constant during the creation of a Markov chain. This SA method with homogenous Markov chains is based on the fact that at every temperature the asymptotic probability distribution is obtained. To reach this asymptotic distribution the Markov chain has to be infinitely long. If the temperature is slowly lowered, such that there is a smooth change in the asymptotic probability distribution towards the most probable states, the system will evolve to the most optimal solution i.e. the one with the lowest cost value.

In order to converge to the asymptotic distribution the conditions on the generation and acceptation methods have to be met. In addition to this we have to make sure that the asymptotic distribution goes to the most optimal solution if the temperature goes to zero. Sufficient conditions for this are (see [2]):

$$\forall i, j : C(X_i) \geq C(X_j) \Rightarrow A_{ij}(T) = 1,$$

$$\forall i, j : C(X_i) < C(X_j) \Rightarrow \lim_{T \downarrow 0} A_{ij}(T) = 0 .$$

These equations make sure that steps to states with a lower cost are always accepted and that in the final stages of the annealing process steps that increase the cost are practically not accepted anymore.

Theoretically the Markov chains have to be infinitely long to be sure that the asymptotic distribution is reached, and the annealing process has to make infinitely many Markov chains (limit of $T \rightarrow 0$) to ensure that the most optimal solution is reached. This is in practice not possible since we cannot wait for an infinitely long time to get the answer to our optimization problems. So, a practical version of the algorithm will have Markov chains of a certain number of Metropolis steps and a stop criterion is needed to stop the annealing process. The practical version is therefore not guaranteed to find the global optimum. This means that the annealing has to be repeated several times with different starting states (different random seeds) and hopefully the solution with the least cost will be among them.

The lowering of the temperature is related to the length of the Markov chains. If we are in a situation that at a certain temperature the probability distribution is very close to the asymptotic distribution, then a large change in the temperature will bring the system far from the new asymptotic distribution and a Markov chain of many steps is needed to come close again. If the temperature change is only small, then we are still reasonably close to the asymptotic distribution and a small Markov chain is sufficient to regain the asymptotic distribution.

The starting temperature, the method of lowering the temperature and the stop condition for the annealing process together are called the cooling schedule. There are several methods of making a cooling schedule and often they are problem dependent. The starting temperature can be determined by looking at the acceptance rate in the Metropolis algorithm during the Markov chain. The acceptance rate should have a high value since at a high temperature it should not really matter where we are in the phase-space. The stop condition is often chosen to be such that the annealing is stopped if no states with lower energy have been found in a certain number of steps.

The lowering of the temperature is usually based on one of the following two equations:

$$T^* = crT\,, \qquad [5]$$

$$T^* = T\left(1+\frac{\ln(1+\delta)T}{3\sigma(T)}\right)^{-1}. \qquad [6]$$

T is the temperature of the current chain, T^* is the temperature of the next chain, cr is the cool-rate and determines how fast the system is cooled, $\sigma(T)$ is the standard deviation in the value of the cost function of the current chain and δ controls how much the probability functions may differ between two chains. Variations are possible see for example [Ingber, 1989 #38]. Equation [5] is only dependent on the temperature of the current chain in order to calculate the temperature of the next chain. Equation [6] is dependent on the temperature *and* the standard deviation of the cost function of the current chain. A practical version of SA with homogenous Markov chains is given in Algorithm 5.

```
Put the system in a random state
Start with a high temperature
while the stop criterion is not met
   Make a Markov chain with the Metropolis algorithm
   Lower the temperature
Endwhile
```

Algorithm 5: Pseudo code for Simulated Annealing 2.

In a practical Simulated Annealing we need to prescribe a *cooling schedule*. This schedule contains the initial value of the control variable (e.g. such that at the initial value a large percentage, say 50 to 80%, of newly generated configurations is accepted), a final value of the control variable (stop condition, e.g. for a fixed number of Markov chains, or if after a certain number of Markov chains the final configuration is not improved, the length of the Markov chains (e.g. the length depends on the size of the proble|m), and a rule to decrease the control variable (e.g Eq. [5] or [6]).

The inhomogeneous Markov chain method

In the previous section on homogenous Markov chains the temperature was decreased after the completion of the Markov chain. In the inhomogeneous Markov chain method the temperature is decreased after *every* Metropolis step.

For the convergence of the inhomogeneous Markov chain method, the conditions discussed in the previous section are not enough. Since the temperature is lowered after each step, the temperature decrement is restricted. Using ergodicity theorems (see [2]) it can be shown that the cooling should not be faster than

$$T = \frac{C}{\log k} \qquad [7]$$

where C is a problem dependent constant and k the number of the Metropolis step in the inhomogeneous Markov chain.

The TSP example, the final step

We can now apply Simulated Annealing to TSP, and compare to greedy iterative improvement. We have taken a TSP with 70 randomly placed cities (so, with in total 69! = 1.7×10^{98} possible tours), and applied greedy iterative improvement and Simulated Annealing to find the optimal tour. We use the Lin 2-opt method to generate a new tour from a given tour. The initial tour was 1 – 2 – 3 … – 69 – 70 – 1. In Simulated Annealing the initial control value was 0.3 (56 % of generated tours were accepted at this value). We generated 200 Markov chains with a length of 500. We used the constant temperature decrement rule (Equation [5]) with cr = 0.975. The greedy iterative improvement algorithm was allowed to take the same amounts of trials as in Simulated Annealing, i.e. 500 × 200 trials. The resulting final tours are shown in Figure 15. The final tour after the greedy iterative improvement has a cost of 7.70. Simulated annealing found quite a different tour, with a substantial lower cost of 6.26. It is clear that greedy iterative improvement got stuck in a local minimum. Figure 16 shows how the cost decreased while the optimization proceeds. The greedy iterative procedure quickly finds a local minimum, but subsequently gets stuck and stays there. On the other hand, the cost in the Simulated Annealing decreases, on average, quite slowly and only at very small value of the control parameter finds a tour with a very small cost. Off course we are not sure that the tour that we found with simulated annealing is the global minimum. To convince ourselves that we found a very deep minimum, or maybe even the wanted solution, we need to rerun the algorithm a few time with different initial conditions, of different cooling schedules.

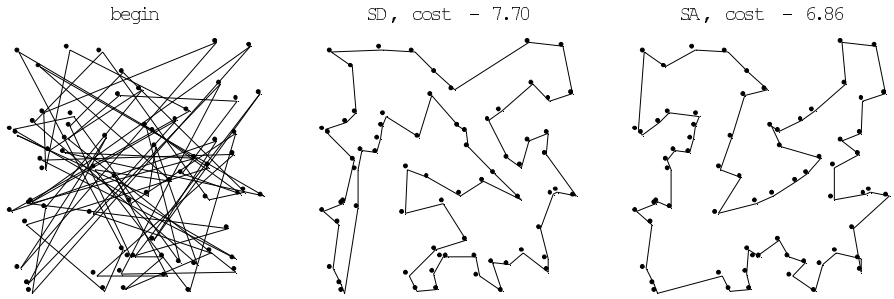


Figure 15: A TSP with 70 randomly placed cities. On the left the initial tour, in the middle the final tour in the greedy iterative improvement method, on the right the final tour after Simulated Annealing.

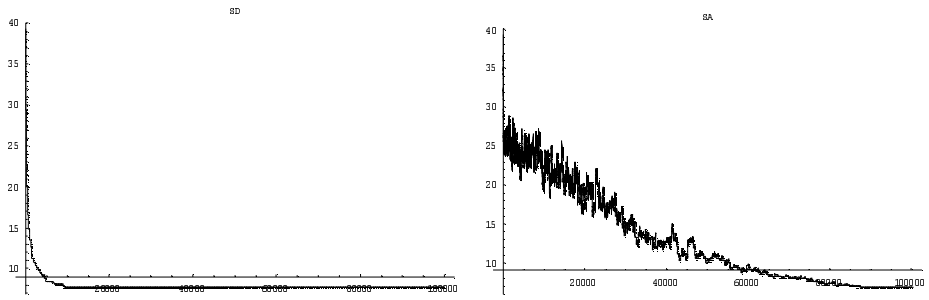


Figure 16: The cost of the generated tours, as a function of the iteration number for the greedy iterative improvement (left) and Simulated Annealing (right).

C.5. Parallel Simulated Annealing

Introduction

Simulated Annealing looks like an inherently sequential process, see Figure 17. Simulated Annealing is a sequence of Markov chains, each Markov chain is a sequence of Metropolis steps and each Metropolis step is a strict sequential algorithm where one first proposes a new configuration, next calculates the associated cost (difference), and finally decides if the new configuration will be accepted or rejected. Despite its inherent sequential nature it turns out that on each level of the Simulated Annealing algorithm we may try to find some room for parallelism. In most cases this room for parallelism turns out to be quite small, resulting in the possibility for nice performance gains, but certainly not in using massively parallel computers. However, at the highest and lowest level there is some room for massive parallelism, as will become clear later on.

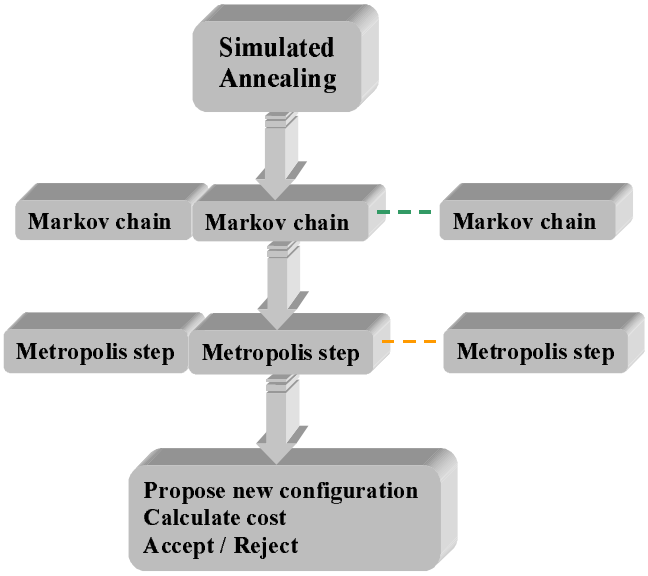


Figure 17: A scheme of the simulated annealing algorithm.

Parallelism within a Metropolis Step

A Metropolis step is a strict sequential algorithm where one first proposes a new configuration, next calculates the associated cost (difference), and finally decides if the new configuration will be accepted or rejected. It may happen that for certain applications the calculation of the cost or the cost difference is a very expensive operation that dominates the computational time. One could then try to run the cost computation in parallel, and keep all the rest sequential. This could e.g. be implemented in a Master/Slave model, where the Master generates the new configuration, which is sent to the slaves, that subsequently calculate the cost (difference) in parallel. Depending on the exact nature of the optimization process this parallel cost calculation may provide lots of room for (massive) parallelism.

Parallel Metropolis Steps

We will now discuss two more parallel Metropolis algorithms that are not parallel within the steps but execute the steps as a whole in parallel. The philosophy is to calculate, in an optimistic way, many Metropolis steps in parallel and only using those results that are required and discarding the rest (a bit like time warp for optimistic parallel discrete event simulation). We will discuss the *binary speculative computing method* and the *generalized speculative computing method* [9].

The binary speculative computing method will evaluate (in parallel) new states as soon as they become available without knowing in advance if this new state will be accepted or rejected. If the system is in a certain state and a new state is proposed, there are two

possibilities: the proposed state can be accepted or the current state is reused. As soon as the state that is proposed is generated, we can start a new Metropolis step, and now two Metropolis steps are active in parallel. One is based on the assumption that the old state is reused, and the other is based on the assumption that the proposed state will be accepted. Of course, after completion of the first Metropolis step we know which path was the right one and the other will have to be discarded (i.e. pruning of a part of the tree) and a message has to be sent to do this. The processors that are stopped can again be used in a part of the tree that is still calculating possible states. This method can execute $^{2}\log n$ steps concurrently on n processors.

Let us consider an example where we take the (unrealistic) assumption that the three parts in the Metropolis step (i.e. generate the new state, calcuate the cost, accept/reject decision) take the same amount of processor time. Furthermore, let's assume that starting up a new Metropolis step on another processor also takes time, and that this is also the same as the other three steps. We will now follow the system during 7 of the time steps and observe how the binary tree of parallel Metropolis steps develops and is pruned after each accept/reject decision. Figure 18 explains how we follow the computation in a time diagram. On time zero the first Metroplis step is started. After one time step the first thing that happens is to start a next Metropolis step that continues to work with the current configuration. Next the new configuration is generated, after which immediately a new Metropolis step is started that assumes that this new configuration will be accepted. Next the cost of the new configuration is obtained, and finally the decision to accept or reject the new configuration is made.

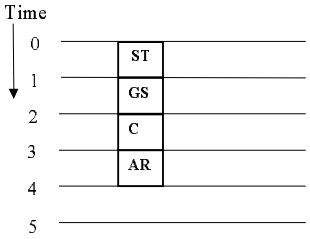


Figure 18: The example of Binary speculative computing. The Metropolis step consists of four parts, that all take one unit of time. These parts are ST – start up of a parallel Metropolis step; GS – generate a new state; C – calculate the cost; AR – Accept/Reject decision. Time flows from top to bottom, and parallel Metropolis steps will be shown by drawing them besides each other.

Figure 19 show the evolution after 4 time steps. It shows all new Metropolis steps that were started. In this and the following pictures the filled blocks refer to those Metropolis steps that assume that the original configuration will be accepted and the open blocks are those Metropolis steps that assume that the new configuration will be accepted. So, after 4 time steps already 11 new Metropolis steps have been started. Now assume that the original Metropolis step reject the new configuration. That means that all Metropolis steps that were started under the assumption that the new configuration would be accepted must be discarded. This means that the Metropolis tree must be pruned, as indicated by the large black cross in Figure 19. The computation now proceeds to step number 5. Again, many new Metropolis steps have been started (in the still active part of the tree), as is drawn in Figure 20. At time step 5 the Metropolis step that was started at time 1 is now ready and assume that the new configuration is accepted. This means that the complete right part of the tree can be pruned now. The earliest accept/reject decision will now be on time 7, by the Metropolis step started at time 3. The layout of the tree at time 7 is drawn in Figure 21. If we again assume that the new configuration was accepted, the complete left part of the tree is pruned again.

After 7 time steps we have concluded three Metropolis steps. If this was done sequentially, the three Metropolis steps would have taken 9 time steps. So, we got a small speedup. This parallelism strongly resembles the pipellined parallelism that was introduced in "Introduction Parallel Computing", see [10]. Using the formula for obtained

speedup for pipelined parallelism it is easy to show that for this specific example the speedup can be anywhere between 1.5 and 3 in the case of long Markov chains. Explain this yourself (and if necessary, go back to the theory of pipelined parallelism to do this)!

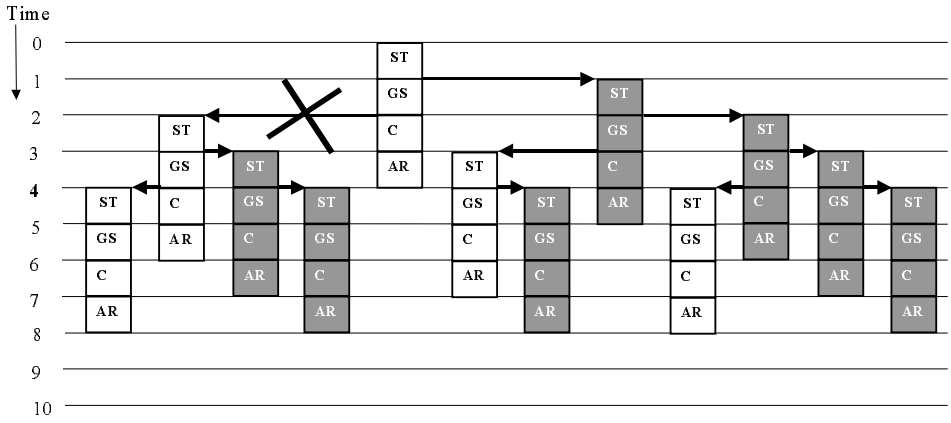


Figure 19: The evolution of the binary speculative computing, after 4 time steps.

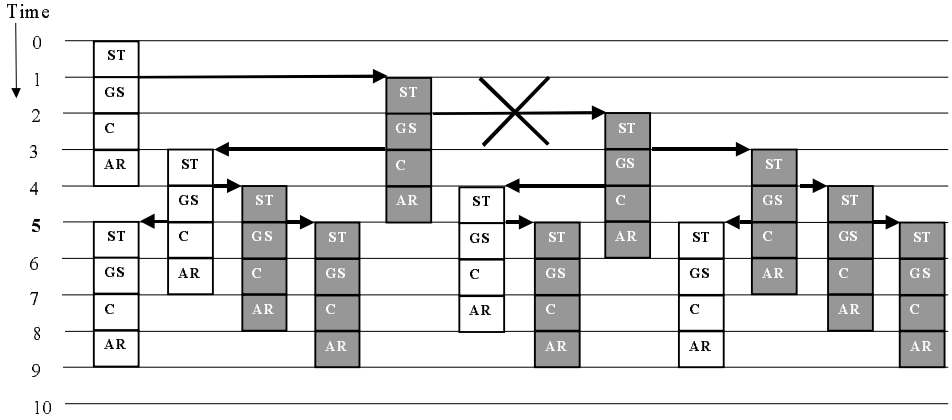


Figure 20: The evolution of the binary speculative computing, after5 time steps.

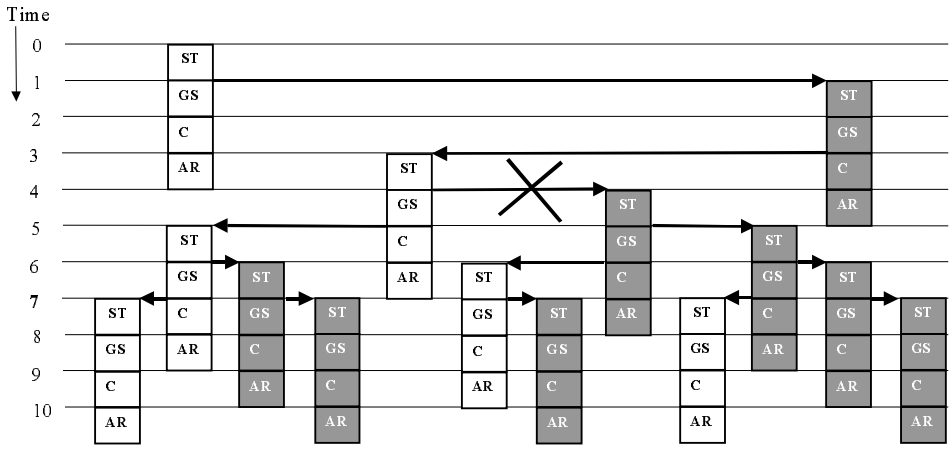


Figure 21: The evolution of the binary speculative computing, after7 time steps.

Finally note that in our example we have made some unrealistic assumptions about the time duration of each part in a single Metropolis step. In reality the execution times for all parts will differ. Especially if e.g. C is relatively large, much can be gained (again, explain this yourself).

Is the generalized speculative computing method [9] we also optimistically calculate a number of independent Metropolis steps in parallel, and try to use as much as possible. The idea is that we generate not one, but a number of new states based on the current state of the system. Of all these new proposed states we check which ones would be accepted. If for example, as in Figure 22, the first and second generated states (1a and 1b) would not be accepted but the third and fifth ones (1c and 1e) are, then the new state of the third trial is used for the computation of the next set of independently generated states. The trials after the first one that is accepted, are discarded (1d and 1e). So, in the first set of trials we have made three Metropolis steps (1a, 1b and 1c), the first two were not accepted (the observable should count the original state three times) and the third Metropolis step changed the state of the system (this is the reason that the other trials (1d, 1e) are not counted). In the second series of trials the state of the third trial of the first series is used, see the arrows. In the figure the second trial of the second series is accepted so two Metropolis steps are generated and the third series use the new state of the second trial of the second series. In this example we were able to calculate 9 Metropolis steps in the time for three Metropolis steps. If the overheads associated with the generalized speculative computing method can be kept small, the obtained speedup can be quite large, especially if the acceptance ratio is small as compared to the number of processors.

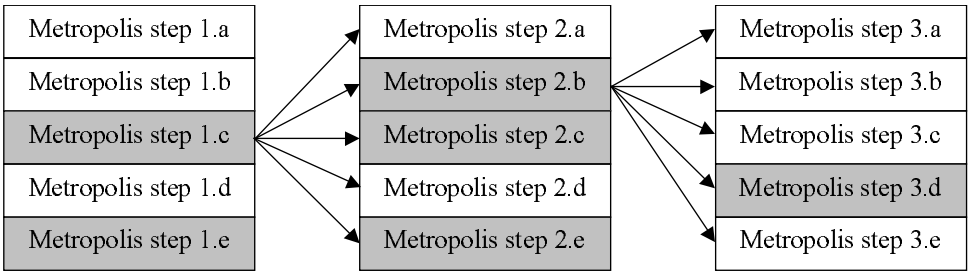


Figure 22: The Generalized speculative computing Metropolis Algorithm, the filled boxes denote Metropolis steps in which the newly generated configuration is accepted, in the white boxes it is accepted.

If we assume an acceptance probability of a in a Metropolis algorithm, we immediately see that the number of Metropolis steps needed for the first accept is a discrete random variable that is geometrically distributed. Therefore (see Chapter 1, appendix) the average number of Metropolis steps needed for an accept is $1/a$ and the variance is $(1-a)/a^2$. If we have enough processors available and ignore communication overhead, the mean speedup of the generalized speculative computing method will be $1/a$. For an acceptance ratio of 5% this will be 20. The smaller the acceptance ratio, the better the expected speedup will be. An interesting question would be the optimal amount of processors, taking communication into account, with optimal in the sense of minimal mean execution time. Try to derive some expressions for this yourself!

Parallel Markov Chains

We now move one level up in Figure 17 and concentrate on parallelism on the level of the Markov chains. Here the idea is to run instances of Markov chains in parallel and somehow combine the results, such that speedup can be obtained.

In the *clustered algorithm* several Markov sub-chains, simulating at the same temperature, are calculated in parallel on different processors. At the end of the sub-chains the intermediate results are compared. From these intermediate results one is chosen and is used as the starting situation for the next sub-chain on all processors at the same temperature (see Figure 23). One of the possible criteria for choosing a configuration from the intermediate results is to choose the most optimal one. After the complete Markov chain has been calculated the temperature is decreased and a new chain is started.

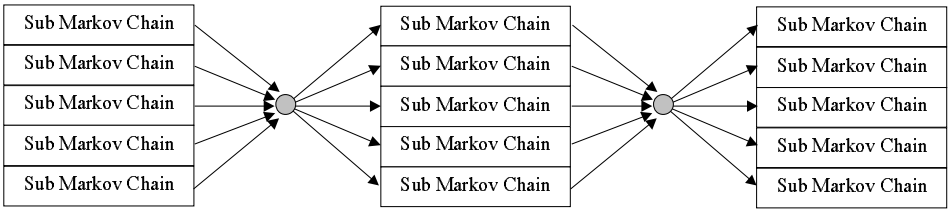


Figure 23: Schematic representation of the clustered algorithm at one temperature.

If one is simulating a system in which the generation of a new trial state can not be adapted for the different temperatures, this method can be used to generate a number of (independent) situations such that the number of accepted steps is higher than with just one Markov chain see [2]. If acceptance ratio is low, a lot can be gained again, and also by choosing suitable lengths for the sub-chains, even for higher acceptance ratios this parallelism will result in gains in performance.

A parallel algorithm that does not mimic sequential annealing is *systolic simulated annealing*. In systolic simulated annealing the annealing schedule itself is parallelized. This is accomplished by assigning a Markov chain to each of the available processors. All chains have equal length and simulate different (in decreasing order) temperatures, see Figure 24. The Markov chains are divided into a number of sub-chains and a new chain (at a lower temperature) is started as soon as the first sub-chain of the previous chain has ended. Equilibrium in the first chain is not yet established by then. In order to reach equilibrium in the Markov chains it is allowed that chains can adopt states of the system from chains with a higher temperature (dotted lines in Figure 24). The sub-chain has the choice of continuing with the final state of the previous sub-chain or adopting the state of the last generated sub-chain from the previous chain calculated in parallel. The choice is made on the basis of the standard Metropolis acceptance rule, see [2].

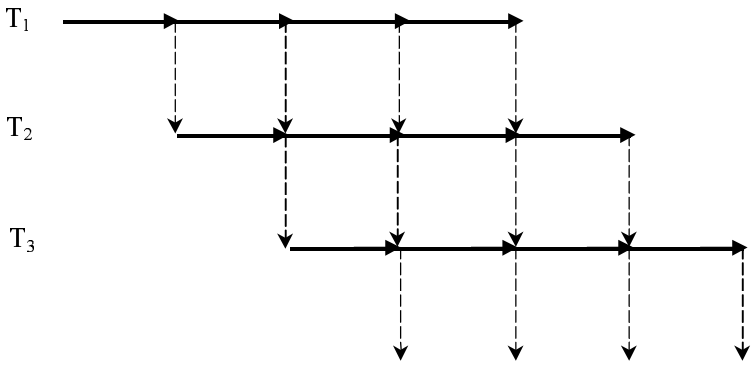


Figure 24: Schematic representation of the systolic algorithm.

The systolic algorithm has a simple communication pattern that can be efficiently implemented as a ring. The communication overhead is small since each processor contains a complete independent database for the optimization problem. To interchange information about the intermediate state it only has to send and receive at the end of each sub-chain. The load for the processors is about equal.

The Simulated Annealing algorithm is an inherently sequential scheme. Parallelizing it by the systolic Simulated Annealing method will introduce a functional difference to the sequential version. This parallelization has consequences for the accuracy of the iterative processes so that more chains have to be generated. Using a larger chain length than in the sequential case can counterbalance this.

If we determine what the chain length as a function of the problem size and the number of processors should be to keep the quality of the solutions the same as in the sequential case, it is found that the length increases with the number of processors in such a way that

the speedup that is gained by the parallel version is lost by calculation of longer chains [6]. At a certain number of processors the speedup is completely lost and the execution time rises.

Hybrid Implementations

In the section on parallel Metropolis algorithms it was discussed how the steps in the Markov chain can be calculated with the help of a parallel algorithm. Next the simulated annealing method itself was parallelized. It is possible to combine these methods to form a hybrid algorithm. The Markov chains are parallelized by the cluster or systolic implementation and the steps in the Markov chains are generated with the help of for example a parallel calculation of the cost-function (see [ter Laak, 1992 #32]).

Parallelism at the Top Level: Farming

Finally we should consider the possibility that we need to run an enormous amount of independent optimizations using Simulated Annealing. In that case we can also apply the ideas of farming parallelism (see [10]). This is the field of High Throughput Computing, and one should use dedicated tools such as Condor, Dynamite, or Nimrod to automate the process of scheduling and managing all the processes in the farm in an optimal way.

D. Genetic Algorithms

D.1. Inspiration from Nature

Evolution has shaped our natural environment and life itself into an always amazing complexity with highly adapted, i.e. optimized, life forms. Through an interplay of constantly mixing genetic material and survival and reproduction of the fittest individuals populations change, thus surviving in specific, possibly dynamic environments. In nature evolution results in highly adapted and very fit solutions to what one could call optimization problems. Although the "natural optimization" through evolution is not goal directed, evolution did provide an inspiration for another natural solver, the so-called *Genetic Algorithms*. Unlike nature Genetic Algorithms are goal directed optimization algorithms.

In most organism the genetic material (or the *genotype*) is organized in a set of *chromosomes* (23 in Humans), that contain the *genes*, which are the templates for production of proteins needed in body cells. Genes may come in different forms, called *alleles*. So, a gene may encode for the color or a flower, and it may come in two variants, for the color white and the color purple. Each cell has 2 copies of each chromosome[1]. Both chromosomes need not be exactly the same, but may contain different alleles of genes. During cell division (called *mitosis*) the DNA content of a cell is completely copied into the daughter cells. So, in principle each cell in an organism contains exactly the same DNA. Due to the possibility of little *mutations* differences in DNA may appear.

The big exception to all this are the reproductive cells, or *gametes* (sperm, ova). They contain one single copy of each chromosome. This single chromosome was taken randomly from the two available copies. So in principle an organism would contain 2^N different gametes, where N is the number of chromosomes in the genotype. It turns out that there is much more diversity in the DNA content of the gametes, due to an important process known as *crossover*, which occurs during *meiosis*, the very special cell division that leads to gametes, and where the random picking of chromosomes occurs. During meiosis all equal chromosome line up, and the equal chromosomes may break up and exchange bits of themselves with each other. This exchange of parts between chromosomes is crossover.

The single set of chromosomes in sperm and ova fuse again during fertilization. The resulting *zygote* develops into a new individual (a new *phenotype*). This complete process

[1] Note the exception for the sex chromosomes, with XX for female and XY for male.

results in a very efficient mixing of DNA from parents. *Selective pressure* and *relative fitness* of individuals in a population will dictate their success in *reproduction* into a next generation.

D.2. From Nature to a Natural Solver

Genetic Algorithms [11-13] are search algorithms and refer to a family of *computational models* inspired by evolution. They are based on the *mechanics* of natural selection and natural genetics. Potential solutions to specific problems are encoded as simple chromosome-like data structures. The *evolution* of the algorithm proceeds in combining chromosomes or *individuals* of a previous *generation* in order to form a next generation. In a sense a Genetic Algorithm can be regarded as a *guided random walk*, using historical information to effectively exploit new *search points*. A Genetic Algorithm can be applied to a variety of combinatorial, i.e. NP-hard and function optimization problems. By regarding Genetic Algorithms as optimization algorithms we can put Genetic Algorithms next to Simulated Annealing.

We will use the ideas from evolution as a metaphor for a procedure for optimization leading to Genetic Algorithms. We start with a population of individuals (phenotypes, e.g. tours in a TSP), each with a certain fitness (calculated from a cost function). The individuals reproduce with a probability based on their fitness in which the DNA of the parents (genotype) is mixed including possible mutations. A population evolves with increasing fitness in subsequent generations, i.e. optimization occurs.

D.3. The basic Genetic Algorithm

An implementation of a Genetic Algorithm begins with a population of random individuals, i.e. possible solutions. This collection of phenotypes is called a *generation* and the Genetic Algorithm evolves generation after generation. Each phenotype has a associated genotype. Usually in Genetic Algorithms the genotype is an encoding in a bit string. The chromosome consists of "genes" (bits), each gene being an instance of a particular "allele" (i.e. 0 or 1), e.g. 0111001010010. Next, the individuals are evaluated according to a so-called *objective function* (or *cost function*), which in turn is used to calculate the *fitness* of the individual in the current population (the higher the fitness the "better" the solution). The individuals are given reproductive opportunities related to their fitness. These reproductive opportunities are actually probabilities assigned to individuals to indicate the chance to get selected in the next generation. Individuals with a high fitness are given high probabilities. The fitness of a solution is often defined with respect to the current population.

To clarify some of the terminology we introduce a very trivial example. We will apply a genetic algorithm to the "problem" of finding the maximum of $f(x) = x^2$, where x is an integer and $x \in [0,31]$. Solutions are encoded as bit strings of length 5, hence the entire domain is covered. So, the phenotype '9' would correspond to the genotype '01001'. We take the fitness of each phenotype x to be $f(x)$, i.e. the fitness of the phenotype '9' is 81. To start off the population is randomly initialized with four individuals. This initial generation is shown in Table 2.

individual	*initial population*	x	$f(x)$
1	0 1 1 0 1	13	169
2	1 1 0 0 0	24	576
3	0 1 0 0 0	8	64
4	1 0 0 1 1	19	361
Sum			1170
Average			293
Max			576

Table 2: The initial generation for the optimization example.

Before continuing the example we must first further clarify the basic genetic algorithm. It is useful to view the execution of a genetic algorithm as a two-stage process. We start with the *current population*. *Selection* is applied to form an intermediate population. Subsequently *reproduction* is used to form the next generation.

Let us call f_i the value of the objective function of individual i and $\langle f \rangle$ the average value of the fitness function over the total population. We can now define the fitness of an individual as $f_i/\langle f \rangle$. We can regard this fitness as the probability of an individual to be selected in the intermediate generation. This selection step is often called *reproduction*. After the reproduction the *recombination* step is applied, which in turn can be divided in a *crossover* and a *mutation* step. During the recombination selected individuals are pair wise crossed with a certain large probability which results in two *offspring*, which both have features of their parents. Also there is a chance that some bits in the bit string are flipped with a low probability, this is called *mutation*. The resulting individuals are placed in the new generation. This is repeated until the new population is full, for example if it is as large as the previous one. Then the cycle starts all over until some *stop-criterion* is fulfilled. Summarizing, the main cycle of the GA consists of three operators:

- Selection (or reproduction)
- Crossover
- Mutation

These are the main operators in a genetic algorithm, they are found almost in any GA. These operators jointly determine the efficiency of the algorithm. It is possible to add other operators in this cycle, but they often only have marginal differences in the original operators. An example can be a restriction in the selected offspring for the next generation, where only the better of the two individuals is selected. The basic Genetic Algorithm is shown in Algorithm 6. Note that selection must always occur to produce offspring, but that crossover and mutation only occur with a certain probability p_c and p_m respectively.

1. Start with a randomly generated population of n l-bit chromosomes.
2. Calculate the fitness $f(x)$ of each chromosome x in the population.
3. Repeat until n offspring have been created:
   a. Select a pair of parent chromosomes
      selection based on fitness, same chromosome may be selected more than once
   b. With probability p_c (cross over probability) cross over the pair at a randomly chosen point to form two offspring.
      If no cross over takes place the parents become the offspring
   c. Mutate the offspring at each bit position with a mutation probability p_m.
4. Replace current population with new population
5. Go to step 2.

Algorithm 6: The basic Genetic Algorithm.

We will now continue our example optimization problem. Table 2 showed the initial population for this example, which was just a randomly initialized population with four individuals. The next step, as shown in Algorithm 6, is the calculation of the fitness of the individuals in the population. This is done by calculating $f_i/\langle f \rangle$. The result is shown in Table 3. The next step is the selection (or reproduction). In this example we do this by first creating a mating pool, and next let pairs of individuals recombine. Note that multiple copies of a single individual may end up in the mating pool. The fitness determines the chance number of times an individual will end up in the mating pool. Many techniques exist for the selection operator, and they will be discussed in detail in later sections. Here we just assume that we have such a technique available, and the

column 'actual count' in Table 3 shows how many copies of each individual actually end up in the mating pool.

Individual	initial population	x	$f(x)$	$\frac{f_i}{\sum f_i}$	$\frac{f_i}{< f >}$	actual count
1	0 1 1 0 1	13	169	0.14	0.58	1
2	1 1 0 0 0	24	576	0.49	1.97	2
3	0 1 0 0 0	8	64	0.06	0.22	0
4	1 0 0 1 1	19	361	0.31	1.23	1
Sum			1170	1.00	4	
Average			293	0.25	1.00	
Max			576	0.49	1.97	

Table 3: The first steps in a Genetic Algorithm for the optimization example; calculation of the fitness and selection of individuals into a mating pool.

Next the actual recombination takes place. First, consider the mating pool in Table 4. If you discard the — symbol for a while, you immediately recognize the individuals of the initial population. Individual '1' is encountered one time, individual '2' is encountered two times, etc. Next, the mating pool is grouped into pairs of individuals. The first two will be recombined, and the third and fourth individual will recombine (again, note, this is just a constructed example, in real Genetic Algorithms all this may, and will be done in other ways). As explained above, recombination in Genetic Algorithm means crossover and mutation. Let us assume that crossover always occurs (i.e. p_c =1). The — symbol shows the location where the two individuals will be crossed over. The idea is, like in real crossover during meiosis, that the chromosomes break at a specific location (the — symbol), and exchange there genetic material. This is exactly what happens, and these are the individuals in the 'new population' column in Table 4. In this example we assume no mutation, i.e. p_m = 0. Finally the genotypes of the new population are translated to the phenotypes, and the associated objective function $f(x)$ is calculated. The sum, average, and maximum objective function values of the new population have increased, i.e. we have found more optimal solutions to our optimization problem. This single iteration is repeated over and over again, and according to some stopping criterion (e.g. the increase in maximum fitness falls below a certain value) the iterations are stopped and the most fit individual is taken as the solution.

mating pool	mate	new population	x	$f(x)$
0 1 1 0—1	2	0 1 1 0 0	12	144
1 1 0 0—0	1	1 1 0 0 1	25	625
1 1—0 0 0	4	1 1 0 1 1	27	729
1 0—0 1 1	3	1 0 0 0 0	16	256
Sum				1754
Average				439
Max				729

Table 4: The final steps in a Genetic Algorithm for the optimization example; recombination into a new population.

Before discussing all details related to the operators in a Genetic Algorithm and parallel Genetic Algorithms, we first explore some of the theory behind Genetic Algorithms.

D.4. Mathematical Foundations

One may wonder why and how genetic algorithms really work. In this section we will focus a little on the so-called *schema theorem* and the *building block hypothesis*. Together they form the classical Genetic Algorithms theory developed by Holland. [13] The central

idea is that "GA work by discovering, emphasizing and combining good building blocks in a highly parallel manner". [12] This may sound magical and actually in a way it is, but it is possible to formalize the processing power of a Genetic Algorithm.

First, we have to formally define a building block by a *schema*. A schema is a similarity template describing a subset of strings with similarities at certain positions. The template consists of members of the coding alphabet and the *don't care symbol* : *. At positions in a schema with a don't care symbol, any member of the alphabet can be positioned. In our case a schema therefore is a bit string, containing '0', '1' and '*'. Schemas are represented by the letter H. For instance, the schema H = 1**1 is the set of all 4-bit strings that begin and end with a '1'.

The number of fixed positions in a schema is called the *order* of the schema, and is denoted by o(H). A fixed position is everything but the don't care symbol. The *defining length* of a schema is the distance between the first and last fixed string positions, and is denoted by d(H).

As an example, consider H = 1*1*. There are exactly four strings that match this schema: 1010, 1011, 1110 and 1111. There are also other schemata that these strings have in common: 1*** and **1*. The order of H is 2. The defining length is 3-1=2.

With these definitions we can formulate the central idea of the theory as "Genetic Algorithms process schemata efficiently under selection, crossover, and mutation". So, although a Genetic Algorithm processes a relative small number of individuals, it processes many more schemata of which the individuals are just a single representative. This very important notion is called the *implicit parallelism* in Genetic Algorithms. Suppose the individuals are encoded by a bit string of length l. This means that each individual is an instance of 2^l different schemata. Therefore, a population of n individuals contains instances of between 2^l and $n \times 2^l$ different schemata. In other words, at any generation, a Genetic Algorithm is explicitly evaluating the fitness of n strings, but implicitly it is evaluating the average fitness of a much larger set of schemata, as we will prove below. E.g., in a population of random strings, about $n/2$ are instances of 1**…* and about $n/2$ are instances of 0**…*. Without actually calculating the average fitness of these schemata, Genetic Algorithms do process them. This is the implicit parallelism introduced above.

Strings that match a certain schema are said to *sample* this schema. This sampling is often referred to as *schema processing*. It is interesting to explore the effects of reproduction, crossover and mutation on this schema processing, i.e. to calculate the dynamics of increase and decrease of schema instances under the operation of selection, crossover, and mutation. So assume that H is a schema with at least one instance present in the population at time t. Define $m(H(t))$ as the number of instance of H at time t and $\mu(H(t))$ as the average fitness of H at time t, i.e.

$$\mu(\mathrm{H},t) = \frac{1}{m(\mathrm{H},t)} \sum_{x \in \mathrm{H}} f(x) . \qquad [8]$$

Our task is to try to calculate $E[m(\mathrm{H},t+1)]$, i.e. the expectation of the number of instances of H in the next generation, under the influence of selection, crossover, and mutation. Let us first analyze the effect of selection. The expected number of offspring of a string x is equal to $f(x)/\bar{f}$ with $\bar{f}$ the average fitness in the population. Therefore, we can immediately conclude that because of selection alone

$$E[m(\mathrm{H},t+1)] = \frac{\sum_{x \in H} f(x)}{\bar{f}(t)} = \frac{\mu(\mathrm{H},t)}{\bar{f}(t)} m(\mathrm{H},t) . \qquad [9]$$

Note from Equation [9] that m(H,t) is not calculated explicitly in GA, but dynamics of m(H,t) depends on it. A particular schema grows as the ratio of the average fitness of a schema to the average fitness of the population. Fitter schemata will be given an increasing number of samples in the next generation. In other words, above-average schemata grow and below-average schemata die off. What if a particular schema H remains above average with an amount $c\bar{f}$ (c is a constant). Equation 9 can be rewritten as

$$E[m(H,t+1)] = m(H,t)\frac{\bar{f}+c\bar{f}}{\bar{f}} = (1+c)m(H,t). \qquad [10]$$

If we start at $t = 0$, we obtain a kind of compound interest equation

$$E[m(H,t)] = m(H,0)(1+c)^t. \qquad [11]$$

This quantifies the effect of reproduction, the number of schemata with a larger than average fitness will exponentially grow. As we know, reproduction alone does nothing to promote exploration of new regions of the search space, it only exploits existing points. We need crossover and mutation for the exploration part.

Crossover (and for that matter mutation also) can either destroy or create schemata. We will only consider the destructive effects Crossover can occur on l - 1 sites. There are d(H) sites where the crossover is destructive, hence the probability that crossover destroys a schema isd(H)/(l-1). Let $S_c(H)$ be the probability that a schema survives single point crossover. If p_c is the probability that crossover occurs, an upper bound for S_c is given by

$$S_c(\mathrm{H}) \geq 1 - p_c \frac{d(\mathrm{H})}{l-1}. \qquad [12]$$

We can only give this upper bound, because the possibility exists that crossover, if it occurs, will not destroy the schema. Probability of crossover survival is higher for shorter defining length schemata.

Now we can combine Equations [9] and [12] to get the combined effect of crossover and selection:

$$E[m(\mathrm{H},t+1)] \geq \frac{\mu(\mathrm{H},t)}{\bar{f}(t)} m(\mathrm{H},t)\left(1 - p_c \frac{d(\mathrm{H})}{l-1}\right). \qquad [13]$$

It is clear that the combined effect is obtained by multiplying the expected number of schemata for reproduction alone by the survival probability under crossover. Those schemata with both above average fitness *and* short defining length are going to be sampled at exponentially increasing rates.

This leaves us only with mutation. Mutation is defined as the random alteration of a single position with probability p_m. A single bit position survives with a probability (1 - p_m), hence a complete schema H of order o(H) survives with probability (1 - p_m)$^{o(H)}$. Combining mutation with Equation [13] results in the *schema theorem*:

$$E[m(\mathrm{H},t+1)] \geq \frac{\mu(\mathrm{H},t)}{\bar{f}(t)} m(\mathrm{H},t)\left(1 - p_c \frac{d(\mathrm{H})}{l-1}\right)(1-p_m)^{o(\mathrm{H})}. \qquad [14]$$

The addition of mutation changes the previous conclusions little. Still short-, low-order, above-average schemata are sampled with exponentially increasing rates, because of their higher probability of surviving the recombination process. The real strength of a GA comes from the observation that an encoded solution matches not one, but many different schemata. This means that fit strings sample simultaneously from several building blocks.

This event is called implicit parallelism. The number of schemata that are processed is thus large, compared to the number of strings. Holland [13] stated that if a population consists of N strings, the number of schemata that are usefully processed is about N^3. Where "useful" means that these schemata are not destroyed in the recombination process.

Now the definition of a building block can be given: a building block is a low-order schema with a short defining length which is highly fit, i.e. the strings that match the schema are relatively fit. It is crucial that the defining length of a building block is short because crossover destroys schemata with a long defining length more often than those with a short defining length.

Note however that so far, in deriving the schema theorem, we only considered the destructive effects, i.e. the survival rates of schemata against crossover. This would suggest that crossover is bad thing that might as well be removed from the algorithm. However, this is definitely not true. A Genetic Algorithm without crossover will, in general, fail. Crossover is believed to be the major source of the power of Genetic Algorithms, with the ability to recombine instances of good schemas to produce instances of equal or even better schemas. This hypothesis that Genetic Algorithms work because of this constructive power is known as the Building Block Hypothesis.

What is the role of mutation? Mutation is seen as an 'insurance policy' against the loss of diversity. Consider for instance the situation that a population ends up with bit strings that all start with a '1'. Crossover alone cannot escape from this situation and half of the search space is therefore no longer accessible, which means a drastic loss of diversity in the population. Mutation can repair this, and escape from this situation.

We might now go back to our original question: "why do Genetic Algorithms work". We quote M. Mitchell [12] who states "The Genetic Algorithm increases the number of instances of low-order, short-defining length, high observed fitness schemata (the schema theorem), and these schemata serve as building blocks that are combined, via crossover, into candidate solutions with increasingly higher order and higher observed fitness (building block hypothesis)". In other words, the schema theorem suggests that selection focuses the search on subsets of search space with estimated above average fitness. The building block hypothesis suggests that crossover puts high-fitness building blocks together to create strings of increasingly higher fitness. Finally, Mutation provides insurance to preserve diversity at any locus. Genetic Algorithms therefore seem to find a good balance between *exploitation* (the use and propagation of fit schemata) and *exploration* (the search for new, fitter schemata). This belief is among other based on the theory of Holland (the analogy with the two-armed bandit problem, see [12]) and on a large body of experience with Genetic Algorithms for a wide range of applications. However, Genetic Algorithms will not always work and there is much debate on the validity of theory of Genetic Algorithms. It still is a wide-open research topic why and how Genetic Algorithms work. The interested reader is strongly advised to read the book of Mitchell [12] to learn more about this exiting topic.

D.5. Practical Matters

Although the principles of Genetic Algorithms will be clear now, we did not discuss the practical realization of the main operators. First we will spent some time on the issue of representing a phenotype in a genotype, i.e. in some (binary) encoding. Next we will discuss reproduction (or selection) and recombination.

Genotype Encoding and the Fitness Function

Only two components of a Genetic Algorithm are problem dependent: the encoding of a solution and the fitness-function. A solution to a problem can be described by a set of parameters, which are given some value. If we take for example TSP, the parameters consist of the order of cities in a given tour. Most often the variables representing the parameters are represented by *bit strings*. As such the variables are discretised in an a-priori fashion and the range of discretisation corresponds to some power of 2. If we use for example 10 bits per parameter, we obtain a range of 1024 discrete values.

Sometimes it can be quite difficult to properly encode a given parameter. What if a parameter can only be assigned an exact number of X values while X is not a power of 2? For example, our familiar TSP problem, which consists of N parameters representing the number of cities, which can only take N different values per parameter. Of course it is possible to code this also in bit strings, (which is naturally always the final coding scheme in a digital computer) but a different representation can often be more useful.

Usually fixed length fixed order bit strings are used for encoding, mainly for historical reasons, but also because crossover and mutation are very easy on bit strings. Furthermore, most Genetic Algorithm theory is based on this encoding. However, for many applications this representation is unnatural. As an alternative one my use a large alphabet for the chromosomes (i.e. many-character encoding) or even real-value encoding. To date there still is much debate if these alternative are better as compared to binary encoding. One can even go one step further and consider open ended or adaptive encoding, such as e.g. a growing tree, or slowly growing chromosomes. These issues are still an active research topic, and more information can be found in [12].

The other issue is the objective function that is usually given as part of the problem description. The formulation of an objective function is not always trivial. Sometimes it can be quite difficult to assign a value to a specific solution. For example if there are many competing factors which determine the "goodness" of a solution, as can be the case in process scheduling on a multiprocessor. Both communication- and calculation-costs must be minimized, but these decisions can not be taken separately, because they are interrelated. [14] On the other hand if we are given a specific function and we are interested in finding its minimum, the fitness can be directly related to this function value. An important criterion for the fitness function is that it must be relatively fast to calculate. This is especially important for genetic algorithms, since they work with a large population of potential solutions.

Selection

Before entering the next generation there is a moment in which the Genetic Algorithm keeps a temporary or intermediate population of individuals that may survive the next generation. This temporary population is called the *mating pool*. The mating pool consists of copies of individuals from the previous generation. A selection process is responsible for selecting those individuals. Individuals with a higher fitness have a larger probability to get in the mating pool.

Selection should be balanced with variation. Too strong selection will result in sub optimal highly fit individuals that may take over the population, reducing diversity needed for further progress. On the other hand, too weak selection results in very slow convergence.

We will discuss the bulk of selection methods used by the GA-community. They can be subdivided into a) fitness proportionate selection methods (*roulette wheel* and *stochastic universal sampling*); b) *tournament selection*; and c) *elitism*. First, we will explain how a selection operator can differentiate between good and bad solutions.

For the sake of simplicity we will only consider bit strings to represent solutions of an optimization problem. In particular we will only treat optimization of functions. Given an encoded solution of a function, how can one decide if it is near the optimum, or far from the optimum, if you have no knowledge about this optimum at all? The solution to this problem in GA is to compare individuals within the current population. Instead of absolute fitness, GA uses relative fitness. Better individuals receive a higher fitness. In order to obtain the fitness, a fitness function must be designed, which quantifies the essential features of a solution to a problem. In function optimization this is actually quite simple.

Consider the following example. Let our objective function be $G(x) = x^2$, we can equal our fitness function $F(x)$ to $G(x)$. Let individual x correspond to bit string 0010, then decoding of the bit string gives $x = 2$ and $F(x) = 4$. Let there be only one other individual

y with $F(y) = 9$. Suppose we are interested in finding the minimum of this function, hence we must assign probabilities accordingly. The normalized fitness is usually expressed as a number between 0 and 1, which equalizes it to a so-called selection probability. Now we can assign the probability f_x = 1.0 - 4/(4+9) to individual x and f_y = 1.0 - 9/(4+9) to individual y. It is clear that individual x has a higher selection probability than y because it has a lower fitness value.

The following objective-to-fitness transformation is also commonly used

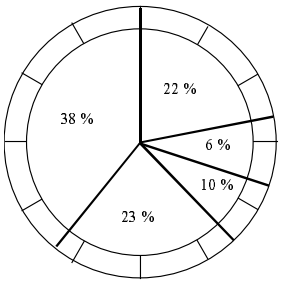
$$F(x) = C_{max} - G(x), \quad if\ G(x) < C_{max},$$

$$F(x) = 0, \quad \text{otherwise}.$$

There are a variety of ways to choose the coefficient C_{max}. It may be taken as an input coefficient, as the largest value of G observed thus far, as the largest value of G in the current population, or the largest value of G of the last k generations. A problem may also arise at early and later stages of the Genetic Algorithm cycle. In the beginning differences between individuals are large and selection may result in the dominance of the best string, because that string will automatically be selected often. At later stages, solutions often only differ slightly and the selection probabilities will almost be equal. This way it is hard to differentiate between the solutions. As a result, the best individual is now treated almost equal to less fit individuals. Both events, at the beginning and at the end, are undesirable but can be solved by scaling the fitness values. At the beginning of a run, differences must be scaled down and at the end, differences must be scaled up. A detailed description about scaling can be found in [11]. Note that in function optimization it is less difficult to obtain an objective function than for other applications. In TSP for example the objective function had to be constructed with the mathematical formulation of the problem in our minds. In function optimization the objective function simply equals to function that is optimized.

The first selection methods that will be discussed are fitness proportional methods where the expected number of times an individual is selected to reproduce equals that individual's fitness divided by the average fitness of the population (see e.g. Table 3 and Table 4).

After the assignment of a fitness value to every individual in the population, a mechanism has to be used to select certain individuals for the mating pool. In this section we will discuss some commonly used methods. An exhaustive discussion about different selection methods can be found in [Goldberg, 1989 #40].



individual	bit string	fitness	selection probability
1	001010	119	0.22
2	011101	34	0.06
3	101101	56	0.10
4	000011	128	.23
5	100001	211	.38
Total		548	1.0

Figure 25: Roulette wheel selection, the table shows arbitrary individuals and fitness values.

With *roulette wheel selection* each individual of the population is related to a part of a roulette wheel. The size of this part is proportionate to the fitness of the individual. The wheel will spin N times, where N is the population size. Every spin results in copying one string to the mating pool. As the roulette wheel consists of different parts, it is biased towards fitter individuals. The mating pool will thus show up an average fitness, which is most probable to be higher than that of the original population. This is exactly the goal of the selection. A pseudo code for roulette wheel selection is given in Algorithm 7.

```
/* pseudo code for roulette wheel selection */
Repeat N times
Choose a random number r between 0 and N.
Loop through individuals, summing expected values, until sum is
greater or equal than r. The individual who puts the sum above
r is the one selected.
```

Algorithm 7: Roulette wheel selection.

The main problem with roulette wheel selection lies in the fact that the actual number of selected individuals may deviate significantly from expected value, especially if the populations are relatively small. *Stochastic Universal Sampling* repairs this by guaranteing that if the expected value of individual i equals $E[i]$ at least $\lfloor E[i] \rfloor$ individuals will be selected, but no more than $\lceil E[i] \rceil$.

Stochastic universal sampling or more complete, stochastic remainder sampling without replacement, calculates the expected number of copies. This is the selection probability p multiplied by the population size. This means that with a population size N, an individual with $p > 1/N$ will have an expected number larger than 1 and is said to have an above average fitness. Individuals below average fitness will have an expected number smaller than 1. In all cases, the expected number will have an integer part and a fractional part. This process proceeds in two phases. In the first phase the population will be filled with an integer part number of copies of every individual. After this first step, the mating pool is not yet completely filled. The pool will be filled using the fractional part of the expected number. By weighted coin tossing it is decided whether a copy of an individual will enter the pool. This is repeated until the mating pool is completely full. A pseudo code is provided in Algorithm 8.

```
/* pseudo code for Stochastic Universal Sampling */
r is a real random number between 0 and 1
for(sum = i = 0; i < N; i++)
      for(sum += E[i]; sum > r; r++)
        select(i)
```

Algorithm 8: Stochastic Universal Sampling.

The main problem in fitness proportionate selection lies in premature convergence. If the initial population contains a few relatively fit individuals, they quickly dominate future populations, thus preventing exploration of search space. Possible solution are "scaling methods", that were already introduced shortly above. The idea is to map raw fitness to expected values such that the Genetic Algorithm is less susceptible to premature convergence. One possibility is sigma scaling, i.e.

$$E[i] = \begin{cases} 1 + \dfrac{f(i) - \bar{f}(t)}{2\sigma(t)} & \text{if } \sigma(t) \neq 0, \\ 1.0 & \text{if } \sigma(t) = 0, \end{cases}$$

where $\sigma(t)$ is the standard deviation in the fitness of the population at time t. Another possibility is rank selection. Here individuals are sorted according to their fitness. The resulting rank is then mapped to expected values.

Beside the problem of premature convergence, fitness proportional selection also has computational problems. Fitness proportionate selection requires two passes through the population, one to compute the mean fitness and one to compute the expected value of each individual. Rank scaling requires sorting the entire population, potentially a very time consuming process. *Tournament selection* does not have these computational problems. It is similar to rank selection, but computationally more efficient and amenable to parallel computing.

Tournament selection picks K strings at random. These individuals will compete to get into the mating pool. The winner of the tournament will be the individual with the highest fitness. This process is repeated until the pool is completely filled. The size of the tournament K varies between 2 and N (the size of the population). If K is close to N the probability of only one individual being selected is getting high. This is not a desirable situation, because we do not want our mating pool to be completely filled with N copies of only one individual. Hence K has to be considerably smaller than N. If $K = 2$, on average, all strings are selected to enter the tournament twice. This would implicate, that above average individuals will get two copies in the mating pool, because they will win both tournaments they compete in. On the other hand, below average individuals will not get in the mating pool at all. This implies that the *ranking* of the population has become more important than its fitness. Individuals that are much better than the rest will only receive a limited number of copies in the mating pool, hence fitness scaling is not necessary in this case. Instead of always choosing the most fit individual as the winner of the tournament, one can also choose a random parameter r between 0 and 1, if $r < k$ (where k is, for example, 0.75), the more fit individual goes into the mating pool.

An important factor in deciding what selection mechanism to use is called *selection pressure*. The higher the selection pressure the shorter it takes for the current best individual to dominate the complete population. In the discussed selection methods it is obvious that roulette wheel selection pressure is high, because on average, there will be a large number of copies of the best individual in the mating pool. On the other hand, tournament selection, with low tournament size, has little pressure, because there will be approximately 2 copies of the best individual in the mating pool. The *convergence* of a Genetic Algorithm is highly influenced by the value of this measure. A high selection pressure can decrease the execution time considerably. To make a choice between high or low selection pressure, one must know which performance is desired, i.e. fast execution time or good solution quality. Experiments with different methods must reveal the best method for the problem at hand.

When no precautions have been taken, there can be no guarantee that the best individual will survive the next generation. As we shall later discover, it is essential for asymptotic convergence that the best individual so far is kept in the population. To prevent the loss of the highest fitness, an *elitist strategy* can be used. [12, 15] Elitism means that the best individual in generation t will always be present in generation $t + 1$. This is achieved by explicitly copying the best individual in the next generation, once it is formed. Therefore another individual must be replaced if it was not already present. A disadvantage of elitism could be possible slower convergence, because this best individual could be a local optimum far away from the global optimum. The evolution of a Genetic Algorithm is now partly governed by the presence of this individual. Many examples have shown that elitism significantly improves the performance of Genetic Algorithms. [12]

Recombination

Recombination is the most essential feature of a Genetic Algorithm. Without recombination operators there would be no exploration of the phase space. Recombination in Genetic Algorithms is essentially different from the random walks in Simulated Annealing. In Genetic Algorithms the information in the individuals is used to guide the search for new exploration points. By combining information from individuals the offspring are "hoped" to receive the best of both worlds. This combining of individuals is called *crossover*. Another essential operator is *mutation*, which has to ability to reintroduce previously extinct individuals by a noise generation mechanism.

Crossover

The crossover operator picks two strings out of the mating pool and crosses them with a probability p_c. Usually p_c is quite high and somewhere between 0.6 and 0.9. This range has been established experimentally, a rigorous theoretical explanation is missing. In order to cross two individuals x and y, a bit position between 1 and l-1 is chosen randomly, where l is the length of a bit string. Both individuals are split and two new individuals (the offspring) are formed from the four pieces of the two parents. Often only

one the offspring with the highest fitness is selected, but this is not always the case. The new individual(s) are put in the new population. The crossover process is repeated until the new population is full. Figure 26 shows an example of the crossover operations.

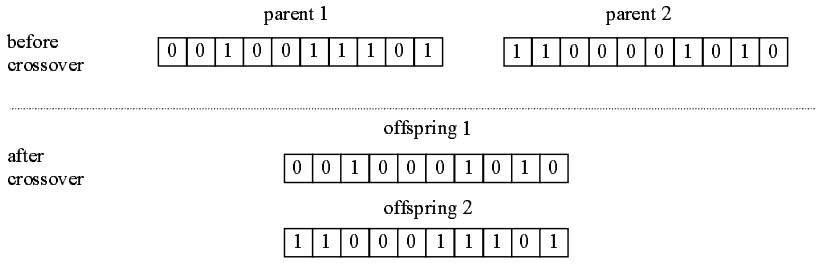


Figure 26: Crossover operator.

Variations on the crossover operator are possible. We have just discussed the so called *1-point crossover*, where only one splitter is chosen in the two individuals. A generalization of 1-point crossover is *m-point crossover*, where *m* splitter points are picked. Note that the crossover operator produces only legal individuals if the complete range of bit strings is valid. As discussed earlier, it is possible that for example only 450 out of 1024 values are valid. It is obvious that there is a large possibility that crossover produces non-valid individuals. If we take TSP as an example, it is not possible to just cut two tours and glue the ends together, because tours can be produced where cities appear more than once. A solution to this problem is the so called PMX-crossover (Partially Matched Crossover) which is described in [Goldberg, 1989 #11.

Crossover is responsible for the *exploration* of the search space. It combines partial solutions to form new solutions. It explores new points, as opposed to the selection mechanism, which is responsible for the *exploitation* of the search space, new points are tested and exploited.

Mutation

In the reproduction and crossover process certain bit values at specific sites may become extinct. Mutation can bring those lost bits back. It is the simplest genetic operator. It operates on all bits of the bit string. It randomly flips all bits of a bit string with a probability p_m, which is usually small. A commonly found value for p_m is approximately $1/l$, with an expected number of bit flips per bit string of 1. Again, this is an experimentally established value.

Mutation is a very valuable operator, it can prevent the algorithm from getting stuck in local optima. One can imagine that selection and crossover alone could easily guide the population to sub-optimality from which it can not escape without a mutation operator. Even sudden increases in the fitness of individuals may be caused by mutation.

D.6. Convergence

An optimization algorithm terminates when the optimum is found. This argument is of course only realistic in the case of known global optima. But it is obvious that we do not know these solutions when dealing with a "real" application. Known local optima can however by used when it is the GA itself we are interested in. A stopping criterion has to be found to prevent the GA from running indefinitely. A possible scheme is to wait for a number of X consecutive generation where the highest fitness in the population does not change. Another version is to wait for the moment when the difference between the current best optimum and the previous best optimum gets smaller than some small value ε. There is no way to deterministically establish how many generations it will take for a GA to terminate, because of the stochastic nature of the algorithm itself. Maybe it is possible to find statistical proofs for convergence, like the ones known for Monte Carlo methods. Because of the lack of a proper theory for GA research in this direction can become quite difficult.

D.7. Parallel GA

Introduction

Despite a GA's capability to implicitly process N^3 schemata at the same time, which makes it implicitly parallel, we would like to exploit some explicit parallelism to efficiently execute a GA on parallel architectures. At a first glance it seems difficult to exploit such explicit parallelism, because of the global knowledge that is needed for the reproduction step. In order to select individuals for the mating pool one has to calculate the average fitness of the entire population. In other words, it is difficult to localize a GA.

Yet it is possible to introduce parallelism into Genetic Algorithms. Three approaches are knows, that are based on (changing) the population structure and the selection strategies. The first strategy is called *global parallel Genetic Algorithm*, and views the total population as a single breeding unit. Parallelism is found on the level of the calculations needed for selection. A second solution could be a selection scheme that does not depend on global knowledge, but uses a local selection scheme. An example of a local selection scheme, i.e. one that does not depend on global knowledge, is tournament selection. Other techniques include the introduction of a *neighborhood structure*, where individuals can only mate with other individuals in their direct neighborhood. This leads to so-called *migration Genetic Algorithms*. One could say that the idea of migration Genetic Algorithms is also based on another observation from nature. In natural evolution, species tend to reproduce within subgroups, *isolated* from each other, in a total population, but with the possibility of mating occurring across boundaries of subgroups. So, we add a feature to the original Genetic Algorithm (isolated subgroups with migration). This addition may improve the performance of Genetic Algorithms (and it does) and it gives us parallelism on the level of the subgroups. In the extreme case one could take very small separate breeding units, i.e. the individual and its immediate neighborhood. This leads to the *diffusion Genetic Algorithms*. We will briefly discuss all three approaches. Parallelization of a Genetic Algorithm does not affect the schema theorem, which implies that exponentially increasing samples are given to the observed best building blocks.

Global parallel Genetic Algorithms

A first approach towards parallelizing Genetic Algorithms begins with the observation that for each individual the fitness must be calculated and that in principle all these fitness calculations are independent from each other. If the fitness calculations take a large amount of time relative to the other calculations (this is off course highly dependent on the application) it might pay off to parallelize the fitness calculations, and keep the rest of the calculations sequential. Knowing the basic speedup law of Amdahl that relates the asymptotic speedup S_∞ to the sequential fraction α of the parallel code as $S_\infty = 1/\alpha$ we can easily judge if or when parallelizing the fitness calculations pays off. If for instance 90% of the execution time is due to fitness calculations, then α would be 0.1, and the maximum speedup that can ever be obtained will be 10. So, on say 4 processors one might hope to achieve some nice speedup (remember, Amdahl did not include any other overheads). However, it would be ridiculous to use for instance 100 processors in this specific example. Given the population size and some approximation of the computational complexity of the fitness calculation, one can easily find an approximation of α and judge to what extent global parallelism pays off.

The global parallel Genetic Algorithm is easily implemented using a Master/Slave model, as drawn in Figure 27. Here the master is responsible for selection, crossover and mutation, and the slaves calculate the fitness of individuals they receive from the master.

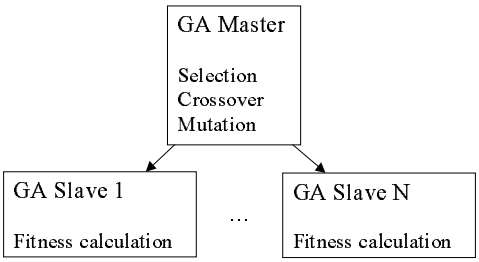


Figure 27: A global parallel Genetic Algorithm in the Master/Slave model.

If Genetic Algorithms are used in optimization of e.g. structures in Engineering (such as the optimal shape of airfoils) the actual calculation of the fitness function can be very expensive, and involve some parallel calculations in itself. Especially for such applications the global parallel Genetic Algorithm is the most straightforward approach to obtain an efficient parallel Genetic Algorithm.

Migration models

The most common way to parallelize a GA is to divide the total population of individuals into sub-populations. The sub-populations are assigned to distinct (virtual) processors. On each processor runs a Genetic Algorithm acting on its assigned sub-population. The fitness of an individual is determined with respect to the sub-population. This approach, drawn schematically in is called the migration Genetic Algorithm, also called the *island model* or the *deme model*.

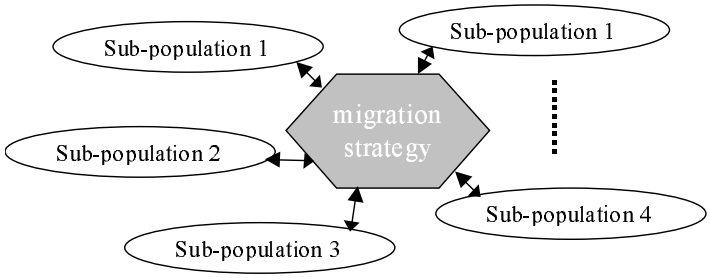


Figure 28: The Migration Genetic Algorithm.

```
/* pseudo code for a (parallel) migration GA for a */
/* single sub population */
initialize sub-population
while not finished {
   evaluate fitness
   selection
   crossover and mutation
      after every m iterations {
      send emigrants
      receive immigrants
   }
}
```

Algorithm 9: pseudo code for a (parallel) migration Genetic Algorithm.

The pseudo code of the Migration Genetic Algorithm is given in Algorithm 9. Each sub population is evaluated on a different processor, and after every m iterations a migration exchange is carried out. Many choices must be made:

- size of sub-populations (not necessarily n/p)
- Number of islands
- Connectivity of islands (totally connected, mesh, torus, ring, etc.)

- Migration strategy
- Migration interval
- Selection of emigrants, and their destination
- How immigrants are inserted

All these details will determine the behaviour of the migration Genetic Algorithm.

A possible implementation of sub-populations is done in the island model. Every individual can mate only with individuals in the sub-population. Once in a while the sub-populations exchange individuals, the sub-populations are able to share genetic material via migration. The island model is able to exploit differences in various sub-populations, which leads to genetic *diversity*. This genetic diversity is an important factor in Genetic Algorithms, because it prevents the algorithm from premature convergence. One can imagine that this genetic diversity is more present in parallel Genetic Algorithms than in sequential Genetic Algorithms. Imagine a parallel Genetic Algorithms as an algorithm with relatively independent sub-populations. A sub-fit individual is more likely to dominate the complete population in a sequential Genetic Algorithms than in a parallel Genetic Algorithms. The migration frequency should be initialized with care: for high frequencies global mixing will occur and differences between islands will fade out; for low frequencies, the migration can cause the sub-populations to converge prematurely, because the sub-population will effectively behave as very small sequential genetic algorithms.

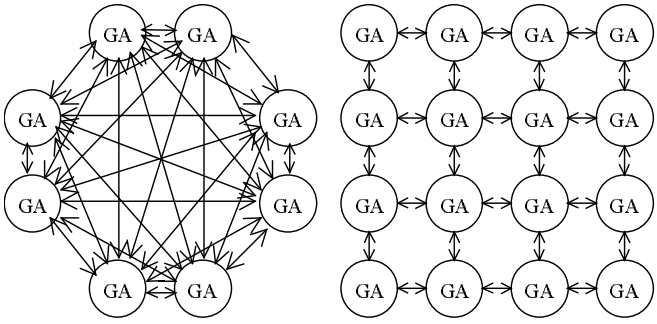


Figure 29: The left figure illustrates the island model, the right one the stepping stone model.

Note that the implementation of an island model yields migration between all processors. Hence performance results are dependent on the connectivity of the multiprocessor used. Distant processors should be able to communicate efficiently, or else the overhead will be large. Putting restrictions on the migration process can prevent both the latter problem and the problem of premature convergence by global mixing. A possible restriction is using only local migration between sub-populations. By local migration we indicate that migration is restricted to geographically nearby sub-populations. A model, which implements this restriction, is called the *stepping stone model*. Now, it will take a fit string longer to propagate through all sub-populations, it will first only dominate the sub-population it came from and its nearest neighbors. Different sub-populations emphasize different characteristics in the individuals and this differentiation is needed to prevent premature convergence. The island model and the stepping stone model are illustrated in Figure 29.

Finally, hands-on experience with the migration model shows that it typically has an improvement of performance of standard Genetic Algorithms. The quality of solution is better, i.e. superior solutions are found, with higher average fitness of population, due to higher genetic diversity of the sub-populations and avoidance of premature convergence (also called the *niching effect*). Note that this is concluded on many case studies, however, no general conclusions ! Furthermore, the total number of function evaluations needed in migration Genetic Algorithms can be lower than in the standard Genetic Algorithm. Finally, high parallel efficiencies can be obtained (explain why !).

Diffusion Genetic Algorithms

The diffusion Genetic Algorithms, also known as *isolation by distance* model, and *neighborhood* -, *cellular* -, or *fine-grained* Genetic Algorihtms., view the population as single continuous structure, with strict *geographic location* for *each individual*, and reproduction for all individuals in a *small local neighborhood*. Diffusion Genetic Algorithms have the same computational structure as e.g. a Cellular Automaton or stencil-based operations on a grid (remember the diffusion in the numerical solvers). Therefore, parallelism is now easily introduced by spatial decomposition of the population. The Diffusion Genetic Algorithms are an extreme case of the migration model, with one individual per island, individuals arranged on some grid, and always migration between local individuals. An example of a spatial structure of individuals, together with some possible interaction neighborhoods (or stencils) are shown in

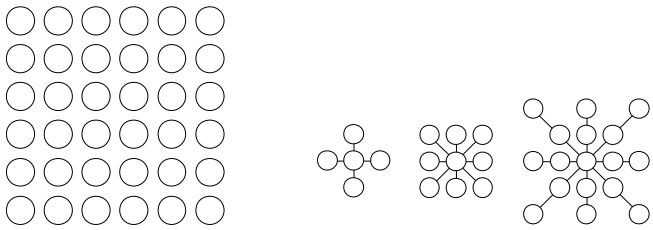


Figure 30: An example of a spatial structure of individuals (left) and three possible local interaction neighborhoods (right) for the Diffusion Genetic Algorithm.

Boundary conditions can be made periodic, which is also often done with cellular automata, which maps a 2-dimensional population on a torus topology. Each individual is allowed only to interact with its direct neighbors. By allowing individuals only to mate with other individuals in their neighborhood, one obtains a continuum model in which the genetic information frequency changes smoothly. This process resembles a diffusion process, for this reason the name diffusion Genetic Algorithms was coined. The diversity in this model is even higher than in the stepping stone model, because mating is even more restricted. Note that every generation migration takes place, which imposes heavy communication loads on multiprocessor architectures. The number of virtual processors that reside on a physical processor (i.e. the granularity) is an important factor that has to be considered. There is another important factor in the number of communicated individuals, which is given by the interaction range or locality. The locality of the algorithm can be tuned, i.e. the neighborhood range can vary between 1 and $\sqrt{N}/2-1$ individuals (where N is the population size). The most local, short-range algorithm is obtained with the minimum neighborhood range. All individual interactions are restricted to their nearest neighbors. A maximum neighborhood range implies in fact a fully connected neighborhood structure, hence we obtain a global, long range algorithm.

Finally some observations from running the Diffusion Genetic Algorithms for some specific examples. The diversity is typically higher than in migration model. *Virtual islands* tend to form, i.e. the population is initialized randomly, but after a few generations, local clusters of individuals with similar genetic material may appear, giving rise to virtual islands. The number of islands tends to reduce, and those with high fitness tend to increase in size. Good results, both in convergence, and parallel efficiency, have been reported.

References

1. Sloot, P.M.A.: Computational Physics: Stochastic Simulation. Faculty of Science, University of Amsterdam, The Netherlands, (2002)

2. Laarhoven, P.J.M.,Aarts, E.H.: Simulated Annealing: theory and applications. D. Reidel Publishing Company, (1987)
3. Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A.,Teller, E.: Equation of state calculations by fast computing machines. Journal of Chemical Physics **21** (1953) 1087-1092
4. Kirkpatrick, S., Gelatt jr., C.D.,Vecchi, M.P.: Optimization by Simulated Annealing. Science **220** (1983) 671-680
5. Cole, J.B.: The Statistical Mechanics of Image Recovery and Pattern Recognition. Am. J. Phys. **59** (1991) 839-842
6. Voogd, J.M., Sloot, P.M.A.,van Dantzig, R.: Comparison of vector and parallel implementations of the simulated annealing algorithm. Future Generation Computer Systems **11** (1995) 467-475
7. Seifert, G.,Jones, R.O.: Structure of phosphorus clusters by simulated annealing. Z. Phys. D **26** (1993) 349-351
8. Mansour, N.,Fox, G.C.: Allocating data to multicomputer nodes by physical optimization algorithms for loosely synchronous computations. Concurrency: practice and experience **4** (1992) 557-574
9. Sohn, A.: Parallel speculative computation of simulated annealing. In (Eds.): 1994 International conference on parallel processing. (1994)
10. Hoekstra, A.G.: Introduction Parallel Computing. Faculty of Science, University of Amsterdam, The Netherlands, (2001)
11. Goldberg, D.E.: Genetic Algorithms in search, optimization and machine learning. Addison-Wesley, (1989)
12. Mitchell, M.: Introduction to Genetic Algorithms. (1996)
13. Holland, J.H.: Adaptation in Natural and Artificial Systems. University of Michigan, Ann Arbor (1975)
14. de Ronde, J.F.: Mapping in High Performance Computing, a case study on finite element simulation. University of Amsterdam (1998).
15. De Jong, K.A.: An Analysis of the Behavior of a class of Genetic Adaptive Systems. University of Michigan (1975).