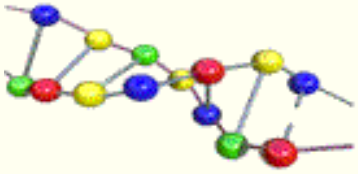
Introduction to Genetic Algorithms



GENETIC

ALGORITHMS

These pages **introduce** some of the fundamentals of genetics algorithms. These pages are intended to be used for learning about genetics algorithms **without any previous knowledge** from this area. Only some knowledge of computer programming is assumed. Several interactive **Java applets** have been included to demonstrate basic concepts of

genetic algorithms.

As the area of genetic algorithms is very wide, it is not possible to cover everything in these pages. But you should get some idea what genetic algorithms are and what they could be useful for. Do not expect any sophisticated mathematics theory here.

Now please choose next to continue or you can choose any topic from the menu on the left side. If you do not want to read all the introductory chapters, you can skip directly to genetic algorithms and return later.

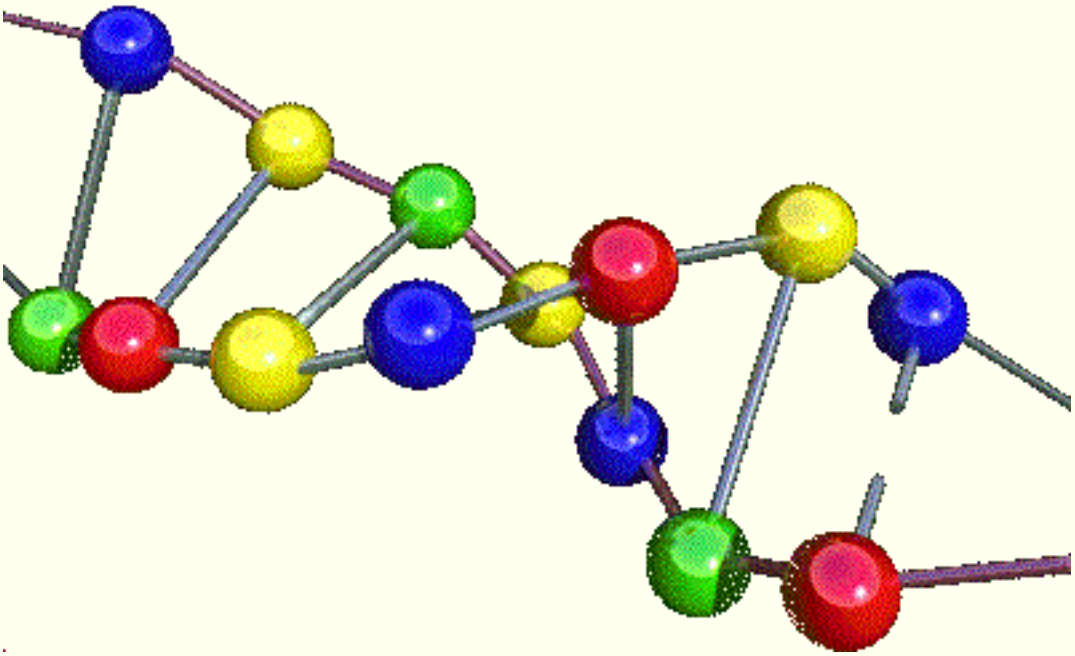
Translations of these pages are also available - you may go to the Japanese translation (translated by Ishii Manabu) or to the Bulgarian translation (translated by Teodor Gig). The English version should be now more readable thanks to the corrections provided by Bob Owen.

You can also check recommendations for your browser. Go here for a reference if you want to cite these pages.

[This page without frames] [This page with frames]

DNA (Deoxyribonucleic acid)

This is a part of DNA. More pictures are available.



Back

GENETIC ALGORITHMS

These pages **introduce** some of the fundamentals of genetics algorithms. These pages are intended to be used for learning about genetics algorithms **without any previous knowledge** from this area. Only some knowledge of computer programming is assumed. Several interactive **Java applets** have been included to demonstrate basic concepts of genetic algorithms.

As the area of genetic algorithms is very wide, it is not possible to cover everything in these pages. But you should get some idea what genetic algorithms are and what they could be useful for. Do not expect any sophisticated mathematics theory here.

Now please choose next to continue or you can choose any topic from the menu on the left side. If you do not want to read all the introductory chapters, you can skip directly to genetic algorithms and return later.

Translations of these pages are also available - you may go to the Japanese translation (translated by Ishii Manabu) or to the Bulgarian translation (translated by Teodor Gig). The English version should be now more readable thanks to the corrections provided by Bob Owen.

You can also check recommendations for your browser. Go here for a reference if you want to cite these pages.

[This page without frames] [This page with frames]

I. Introduction

First Words

Genetic algorithms are a part of **evolutionary computing**, which is a rapidly growing area of artificial intelligence.

As you can guess, genetic algorithms are inspired by Darwin's theory of evolution. Simply said, problems are solved by an evolutionary process resulting in a best (fittest) solution (survivor) - in other words, the solution is evolved.

History

Evolutionary computing was introduced in the 1960s by I. **Rechenberg** in his work "***Evolution strategies***" (*Evolutionsstrategie* in original). His idea was then developed by other researchers. **Genetic Algorithms** (GAs) were invented by John **Holland** and developed by him and his students and colleagues. This lead to Holland's book "*Adaption in Natural and Artificial Systems*" published in 1975.

In 1992 John **Koza** has used genetic algorithm to evolve programs to perform certain tasks. He called his method "**genetic programming**" (GP). LISP programs were used, because programs in this language can expressed in the form of a "parse tree", which is the object the GA works on.

Previous Next

II. Biological Background

Chromosome

All living organisms consist of cells. In each cell there is the same set of **chromosomes**. Chromosomes are strings of DNA and serve as a model for the whole organism. A chromosome consists of **genes**, blocks of DNA. Each gene encodes a particular protein. Basically, it can be said that each gene encodes a **trait**, for example color of eyes. Possible settings for a trait (e.g. blue, brown) are called **alleles**. Each gene has its own position in the chromosome. This position is called **locus**.

Complete set of genetic material (all chromosomes) is called **genome**. Particular set of genes in genome is called **genotype**. The genotype is with later development after birth base for the organism's **phenotype**, its physical and mental characteristics, such as eye color, intelligence etc.

Reproduction

During reproduction, **recombination** (or **crossover**) first occurs. Genes from parents combine to form a whole new chromosome. The newly created offspring can then be mutated. **Mutation** means that the elements of DNA are a bit changed. This changes are mainly caused by errors in copying genes from parents.

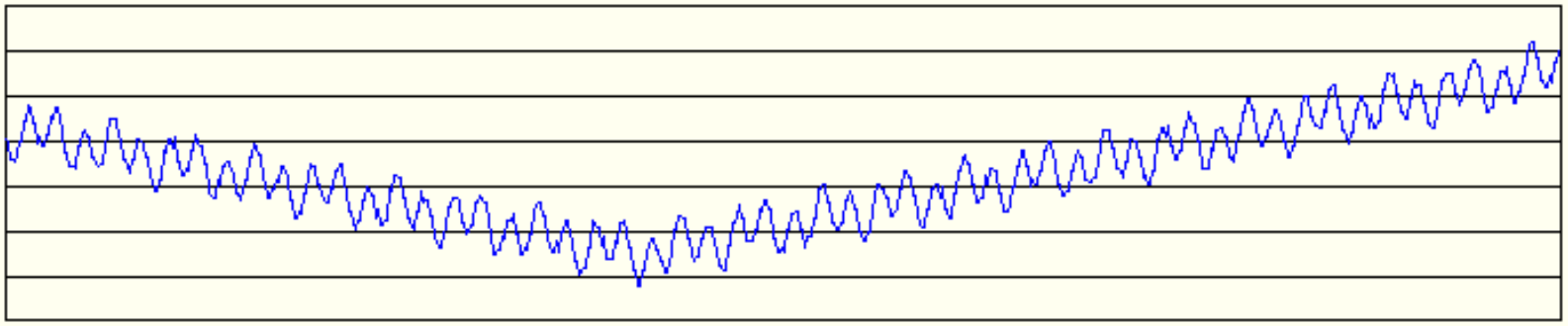
The **fitness** of an organism is measured by success of the organism in its life (survival).

III. Search Space

Search Space

If we are solving a problem, we are usually looking for some solution which will be the best among others. The space of all feasible solutions (the set of solutions among which the desired solution resides) is called **search space** (also state space). Each point in the search space represents one possible solution. Each possible solution can be "marked" by its value (or fitness) for the problem. With GA we look for the best solution among among a number of possible solutions - represented by one point in the search space.

Looking for a solution is then equal to looking for some extreme value (minimum or maximum) in the search space. At times the search space may be well defined, but usually we know only a few points in the search space. In the process of using GA, the process of finding solutions generates other points (possible solutions) as evolution proceeds.



Example of a search space

The problem is that the search can be very complicated. One may not know where to look for a solution or where to start. There are many methods one can use for finding a **suitable solution**, but these methods do not necessarily provide the **best solution**. Some of these methods are **hill climbing**, **tabu search**, **simulated annealing** and the **genetic algorithm**. The solutions found by these methods are often considered as good solutions, because it is not often possible to prove what the optimum is.

NP-hard Problems

One example of a class of problems which cannot be solved in the "traditional" way, are NP problems.

There are many tasks for which we may apply fast (polynomial) algorithms. There are also some problems that cannot be solved algorithmically.

There are many important problems in which it is very difficult to find a solution, but once we have it, it is easy to check the solution. This fact led to **NP-complete problems**. NP stands for nondeterministic polynomial and it means that it is possible to "guess" the solution (by some nondeterministic algorithm) and then check it.

If we had a guessing machine, we might be able to find a solution in some reasonable time.

Studying of NP-complete problems is, for simplicity, restricted to the problems where the answer can be yes or no. Because there are tasks with complicated outputs, a class of problems called **NP-hard** problems has been introduced. This class is not as limited as class of NP-complete problems.

A characteristic of NP-problems is that a simple algorithm, perhaps obvious at a first sight, can be used to find usable solutions. But this approach generally provides many possible solutions - just trying all possible solutions is very slow process (e.g. O(2^n)). For even slightly bigger instances of these type of problems this approach is not usable at all.

Today nobody knows if some faster algorithm exists to provide exact answers to NP-problems. The discovery of such algorithms remains a big task for researchers (maybe you! :-)). Today many people think that such algorithms do not exist and so they are looking for an alternative method. An example of an alternative method is the genetic algorithm.

Examples of the NP problems are satisfiability problem, travelling salesman problem or knapsack problem. Compendium of NP problems is available.

Previous Next

IV. Genetic Algorithm

Basic Description

Genetic algorithms are inspired by Darwin's theory of evolution. Solution to a problem solved by genetic algorithms uses an evolutionary process (it is evolved).

Algorithm begins with a **set of solutions** (represented by **chromosomes**) called **population**. Solutions from one population are taken and used to form a new population. This is motivated by a hope, that the new population will be better than the old one. Solutions which are then selected to form new solutions (**offspring**) are selected according to their fitness - the more suitable they are the more chances they have to reproduce.

This is repeated until some condition (for example number of populations or improvement of the best solution) is satisfied.

> ***Example***
> *As you already know from the chapter about search space, problem solving can be often expressed as looking for the extreme of a function. We solve exactly this problem here - a function is given and GA tries to find the minimum of the function.*
> *Try to run genetic algorithm in the following applet by pressing the Start button. The graph represents a search space and vertical lines represent solutions (points in search space). The red line is the best solution, green lines are the other ones.*
> *The Start button starts the algorithm, Step button performs one step (i.e. forming one new generation), Stop button stops the algorithm and Reset button resets the population.*

Here is applet, but your browser does not support Java. If you want to see applets, please check browser requirements.

Outline of the Basic Genetic Algorithm

1. **[Start]** Generate random population of *n* chromosomes (suitable solutions for the problem)
2. **[Fitness]** Evaluate the fitness *f(x)* of each chromosome *x* in the population
3. **[New population]** Create a new population by repeating following steps until the new population is complete
    1. **[Selection]** Select two parent chromosomes from a population according to their fitness (the better fitness, the bigger chance to be selected)
    2. **[Crossover]** With a crossover probability cross over the parents to form new offspring (children). If no crossover was performed, offspring is the exact copy of parents.
    3. **[Mutation]** With a mutation probability mutate new offspring at each locus (position in chromosome).
    4. **[Accepting]** Place new offspring in the new population
4. **[Replace]** Use new generated population for a further run of the algorithm
5. **[Test]** If the end condition is satisfied, **stop**, and return the best solution in current population
6. **[Loop]** Go to step **2**

Some Comments

As you can see, the outline of the Basic GA is very general. There are many parameters and settings that can be implemented differently in various problems.

The first question to ask is how to create chromosomes and what type of encoding to choose. We then address Crossover and Mutation, the two basic operators of GA. Encoding, crossover and mutation are introduced in next chapter.

The next question is how to select parents for crossover. This can be done in many ways, but the main idea is to select the better parents (best survivors) in the hope that the better parents will produce better offspring.

You may think that generating populations from only from two parents may cause you to loose the best chromosome from the last population. This is true, and so **elitism** is often used. This means, that at least one of a generation's best solution is copied without changes to a new population, so the best solution can survive to the succeeding generation.

You might be wandering, *why* genetic algorithms work. It can be partially explained by the **Schema Theorem** (Holland), however, this theorem has been criticized in recent times. If you want to know more, check other resources.

V. Operators of GA

Overview

As you can see from the genetic algorithm outline, the crossover and mutation are the most important parts of the genetic algorithm. The performance is influenced mainly by these two operators. Before we can explain more about crossover and mutation, some information about chromosomes will be given.

Encoding of a Chromosome

A chromosome should in some way contain information about solution that it represents. The most used way of encoding is a binary string. A chromosome then could look like this:

Chromosome 1	1101100100110110
Chromosome 2	1101111000011110

Each chromosome is represented by a binary string. Each bit in the string can represent some characteristics of the solution. Another possibility is that the whole string can represent a number - this has been used in the basic GA applet.

Of course, there are many other ways of encoding. The encoding depends mainly on the solved problem. For example, one can encode directly integer or real numbers, sometimes it is useful to encode some permutations and so on.

Crossover

After we have decided what encoding we will use, we can proceed to crossover operation. Crossover

operates on selected genes from parent chromosomes and creates new offspring. The simplest way how to do that is to choose randomly some crossover point and copy everything before this point from the first parent and then copy everything after the crossover point from the other parent.

Crossover can be illustrated as follows: (| is the crossover point):

Chromosome 1	11011 | 00100110110
Chromosome 2	11011 | 11000011110
Offspring 1	11011 | 11000011110
Offspring 2	11011 | 00100110110

There are other ways how to make crossover, for example we can choose more crossover points. Crossover can be quite complicated and depends mainly on the encoding of chromosomes. Specific crossover made for a specific problem can improve performance of the genetic algorithm.

Mutation

After a crossover is performed, mutation takes place. Mutation is intended to prevent falling of all solutions in the population into a local optimum of the solved problem. Mutation operation randomly changes the offspring resulted from crossover. In case of binary encoding we can switch a few randomly chosen bits from 1 to 0 or from 0 to 1. Mutation can be then illustrated as follows:

Original offspring 1	1101111000011110
Original offspring 2	1101100100110110
Mutated offspring 1	1100111000011110
Mutated offspring 2	1101101100110110

The technique of mutation (as well as crossover) depends mainly on the encoding of chromosomes. For example when we are encoding permutations, mutation could be performed as an exchange of two genes.

VI. GA Example
Minimum of a Function

About the Problem

As you already know from the chapter about search space, problem solving can be often expressed as looking for extreme of a function defined over the search space. We solve exactly this problem here.

A function is given and GA tries to find minimum of the function. For other problems we just have to define search space and the fitness function (for which we want to find an extreme).

Example

Try to run the genetic algorithm in the following applet by pressing Start button. The graph represents a search space and vertical lines represent solutions (points in search space). The red line is the best solution, green lines are the other ones. Old and new population is shown at the top of the applet. Each population consists of binary chromosomes - red and blue points mean zeros and ones. You can see the process of forming the new population in steps in the applet.

Start button starts the algorithm, Step button performs one step (i.e. forming one new generation), Stop button stops the algorithm and Reset button resets the population.
It is suggested to start with pressing the Step button and to watch how GA works in details. The outline of GA has been introduced in one of the previous chapters. You can see elitism for the first chromosomes and then forming new offspring by crossover and mutation until the new population is completed.

Here is an applet, but your browser does not support Java. If you want to see applets, please check browser requirements.

VII. Parameters of GA

Crossover and Mutation Probability

There are two basic parameters of GA - crossover probability and mutation probability.

Crossover probability: how often crossover will be performed. If there is no crossover, offspring are exact copies of parents. If there is crossover, offspring are made from parts of both parent's chromosome. If crossover probability is **100%**, then all offspring are made by crossover. If it is **0%**, whole new generation is made from exact copies of chromosomes from old population (but this does not mean that the new generation is the same!).
Crossover is made in hope that new chromosomes will contain good parts of old chromosomes and therefore the new chromosomes will be better. However, it is good to leave some part of old population survive to next generation.

Mutation probability: how often parts of chromosome will be mutated. If there is no mutation, offspring are generated immediately after crossover (or directly copied) without any change. If mutation is performed, one or more parts of a chromosome are changed. If mutation probability is **100%**, whole chromosome is changed, if it is **0%**, nothing is changed.
Mutation generally prevents the GA from falling into local extremes. Mutation should not occur very often, because then GA will in fact change to **random search**.

Other Parameters

There are also some other parameters of GA. One another particularly important parameter is population size.

Population size: how many chromosomes are in population (in one generation). If there are too few chromosomes, GA have few possibilities to perform crossover and only a small part of search space is explored. On the other hand, if there are too many chromosomes, GA slows down. Research shows that after some limit (which depends mainly on encoding and the problem) it is not useful to use very large populations because it does not solve the problem faster than moderate sized populations.

Some recommendations for all parameters can be found in one of the following chapters.

Example
Here you can see example similar to previous one. But here you can try to change crossover and mutation probability. You can also control elitism.
On the graph below you can see performance of GA. Red is the best solution, blue is average value (fitness) of all population.
Try to change parameters and look how GA behaves.

Here is applet, but your browser does not support Java. If you want to see applets, please check browser requirements.

Question: *If you try to increase* ***mutation probability*** *to* ***100%,*** *GA will start to behave very strange, nearly like if the mutation probability is 0%. Do you know why? You can use a hint and if you still do not know, look at solution!*

VIII. Extreme of a Function

About the Problem

The problem is again looking for the extreme of a function. However in this case one more dimension is added.

Example

Graph represents a search space and lines represent solutions (points in the search space). The red line is the best solution, blue lines are the other ones.
*You can enter your own function in a text field below graph (after change press enter or button Change). Below it you can define limits of function. Function can consist of x, y, pi, e, (,), /, *, +, -, !, ^ and functions abs, acos, acosh, asin, asinh, atan, atanh, cos, cosh, ln, log, sin, sinh, sqr, sqrt, tan and tanh.*
The graph can be rotated by dragging it with a mouse.
You can also change crossover and mutation probability. By checkboxes you can specify whether you want to use elitism and whether you look for minimum or maximum.
Try to change the function and look how GA works. If you find some interesting function, where GA behaves very good or very strange, you can email it to me.

Here is applet, but your browser does not support Java. If you want to see applets, please check browser requirements.

IX. Selection

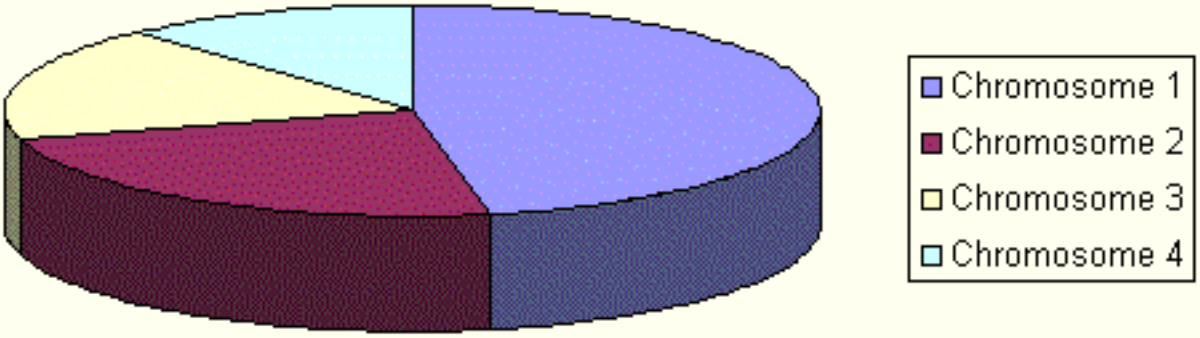
Introduction

As you already know from the GA outline, chromosomes are selected from the population to be parents for crossover. The problem is how to select these chromosomes. According to Darwin's theory of evolution the best ones survive to create new offspring. There are many methods in selecting the best chromosomes. Examples are roulette wheel selection, Boltzman selection, tournament selection, rank selection, steady state selection and some others.

Some of them will be described in this chapter.

Roulette Wheel Selection

Parents are selected according to their fitness. The better the chromosomes are, the more chances to be selected they have. Imagine a **roulette wheel** where all the chromosomes in the population are placed. The size of the section in the roulete wheel is proportional to the value of the fitness function of every chromosome - the bigger the value is, the larger the section is. See the following picture for an example.



A marble is thrown in the roulette wheel and the chromosome where it stops is selected. Clearly, the chromosomes with bigger fitness value will be selected more times.

This process can be described by the following algorithm.

1. **[Sum]** Calculate the sum of all chromosome fitnesses in population - sum ***S***.
2. **[Select]** Generate random number from the interval ***(0,S)*** - ***r***.
3. **[Loop]** Go through the population and sum the fitnesses from ***0*** - sum ***s***. When the sum ***s*** is greater then ***r***, stop and return the chromosome where you are.

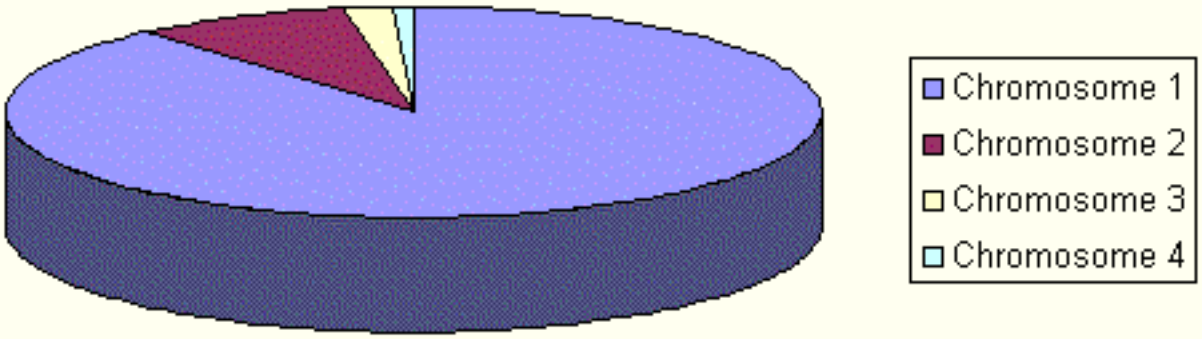
Of course, the step **1** is performed only once for each population.

Rank Selection

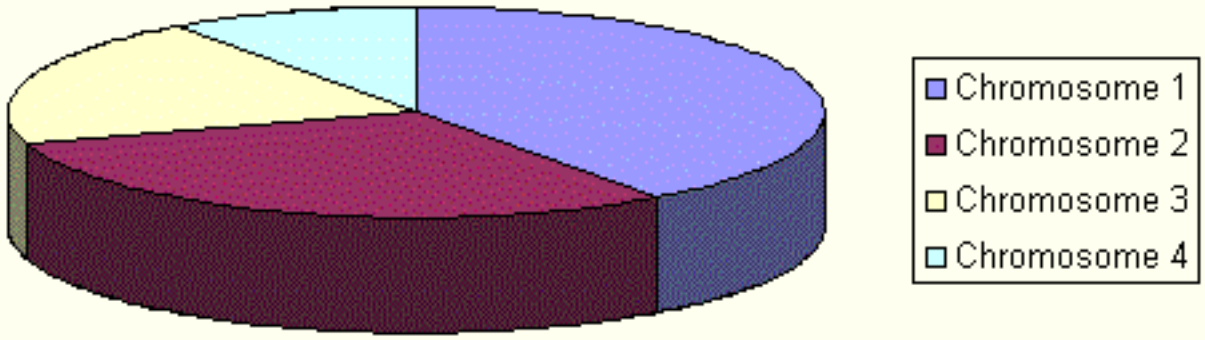
The previous type of selection will have problems when they are big differences between the fitness values. For example, if the best chromosome fitness is 90% of the sum of all fitnesses then the other chromosomes will have very few chances to be selected.

Rank selection ranks the population first and then every chromosome receives fitness value determined by this ranking. The worst will have the fitness ***1***, the second worst ***2*** etc. and the best will have fitness ***N*** (number of chromosomes in population).

You can see in following picture, how the situation changes after changing fitness to the numbers determined by the ranking.



Situation before ranking (graph of fitnesses)



Situation after ranking (graph of order numbers)

Now all the chromosomes have a chance to be selected. However this method can lead to slower convergence, because the best chromosomes do not differ so much from other ones.

Steady-State Selection

This is not a particular method of selecting parents. The main idea of this type of selecting to the new population is that a big part of chromosomes can survive to next generation.

The stady-state selection GA works in the following way. In every generation a few good (with higher fitness) chromosomes are selected for creating new offspring. Then some bad (with lower fitness) chromosomes are removed and the new offspring is placed in their place. The rest of population survives to new generation.

Elitism

The idea of the elitism has been already introduced. When creating a new population by crossover and mutation, we have a big chance, that we will loose the best chromosome.

Elitism is the name of the method that first copies the best chromosome (or few best chromosomes) to the new population. The rest of the population is constructed in ways described above. Elitism can

rapidly increase the performance of GA, because it prevents a loss of the best found solution.

X. Encoding

Introduction

Encoding of chromosomes is the first question to ask when starting to solve a problem with GA. Encoding depends on the problem heavily.

In this chapter some encodings will be introduced that have been already used with some success.

Binary Encoding

Binary encoding is the most common one, mainly because the first research of GA used this type of encoding and because of its relative simplicity.

In **binary encoding**, every chromosome is a string of **bits - 0** or **1**.

Chromosome A	1011001011001010111100101
Chromosome B	1111111000001100000111111

Example of chromosomes with binary encoding

Binary encoding gives many possible chromosomes even with a small number of alleles. On the other hand, this encoding is often not natural for many problems and sometimes corrections must be made after crossover and/or mutation.

Example of Problem: Knapsack problem
The problem: There are things with given value and size. The knapsack has given capacity. Select things to maximize the value of things in knapsack, but do not extend knapsack capacity.
Encoding: Each bit says, whether the corresponding thing is in knapsack.

Permutation Encoding

Permutation encoding can be used in ordering problems, such as travelling salesman problem or task ordering problem.

In **permutation encoding**, every chromosome is a string of numbers that represent a position in a **sequence**.

Chromosome A	1 5 3 2 6 4 7 9 8
Chromosome B	8 5 6 7 2 3 1 4 9

Example of chromosomes with permutation encoding

Permutation encoding is useful for ordering problems. For some types of crossover and mutation corrections must be made to leave the chromosome consistent (i.e. have real sequence in it) for some problems.

Example of Problem: Travelling salesman problem (TSP)
The problem: There are cities and given distances between them. Travelling salesman has to visit all of them, but he does not want to travel more than necessary. Find a sequence of cities with a minimal travelled distance.
Encoding: Chromosome describes the order of cities, in which the salesman will visit them.

Value Encoding

Direct value encoding can be used in problems where some more complicated values such as real numbers are used. Use of binary encoding for this type of problems would be difficult.

In the **value encoding**, every chromosome is a sequence of some values. Values can be anything connected to the problem, such as (real) numbers, chars or any objects.

Chromosome A	1.2324 5.3243 0.4556 2.3293 2.4545
Chromosome B	ABDJEIFJDHDIERJFDLDFLFEGT
Chromosome C	(back), (back), (right), (forward), (left)

Example of chromosomes with value encoding

Value encoding is a good choice for some special problems. However, for this encoding it is often necessary to develop some new crossover and mutation specific for the problem.

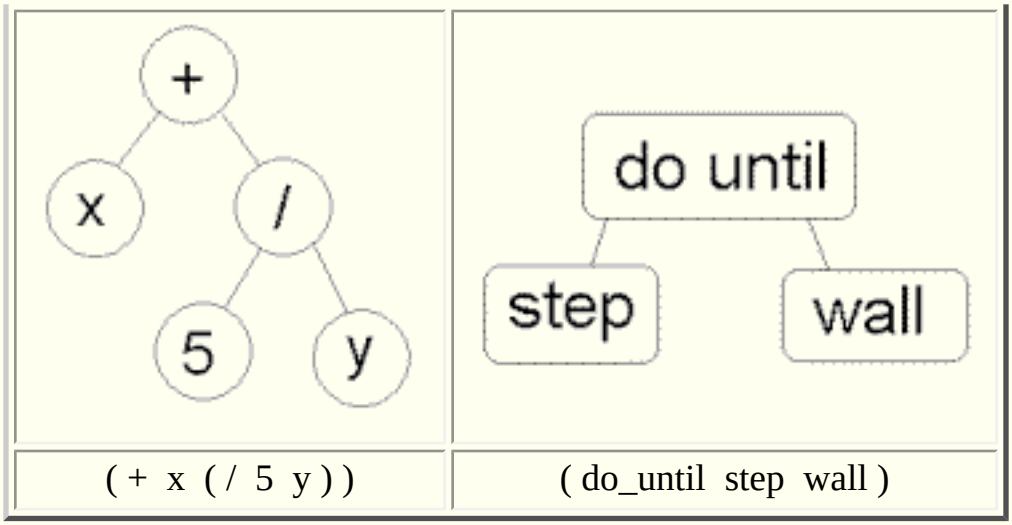
Example of Problem: Finding weights for a neural network
The problem: A neural network is given with defined architecture. Find weights between neurons in the neural network to get the desired output from the network.
Encoding: Real values in chromosomes represent weights in the neural network.

Tree Encoding

Tree encoding is used mainly for evolving programs or expressions, i.e. for **genetic programming**.

In the **tree encoding** every chromosome is a tree of some objects, such as functions or commands in programming language.

Chromosome A	Chromosome B



Example of chromosomes with tree encoding

Tree encoding is useful for evolving programs or any other structures that can be encoded in trees. Programing language LISP is often used for this purpose, since programs in LISP are represented directly in the form of tree and can be easily parsed as a tree, so the crossover and mutation can be done relatively easily.

Example of Problem: Finding a function that would approximate given pairs of values
The problem: Input and output values are given. The task is to find a function that will give the best outputs (i.e. the closest to the wanted ones) for all inputs.
Encoding: Chromosome are functions represented in a tree.

Previous Next

XI. Crossover and Mutation

Introduction

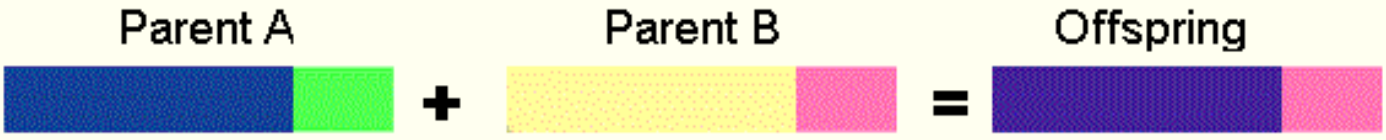
Crossover and mutation are two basic operators of GA. Performance of GA depends on them very much. The type and implementation of operators depends on the encoding and also on the problem.

There are many ways how to perform crossover and mutation. In this chapter we briefly describe some examples and suggestions how to perform them several encoding.

Binary Encoding

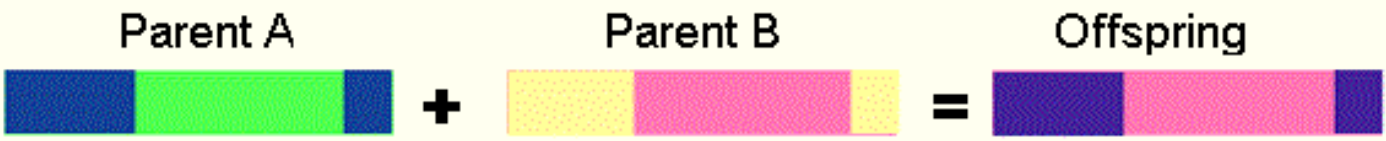
Crossover

Single point crossover - one crossover point is selected, binary string from the beginning of the chromosome to the crossover point is copied from the first parent, the rest is copied from the other parent



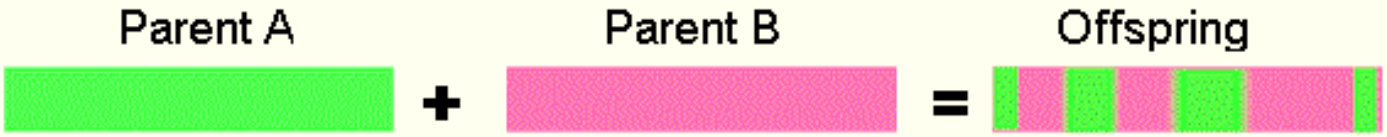
11001011+11011**111** = **11001111**

Two point crossover - two crossover points are selected, binary string from the beginning of the chromosome to the first crossover point is copied from the first parent, the part from the first to the second crossover point is copied from the other parent and the rest is copied from the first parent again



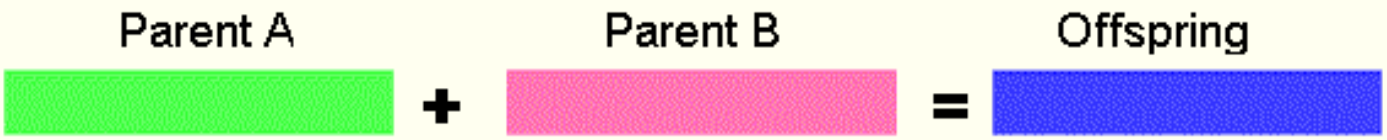
11001**011** + 11**011**11 = **11011111**

Uniform crossover - bits are randomly copied from the first or from the second parent



1**10**010**11** + **1**10**111**01 = 11011111

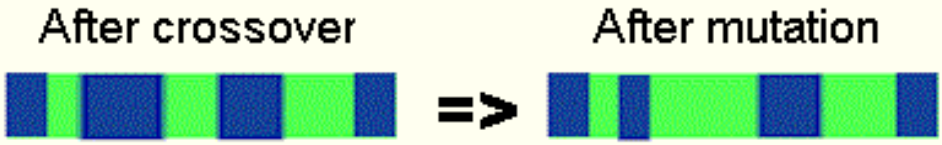
Arithmetic crossover - some arithmetic operation is performed to make a new offspring



11001011 + 11011111 = 11001001 (AND)

Mutation

Bit inversion - selected bits are inverted



1**1**001001 => 1**0**001001

Permutation Encoding

Crossover

Single point crossover - one crossover point is selected, the permutation is copied from the first parent till the crossover point, then the other parent is scanned and if the number is not yet in the offspring, it is added

Note: there are more ways how to produce the rest after crossover point

(**1 2 3 4 5** 6 7 8 9) + (**4 5 3 6 8** 9 7 2 1) = (**1 2 3 4 5** 6 8 9 7)

Mutation

Order changing - two numbers are selected and exchanged

(1 **2** 3 4 5 6 **8** 9 7) => (1 **8** 3 4 5 6 **2** 9 7)

Value Encoding

Crossover

All crossovers from **binary encoding** can be used

Mutation

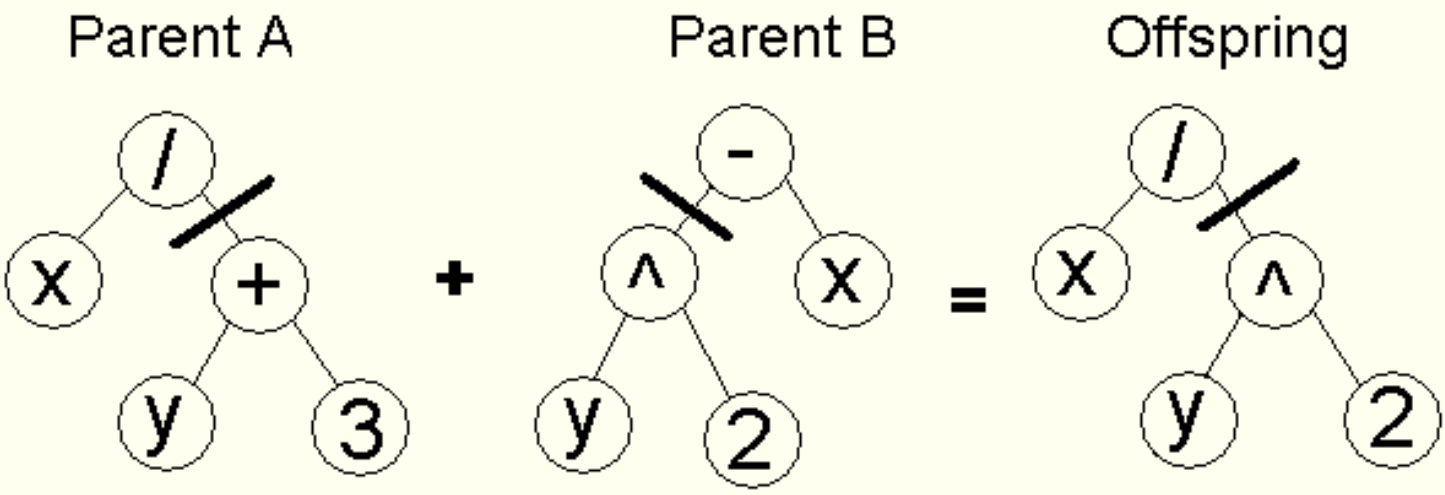
Adding a small number (for real value encoding) - a small number is added to (or subtracted from) selected values

(1.29 5.68 **2.86** **4.11** 5.55) => (1.29 5.68 **2.73** **4.22** 5.55)

Tree Encoding

Crossover

Tree crossover - one crossover point is selected in both parents, parents are divided in that point and the parts below crossover points are exchanged to produce new offspring



Mutation

Changing operator, number - selected nodes are changed

XII. Travelling Salesman Problem

About the Problem

Travelling salesman problem (TSP) has been already mentioned in one of the previous chapters. Just to remind, there are cities and given distances between them. Travelling salesman has to visit all of them, but he does not want to travel very much. The task is to find a sequence of cities to minimize travelled distance. In other words, find a minimal Hamiltonian tour in a complete graph of ***N*** nodes.

Implementation

Population of 16 chromosomes is used. For encoding these chromosomes permutation encoding is used - you can find in the chapter about encoding, how to encode permutation of cities for TSP. TSP is solved on complete graph (i.e. each node is connected to each other) with euclidian distances. Note that after adding and deleting city it is necessary to create new chromosomes and restart whole genetic algorithm.

You can select crossover and mutation type. The description of their meaning follows:

Crossover

- One point - part of the first parent (i.e. part of the cities order) is copied and the rest of cities is taken in the same order as in the second parent
- Two point - two parts of the first parent are copied and the rest between these two parts is taken in the same order as in the second parent
- None - no crossover, offspring is exact copy of parents

Mutation

- Normal random - a few cities are chosen and exchanged
- Random, only improving - a few cities are randomly chosen and exchanged only if they improve solution (increase fitness)
- Systematic, only improving - cities are systematically chosen and exchanged only if they improve solution (increase fitness)
- Random improving - the same as "random, only improving", but before that the "normal random"

mutation is performed
- Systematic improving - the same as "systematic, only improving", but before that the "normal random" mutation is performed
- None - no mutation

Example

The following applet shows GA optimizing the TSP. Button "Change View" changes view from whole population to the best solution and vice versa. You can add and remove cities by clicking on the graph. After adding or deleting cities a random tour will appear between them as new population with new random chromosomes is created. Also note that we are solving TSP on complete *graph.*
Try to run GA with different types of crossover and mutation and note how the performance (and speed - add more cities to see it) of GA changes.

***Known bug**: If you are using older versions of Java in Netscape, please press button "Change View" before doing anything else otherwise some graphs will not respond.*

Here is applet, but your browser does not support Java. If you want to see applets, please check browser requirements.

XIII. Recommendations

Parameters of GA

This chapter should give you some basic recommendations if you have decided to implement your genetic algorithm. These recommendations are very general. You will probably want to experiment with your own GA for a specific problem, because there is no general theory available that would help you to tune GA parameters for *any* problem.

Recommendations are often results of empiric studies of GAs that were often performed on binary encoding only.

- **Crossover rate**
  Crossover rate should be high generally, about **80%-95%**. (However some results show that for some problems crossover rate about 60% is the best.)
- **Mutation rate**
  On the other side, mutation rate should be very low. Best rates seems to be about **0.5%-1%**.
- **Population size**
  It may be surprising, that very big population size usually does not improve performance of GA (in the sense of speed of finding solution). Good population size is about **20-30**, however sometimes sizes 50-100 are reported as the best. Some research also shows, that the best population size depends on the **size of encoded string** (chromosomes). It means that if you have chromosomes with 32 bits, the population should be higher than for chromosomes with 16 bits.
- **Selection**
  Basic **roulette wheel selection** can be used, but sometimes rank selection can be better. Check the chapter about selection for advantages and disadvantages. There are also some more sophisticated methods that change parameters of selection during the run of GA. Basically, these behave similarly like simulated annealing. **Elitism** should be used for sure if you do not use other method for saving the best found solution. You can also try steady state selection.
- **Encoding**
  Encoding **depends on the problem** and also on the size of instance of the problem. Check the chapter about encoding for some suggestions or look to other resources.
- **Crossover and mutation type**
  Operators depend on the chosen encoding and on the problem. Check the chapter about operators for some suggestions. You can also check other sites.

Applications of GA

Genetic algorithms have been used for difficult problems (such as NP-hard problems), for machine learning and also for evolving simple programs. They have been also used for some art, for evolving pictures and music.

The advantage of GAs is in their parallelism. GA is travelling in a search space using more individuals (and with genotype rather than phenotype) so that they are less likely to get stuck in a local extreme like the other methods.

They are also easy to implement. Once you have the basic GA algorithm implemented, you have just to write a new chromosome (just one object) to solve another problem. With the same encoding you just change the fitness function - and you are done. However, for some problems, choosing and implementation of encoding and fitness function can be difficult.

The disadvantage of GAs is in the computational time. GAs can be slower than other methods. But sice we can terminate the computation in any time, the longer run is acceptable (especially with faster and faster computers).

To get an idea about some problems solved by GAs, here is a short list of some applications:

- Nonlinear dynamical systems - predicting, data analysis
- Designing neural networks, both architecture and weights
- Robot trajectory
- Evolving LISP programs (genetic programming)
- Strategy planning
- Finding shape of protein molecules
- TSP and sequence scheduling
- Functions for creating images

More information can be found through links in the appendix.

Appendix: Other Resources

Some selected links to web sites or ftps are listed here. You can use them for finding more information about genetic algorithms and concerning stuff.

ENCORE, the EvolutioNary COmputation REpository network
ftp://alife.santafe.edu/pub/USER-AREA/EC/ (there are also some others nodes)

FAQ - The Hitch-Hiker's Guide to Evolutionary Computation
ftp://alife.santafe.edu/pub/USER-AREA/EC/FAQ/www/index.html

FAQ - Genetic programming
http://www-dept.cs.ucl.ac.uk/research/genprog/gp2faq/gp2faq.html

The Genetic Algorithms Archive - many links, information about mailing list, some fun stuff
http://www.aic.nrl.navy.mil:80/galist/

Artificial Life Online - links, if you are looking for some introductory materials, look here
http://alife.santafe.edu/

Yahoo! Science:Computer Science:Algorithms:Genetic Algorithms - directory of other links
http://www.yahoo.com/Science/Computer_Science/Algorithms/Genetic_Algorithms/

Usenet groups comp.ai.genetic and comp.ai.alife

Note: All links were checked at the time of creating. If you find any broken link, please inform me.

Previous Next

Browser Requirements

For best viewing of these pages a browser with support of frames is recommended, as well as with support of JavaScript and Java 1.1 or higher (if you see errors instead of applets, your browser supports Java 1.0). Netscape Navigator from version 4.07 is recommended. You can also use Microsoft Internet Explorer 4.0 or higher, but the support of Java is strange in this browser (you may experience problems with redrawing and controlling applets).
However, if you do not need to see Java Applets, any older browser (even without frames support) can be used.

FAQ - Frequently Asked Questions

Questions

Answers

1. Question: I would like to cite these pages, is there any paper I can refer to?

Answer: If you want to cite a paper, use this reference:

> Obitko, Marek and Slavík, Pavel. Visualization of Genetic Algorithms in a Learning Environment. In: Spring Conference on Computer Graphics, SCCG'99. Bratislava : Comenius University, 1999, p. 101-106. ISBN 80-223-1357-2.

You may find the paper online here or here. Of course, you are free to link this site from anywhere you want and you may also refer to this site as to

> Marek Obitko: An introduction to genetic algorithms with Java applets, [URL of this site], 1998.

---

2. Question: Applets are not working, I see only errors. What shoud I do?

Answer: First look at browser recommendations.

You have probably browser with other version of Java than Java 1.1. Even some browsers, which have in description, that they support Java 1.1., really support Java 1.0 (without new event model). I suggest you upgrade your browser (sometimes is enough to change certain libraries).

Sometimes this error is caused by error in class transmition. In this case try reload the class or simply try it again after some time.

---

3. Question: I have a question concerning GA, can you help me with it?

Answer: I am not currently active in the GA field, so I probably can't help you with e.g. finding up to date articles concerning some special problem. Try to find answer to your question by e.g. Google and if it doesn't help, you may try to send me your question. I will try to answer within a few days. I would also recommend asking in comp.ai.genetic - there you have bigger chance that your question will be answered.

---

4. Question: Can this site be downloaded as a single file?

Answer: Yes, as a zipped pdf file. Get the Acrobat Reader to read it. But of course, you will see no applets in this version.

---

5. Question: Will you create a site like this about neural networks?

Answer: I will try to find a time to do that. Meantime you can look at a page with one applet illustrationg prediction by means of backpropagation neural network.

---

6. Question: Will there be any translation of this site to other languages?

Answer: There is a Japanese translation translated by Ishii Manabu and a Bulgarian translation translated by Teodor Gig. Other translations are welcomed - please contact me if you would like to make a translation.

---

4. Question: Is there any access statistics for this page?

Answer: Yes, some statistics is available here.

About These Pages

About

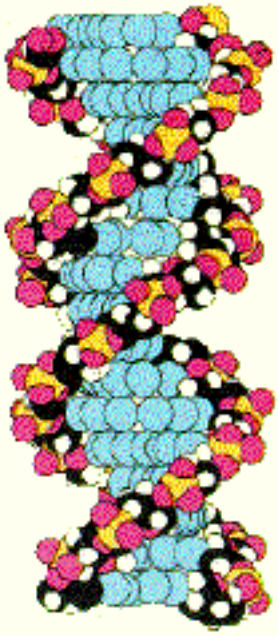
These pages were developed during August and September of 1998 at the Hochschule für Technik und Wirtschaft Dresden (FH) (University of Applied Sciences) by Marek Obitko, student of the Czech Technical University in Prague.

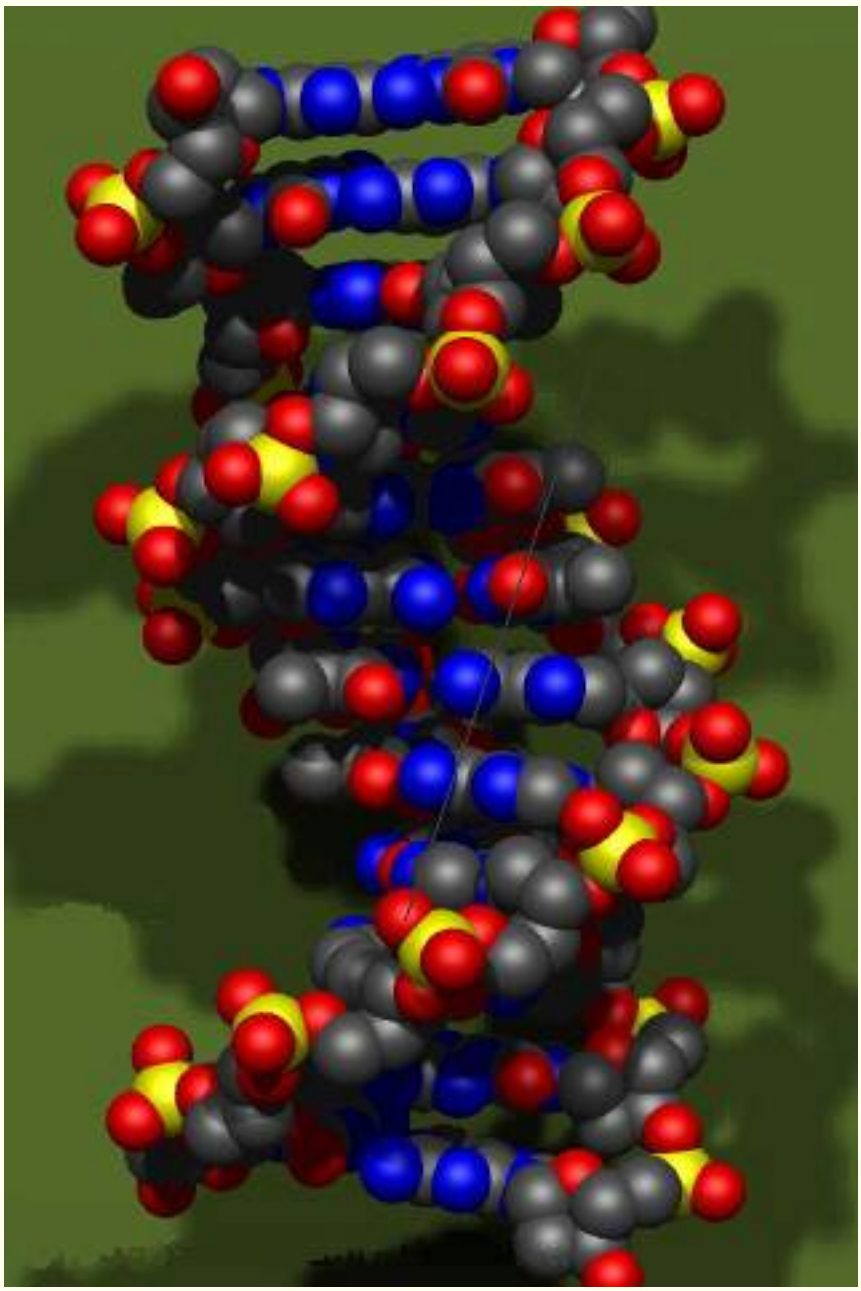
The first version of the applets was written during summer semester of 1998 at the Czech Technical University, supervised by Assoc. Professor Pavel Slavík. During the stay in Dresden the project was supervised by Professor Walter Pätzold from the Hochschule für Technik und Wirtschaft Dresden.

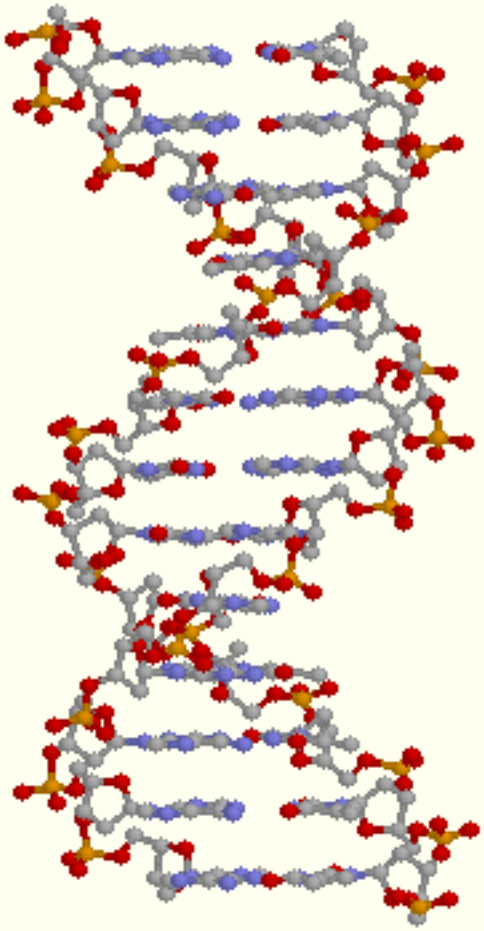
Pages and Java Applets were all created by Marek Obitko, (c) 1998. If you have any comments, questions or suggestions, you can send them to the author.

DNA
(Deoxyribonucleic acid)

Here you can see some pictures to get an idea how the DNA structure looks like. Some basic information about the biological background is also available.







Back
