Application of Evolutionary Algorithms to solve complex problems in Quantitative Genetics and Bioinformatics

27 to 29 January 2010

Armidale Animal Breeding Summer Course 2010
University of New England

by

Cedric Gondro and Brian Kinghorn

The Centre for Genetic Analysis and Applications
University of New England

Course schedule

Wednesday - 27/01	
09:00-09:10	**Introduction to the course (CG)**
09:10 - 09:30	**A brief overview of evolutionary algorithms (BK)**
	Problems not for EC
	Problems for EC
	Architecture for solving problems
09:30 - 10:30	**Heuristics and evolutionary computation (CG)**
	What are heuristics?
	Deterministic versus stochastic methods
	From random search to simulated annealing
	Evolution of evolutionary computation
	Types of EC and choosing a method
10:30 - 11:00	**Real life applications - demonstration of software in practice (BK)**
11:00 - 11:30	**Morning tea**
11:30 - 12:30	**EC framework (CG)**
	Evolution and population genetics 101
	Skeleton of an EA (canonical GA) Parameters: population size, mating schemes, selection, mutation, recombination
	Introduction to fitness and objective functions
12:30 - 13:30	**Lunch**
13:30 - 15:00	**Practical 1 (CG, BK)**
	A trivial GA example - play with different parameters
	Modify objective functions in code (maximize, minimize, etc)
15:00 - 15:30	**Afternoon tea**
15:30 - 17:00	**Differential Evolution (BK)**
	How does DE work?
	Parameter settings

Thursday - 28/01	
09:00 - 11:00	**Practical 2 (BK, CG)**
	In vivo demonstration of Differential Evolution
	Simple application of Differential Evolution
11:00 - 11:30	**Morning tea**
11:30 - 12:30	**Genetic Programming (CG)**
	Overview of GP
	Gene Expression Programming
	Selection and search operators
	Fitness and bloat
12:30 - 13:30	**Lunch**
13:30 - 14:15	**Problem representation (BK)**
	Moulding the solution space
14:15 - 15:00	**Managing constraints (BK)**
	The need for constraints
	How to apply constraints
	Hard and soft constraints
15:00 - 15:30	**Afternoon Tea**
15:30 - 17:00	**Practical 3 (BK, CG)**
	Discuss and start work on group project topic

Friday - 29/01	
09:00 - 10:00	**Changing the goal posts (BK)**
	Changing on the fly
	Managing change of direction
	Opportunities
10:00 - 10:45	**Improving performance (CG)**
	Self-evolving parameters
	Multicriteria optimization
	Hybrid algorithms
	Parallelization
10:45 - 11:15	**Morning tea**
11:15 - 12:15	**Diagnosing convergence (BK)**
	Criteria for stopping
	Maximum solutions
12:15 - 13:15	**Lunch**
13:15 - 14:30	**Applications in bioinformatics, systems biology and Alife (CG)**
	multiple sequence alignment
	optimization of microarray experimental designs
	model discovery and parameterization
	Alife virtual agents
14:30 - 15:00	**Afternoon Tea**
15:00 - 16:45	**Practical 3 continued (CG, BK)**
	Continue work on group project topic
16:45 - 17:00	**Course wrap up**

Lecturers

Cedric Gondro is a Lecturer at the University of New England in Australia.

Cedric's research interests include:

- Evolutionary Computation and Artificial Life for optimization of biological problems
- Computational methods and statistical analysis of high-throughput genomic data
- Systems biology
- Bioinformatics, evolution and population genetics

Brian Kinghorn is a Professor at the University of New England in Australia.

Brian's research interests include:

Theoretical quantitative genetics and design of animal breeding programs including

- Integration of different technical, logistical and cost issues into a unifying dynamic decision framework.
- Information systems to aid detection of genes of major effect and to exploit molecular techniques.
- Exploiting novel reproductive techniques.
- Computer simulation of animal breeding programs

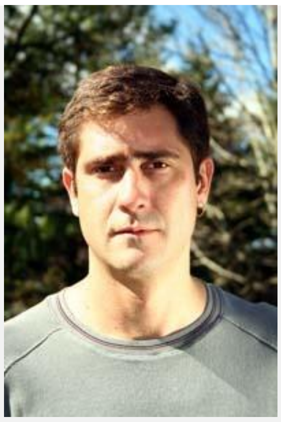
Industry application of breeding programs
And, of course Evolutionary Computation

Brian Kinghorn

The Centre for Genetic Analysis and Applications

University of New England
Homestead 32
Armidale, NSW
2351 Australia

bkinghorn@une.edu.au
+61 2 6773 2718



Cedric Gondro

The Centre for Genetic Analysis and Applications

University of New England
Homestead 35
Armidale, NSW
2351 Australia

cgondro2@une.edu.au
+61 2 6773 3978

And most importantly, both are aficionado motorcyclists!

Table of contents

Chapter 1: A brief overview of evolutionary algorithms

Brian Kinghorn

Seek, and you shall find

Pontifications

This course aims to empower its participants by sharing some knowledge and skills that have greatly benefitted the presenters.

The development of evolutionary algorithms is not a hard science – there are tips and tricks, and opportunities for tweaking. There is an art to developing and using these algorithms, and much of the content of this course relates to personal experience and discoveries made in applications related to quantitative genetics and bioinformatics. However applicability is very wide indeed.

The process of evolution has brought about great complexity in a simple manner. Biology is mind-bogglingly complex, even to our shallow level of understanding. How could all of this have happened? How could massive sequence and other components be derived to result in life as we see it? Working with evolutionary algorithms to solve complex problems gives some intriguing insight – it helps to give realization that indeed complex outcomes can come from a relatively simple process.

As practitioners setting out to solve complex problems, we are not constrained to the systems of replication and propagation that we see in biology. We can engineer our own systems to give faster and more robust evolutionary change towards solutions for the prevailing problem.

We can target our outcomes in a more controlled yet flexible manner than in biological evolution. We can explore the changes that could be made, and use this information to help set target outcomes for our prevailing problems – we can make a steering wheel.

Problems *not* for evolutionary algorithms

Many problems in quantitative genetics and bioinformatics can be solved by an appropriate evolutionary algorithm. We could *find* the mean, rather than *calculating* it. We could solve BLUP by *finding* (rather than numerically *calculating*) the set of solutions that minimizes least squares while satisfying constraints on variance. But these would not be the best uses of evolutionary algorithms.

It is usually the case that if a solution can be calculated using a closed-form equation, or derived numerically through integration or some form of iteration, then that calculation or derivation will be the method of choice, because of speed if not precision. Exceptions can occur when the method makes assumptions that may be unreasonable.

Moreover, problems that can be solved by iterative sampling methods, such as Gibbs Sampling, might be hard to beat with an Evolutionary Algorithm.

Exceptions to all of the above might occur when the calculation method is complex, or otherwise difficult to code into a program for analysis. As will be seen, with some EA code in your toolbox there can be surprisingly little work required to solve, for example, a multiple regression.

If you don't know whether there is a method to calculate an answer to your problem, then you can be lazy, and resort directly to using an evolutionary algorithm.

Problems for evolutionary algorithms

Evolutionary algorithms are useful for problems for which no mechanistic method is available – or when unreasonable assumptions need to be made. Problems for which you cannot *calculate* or *derive* a solution. Problems for which you have to *find* a solution.

[Could <u>we</u> *calculate* the nucleotide sequence required to make a given organism? Much easier to *find* it ... for an organism with the same properties.]

Aspects of problems that lend them to EA solution include:

> Assignment of individuals into groups, wherever simple ranking and truncation will not work. This usually involves interactions, whereby whether an individual should be in a group depends on what other individuals will be in that group. Example: developing multiplex groupings for genotyping.
>
> Problems where thresholds are involved, for example when the value of a solution depends on whether certain thresholds have been passed, or the fate of an individual depends on passing one or more thresholds. Example: Supply chain optimizing to target multiple product end-points and/or turnoff dates.
>
> Combinatorially tedious problems, where many combinations of components can exist. Example: The setting up of animal matings.

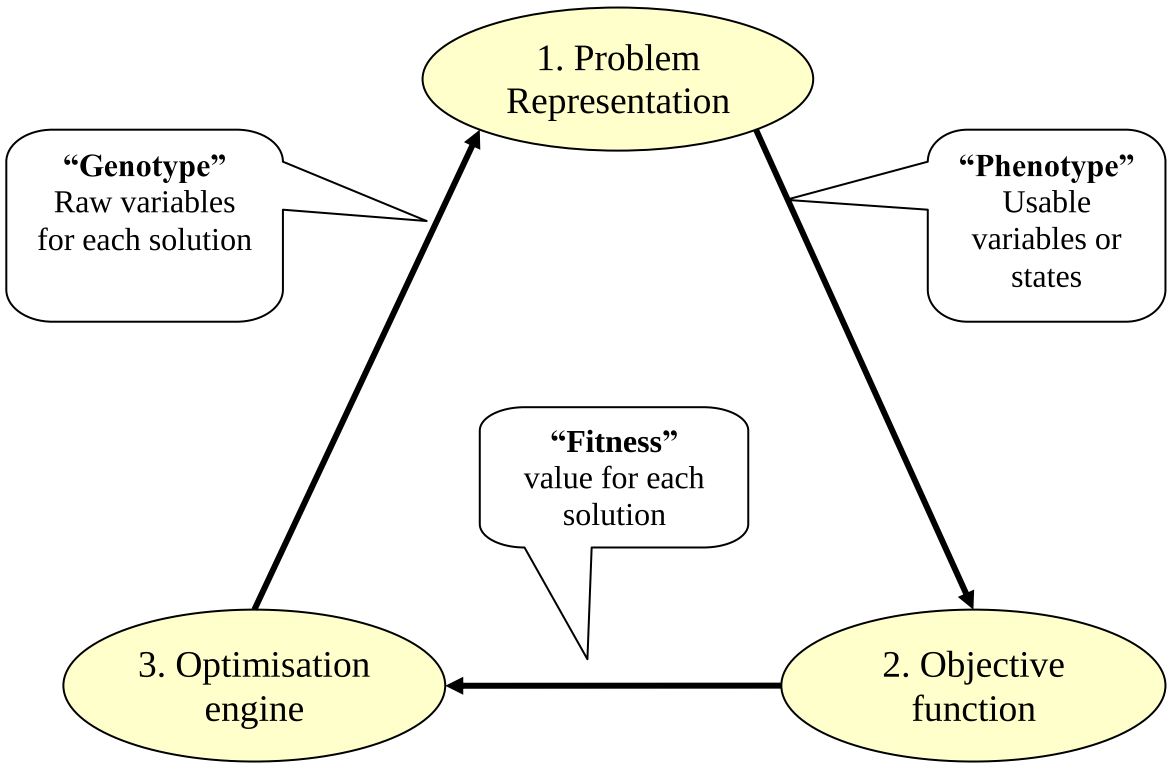
The architecture of an evolutionary algorithm, in a nutshell.

1. <u>Problem representation:</u> If needed, write an algorithm to produce the input variables/states ("Phenotypes") from a vector of simple numbers ("Genotype"). An example is given by Kinghorn and Shepherd (1999) who convert such a vector into a

pattern of mating and selection (see Chapter 6). This algorithm should ideally produce only legal solutions to the problem.

2. Objective function: This should be able to return a single value ("Fitness") that represents the value of a single solution. The single solution is represented by variable input values (eg. selection index weights) and/or states (eg. a vector of animals that should be selected).

3. Optimization engine: This is the heart of an Evolutionary Algorithm – it is where operations such as recombination and mutation are carried out, to make genotypes of progeny out of the genotypes of parents. However, the optimization engine is quite simple 'on the outside'. It generates vectors of numbers ("Genotypes") and seeks the vector that gives the highest fitness. The next four chapters will introduce you to these methods and related approaches.


1. Problem Representation
"Genotype" Raw variables for each solution
"Phenotype" Usable variables or states
"Fitness" value for each solution
3. Optimisation engine
2. Objective function

Aim of the course

We hope that you leave this course with the knowledge and skills needed to apply an evolutionary algorithm to any suitable problem that confronts you. There is now a small band of animal breeding scientists who use these tools on a regular basis, and we hope you will join us.

Postscript: Let your computer make you famous.

The author was writing a program to optimize development of a new composite breed. But it was not going well. Some bugs were causing breed proportions to jump all over the place – not settling down nicely to optimal proportions. After some detective and debugging work there was a sudden understanding – the program was thinking outside the box and doing the best thing possible – a periodic rotational cross.

Just think of that – your optimization program finds a way of doing things – perhaps a concept that has not been thought of before. You get the Nobel Prize while the poor computer is being carted off to the dump.

Chapter 2: Heuristics and evolutionary computation

Cedric Gondro

Don't think just guess - but be smart about it

Introduction

At the end of the day, this course is about solving problems. Most people are interested in the solution and don’t much care about how to get to it, even though much time and energy can be spent trying to solve the problem.

In the previous chapter we saw problems that are suitable for evolutionary computation (EC) and problems that are not. Part of the art of problem solving is identifying the best way to do it: most accurate, most repeatable, fastest and most computationally efficient and, why not, easiest. Before you start coding an evolutionary algorithm (EA) make sure this is the best approach. Maybe the most efficient way of solving your problem is just to knock on the door of the next office!

But some problems are really hard to solve. Not infrequently because we do not understand what we are really trying to solve. A very nice side effect of EAs is that they can help us understand the nature of the problem we are tackling – more about this in the end of the chapter. Back to the subject. Problems can be complex because the number of potential solutions is astronomical – it is impossible to evaluate all possible solutions and pick the best one. The relationships, the interactions, the structure of the problem is so convoluted that a model of the problem (remember that a model is a simplification of reality) does not reflect the properties of the problem, thus is useless. The solution changes over time or in different scenarios (GxE anyone?). Too many constraints in the system. For example, female matings in a breeding program; or if you think back to your college days, the juggling between the subjects you wanted to take and the days/times of the week they were offered. Add in the prereqs and all the times that you did not want have classes (it was college after all!) and you have a heavily constrained problem. For these types of problems, chances are that an EA is a good alternative.

[An aside: Evolutionary Computation is the field of study and an Evolutionary Algorithm is a method. As an analogy, think of Animal Breeding and BLUP).

Evolutionary algorithms are heuristic methods. Hmm, what are heuristics?

Evolutionary algorithms are heuristics. The term heuristic comes from the Greek word *heuriskein* which means to discover. It is associated with the process of gaining knowledge or some desired result by intelligent guesswork rather than by following some pre established formula. Think of it as a trial-by-error approach to learning. It essentially is an approach to learning by trying without necessarily having an organized hypothesis or way of proving that the results proved or disproved the hypothesis. The later is probably the key message here: with EAs you do not know if your solution is the best possible one, and even if it is, you cannot prove it (except for trivial examples).

Deterministic versus stochastic methods

Heuristic methods can be based on deterministic or stochastic methods. Deterministic methods behave predictably, which basically means that you will always get the same output for a particular input. A mathematical function is a typical example: for any given input the function will yield the same output. If this is translated to a computer you have a deterministic algorithm, which is just a description (the sequence of steps/states) of how the above mentioned function will obtain the output. Deterministic algorithms are ideal to solve problems, they provide a cookbook view to problem solving (who does not have a copy of Numerical Recipes?), all you have to do is go to the kitchen and pick the subroutine. But functions are not heuristics. In heuristic terms think of an exhaustive search starting at zero and testing all possible integers up to 10000, one at a time- it's deterministic in the sense that you can predict the states of the algorithm and the output will always be the same.

The downside of a deterministic approach is that frequently there is no algorithm for a specific problem or it is computationally unfeasible – increase the limit of the solution space in the previous example to all positive integers (that will surely give you some time to fetch a cup of coffee!). This later class of problems is referred to as NP-complete. NP stands for non-deterministic polynomial time and it just means that as the problem increases in size the resources needed to solve the problem grow very rapidly. This term is commonly associated with EAs, but all it boils down to is that the problem has a large potential solution space.

On the other hand, stochastic methods use algorithms for which, given a certain input state, it is not possible to determine the output state. This just means that the results are not predictable. EAs are stochastic algorithms and that is why the solution cannot be proved right or wrong. And this is of course the main negative aspect of stochastic methods: there is always some degree of uncertainty (distrust if you prefer) in the results, a lingering feeling that maybe it could just do a bit better. Another problem with stochastic methods is that in general they are slower than their deterministic counterpart. And last, the lack of repeatability of results – many runs should be performed to get a better estimate of the range of results and their repeatability. Narrow spread of results and repeatability are two key features that should be considered when selecting an EA.

Types of optimization strategies – how to solve a problem

If asking the guy in the office next door does not work, then you might actually have to try to solve the problem yourself. Most of the problems we encounter can be represented as an optimization problem.

There are different classes of optimization problems: assignment, model parameterization, model discovery, combinatorial and minimax, just to name a few. Selection of the best method for a certain optimization problem is not trivial. The no-free-lunch (NFL) theorem (Wolpert and Macready 1996) essentially states that there is no optimal method for solving all types of optimization problems. A method adequate for a certain class of problems may breakdown with a different problem. A wide range of global optimum finding methods such as dynamic programming and linear programming are available for specific types of optimization problems, these are traditional deterministic methods that are guaranteed to converge on the global optimum. Traditional methods can be split into two main classes, the algorithms that evaluate complete solutions and the algorithms that evaluate partial solutions (Michalewicz and Fogel 2000). In the first class are exhaustive search, gradient methods and linear programming. These can be applied to specific domains; for smooth differentiable problems a gradient approach is suitable, linear programming is well tailored for a problem of linear variables. As mentioned above, for a small confined search space exhaustive search is adequate.

The second class of methods evaluates partial solutions and builds on them; these include greedy algorithms, dynamic programming and branch-and-bound algorithms. As an example dynamic programming is efficient in pair-wise alignment of amino acid or nucleotide sequences.

Regrettably most of the above algorithms are confined to particular classes of problems which frequently cannot be applied to real-world problems. Real-world optimization problems tend to exist in a non-linear, discontinuous and complex landscape. Further, the systems can be dynamic with objectives shifting over time thus demanding that the optimization method be capable of modifying its strategy in situ. Other common difficulties are the presence of noise in the available information and lack of knowledge about the nature of the problem itself (Michalewicz and Fogel 2000).

These difficulties can hinder the adoption of global optimum finding algorithms, even though they have been successfully adopted to solve a wide range of specific problems. EC algorithms offer a compromise solution. Whilst not guaranteed to converge on optimal solutions they can be used on a wider range of problems albeit at the sacrifice of efficiency. This loss of efficiency is particularly evident in simple scenarios for which specifically tailored optimum seeking algorithms are available. Thus, classes of problems for which specific algorithms are available should not be solved through ECs for they will not be able to solve these problems faster or with higher precision (Schwefel 2000). A common approach to solving a complex problem is to linearize it or make use of some other simplification method that will allow the problem to be solved by an available optimum seeking method. However these simplifications can yield results that are further away from the true answer than an approximate result obtained through an EC approach. Even with quite incorrect initial conditions, an EC approach can still produce reasonable results (Schwefel 2000). EC methods are not guaranteed to converge on an optimal solution. But worse, it is also not possible to evaluate how close to optimal a result is. Attempts to develop a formal analytical framework for EC are still incipient, mainly due to the fact that EC methods present the same analytical difficulties of the problems they are used to solve – complex, noisy, dynamic and non-linear (Forrest 1993). Probably the greatest limitation to the use of EC methods is the dimensionality problem. As the number of variables increases the computational effort can increase exponentially. But, since by their very nature EC methods are well suited for parallelization and there is a growing interest in developing parallel algorithms, the importance of this problem is becoming less significant.

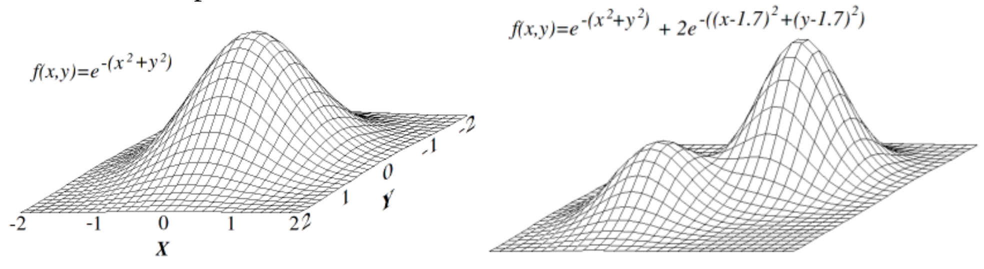
Non EC stochastic heuristic methods

Before delving into EC it is worth briefly mentioning some related methods. There are many others (tabu search, ant colony optimization, swarm optimization, etc), I have selected the simplest ones because they build naturally into the EC framework.

Random Search – this is probably the simplest implementation of a stochastic algorithm. It is, as the name implies, simply a random search of the solution space. If exhaustive search is unfeasible, one could try to sample a subset of the solution space at different points and try/hope to obtain a reasonable answer. This is of course evidently suboptimal and various immediate improvements can be considered. For example, do not select the same sample twice, or even better, try to divide the solution space into regions and sample from each of these regions to obtain a better coverage.

Random Walk – can be seen as an extension of random search. Instead of selecting a single point in the solution space and then moving on to another independent point, one could select a point and then randomly move away from it while the solution is improving (obviously questions such as step size and in which dimension to move creep to mind).

Hill Climbing – with random walk the most obvious problem is that it is very easy to overshoot the solution or choose a bad path. This can be balanced out with hill climbing in which from a starting point the algorithm will test the nearest neighbors (not necessarily all if the dimensionality is too high) and move in the best direction until it cannot find any neighboring solution better than the current one. Of course overshooting is a concern because the concept of *neighbor* depends on the step size. The first improvement that can be considered is to reduce the step size and start again from the previous best point and repeat this process until the step size is zero and no further improvements are attainable. Of course this works fine for smooth surfaces with a single peak, as soon as the adaptive landscape becomes more complex, hill climbing can get stuck at a local optimum. In the figure below it is easy to see that in the first scenario hill climbing would be efficient, but this is not the case for scenario 2. A work around local optima entrapment in hill climbing is to use stochastic hill climbing. Implementations vary widely, but consider starting many climbs from different regions of the solution space or from time to time add a bit of random walking to try to break out of the entrapment.



Simulated Annealing (SA) – an adaptation of the Metropolis-Hastings algorithm, it is probably the most widely used optimization heuristic. SA uses an analogy to metallurgy which involves heating and controlled cooling of a metal. An easy example is shown by Palshikar (2001). The basic concept is to start with a random solution and replace it by a randomly chosen neighbor with a given probability. This probability depends on the difference in fitness of each of these solutions (more on fitness in the next chapter) and a *temperature* parameter. Initially the temperature is set very high (the probability of selecting the alternative solution is very high, or even one) and gradually decreased so that the probability of choosing the alternative solution becomes very small. This means that initially only solutions that are very significantly better than the current solution are accepted and as the temperature decreases the algorithm tends to behave more and more as a hill climbing algorithm. But let's get started in EC…

Evolution of evolutionary computation

Computational intelligence is a field of computer science that uses biology as a source of inspiration for solving real world problems or mimicking biology in silico. Evolutionary Computation, alongside Artificial Life, Swarm Optimization, Fuzzy Systems and Artificial Neural Networks is a subfield of Computational Intelligence and the umbrella under which reside closely related stochastic methods of simulating evolution. The main branches of EC are Evolutionary Programming (EP), Evolution Strategies (ES), Genetic Algorithms (GA), Genetic Programming (GP) and Learning Classifier Systems (LCS). EC is a young field; the term was coined in 1991 in an attempt to unite the different branches. The origins of using computers as a means to emulate and understand evolution dates back to the late 1950s and early 60s through the pioneering work of Bremermann, Fraser and Friedberg. The different EC branches evolved quite independently from each other, despite having much in common. The current trend is to merge the best aspects of the different branches and develop solid theoretical foundations for the entire field (De Jong *et al*. 2000).

De Jong (2006) poignantly split the history of EC into the Catalytic 60s – when computers started to become available, the Explorative 70s – development of what was to become the various branches of EC, the Eplorative 80s – practical application of the methods, the Unifying 90s – pragmatism over ideology, the best aspects of each method are integrated into unified algorithms. Currently EC is not only a rich community in its own right but it is also widely adopted across the most diverse disciplines.

Types of Evolutionary Algorithms and problem matching

Evolutionary Computation is about solving problems. An optimal solution implies absolute knowledge of the problem, but frequently knowledge of the problem's domain is incomplete and sparse, the solution space is unknown and data sources are noisy. In other types of problems optimality may vary over time or there are multiple optimal solutions. And, sometimes problems are heavily constrained and simply finding a single set of parameters that comply with the bounds is unwieldy. Biological problems seem to fall under all these categories, they have large search spaces, are non-linear, complex and open-ended, available data is noisy and incomplete and, they are interlinked and constrained by the components of the system. For such complex problems a compromise must be sought. Instead of deterministic algorithms which yield optimal solutions but can only be used in specific and frequently unrealistic problems, stochastic algorithms that are not mathematically guaranteed to converge on a optimal solution can be adopted (Michalewicz and Fogel 2000). Evolutionary Computation falls under this last category. EAs are a group of stochastic problem solving methods loosely inspired on evolutionary processes such as selection, mutation and crossover. All algorithms have in common the use of populations of candidate solutions which reproduce, compete, and are subjected to selective pressures and random variation – the four basic elements of evolution (Atmar 1994).

Below we briefly describe the main branches of EC. The purpose is neither to provide a comprehensive overview nor to suggest that these are the correct methods for a given EA; the sole intent is to give a taste for the different variants. Keep in mind that EC is evolving rapidly, the binning below is simply for didactic purposes, at the end of the day you should pick the aspects of each method that seem best suited to your needs and use them. Remember: *adapt the method to the problem and not the problem to the method*.

Evolutionary Programming (EP)

EP in its basic form consists of generating an initial population μ and a fitness value is assigned to each individual. The iterative loop (each loop is commonly referred to as a generation) usually consists of duplicating each parent μi until a predefined number λi of offspring are generated. The offspring are modified through a mutation process – commonly a Gaussian distribution with zero mean and variance of one, crossover is not used in classic EP. All offspring are evaluated as to their fitness and along with the parental population a selection operator is used to cull the population size back to μ. EP shares more than a few similarities with ES. The main difference between EP and other EC methods is the global optimization method employed by EP. No attempt is made to break the problem down into subcomponents; the fitness evaluation is based solely on the whole organism. In this sense the genotype is of little importance and focus is on optimization of the phenotype, for this reason crossover is not used in EP. A further distinction is that traditionally EP uses continuous-valued variables instead of the discrete representation common in GA. Current versions of EP are self-adaptive, with the mutation parameters (variance, covariance) adapting to the current state of the population. A good overview of EP is given by Bäck (1996), Fogel (1999) and Porto (2000).

Evolution Strategies (ES)

ES were initially developed to solve technical optimization problems. There are two main general notations for the strategy: (μ + λ) where the ES generates λ offspring from a parental population μ and selects the best μ from all μ+λ individuals. Alternatively the (μ , λ) strategy generates λ offspring from μ parents and selects the μ best from the λ offspring. Weak selective pressures seem to yield a better response thus the μ/λ ratio should not be too small. Of course, a 1:1 mapping of μ:λ reduces the algorithm to a random walk. Typically ES use crossover between two randomly selected parents to generate the offspring; commonly adopted is the multipoint crossover. After crossover the offspring are mutated. ES mutation schemes are not particularly straightforward; each organism, besides the element that maps their position in the search space, can have several parameters controlling the mutation distribution which customarily follows a multivariate normal distribution with zero mean and a covariance matrix that is symmetric and positive definite. At least two mutation parameters are commonly used: angles (σ) and standard deviations (ω). These mutation parameters can be self-adaptive as in EP algorithms. The original ES strategy was (μ + 1) with a single replacement per iteration loop which is called steady-state (an analogy to overlapping generations) in opposition to the generational approach. Even though the steady-state approach is the preferred choice for other EC methods, modern ES adopt a generational approach similar to EP.

As with Evolutionary Programming, ES does not attempt to break down the problem into smaller subcomponents. Optimization is solely based on the phenotypic values of the organism. Schwefel and Rudolph (1995) and Rudolph (2000), provide an overview of ES.

Genetic Algorithms (GAs)

The most widely disseminated EC branch, GAs date back to the seminal work of Holland (1975). GAs distinguish themselves from the other methods by the emphasis that is placed on crossover. Traditionally GA organisms are represented as linear bitstrings which are referred to as chromosomes; this is the canonical GA (Holland 1975; Goldberg 1987). The value in each position of the bitstring is an allele (0 or 1) and the position itself is a gene or locus. The combination of values (alleles) in the bitstring (chromosome) maps to a phenotypic expression, such as a parameter to be optimized. From the above it is clear that GAs operate at two structural levels: a genotypic and a phenotypic one. Selection operators are carried out based on the overall chromosome value (phenotype) while search operators act on the genotype, modifying the chromosome which may or may not change the phenotypic expression.

GAs are the class of EC which most closely mimic evolutionary processes at a genetic level. Crossover swaps chromosome parts between parents to form the offspring and mutation changes the value of alleles at randomly selected loci. From this notion derives the concept of schema in GAs (Holland 1975); a good solution consists of a set of good small building blocks. Thus, the assumption is that the chromosomes in the population are formed by small schemas that add up to yield the final fitness. Under the schema model, crossovers between good schemas are preserved since they increase the overall fitness of the chromosome while crossovers which break up a good schema are eliminated since they reduce the fitness. This assumes that the GA will concentrate on searches which are optimally allocated. The schema theory has been under attack recently, with many arguments for and against but still limited solid proofs (Whitley 2001; Langdon and Poli 2002). To the author's knowledge no work has attempted to describe schema theory within the framework of quantitative genetics; such a study might evidence the appearance and maintenance of haplotypes and linkage disequilibrium studies could provide a more in-depth understanding of the dynamics of GAs.

Crossover is often regarded as the main search operator of a GA, with mutation seen more as a mechanism of ensuring a robust gene pool to be explored by crossover (Eshelman 2000).

Genetic Programming (GP)

Often regarded as a specialization of Genetic Algorithms, GP has evolved to become a branch of EC in its own right. Initially GP was devised as a method to optimize data structures as executable computer programs with the fitness value assigned based on the results obtained when executing the instructions contained in each member of the population. In this context, GP evolves populations of computer programs or other algorithmic processes to solve a specific problem (Banzhaf et al. 1998; Koza 1989; Koza 1992; Koza 1994; Koza et al 1999; Koza et al 2003).

We will be discussing GP in details in chapter 5 and particularly Gene Expression Programming which is the author's favorite flavor of GP.

Learning Classifier Systems (LCS)

Good references to LCS are Wilson (1994), Smith (2000) and Vlachos et al. (2004) for an application of LCS in biology. LCS are more of a concept than an actual algorithm; they can be seen as a hybrid between a machine learning technique and a GA. LCS is a rule-based system which uses agents that receive an external signal (message), process it and produce a response. The genetic material consists of a set of rules which map an input (detector) to an output (effector), modifying the behaviour of the agents to environmental changes. The GA component of the LCS is to adapt the rules to obtain a desired behaviour from the agents.

Differential Evolution (DE)

DE (Storn and Price 1997) is not a branch of EC in its own right, but it has been widely adopted in our group to solve quantitative genetics problems. DE can be seen as a parallelized simulated annealing algorithm that lies on the intersect between real-valued genetic algorithms and evolution strategies. In chapter 4 DE will be discussed in-depth.

Which one do I use?

The NFL theorem applies to EAs. No single approach is always superior for all problems or can solve any type of problem. The choice of an appropriate EA depends on the nature of the problem at hand. There have been advances in developing a formal framework for EC but largely the field is still anchored on a trial-and-error approach. There are no widely applicable rules for selection of population parameters apart from the collective empirical experience of practitioners. The current drive is to use self-evolving parameters and let the system figure out what is best (more in chapter 9).

On the bright side, the methods are robust and even suboptimal parameter selection can still lead to good results. As a rule of thumb, GAs are well suited for discrete problems such as sorting, ranking or allocation problems; EP and ES are a good first choice for continuous problems such as model parameterization; GP allows tackling problems such as model discovery. Within each EC branch there is vast number of different algorithms. Selecting the best one for a given task can be quite daunting. From a practical standpoint considerations of ease of implementation, computational and convergence speeds and repeatability of results are important.

Concluding remarks

In a nutshell, Evolutionary Computation uses computer algorithms which search through complex solution landscapes. A population of candidate solutions is created, a fitness value is assigned to each member of the population and depending on their fitness value an organism has a higher or lower probability of being selected to remain in the population and generating offspring. New members are created through crossover and mutation thus exploring the solution space.

We have seen when to use and when not to use EAs, but before we get our hands dirty and start coding EAs there's still one last question: why use Evolutionary Algorithms? Basically, it is just because they present various characteristics that are advantageous for solving optimization problems. Michalewicz and Fogel (2000) summarized these advantages as:

1. Simplicity. The concept and the implementation of EAs are simple.
2. Broad applicability. Virtually any problem can be addressed by EAs.
3. Hybrid methods. EAs allow integration with other methods.
4. Parallelism. The structure of EAs makes them particularly well suited for parallelization.
5. Robust to changes. Changes in the target system do not render the algorithm useless.
6. Self adaptation. The parameters of the EA can evolve alongside the solutions.
7. Solve problems with no known answers. Probably the greatest advantage of EAs; if an evaluation of goodness of fit of a solution is possible, EAs can be used.

Nevertheless EAs are not a magic bullet. You will have to think about how to present the problem to the algorithm, in chapter 6 problem representation will be discussed. Even more fundamentally you will have to understand what the problem really is. A nice side effect of EAs is that they show you things that you did not know – as mentioned in the previous chapter, they can show solutions which had not previously been considered or show problems in the model which you were not aware of – at the end of the day you and the computer will have learnt something about the problem!

Chapter 3: From *in vivo* to *in silico*

Cedric Gondro

It's been good for 3 billion years - it's good enough for me

Evolution and population genetics 101

Evolution can be seen as a dynamic and opportunistic optimization process. Effectively it is a method to search through a vast solution space and find a solution that allows organisms to survive and reproduce in a certain environment. It is dynamic in the sense that solutions (organisms) can change to adapt to environmental changes and it is opportunistic in the sense that solutions are not necessarily globally optimal but rather tend to move to the next available solution that ensures viability, even if in detriment of a more globally optimal solution. Interestingly enough, the high-level rules that govern evolution and account for the great variability of organisms are quite straightforward. Organisms – which can be seen as candidate solutions – evolve through random variation due to mutation, crossover and manipulations on their genetic material; these candidates are subjected to selective pressures which evaluate their adaptiveness and determine their capacity of generating descendants, thus propagating better fit genotypes into the future generations. These characteristics are the inspiration of Evolutionary Computation.

Evolutionary Algorithms

EC tries to mimic the mechanisms of biological evolution to solve complex problems (Mitchell and Taylor 1999; Fogel 2000a; Fogel 2000b). Even though specific implementations can vary significantly and algorithms are not constrained to using only biological mechanisms, there are three common features which are shared by the different branches of EC (Bäck 2000):

1. A population. A number (n) of candidate solutions (representations of the problem) compete against each other to remain in the population and generate offspring. Since ECs use populations, they can be seen as a parallelized search of the solution space.

2. Selection. Organisms from the population pool are selected for culling or reproduction based on their fitness. Fitness is a function measurement of how *good* a representation is at solving the problem. The two most adopted methods for assigning fitness are as a direct mapping to the problem or as a relative measurement of performance in relation to the remainder of the population. Arguably, the choice of a fitness function that clearly states the problem is the most important step in determining the success or failure of the EC algorithm.

3. Search operators. EC uses stochastic methods to solve a problem; these biologically inspired operators provide the variability necessary for the EC population to explore different areas of the solution space. The two main sources of variability are mutation, which are randomly generated new sources of variability and crossover, which exploits the available variability within the population to form new combinations of candidate solutions.

Thus, a general EC algorithm combines these features and through iterations improves the overall fitness of the population, gradually converging on a solution. The following steps form the general structure of an EC algorithm:

1. Create an initial population – randomly or based on prior information
2. Assign a fitness value to all organisms (also referred to as chromosomes)
3. Select organisms for reproduction based on their fitness and a selection scheme
4. Create descendants from the selected parents
5. Modify the descendants with the search operators
6. Evaluate the fitness of the descendants
7. Cull organisms from the parental population and replace them with the descendants according to the selection scheme
8. Repeat from step 3 until a termination criterion is met, for example, a specified number of iterations or a predefined fitness value is reached

Computational representation and implementation

An appropriate choice of representation for the populations is crucial for an EA and largely depends on the nature of the problem. A parameterization problem is usually represented as a real-valued vector; if using an ES or EP the vector consists of the solution vector and variability parameters. Finite-state representations are also frequent with EP. A GA classically uses binary strings. GP has to store information on the functions, the terminals and the relations between the two; lists, stacks, parse-trees and vectors are commonly used.

The choice of programming language is of secondary importance to the algorithms and they can usually be easily ported between languages. Without swimming in the dangerous waters of defending a specific language, I am rather fond of C# because it's a modern fully object-oriented language which allows rapid development of GUIs and algorithms with a low overhead. BK prefers VB.Net (and for exactly the same reasons!). On the downside both C# and VB are slower than either Fortran or C under the latest release of the .Net platform. EAs can also be easily written for Matlab, R and even Excel. The bottom line is that EAs can probably be coded into just about anything. The choice depends on what the objectives are.

Population

The population structure and size varies according to the type of EA used. Normally an initial population is randomly created or seeded from some previous set of candidates (e.g. for a multiple sequence alignment problem, an EA converges faster and more efficiently if the initial population is formed by all possible pairwise alignments).

Population sizes also vary according to the type of problem and algorithm. As a rule of thumb Differential Evolution works well with smaller populations (10-20) whilst GAs and GPs need large population sizes (100-1000, or even 10,000).

Selection

Selection is an integral component of all EC methods. Through selection, solutions with a higher fitness are emphasized in the population. There are several selection operators (Bäck et al. 2000a) but all essentially select better solutions for reproduction and delete less fit solutions which are replaced by the offspring of the better performing ones. Selection does not generate new solutions; it simply directs the evolution of the population. Of notice is that not only the best organisms are always selected; the process is stochastic, which can allow inferior solutions to be selected over better ones with a low probability. This preserves the diversity of the population and avoids a premature convergence on local optima. The main selection methods are proportionate, rank-based, Boltzmann and tournament.

Proportionate selection assigns a probability of generating offspring based on the relative fitness of the organism. The simplest form of proportionate selection is roulette wheel; where each solution is assigned an area in the wheel proportional to its fitness – fitter organisms have a bigger area and consequently a higher probability that the wheel when spun will stop in their area.

Rank-based selection ranks the entire population based on their fitness and then assigns a selection probability based on these ranked values.

Boltzmann selection uses a probability distribution with a T term similar to the temperature term in the Boltzmann distribution which decreases as the iterations progress; initially all solutions have similar chances of being selected since a large T is used but as T reduces the stringency increases and only better solutions are chosen.

Tournament selection chooses a certain sample size from the population to compete and the ones with the highest fitness are selected; the selective pressure is defined by the size of the tournament. Tournament selection is rapidly becoming the selection method of choice for EC applications. There is no need to evaluate the entire population or maintain population statistics which makes the selection process faster. For the same reasons it is also well suited for parallel implementations. The major drawback of roulette wheel is avoided; in which the size of the areas rapidly become the same as the population converges on a solution, forcing the use of a fitness scaling mechanism between the upper and lower limits of the fitness range. Tournament selection is inherently noisy and can rapidly lead to a loss of diversity. To counterbalance this effect small tournaments are preferred in association with slightly higher mutation rates. Hancock (2000) presents a comparison of the different selection mechanisms.

Generation structure

EC uses two generational structures: steady-state and generational:

1. Steady-state uses an overlapping generation approach in which parents and offspring simultaneously compete in the population. Tournament selection is typically steady-state with only a few new organisms in each generation.

2. The generational approach uses non-overlapping populations with the offspring entirely replacing the parental population.

Steady-state runs tend to have a higher variance, thus in small populations the effect of drift in more pronounced and can lead to the loss of variability. To counteract this effect, larger populations should be used in steady-state systems.

Fitness

Selection operators act on the fitness of the organisms. Fitness is arguably the most important aspect of any EA, if the fitness function is not well constructed the whole EA will breakdown.

The fitness function can be seen as a measure of the probability that an organism will survive and reproduce in the population. The selection scheme and the fitness function are inextricably connected. A good fitness function should allow for a range of intermediary values which can be explored by the EA. All-or-nothing fitness functions are ineffective for an EA which must be capable of evaluating if a certain solution is better or worse; the less granular the fitness function, the higher the probability that the EA will converge on an adequate solution.

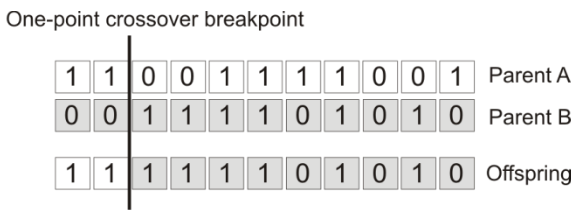
At this point it is important to highlight the distinction between fitness and objective function. The fitness function maps to the objective function which is external to the EC and depends on the nature of the problem. The terms fitness and objective function are frequently used as synonyms, especially when the mapping between the fitness function and the objective function is 1:1. On the other hand, if a population scaling scheme is adopted it is quite clear that fitness and objective function are necessarily distinct. In summary, the objective function assigns a value to an organism which can be directly translated as its fitness or which can be mapped to a fitness value based on the fitness function of the EA. All this might sound rather pedantic, but the message is that you can either directly link selection with the problem or you can use a completely independent method to select the population.

Search operators

Mutation and crossover are the main search operators used in EC. Their main function is to modify candidate solutions to explore the solution space. Frequently both are used in an EA and the parameter settings for these operators are critical for a successful run. High mutation rates can reduce the method to a random search. If too low, there will be little variability or loss of variability in the population. The same applies to crossover, if too high good constructs will be broken up. If too low there will be little exploration of the search space. A balance always has to be achieved between the two search operators as well as the selective pressure that is applied and the population size that is used. These are the four main parameters in an EA.

Crossover

Crossover is a search operator that does not generate new sources of variability in the populations albeit introducing new variation. It operates by combining parts from two or more parents to generate one or more offspring. The drive behind crossover is to generate new variability in the population by manipulating the component sources of variation to explore new combinations which might be better solutions to the problem. There are many different crossover algorithms depending on the EC method and the representation of the problem: binary strings, real-valued vectors, finite-state machines or parse trees. Booker et al. (2000) reviewed different crossover methods. The most straightforward crossover is used in the canonical GA and is a good model to illustrate the principle. In the figure below a one-point crossover in a binary GA is depicted.

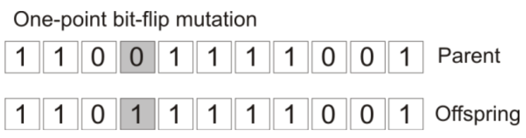

One-point crossover breakpoint
1 1 0 0 1 1 1 1 0 0 1 Parent A
0 0 1 1 1 1 0 1 0 1 0 Parent B
1 1 1 1 1 1 0 1 0 1 0 Offspring

One-point crossover in a binary genetic algorithm. A breakpoint is randomly selected and the two chromosomes swap bitstrings after the breakpoint. Crossover is a search operator which explores available population variability by testing new combinations. No new allelic variability is generated through crossover but it does generate new variation in fitness values.

Briefly, two parents are selected for crossover, a breakpoint in the chromosome is randomly determined, and from the breakpoint onwards the two chromosomes swap the remainder of their bitstrings. The figure shows how crossover can produce an offspring with a higher fitness than the parents. Consider that the trivial objective function of a GA is to maximize the number of ones in the binary string. The parents have respectively seven and six ones in their strings and, in the example, their offspring have eight ones, after crossover.

Mutation

In contrast to crossover, mutation generates new allelic variability in the population. The general principle is that new offspring are created by a stochastic change to a single parent. Like crossover there is a plethora of mutation algorithms for the different EAs (Bäck et al. 2000c). For example, a real-valued EA would change the value at a selected allele with a new randomly selected value. Again, the canonical GA is the most pictorial example of mutation. The following figure shows a point-mutation bit-flip in which an allele of a parent is randomly selected to be flipped. The most common approach is to assign a small uniform probability that mutation will occur and test each position of the bitstring; if the mutation operator returns true the bit at the position is flipped. Notice that mutation can also produce an offspring with a higher fitness than the parent. After mutation the offspring has eight ones instead of seven.



Point-mutation bit-flip in a binary genetic algorithm. New offspring are produced by a random change to the parent. In the example allele at position 4 mutated and flipped from zero to one. Mutation is a source of new variability in a population.

Mutation is an important operator to generate new sources of variability and expose new areas of the solution landscape whilst crossover can only shuffle available variability. Consider that in the example in the figure above no member of the population had a one in allele four; no amount of crossover would generate a solution with one in this position, thus an optimal solution would be unobtainable.

The interplay between mutation and crossover is paramount to the success of the EA. Mutation feeds new variability into the system and crossover tries to combine it into useful combinations.

Concluding remarks

Evolutionary algorithms are primarily computational methods designed for optimization of complex problems with large search spaces. There is no optimal method for solving all types of optimization problems. An algorithm adequate for a certain class of problems may breakdown under a different problem.

The framework for developing suitable EA algorithms for a given problem can be broken down into the following steps:

1. Nature of the problem – to solve a problem it is necessary to understand it. This may sound like a tautology, but EAs need to be able to evaluate how good a solution is, if not in absolute terms, at least in relation to other candidate solutions. If a problem is well understood, a more reliable/realistic method to evaluate solutions can be developed.

2. Modeling of the problem – optimization is not carried out on the problem itself but on a model of the problem. This is an important distinction; a solution can be perfect for the model but, for the real problem, it is only as good as the model itself. Thus again the importance of understanding the nature of the problem. Knowledge of the problem allows the development of a model that captures and reflects its essential characteristics.

3. Objective function – arguably the most important component of EAs. The objective function is a measurement of how well a solution fits the model of the problem and is used to assign a fitness value to candidate solutions, either through, for example,

direct 1:1 mapping or rank based selection (based on the relative performance of solutions within the population).

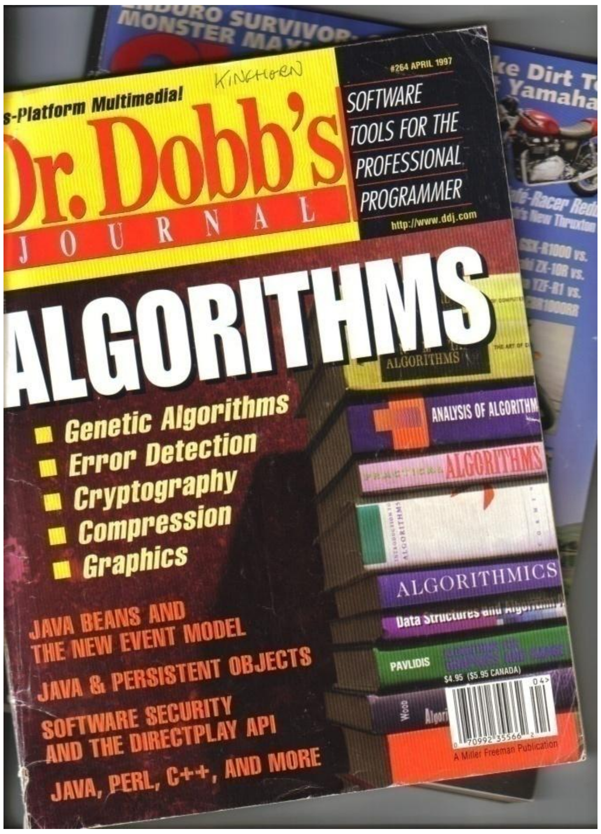
4. Development of an evolutionary algorithm – depending on the problem a certain EA will be better suited than others. For example, it is now generally accepted that the original binary Genetic Algorithm is inefficient to solve real valued numerical problems. Alternative methods such as Differential Evolution, discussed in the next chapter, are faster and yield better solutions. Two other important aspects are the design of efficient search operators (we will see many examples during the course) and how to present the problem to an EA, ideally in a parameterization that *automatically* accommodates constraints (in chapter 7 we will discuss how to manage constraints).

Chapter 4: Differential Evolution

Brian Kinghorn

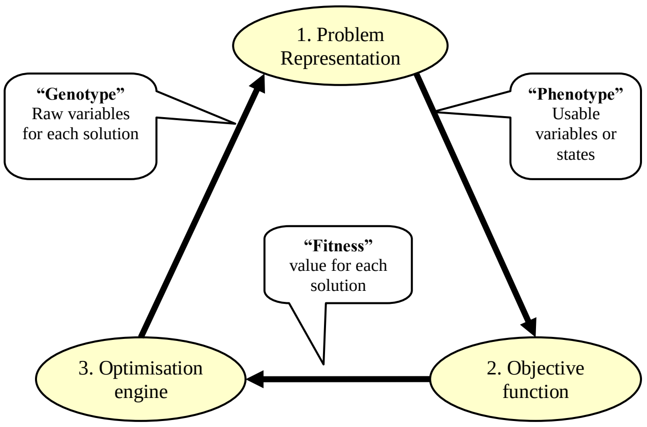
Vive la difference!

Introduction



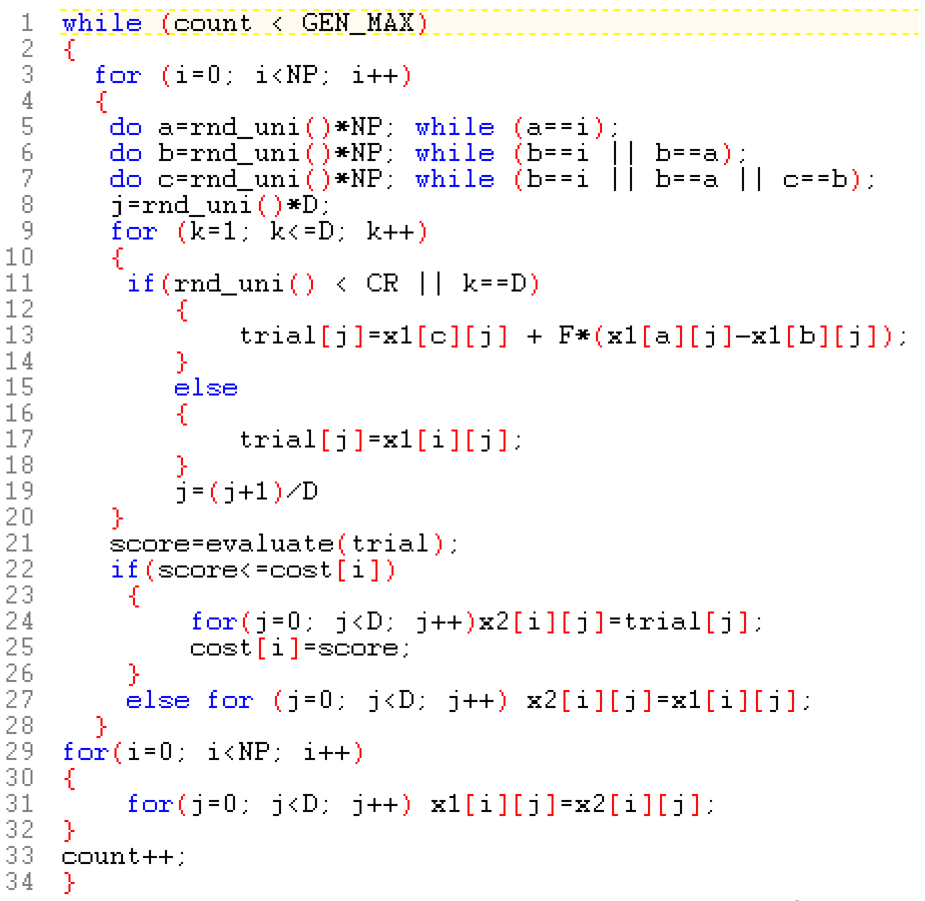
BK escaped from the office for a cup of coffee and a browse through the motorbike magazines at the Student Center, Colorado State University in March 1997. This action changed his life.

Within a week or so the massive Colorado Beef Cow Production Model was being optimized, with relative ease. “With relative ease” are the operative words here. Differential Evolution is an EA that is small – it can be as small as one page of code – and simple to implement. It fulfills the job of “3. Optimisation engine” in this diagram from Chapter 1:


1. Problem Representation
“Genotype” Raw variables for each solution
“Phenotype” Usable variables or states
“Fitness” value for each solution
3. Optimisation engine
2. Objective function

What does “Differential” mean? This comes from the way that the algorithm matches the size of mutations to the amount of variability in the current generation of solutions. This is done individually for each parameter to be optimised, as we will see in more detail in the exercise. This makes the algorithm more adventurous at early stages, while it is looking for general regions in which good solution might lie. As it begins to hone in on one or a few promising regions, it takes smaller and more measured steps. This can give an appropriate blend of robustness and speed of progress at all stages of the optimization. DE is generally found to perform well compared to other more bloated alternatives (eg. Mayer et al 2005).

How simple is DE?



```
while (count < GEN_MAX)
{
  for (i=0; i<NP; i++)
  {
   do a=rnd_uni()*NP; while (a==i);
   do b=rnd_uni()*NP; while (b==i || b==a);
   do c=rnd_uni()*NP; while (b==i || b==a || c==b);
   j=rnd_uni()*D;
   for (k=1; k<=D; k++)
   {
    if(rnd_uni() < CR || k==D)
       {
           trial[j]=x1[c][j] + F*(x1[a][j]-x1[b][j]);
       }
       else
       {
           trial[j]=x1[i][j];
       }
       j=(j+1)/D
   }
   score=evaluate(trial);
   if(score<=cost[i])
    {
        for(j=0; j<D; j++)x2[i][j]=trial[j];
        cost[i]=score;
    }
    else for (j=0; j<D; j++) x2[i][j]=x1[i][j];
  }
for(i=0; i<NP; i++)
{
    for(j=0; j<D; j++) x1[i][j]=x2[i][j];
}
count++;
}
```

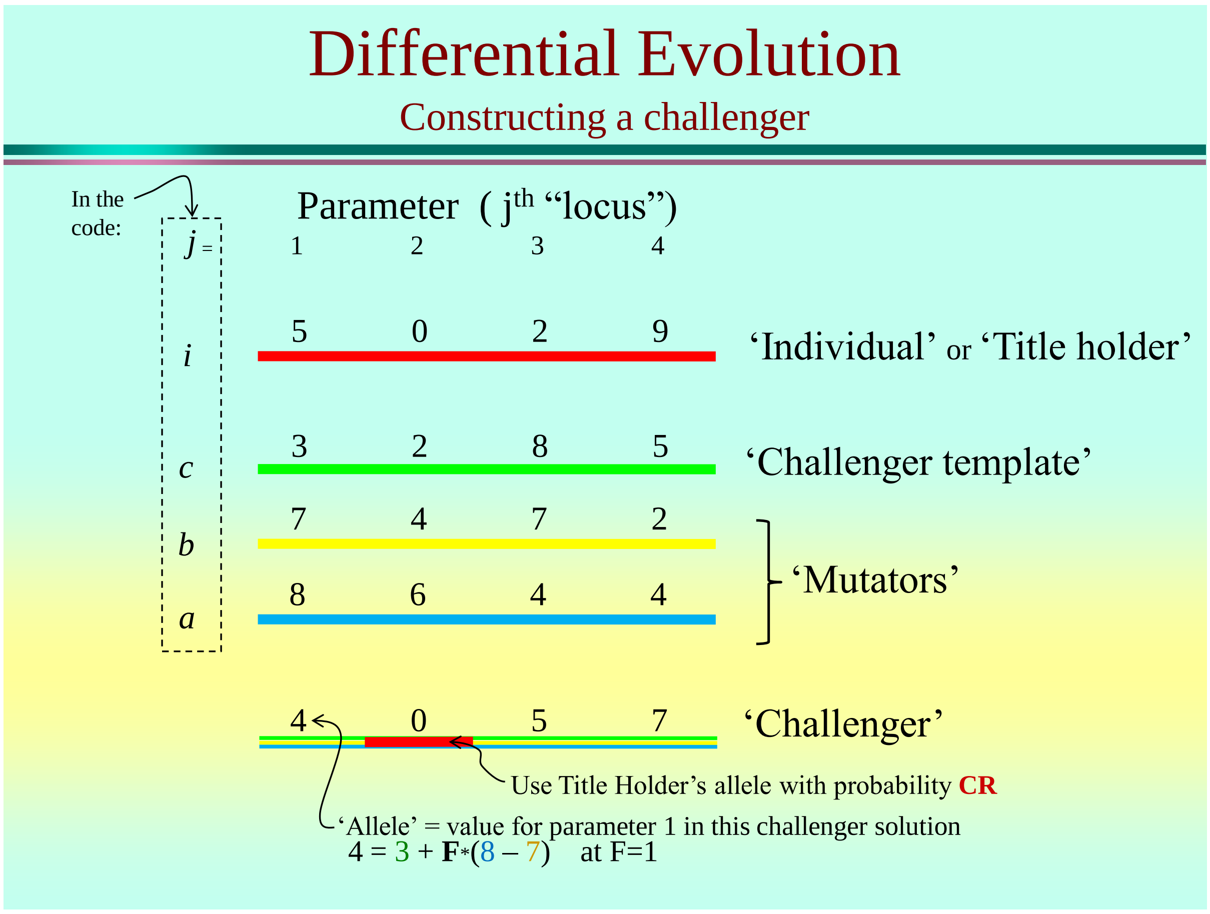
Adapted from Price and Storn (1997)

… small, simple and effective.

How does DE work?

DE uses “tournament selection”. There is a population of solutions – typically of size 10 or so. Think of 10 chairs, each with a “title holder” (actually a solution) standing on that chair. A title holder can keep its position on its chair for as long as it is able, over many cycles or “generations”. In each generation, a challenger solution is made for each of the chairs, to challenge the title holder in a one-on-one tournament. If the challenger is a better solution than the title holder, it climbs on the chair as the new title holder for that position in the population. It is a good thing to let the challenger win even if it is only equal in value to the title holder – better to keep things moving.

All we need is a class full of volunteers, 4 chairs, 4 dice, 2 coins and lots of pieces of paper … later on.


Differential Evolution
Constructing a challenger
In the code:
Parameter (jth "locus")
j = 1 2 3 4
i 5 0 2 9 'Individual' or 'Title holder'
c 3 2 8 5 'Challenger template'
b 7 4 7 2
a 8 6 4 4
'Mutators'
4 0 5 7 'Challenger'
Use Title Holder's allele with probability CR
'Allele' = value for parameter 1 in this challenger solution
4 = 3 + F*(8 – 7) at F=1

How to make a challenger?

A challenger is to be constructed for each title holder. This is where the "Differential" bit comes in. First choose three title holders that are different from the one to be challenged (This makes the minimum populations size = 1+ 3 = 4). Call the solutions for these title holders *A*, *B* and *C*.

We make the solution for the challenger equal to *C*, but then we change (mutate) it according to the ***Difference*** between *A* and *B*:

Challenger solution = $C + F^*(A - B)$

… where F is a tuning parameter described below, typically between 0.2 and 2.0 in value.

This oversimplifies the basic DE as invented by Price and Storn. We still need "Recombination" or "Crossing over".

Solutions A, B and C are actually vectors of parameters that are needed to describe a solution. So, for example, $C = (c_1, c_2, \ldots , c_L)$ where there are L parameters that describe a solution. [L stands for Loci]

A useful strategy is to make some use of parameters of the title holder to be challenged. If this is the i^{th} population member, s/he will have parameters $I = (i_1, i_2, \ldots , i_L)$.

So we cycle through the parameters (j=1 to L) and allocate parameter values to the challenger as follows:

With probability CR : Challenger solution = $c_j + F^*(a_j - b_j)$

With probability (1-CR) : Challenger solution = i_j

The latter scenario has taken place in the example in the PowerPoint slide above, with the allele '0', colored red, being inherited by the challenger directly from the Title holder that it is going to challenge.

Choice of CR and F

If F is large, the algorithm is more adventurous – casting more widely to find solutions that might be good. This is useful in the early generations, helping to give some initial coverage of the solution space. However, it will usually slow down convergence once there is a decent hill to climb – too much distraction in outlandish areas.

One strategy is to keep F high (say between 1 and 2) for the first several hundred generations (more or less for bigger and smaller problems). Much of this tuning work can involve trial and error.

I use a periodic change in F – to "have your cake and eat it". Some generation are adventurous, but most are conservative:

```
'these bits vary 'mutation' every few generations ...
If generation Mod 4 = 0 Then F = 1 Else F = Fhold
If generation Mod 7 = 0 Then F = 4 Else F = Fhold
```

For CR I use about 0.4. Trial and error. But …

```
'Usually good to start higher ...
If generation > 500 Then CR = 0.2 Else CR = 0.4
```

Extra mutation

I find that "Out of the box" Differential Evolution can get stuck on a local optimum rather more easily that is desirable. This is at least partly because all population members converge on this part of the solution space, and the mutation generated is reduced to such small levels that it is essentially impossible to get away from this location.

This is easily fixed by randomly introducing other forms of mutation. First of all, it makes sense to keep “Pure DE” in some generations …

```
' Pure DE most generations
If generation Mod 3 = 0 Then DiffOnly = 1 Else DiffOnly = 0


****************
If MyRnd.NextDouble < CR Or k = loci Then
  If MyRnd.NextDouble < 0.9 Then      ' do DE ~prob=.9
    allele(j) = parentallele(c, j) + F * (parentallele(a, j) - parentallele(b, j))
  ElseIf MyRnd.NextDouble < 0.5 Then  ' make proportional mutation ~prob=0.05 (But
                    no good if stuck at value of zero)
    allele(j) = parentallele(c, j) * (0.9 + 0.2 * MyRnd.NextDouble)
  Else                                ' make absolute mutation ~prob=0.05
    allele(j) = parentallele(c, j) + 0.01 * F * (parentallele(a, j) + 0.01) *
                    (MyRnd.NextDouble() - 0.5)
    ' ... could make this a function of mean solution rather than an arbitrary +0.01
  End If
Else
  allele(j) = parentallele(i, j)     ' or no change depending on CR
End If
```

Note that this is not “The Correct Way” of doing these things. In this business one is free to express one’s own opinion and ideas. But you should test your ideas for speed and robustness of convergence. That can take quite a bit of investigation!

Chapter 5: Genetic Programming

Cedric Gondro

Why not evolve the model as well?

Introduction

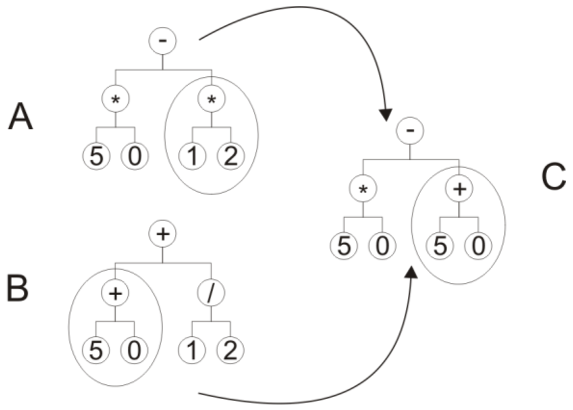
So far we have focused on *numerical optimization*. For example, we have a certain model and we want to parameterize it or we have an assignment problem and we want to allocate resources to the different classes. But what if, say, we have observed data in an experiment but don't have an underlying model to explain the observations? What we want is to discover the model itself. That's were GP comes in handy, using the same EC concepts we have seen so far, it can be used for model construction (and for good measure parameterization as well!).

Overview of genetic programming

Often regarded as a specialization of Genetic Algorithms, GP has evolved to become a branch of EC in its own right. Initially GP was devised as a method to optimize data structures as executable computer programs with the fitness value assigned based on the results obtained when executing the instructions contained in each member of the population. In this context, GP evolves populations of computer programs or other algorithmic processes to solve a specific problem (Banzhaf et al. 1998; Koza 1989; Koza 1992; Koza 1994; Koza et al 1999; Koza et al 2003).

Original implementations of GP used tree-structured representations implemented in LISP. Tree-structures have the terminal nodes of the tree containing inputs (referred to as terminals) and the internal nodes holding functions. This type of construct demands significant overhead to ensure viability of the trees (handle, for instance, division by zero or infinite loops) or correct tree structures which can break-up due to mutation and crossover.

Currently LISP implementations are rarely used and alternative language implementations such as C++ are used. The same applies to new GP structures developed to improve on the original Tree-GP. Alternative methods are numerous (as everything else in EC!) and include Linear-tree Genetic Programming (Kantschik and Banzhaf 2001), Linear Genetic Programming (Nordin 1994), Linear Page-based Genetic Programming (Heywood and Zincir-Heywood 2000) and Gene Expression Programming (Ferreira 2001). GP uses crossover and mutation in a similar fashion as GAs. Commonly crossover involves swapping entire subtrees between parents. The figure below illustrates a typical tree GP and crossover between two parents. Mutation replaces a subtree with a random subtree of a randomly defined depth. GP structures use variable-length representations which have a tendency to grow in tree depth through the addition of subtrees of functions and terminals which contribute only slightly to improve the fitness. This growth is referred to as bloat and is an active field of research in the GP community. A common practice is to place a cap on tree depth (Langdon and Poli 2002).



Crossover in a tree GP. Crossover between parents A and B generate offspring C, the function set is {*,+,/,-} and the terminal set is {2,0,1,5}. The trees code for A=(5*0)-(1*2); B=(5+0)+(1/2); C=(5*0)-(5+0).

Gene Expression Programming

We will focus on Gene Expression Programming – GEP (Ferreira 2001; Ferreira 2002). GEP is very straightforward to implement, fits well with modelling problems and is the author's favourite!

GEP is a variant of Genetic Programming that instead of coding structures as non-linear entities, the typical parse-trees of GP; it uses linear strings of fixed size which are translated into non-linear entities of different sizes and structures. This two-step approach allows GEP to be viewed as a hybrid between Genetic Algorithms and Genetic Programming. GEP has essentially three main advantages over GP. Firstly, it allows implementation of the straightforward search methods common to GAs whilst maintaining the structural complexity attainable through GP. Secondly, all linear strings either code for valid structures or can be repaired with little overhead; such is not the case with GP where the parse-trees can easily breakdown into invalid structures and complex search operators must be used to ensure tree viability. The need to preserve valid tree structures limits search operators mainly to crossover between tree branches of the same arity. The third advantage of GEP is that it allows for more parsimonious solutions in contrast to GP where the entire parse-tree is the solution and the tendency is to bloat the tree to the maximum allowed size once the population stops evolving.

Heads and tails

The linear strings of fixed size in GEP are referred to as chromosomes. Each chromosome has *n* genes with a head and a tail. The head consists of terminals and functions and the tail only of terminals. Terminals are numerical variables or constants, and functions are mathematical operators. The size of the head is a user-defined parameter while the tail is a function of the size of the head such that $t = h(n-1)+1$, where t is the tail size, h is the head size and n is the highest arity of the set of functions.

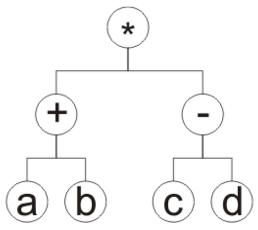
Each gene (h+t) can be translated into an expression tree (ET) from the coding region of the gene, referred to as an open reading frame (ORF) in a loose biological analogy. An ORF can be of the same size of the gene or smaller, in the last case there are non-coding regions downstream of the gene. The size of an ORF depends on the position and relationships between functions and terminals on the gene. This is a simple matter of looking at each position on the gene starting from zero and if in that position there is a function its arity is added to a counter. When the counter value becomes smaller than the current position all terminal nodes in the ET are filled with terminals and no further functions can be added to the ET.

In summary a GEP structure consists of a chromosome with genes of fixed size which code for ORFs of variable sizes that are translated into expression trees (ETs). Search operators act on the gene structure (genotype) while selection acts on the expression trees (phenotype).

To illustrate these concepts consider the following equation:

$$y = (a+b)*(c-d)$$

This simple algebraic expression can be split into a set of terminals T={a,b,c,d} and a set of functions F={+,*,-). This equation can be represented as the ET:



Expression tree of equation *y = (a+b)*(c-d)*. Terminal set T={a,b,c,d) and function set F=(+,*,-}.

The ORF for this equation can be constructed copying the values of the nodes from top to bottom and from left to right, resulting in the string { * + - *a b c d* }.

The figure below shows an example of this ORF as part of a gene of size 21. The head size of this gene is 10 and consists of terminals and functions. The tail (in bold) is formed exclusively with terminals and, from our example is of size 11 since the function of highest arity is two. Note that the break off point in the gene is position 6, since in this position the total arities of the functions add up to 6 and the total number of elements is 7. Evidently from the tree above it is clear that no further function could be added to the expression tree. Thus a gene of size 21 can code for an ORF of size 7.

0	1	2	3	4	5	6	7	8	9	**10**	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18**	**19**	**20**
*	+	-	a	b	c	d	*	-	-	**a**	**a**	**b**	**c**	**c**	**d**	**d**	**d**	**b**	**b**	**a**

GEP gene of size 21 with head (h) of size 10 and tail (t) of size 11. $t = h(n-1)+1$. The highest arity of the function set is 2, thus n = 2.

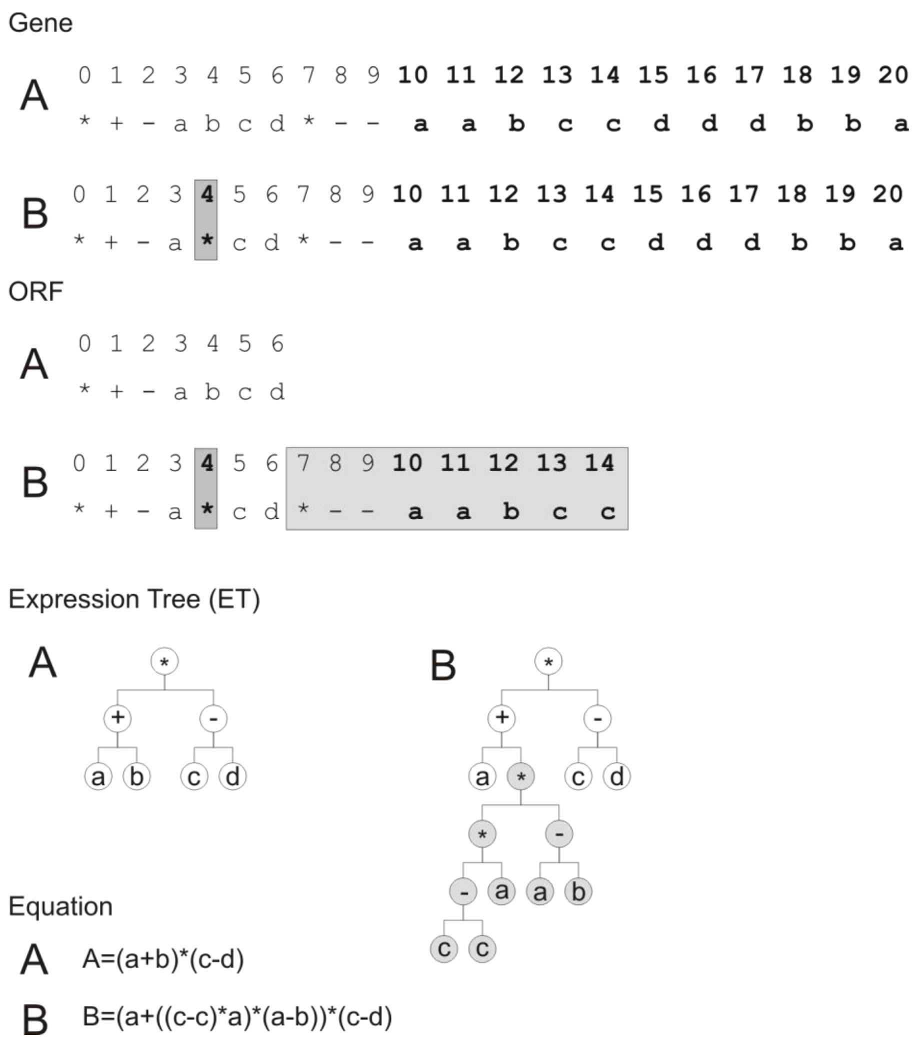
At this point it should be clear that the tail is important to ensure integrity of the tree structure. By having a tail with only terminals, even if the head is solely formed of functions, there are enough terminals available to form complete trees with terminals in the outer nodes. In this scenario the gene and the ORF are of the same size. The value of the function with the highest arity is important to ensure tails with sufficient terminals for any combination of functions.

Selection and Search Operators

Originally GEP uses roulette wheel selection. The author tends to use the currently preferred tournament selection since it permits better control of the selective pressure and allows for a smoother evolution.

As we mentioned the distinction between head and tail is an important component of GEP. Tail integrity (only terminals in tail) must be preserved and search operators need to be used accordingly. GEP uses a wide range of search operators: mutation, insertion sequence transposition, root insertion sequence transposition, gene transposition, one and two-point crossover and gene crossover (Ferreira 2001).

Before discussing search operators let's look at how they can modify the structure of ETs yet still preserve structure integrity. For our example consider a mutation operator. Mutation randomly changes the value in a gene position with another value given a certain probability. Since gene structure must remain intact, mutation in the head can replace the original value with either a terminal or a function; in the tail section only terminals can be used. The following figure shows a point mutation in position 4 in which a terminal (*b*) was replaced by a function (*), as a result instead of the ORF of length 7 a new ORF of length 15 is created. A new expression tree is created and the resulting equation is also very different. Notice that tree viability is still preserved.



Effect of mutation on the GEP structures. A single point mutation can completely change the size of ORFs, the ETs and the final product. In position 4 a terminal (b) mutated to a function (*) as shown in Gene B. From this mutation a new ORF was created of size 15 (shaded grey) replacing the original ORF of size 7, and a new tree (new segments shaded grey). The final products are shown under *Equation*. Tails in bold font.

Even though the original GEP employs several search operators, the operators described below are adequate for most tasks. This choice was based on better convergence in test cases or simple computational efficiency. The operators can be subdivided into mutation and crossover.

Mutation

Two types of mutation work well: point mutation and block mutation. Point mutation replaces a single position with another random element from the function or terminal set in the head or just from the terminal set in the tail. Block mutation replaces a randomly defined number of positions (up to 20% of the gene size is reasonable) with new random elements; again, maintaining tail integrity. The probability of block mutation at 10% of the point mutation probability is an adequate starting point.

Crossover

Crossover can consist of one, two and three-point crossover, with an equal probability for each method.

1. One-point crossover cuts both parents at the same position and the remainder of the gene downstream from the cut point is swapped to form the offspring.

2. Two-point crossover selects a block of the same size, starting at the same position in both parents and this block is swapped in the offspring.

3. Three-point crossover is a simple extension of two-point crossover where instead of one block being swapped, two blocks are swapped (notice that for three-point crossover *four* cut points are necessary).

The first two methods are common operators in GEP, the third is a new extension used by the author. Three-point crossover is the most disruptive operator employed and is used to replace transpositions, gene crossovers and gene transpositions which are excessively disruptive for most cases.

Fitness and Objective Functions

The same guidelines for fitness and objective functions discussed so far are still valid. But GEP (and all GP flavours for that matter) have some additional considerations.

Even though GEP will mostly yield valid trees, it is still possible that non viable structures will appear. These could be a division by zero or the square root of a negative number. These invalid organisms can be assigned a highly negative fitness value so that they are rapidly removed from the population. The same approach can also be employed for unstable equations that overflow. Fitness evaluation tends to be the most time consuming aspect of GEP, mainly because of the structure checks.

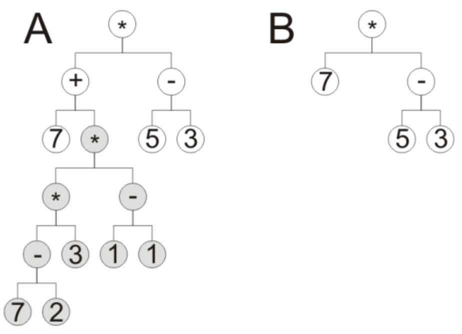
A good example of where tree structures can break down is with the use of time-delays in differential equations. A time-delay can be viewed as an operator that takes two arguments, on the right-hand side a list of, for example, the concentrations of a component over time and on the left-hand side a delay (t-τ), and returns the concentration at the delay point.

An option for incorrectly placed time-delay operators instead of assigning a low fitness value, to the GEP string, is to repair it and ensure a viable solution. Such repairs can be carried out probabilistically, either replacing the time-delay operator with another randomly selected operator or replacing the right-hand side with a concentration and the left-hand side with a valid delay.

A last aspect of the fitness function is constraint-handling. A simple way of handling constraints is to set values that do not meet the constraints with the average value of the constraint range defined for the parameter, instead of penalizing the entire function. This approach ensures that a higher proportion of the population is formed of organisms that meet the constraint criteria. In chapter 7 we will see more advanced approaches to handling constraints.

Bloat

Bloat is a common phenomenon in Evolutionary Algorithms of variable length, particularly Genetic Programming and its variants. This essentially is the growth in the length of organisms which does not necessarily reflect an improvement in their respective fitness. Frequently the growth consists of regions that do not alter the structure of the solution giving raise to inert regions referred to as introns. An example are long trees attached to an addition branch that result in zero: they do not alter the value of the solution.



Bloat. The shaded tree is inert and does not alter the final value of the organism. A) A bloated parse-tree with 15 nodes. B) The same tree can be reduced to 5 nodes.

A common cause of bloating in EAs is attributed to convergence. If the algorithm ceases to find better solutions or evolution slows down, there will be a greater number of similar solutions and the entire population tends to grow to the maximum allowed size.

Several approaches have been suggested to reduce bloat. Common techniques include maximum size and tree depth limits (Koza 1992) used in almost all GP applications, a fitness function weighted by the size of the solution (Banzhaf *et al*. 1998) and code editing to identify non-coding regions and prune them out (Blickle 1996).

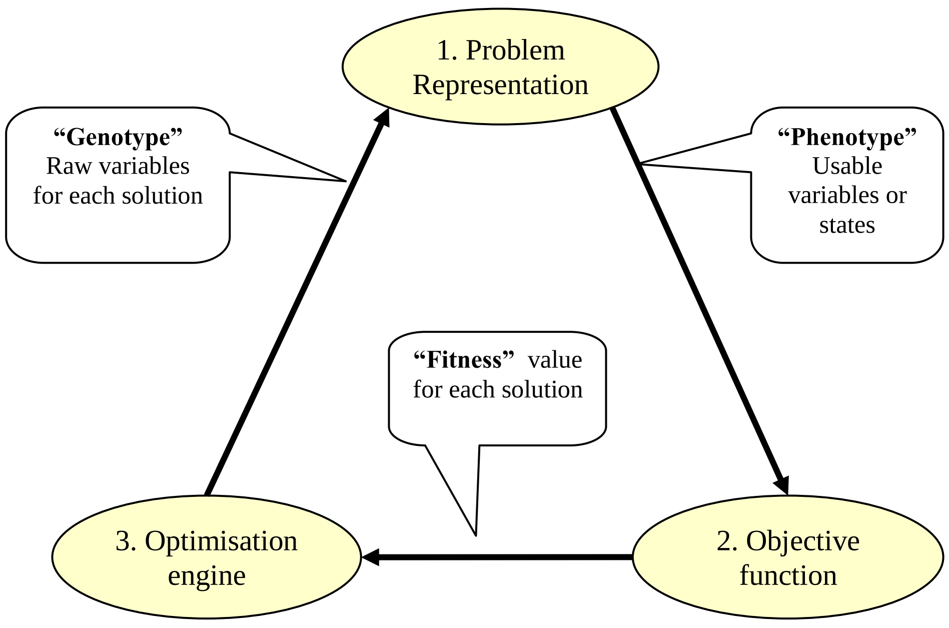
Chapter 6: Introduction to problem representation

Brian Kinghorn

Making complexity out of simplicity

Introduction

This is our pattern for a problem optimization system, from Chapter 1:


1. Problem Representation
"Genotype" Raw variables for each solution
"Phenotype" Usable variables or states
"Fitness" value for each solution
3. Optimisation engine
2. Objective function

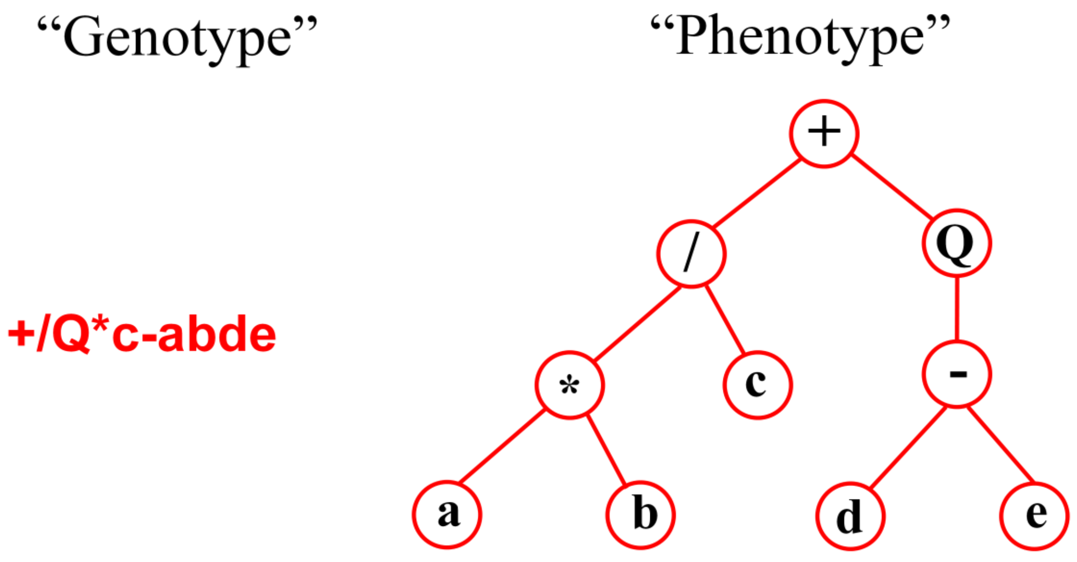
To let the optimization engine operate simply and effectively, we need to let it work with a set of simple variables to represent each candidate solution – such as a vector of real numbers, a vector of integers, or perhaps a mixed vector.

Example 1: No problem representation filter needed

In simple cases, such representations can be used directly to evaluate a solution through the objective function (or the criterion that acts for the objective function) – an example of that is in our first optimization example, to find $\{x_1, x_2\}$ that maximizes $y = -(30 - x_1)^2 - (4 - x_2)^2$. $\{x_1, x_2\}$ is a simple vector of real numbers.

Example 2: Find a good-fitting equation using GEP

However, other problems are more complex and need some form of problem representation. Gene Expression Programming (Chapter 5) is a nice example:



$$\frac{a*b}{c} + \sqrt{d-e}$$

In this case the relatively simple system that the optimization engine handles is the “Genotype” and this represents the “Phenotype” – an equation whose form and parameters are to be optimized.

Problem representation “converts genotypes to phenotypes” and involves some kind of filtering system.

Example 3: A mate selection driver

Here is another example – optimizing Mate Selection using the filter suggested by Kinghorn and Shepherd (1999)

We have to decide which males to mate to which females, subject to a number of constraints, such as the maximum number of mate allocations per candidate. Early efforts by Ross and Brian were not effective – we tried various approaches, such as directly optimizing the matrix of mating instances. However, almost all solutions thrown up by a genetic algorithm broke constrains, and had to be aborted or "fixed up" (see "Managing Constraints" later on).

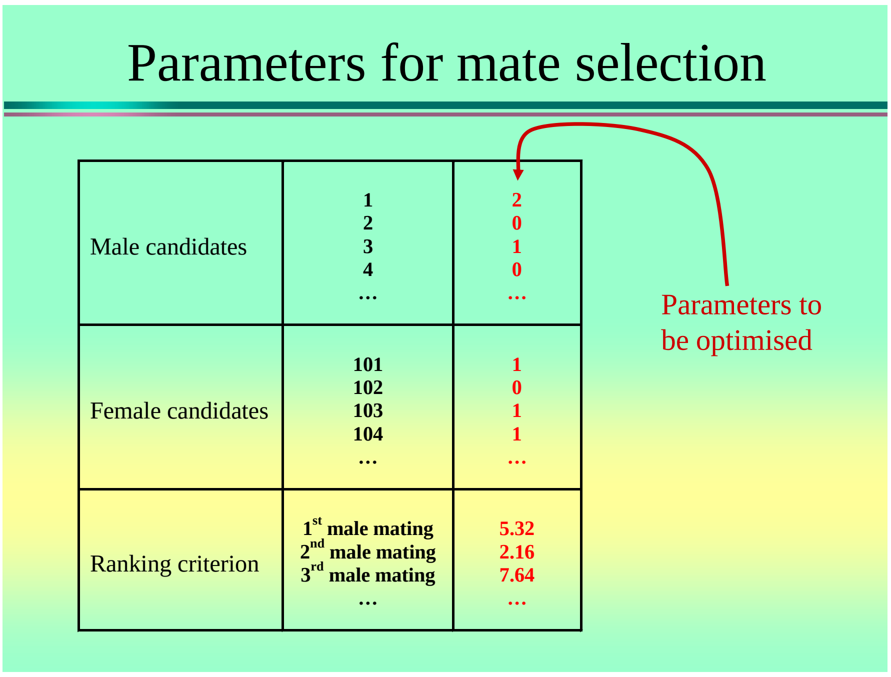
However, the following filtering idea will only result in legal solutions for the pattern of mate allocation:

Table 1. This table illustrates the components to be optimised for mate selection - they are underlined.

			Female →	1	2	3	4 …
Male ↓	No of uses	Ranking criterion	Rank	**1**	**0**	**1**	**1**
1	**2**	**5.32** **2.16**	2 3			✓	✓
2	**0**	-	-				
3…	**1**	**7.64**	1	✓			

From Kinghorn and Shepherd (1999): "The mate selection driver shown in Table 1 was developed to conduct the search across all legal mating sets. The underlined figures in Table 1 drive the three matings noted, and these are the values to be optimised. "No. of uses" (second column for males, second row for females) is the number of matings for which each animal should be used, and this in turn drives selection, including extent of use of each animal. An animal is culled if this is set to zero. "Ranking criterion" is simply a real number. It is ranked to give the column "Rank". This in turn drives the mate allocation. The first ranked male mating is the single mating from male 3. He is thus allocated to the first available female mating (the one nearest to the left) - the one mating from female 1. The second ranked male mating is the first mating from male 1. He is thus allocated to the second available female mating (the one second nearest to the left) - the one mating from female 3. The third ranked male mating is the second mating from male 1. He is thus allocated to the third available female mating - the one mating from female 4."

And this Powerpoint slide underlines the parameters that have to be optimized:



Parameters for mate selection

Male candidates	**1** **2** **3** **4** **…**	**2** **0** **1** **0** **…**
Female candidates	**101** **102** **103** **104** **…**	**1** **0** **1** **1** **…**
Ranking criterion	**1st male mating** **2nd male mating** **3rd male mating** **…**	**5.32** **2.16** **7.64** **…**

Parameters to be optimised

This problem representation works very nicely when coupled to a DE optimizer – as used in Total Genetic Resource Management (See www.xprime.com.au). GenMate, as used at PIC, uses a more involved problem representation that accommodates grouping constraints (eg that certain classes/groups of male can only mate with certain classes/groups of females), while only producing legal solutions that conform to the grouping constraints.

In both these cases, the number of matings per candidate should sum to the overall total target number of matings, or total per group/unit/farm, and this needs to be managed separately. However, given a legal pattern of "No. of uses", a proper mate selection driver will only produce legal mating sets. This greatly increases the robustness and speed of the Mate Selection optimiser.

Example 4: Choosing *p* animals out of a group of size *n*

How should we represent which animals get chosen? We could have a vector with one element for each animal, and values 0 and 1 mean unselected and selected respectively.

Two drawbacks here:

1. We need to constrain to *p* animals chosen. Whether we allow a solution to have a 1 in a certain location depends on the values at other locations.

2. The response surface is not a good shape for efficient climbing, as there are no intermediate values. A "good" animal is either in or out. There is no such thing as a solution being close to letting the good animal in – an attribute for which its progeny solutions might benefit.

A better problem representation involves optimizing a vector of real numbers, one per animal. Here is an example solution at the "Genotype" stage, before the "problem representation filter" is applied:

Animal	1	2	3	4	5	6	7	8	9	10
Real number	6.91	7.43	3.23	1.88	8.97	3.76	6.92	4.46	8.44	2.12

To invoke the filter, rank animals on these numbers:

Animal	5	9	2	7	1	8	6	3	10	4
Real number	8.97	8.44	7.43	6.92	6.91	4.46	3.76	3.23	2.12	1.88

And choose the top *p* animals as the solution to be evaluated. This is the "Phenotype". So if *p* = 4 our vector of numbers represents a solution "Choose animals 5, 9, 2 and 7 to be in the group".

[Note that this could be a good or a bad solution – that is for the objective function to decide. All we are dealing with here is a system to produce "legal" solutions.]

As for the drawbacks noted above.

1. Notice that we do not have to worry about constraining the number of animals chosen to *p*. That happens ~automatically. [In fact we could add *p* to the list of variables to be optimized, and let group size and group membership be co-optimised.]

2. Notice in the example given that animal 1 just lost out on group membership by a small margin of 0.02. However, if this is in fact a good animal that actually belongs in the optimal solution, then this solution will be better exploited. For example, if it is chosen as a template to make a new challenger in DE, then animal 1 has a better chance of making it into new solutions because of its high value in this solution. [A vector of real numbers contains more information than a vector of 0's and 1's, and this is of real value to the optimization system.]

Making a response surface that is better for climbing will be a bit more clear in the next example.

Example 5: Assigning animals into groups

We can base this on the same example. The third row is group number:

Animal	5	9	2	7	1	8	6	3	10	4
Real number	8.97	8.44	7.43	6.92	6.91	4.46	3.76	3.23	2.12	1.88
Group	1	1	1	2	2	3	3	3	4	4

Notice that group sizes can be different – and with a bit more work we can make number of groups and group sizes variable and to be optimised. But let's stick to 4 groups of sizes 3, 2, 3 and 2, as in the table.

In this case we have:

"Genotype": {6.91, 7.43, 3.23, 1.88, 8.97, 3.76, 6.92, 4.46, 8.44, 2.12}

… which filters to …

"Phenotype": (5,9,2), (7,1), (8,6,3), (10,4)

We can take steps to improve the response surface:

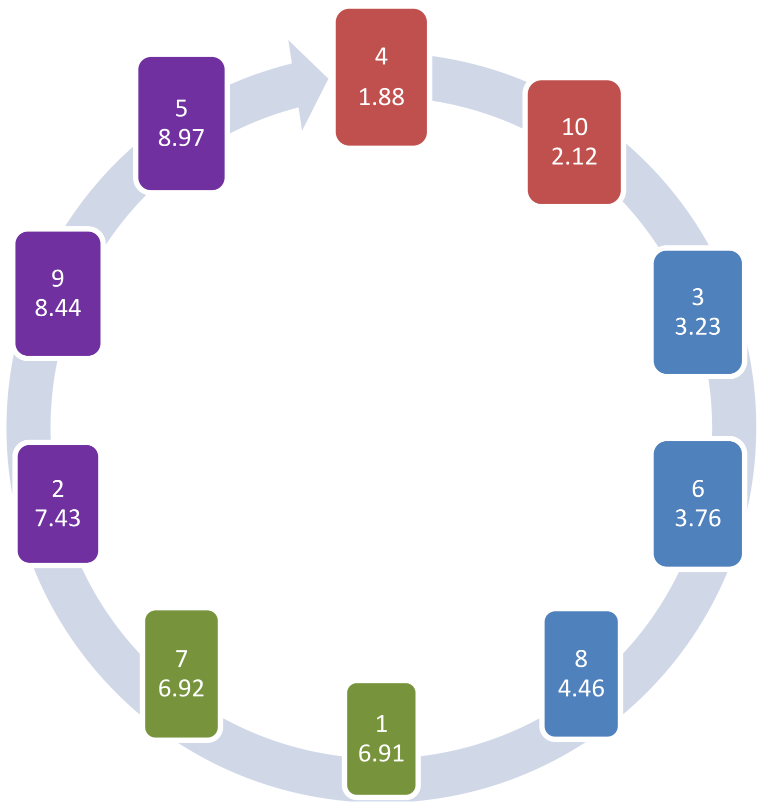
Consider that our example is quite a good solution – the best in the current generation of our optimization engine, and the second best possible solution. Assume that animal 5 should be swapped with animal 4 to get the best solution. To progress to the best solution animal 5 has to be lucky to make the big jump (and both mutation rate and variation among solutions could be low at this stage of convergence), or it has to navigate its way through the possibly deep "valley" of poor solutions while it belongs to groups 2 and 3.

What we can do to help is to order the groups such that groups 1 and 4 sit adjacent to each other. The only basis to do this sort of thing is prior knowledge about the attributes of each group. For example, if groups are farms, we might order these according to pasture quality, average milk yield, mean EBV, or some index of such things. What we are doing here is trying to make a response surface with fewer valleys or less deep valleys. The practical outcome is faster convergence.

The possible downside here is less adventurous coverage of the solution space in early generations (but *after* the optimization has started).

We can take further steps to improve the response surface:

An additional trick that would help solve our hypothetical case is to turn the linear structure into a circle, joining groups 1 and 4:



Here is our solution in a circle, with group shown by color (group 4 is dark red). Upper numbers are animal numbers and lower numbers are their “Genotype” numbers.

To jump directly from group 1 to group 4 we can simply rescale. For example, if the only change were to the Genotype value of animal 5, by adding 2 (from 8.97 to 10.97) then we subtract 10 (taking 10 to be the “12 o’clock value” on our scale) to give 0.97. [Animals with negative numbers get these subtracted from 10, to go anticlockwise on the clock face.]

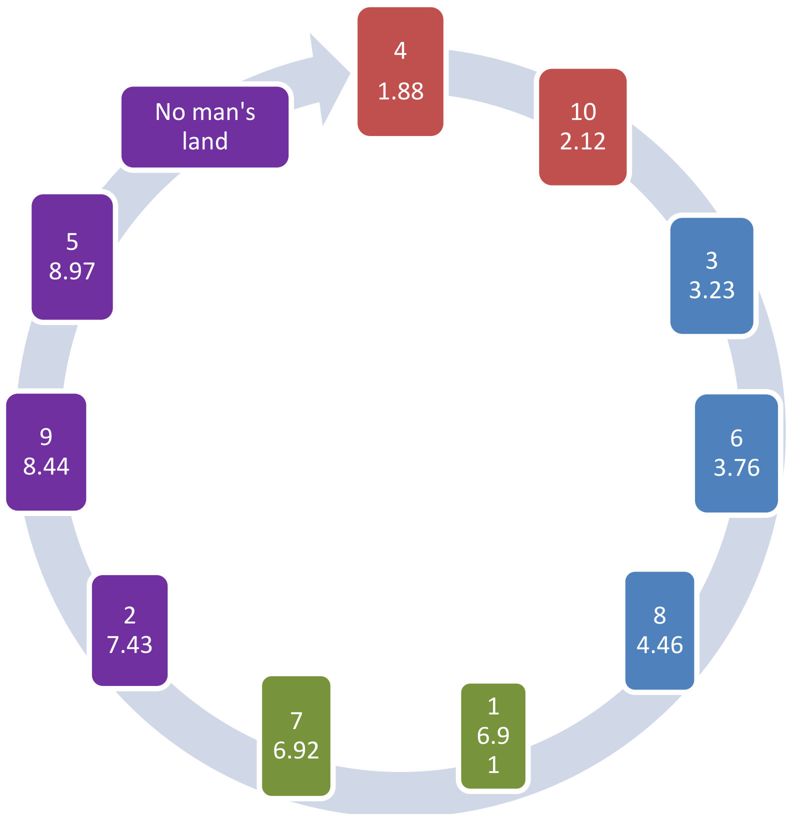
Animal 5 then goes to the bottom of the list and pushes all the other animals up one place – giving some extra work to be done. However, the many small corrections needed can be handled more easily that a big move for that one animal, especially for a problem with many animals.

Such flexibility in the filtering system makes an optimization less prone to getting stuck at a suboptimal solution.

No-Man’s land

The strategy just described can make it too easy for an animal to jump between the two most extreme solutions – if we are close to the optimum, a high proportion of mutations can be deleterious because of this. The solution is no-man’s land.

For example we reserve the values between 9 and 10 as belonging in no man’s land. An animal that ends up here can be allocated to the nearest group (the first or the last).



However, a better strategy is to send him to the nearest group with a probability $d_{farthest}/(d_{farthest}+d_{nearest})$, where d is distance to the boundaries (0 and 9 in our case). Use any chance to use randomness in your algorithm. It does help.

Chapter 7: Managing constraints

Brian Kinghorn

Those that trespass will be penalized. Or otherwise fixed up.

Introduction

Many problems involve constraints on the solutions. Here are some examples that are imposed by simple logic:

1. The number of mates to allocate to a bull should not be negative.
2. The amount of feed to offer an animal should not be negative.
3. The date to wean should not be before birth date.
4. A natural mating bull can be used on a maximum of one farm at a time.

And here are some examples that might be imposed by the stakeholder(s):

1. Maximum breeding herd size should be 200 females mated.
2. The maximum acceptable coancestry among parents in this mating round is 0.1.
3. The maximum number of individuals in a group for pooled tissue quantitative genotyping is ten.
4. The maximum permissible number of genetic marker mismatches per animal (various applications …)

This chapter discusses two key approaches to handling constraints

Ask yourself – do I need this constraint?

Quite often, constraints are requested/applied by stakeholders because they think that solutions that break that constraint will not be so good. That is not "thinking outside the box"! You might get a surprise and discover a solution that breaks your preconceived constraints, but is in fact desirable for reasons that are good and useful (see "Let your computer make you famous." in Chapter 1).

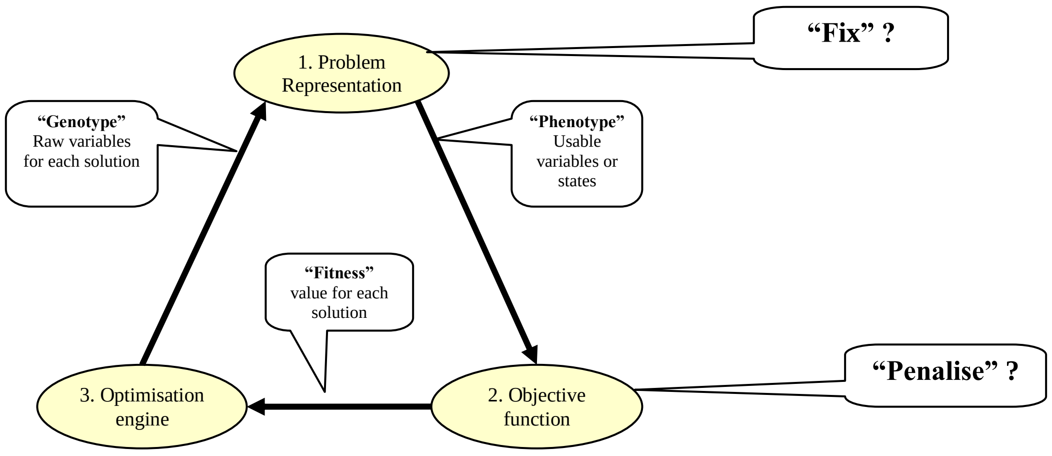
Investigate the impact of removing constraints that are not needed to give a feasible solution. If these constraints were well-founded, then the optimal solution should not break them if the objective function is sufficiently comprehensive.

How to apply a constraint.

There are two approaches to applying a constraint:

Penalise:	Apply penalty in the objective function to solutions that break constraints. The penalty should be sufficiently large to prevent the optimal solution being one that breaks the constraint.
Fix:	If the constraint is on the variables passed to the objective function, you can choose to change these appropriately before passing them to the objective function.

Both of these approaches have their merits, as discussed here:


"Fix" ?
1. Problem Representation
"Genotype" Raw variables for each solution
"Phenotype" Usable variables or states
"Fitness" value for each solution
"Penalise" ?
3. Optimisation engine
2. Objective function

The simplest form of constrain is to set bounds for variables, and these can be easily fixed, as in DE_Demo …

```
For j = 1 To loci
    If allele(j) < MinVal(j) Then allele(j) = MinVal(j)
    If allele(j) > MaxVal(j) Then allele(j) = MaxVal(j)
Next
```

… before allele() gets evaluated in the Criterion subroutine. Setting values exactly to the boundaries is simple but not the best strategy. See "Assigning animals into groups" in Chapter 6.

Table 1 in Chapter 6 gives a more comprehensive example. If we freely optimize the number of uses of each candidate animal, then virtually every solution will be illegal, because the number of uses per sex will only rarely total to the desired number of matings. So, if we use the Penalise strategy, the program will spend ages working on illegal solutions. Better to Fix the variables (number of uses) before hand. This can be done by finding the number of uses over (or under) the total target and implement a repeatable strategy to deal these out to (or withdraw from) the candidates available. This can be done separately for each sex of parent.

"Fixing up" is the better strategy for most "logical" constraints. Fix them up before going to the objective function. However, you must do this in a repeatable manner. If two solutions have exactly the same "unfixed" set of parameters, they should have the same "fixed" set of parameters.

When fixing solutions in this manner there is some danger that the strategy you use prevents good exploration of the solution space. Take care not to "Fix" parameters to the same sort of region, for example at the boundaries of constraints.
The big advantage of the "Penalise" strategy is that it so easy to apply…

```
If (ConstraintBroken) fitness = fitness – 999999
```

… that should cure the problem!

If it is difficult or impossible to logically fix a solution set so that it is rendered legal, or otherwise does not break a constraint, then penalisation can be used. Penalisation is easy to implement, but it makes for much slower convergence if most solutions are illegal.

Hard constrains and soft constraints

In some such cases all solutions break one or more constraints in the first generation(s) of optimization, This is why it can be useful to apply "softer" penalties – so that the optimization engine can give reward to solutions that break fewer constraints, and eventually it finds solutions that break no constraints. You might have to play with constraint penalties to help a complex optimization to "get off the ground". A 'hard constraint' would allocate the smallest possible fitness. That is rarely needed.

Softer constraints can also be useful to get the optimization going well in the middle phase of optimization. For example, there could be a solution that just breaks a constraint, but is much better in other respects than any other fully legal solution. Letting it in to play a role can speed up optimisation. In this case it pays to take account of how much the constraint is broken by – a small misdemeanor is more easily corrected in the game of evolution:

```
If (Para> ParaConstraint) fitness = fitness – 99*(Para – ParaConstraint)
```

If the final best solution breaks a constraint, then this is most easily fixed by changing the "99" weighting on the fly (See Chapter 8).

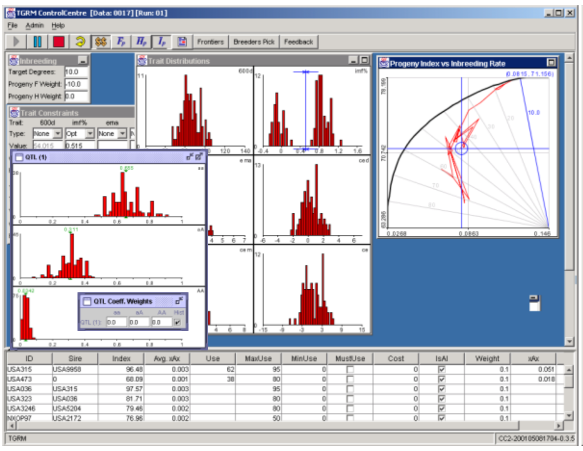
Chapter 8: Changing the goal posts

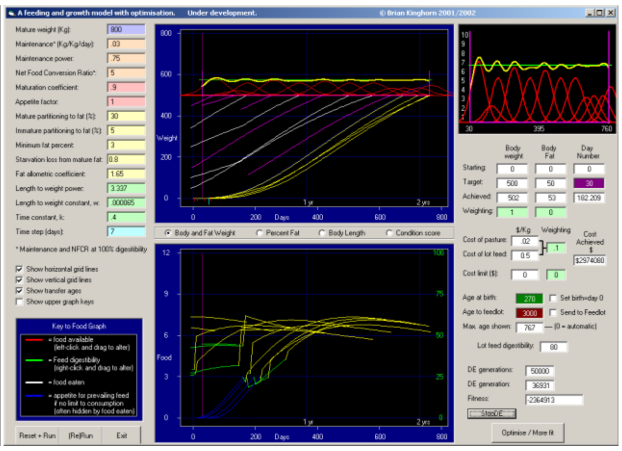
Brian Kinghorn

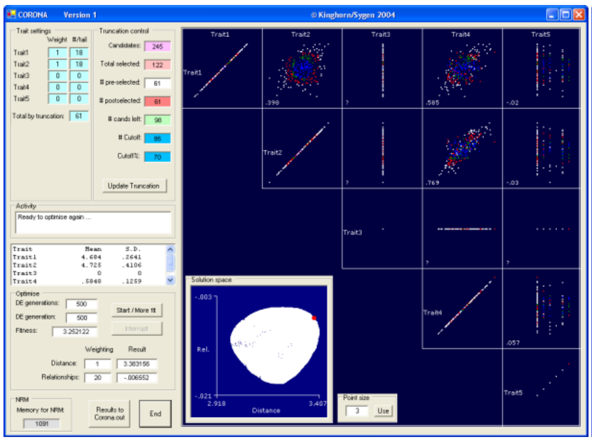
The best direction to take depends on how far you can go in each direction.

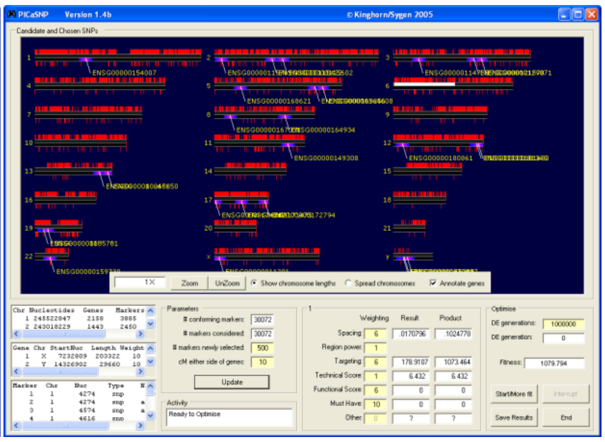
Introduction

While an optimisation is running it is possible to view key aspects of the currently best solution in a visual manner. Here are some example interfaces:


TGRM ControlCentre [Data: 0017] [Run: 01]
Inbreeding
Trait Distributions
Progeny Index vs Inbreeding Rate
Trait Constraints
QTL (1)
QTL Coeff. Weights


A feeding and growth model with optimisation. Under development.
© Brian Kinghorn 2001/2002
Weight
Food
Days
Key to Food Graph
Optimise / More fit





By using a stochastic optimization algorithm, an optimal solution is not immediately ‘calculated’, but intermediate forms are generated as evolution progresses. This is in fact very useful, as the pattern of this progression across the solution space is very informative to the user. It helps in visualizing the domain of possible outcomes (“the Response Surface”), and shows the compromises made when the user is overly dogmatic about achieving preconceived targets or maintaining preconceived constraints ... “Gosh – if I relax my desires a bit on this aspect over here, look how much more I can make on these aspects over there!”

All this is done on the basis of the practitioner's exact prevailing scenario/animals – and it can be very educational indeed.

This sort of exercise often leads the user to doubt the suitability of the objective function initially chosen.

This change of heart rarely comes as a surprise or disappointment. In complex problems the objective function can cover a range of largely disjointed issues, and is often formed with more subjectivity and guesswork than objectivity. It is also true that the best direction to go in depends on how far you can go in each direction – and this does not become evident until an optimisation tool is used flexibly – to "change the goalposts on the fly".

Changing "on the fly"

The process is quite simple. The user can 'change the goalposts' by, for example, altering a constraint or a weighting in the objective function, or setting a target state for a key issue or variable. The software is engineered such that the optimization analysis is interrupted, the objective function changed, all current solutions are evaluated using this new function, and the analysis restarted. This process should be transparent to the user.

Here are some relevant highlights in the DE_Demo example:

Space to store last good solution …

```
    Sub DE()
        Static GoodFit() As Single  ' For reloading new solutions on change
of goalposts
```

Seed the last good solution into the re-started optimisation …

```
        For i = 1 To popsize
            For j = 1 To loci
                parentallele(i, j) = 0 + 10 * Rnd()
                If i = 1 Then parentallele(i, j) = GoodFit(j) ' Make first
population member same as last good solution
```

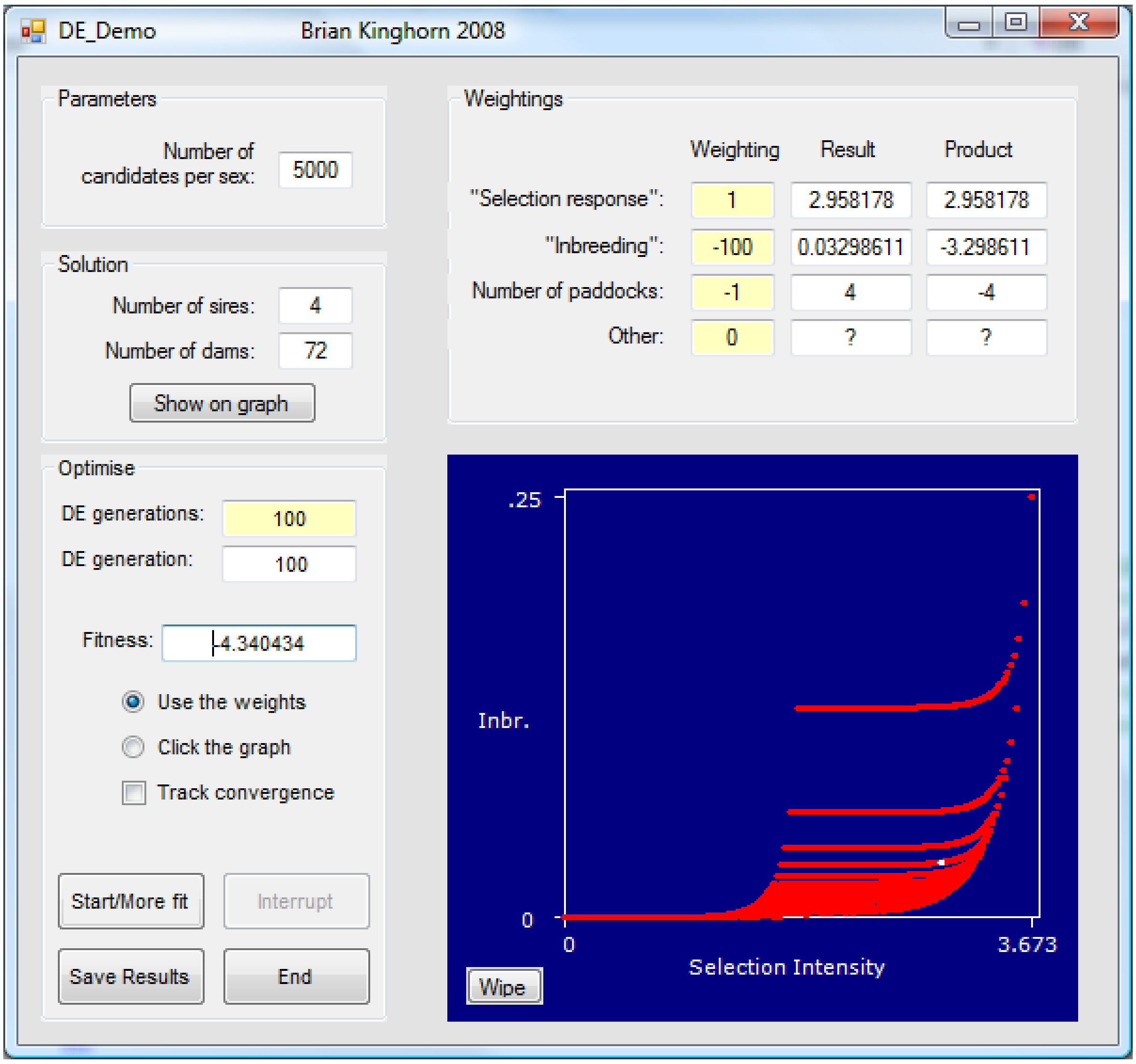
… and note in the code that the fitness value for this "GoodFit" is recalculated using the newly set objective function.

After optimization is interrupted or otherwise finished, store the best solution …

```
        For j = 1 To loci
            allele(j) = parentallele(Best, j)
            GoodFit(j) = progenyallele(Best, j)
        Next
```

How to manage change of direction

The simple approach is to change the weightings in the objective function. Here is an example from the DE_Demo program:


DE_Demo Brian Kinghorn 2008

Parameters

Number of candidates per sex: 5000

Solution

Number of sires: 4

Number of dams: 72

Optimise

DE generations: 100

DE generation: 100

Fitness: -4.340434

Use the weights [✓]

Click the graph

Track convergence

Weightings

	Weighting	Result	Product
"Selection response":	1	2.958178	2.958178
"Inbreeding":	-100	0.03298611	-3.298611
Number of paddocks:	-1	4	-4
Other:	0	?	?

The graphic shows the optimal solution for two key components, selection response and inbreeding, as a small white dot in a sea of red. The red dots have been left by all the candidate solutions that have been tested by the optimization, and this shows the response surface for just those two components – it shows where you can go.

The relevant code in the objective function (subroutine Criterion) is:

```
Fitness = 0
Fitness = Fitness + WeightSel * SelectionResponse
Fitness = Fitness + WeightInb * Inbreeding
Fitness = Fitness + WeightPad * nPaddocks
```

… where eg. `WeightSel` is the weight on selection response and `SelectionResponse` is the selection response that has been calculated for the prevailing solution.

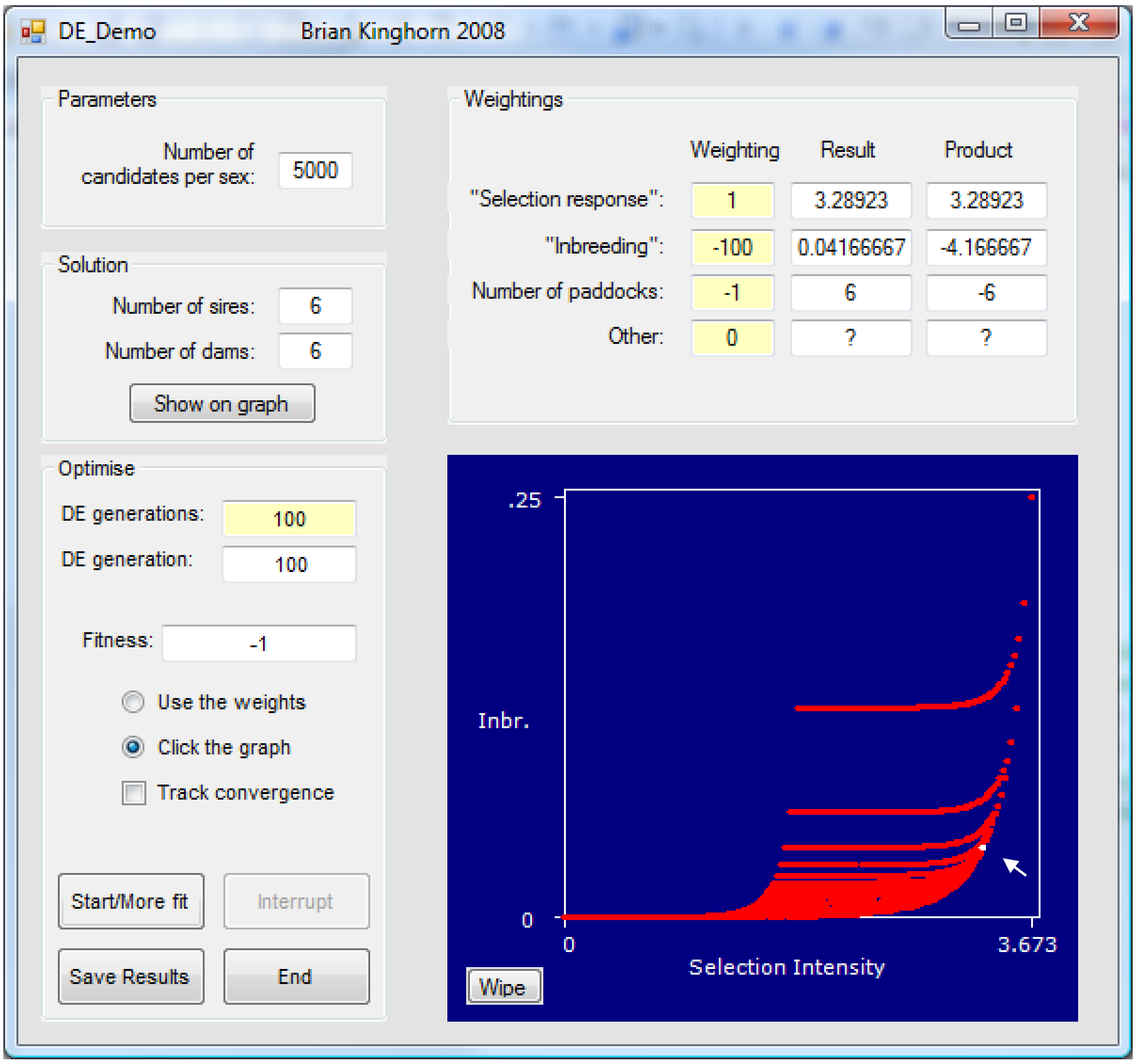
So – how best to change direction? If we want to aim for a solution at a particular place on the response surface, we could try playing with the weightings to try to get there. But there is a much better way of navigating …

GET THE COMPUTER TO DO THE WORK. If you know where you want the white dot to be, make an objective function that will take it there in a direct manner. This is virtually a one-liner …

```
Fitness = -(OptSel - PicX(SelectionResponse)) ^ 2 - (OptInb - PicY(Inbreeding)) ^ 2
```

Here, `PicX(SelectionResponse)` is the X-axis pixel location of the prevailing solution's predicted selection response. `OptSel` is the previously captured X-axis pixel location of the mouse click that the user made on the graph - on or near the response surface. That mouse click also initiated a "Change of goalposts". So Fitness is simply the negative of the Euclidean distance between the solution location and the mouse click location. Maximising Fitness minimizes this distance.

Here is a result:



Parameters

Number of candidates per sex: 5000

Solution

Number of sires: 6

Number of dams: 6

Optimise

DE generations: 100

DE generation: 100

Fitness: -1

( ) Use the weights
(●) Click the graph
[ ] Track convergence

Weightings

	Weighting	Result	Product
"Selection response":	1	3.28923	3.28923
"Inbreeding":	-100	0.04166667	-4.166667
Number of paddocks:	-1	6	-6
Other:	0	?	?

Notice that the white dot is at the edge of the solution space – it has moved to the most extreme point that it can, adjacent to the mouse click location. However, in doing so it has ignored the component objective "Number of paddocks", which settles on a value of 6. This is the same as the number of sires: one sire per mating paddock.

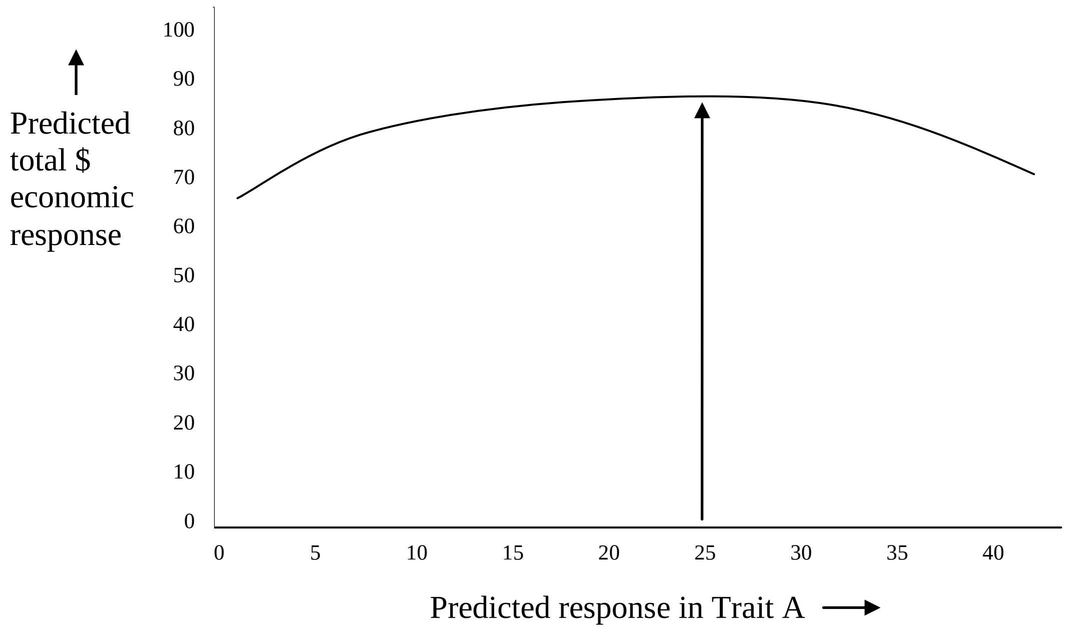
So a simple extension might be to make a tick box that, when checked, constrains the number of paddocks to the value chosen – or two tick boxes to invoke chosen upper and lower limits. Then, with a few clicks on the graph, the user will see the reduced response surface for selection response and inbreeding – reduced because of the constraint(s) on number of mating paddocks.

If you can think of it, you can do it !

> Aside: Note that changing the goal posts is very different from directly fiddling with the parameters to be optimized! Changing the goalposts is much more powerful – the optimization engine does the work of pushing in the direction of the prevailing objective function. We just have to move the target around to get the result we desire. As someone once put it, "Using the tactical approach is like driving a good car in a competitive race. We have control of the steering wheel, accelerator and brakes, and we can drive in a manner that is fast, yet safe, economical and in the proper direction. We no longer need to have our head under the bonnet, monitoring every piston beat, and missing opportunities to overtake or avoid crashes. To make the most of mate selection, we should let it monitor the piston beats, and give it good head to find the best way ahead. There is plenty of opportunity to do test laps of the circuit before committing to a decision - if it does something we do not like, we need to adjust the way we steer it, rather than getting out and pushing it round the track."

Opportunity for changing direction

There can be amazing opportunities to change the solution from what an initial objective function might dictate. Take this hypothetical example:



The selection index calculation tells us that we must aim for a response of 25 units in trait A. But when we look at the graph we can see that we can aim anywhere between about 10 and 35 units of response in Trait A, and suffer little theoretical drop in predicted total dollar response.

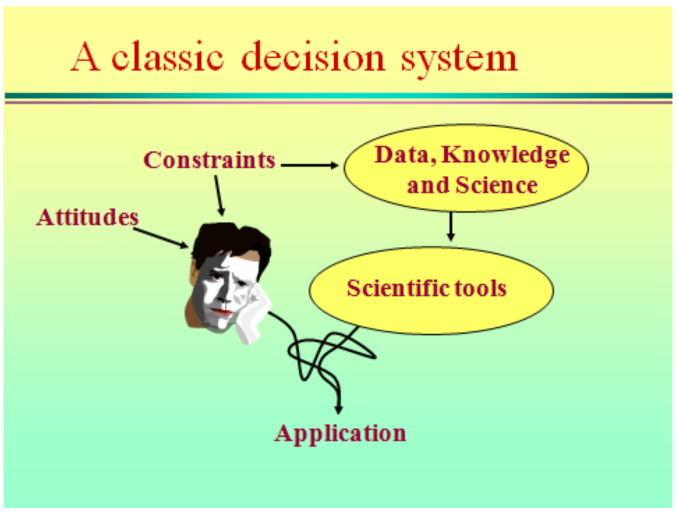
Why would we decide to deviate from the 25 figure? There are many possible reasons, and they mostly have to do with:

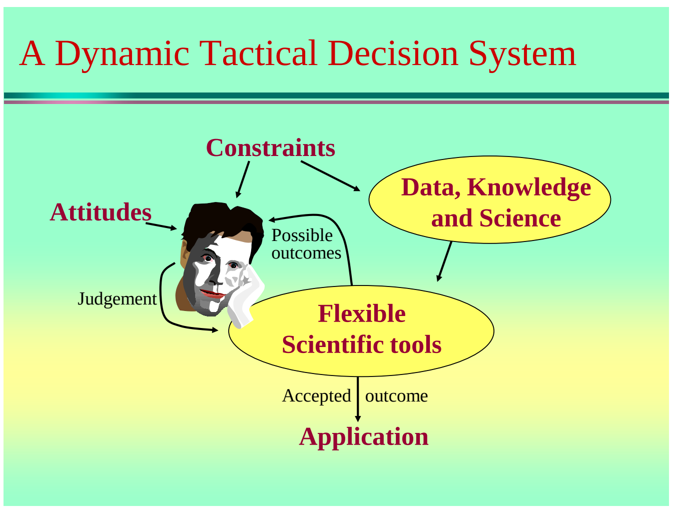
1. Oversimplification of the economic model. We usually assume linearity, when in fact economic contours are more correctly curved. The value of a Kg increase in milk yield depends on what changes are also made in protein percent. A desired gains approach can help resolve this.

2. Probably more importantly, we have additional information that is not available to the analysis. For example, if we adopt the "25" figure, we might also get a negative predicted response in Trait B, or a negative selection index weight on Trait C, and our customers (who are always right) might not understand or like that, and be put off our breeding program. With little compromise in predicted total dollar response, we might be able to keep everyone happy.

Ownership of the solution

With frequent manipulation of the objective function in this way, the user can explore the most exciting parts of the response surface, learn much about the problem in the context of the prevailing example, and develop confidence that the solution finally accepted is a good one. "Ownership" of the accepted solution is a very important phenomenon, especially for thinking practitioners:




A Dynamic Tactical Decision System
Constraints
Data, Knowledge and Science
Attitudes
Possible outcomes
Judgement
Flexible Scientific tools
Accepted outcome
Application

There is a point to underline here. When properly managed, this process constitutes a vehicle whereby scientists can bring the maximum possible power of their science into direct practical application. This is because the 'scientific' components of an objective function will always compete to be exploited as much and as appropriately as possible in the face of compromises, mostly realistic and useful compromises, imposed by practitioners. The alternative is to mix science and practice in a somewhat arbitrary manner, which all too often leaves science both misunderstood and ineffective.

Chapter 9: Improving performance

Cedric Gondro

Evolving evolvability

Introduction

We have covered a wide range of EAs in the previous chapters and some practical considerations on how to get the best out of them. In the next chapter we will cover how to decide that enough is enough - diagnosing convergence. But there still are some pressing issues, for example, what to do when you want the best product and the lowest price!? No, there are no miracles, but you can think of ways to handle multicriteria problems (especially the ones that conflict with each other). And how about making the runs go faster? Your code is already as streamlined as it gets, what can you do? Easy, more computers, of course - parallelization! And when you are fed up of tinkering with population parameters, a bit more mutation, a little less crossover - try self-evolving parameters, let the EA find what the ideal settings are. And our last topic covers what to do when part of your problem is ideal for one method and the other part fits perfectly into another algorithm - hybrid evolutionary algorithms.

Multicriteria optimization

In multicriteria (or multiobjective) optimization the fitness function is even more critical as there usually is no unique solution to a problem but rather a Pareto front of solutions. Since objectives can conflict, improvements in one objective can degrade another one. Different combinations of values for the different objectives can yield the same total fitness; this implies that there is no unique optimal solution to the problem but rather a set of solutions with the same fitness (Pareto-optimal set). More formally a solution is Pareto optimal if there is no feasible set of variables which would improve a criterion without simultaneously decreasing at least one other criterion. An example of such a problem is an optimization problem in microarrays in which a balance must be found between the number of slides (cost constraints) and the experimental questions (information constraints). With few slides the costs are low but there is not enough information to address the experimental questions; on the other extreme there is surplus data but at a very high cost (we will discuss this example in chapter 11).

A common approach to multi-objective optimization is to use a weighting scheme for the different objectives (Zitzler et al. 2000; Van Veldhuizen et al. 2000). There are several approaches to the weighting scheme, these methods range from a fully self adaptive approach – the scheme evolves alongside the EA in the same fashion as mutation parameters in ES – to a user-defined approach where the user modifies weights based on personal preferences. In this case, weightings can be varied in the light of the response surface of component outcomes generated during analysis – as discussed in the last chapter, the best direction to take depends on how far can be gone in each direction.

Multicriteria fitness example - simple scaling

Consider an EA that is trying to simultaneously fit time series data for various correlated functions. A simple fitness function is a measurement of goodness of fit between the predicted values of the model at a given time and the observed values such that

$$f = -1 * \sum_i^n \frac{\sum_j^m (x_{ij} - y_{ij})^2}{\sigma_{x_i}^2}$$

Where the upper term is the sum of the squares of differences between the observed (x_{ij}) and predicted (y_{ij}) values at time point j and the lower term is the variance of the observed data (x_i) for each component (i) of the system. The use of the variance in the lower term scales the sum of squared deviations so that excessive emphasis is not given to a particular equation in detriment of the others. Fitness is treated as a maximization problem with worse solutions having highly negative values (due to the minus one multiplier) and the better organisms having values closer to zero, which is the maximum fitness.

A suggestion

When working with multicriteria problems it is worthwhile storing all equivalent solutions either to make a decision based on any available additional information (or even a whim!) or get a better understanding of the potential scope of solutions. A good tutorial for multicriteria optimization is found in Coello Coello et al (2007) and Zitzler et al. (2004).

Parallelization

Probably the greatest limitation to the use of EC methods is the dimensionality problem. As the number of variables increases the computational effort can increase exponentially. EAs cannot compete in terms of speed with *strong harm* approaches. But by their very nature EC methods are well suited for parallelization - they are commonly referred to as *embarrassingly parallel* due to the ease with which they can be split into smaller problems. This ease meets heads on the current trend of low cost clusters and multi core processors and can potentially shift the time-cost balance since parallelization of deterministic (e.g. dynamic programming) algorithms is not a trivial task.

The main constraints to parallelization are not the EAs but getting processors/computers to communicate with each other. On up side, higher level APIs are making parallel programming easier. Under Windows WMI can be used to connect across computers and under .NET the *remoting* library is very handy.

In EC terms, an algorithm can be parallelized by simply running independent jobs in each machine (yes, this still is parallel computing!). Due to the stochastic nature of EAs multiple runs of a job are always mandatory to ensure reliability of results. It can be very time saving to run all repeats at the same time (especially if the run takes a week or two).

More realistic parallelization can be achieved at the population, individual or fitness level through different models. The two main models are:

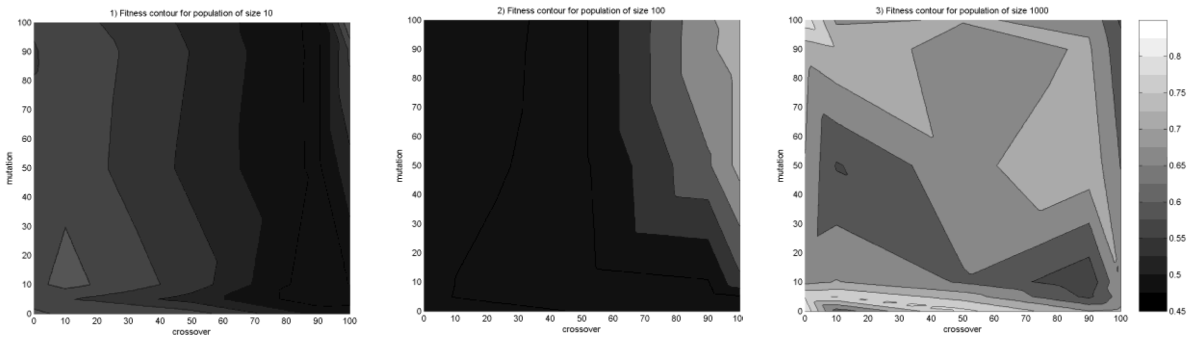
- Master-slave model: run the population on one machine and calculate the fitness on other machines. Here the population manipulations (the EA per se) runs on a single node but the fitness evaluation (which in more cases than not is the most demanding task) is spread out across the computational resources. This model is particularly efficient with overlapping generations since there is no need to keep the population synchronized.

- Island model: each processor runs its own population and from time to time *migrants* move from one machine to the other. This model allows different areas of the search space to evolve concurrently whilst still allowing a certain level of gene flow which will have smaller or larger influence in the acceptor population depending on the differences between fitness. If the migrants move between neighbors the model is termed *stepping stone*.

A complete overview of parallel EAs is given in Nedjah et al. (2006).

Self-evolving parameters

Parameter setting has always been a concern in EAs. The methods are quite robust to parameter settings but nevertheless they can influence convergence times and even define if the algorithm will get entrapped in a local optimum or not, as illustrated in the figure below. Unfortunately, except for simple scenarios there are no formal methods of determining adequate parameters.



Fitness contours for populations of size 10 (1), 100 (2) and 1000 (3).The lower fitness values represent better results.

An alternative is to concurrently evolve the solution and the parameters. In effect this is what DE does (remember the Differential?). Self-evolving parameters are part of evolutionary strategies and evolutionary programming, but not so common in GAs and GPs.

A simple strategy for GAs and GPs is to ensure an adequate balance between new mutations and crossover - recall that mutation creates variability and crossover combines it. This balance can be modified dynamically by changing (evolving!) the mutation and crossover probabilities between generations based on the fitness gain scaled by the population's fitness variance, evolving these parameters alongside the population. If tournament selection is used, the tournament size can easily co-evolve as well. An entire book devoted to parameter setting in EAs is found in Lobo et al. (2007).

Hybrid Evolutionary Algorithms

When we discussed Genetic Programming it was quite clear that the methods are well suited for building structures, but less than ideal for parameterization. Consider for example a problem in which the objective is to discover the underlying function and also the correct parameters that explain a given dataset. Ideally one would want to use a method such as Gene Expression Programming and explore its capacity to construct model structures but instead of parameterization with GEP, use a more robust algorithm such as Differential Evolution for the parameter optimization. Hybrid algorithms are common practice in EC, there are many hybrids out there *in the wild* and of course you can always make your own. Here we will illustrate with a hybrid between GEP and DE.

Hybrid Differential Evolution and Gene Expression Programming Algorithm

A simplified version of the hybrid algorithm is depicted below. Initially random values are assigned to a given set of variables either within the bounds of a set of constraints or with fully unconstrained values (in our tests the later tend to increase the search times). The algorithm iterates between GEP and DE by a user-defined number of iterations.

For the first iteration a random population of models is generated using the initial variable set. GEP is used to select better models. At the end of the GEP run the best model is selected, simplified through a bloat reduction method and used as the model for the DE to optimize the variable set. At the end of the DE run if the optimized variable set has a higher fitness than the original set it replaces it.

From the second iteration onwards, the GEP run will use the optimized variable set. The initial populations of GEP and DE are randomly generated apart from chromosomes zero, into which is respectively copied the current best model and the current best variable set, thus ensuring that the next round starts at least at the current best solution.

Algorithm of the hybrid method using Differential Evolution and Gene Expression Programming.

```
Initialize random values for variables within a set of constraints
Do until (termination criterion)
{
        Iteration i
        {
                GEP
                Initialize random population of models
                Replace chromosome 0 with best model
                Do until GEPGeneration = GEPMaxGenerations
                {
                        Select
                        Crossover
                        Mutate
                        Evaluate
                        Replace
                        Generation++
                }
                If (GEP Best Model Improves Fitness)
                        Replace model with best model from GEP
                Else Keep original model

                Bloat Reduction Method

                DE
                Use Best Model to optimize variables
                Initialize random population of variables within constraints
                Replace chromosome 0 with best variables
                Do until DEGeneration = DEMaxGenerations
                {
                        Select
                        Crossover
                        Mutate
                        Evaluate
                        Replace
                        Generation++
                }
                If (DE Best Values Improve Fitness)
                        Replace variables with best values from DE
                Else Keep original variables
                i++
        }

}
```

Chapter 10: Diagnosing convergence

Brian Kinghorn

At the end of the rainbow is a pot of gold.

Introduction

When will we ever get there? The trouble is that for most practical problems we would not know when we have arrived.

With some searching around the region of the current solution we might be able to detect that we are either at or very close to an optimum. But it could be a local optimum, with a valley to be crossed to get to the global optimum.

In practical situations, we have to stop evolving and accept the result at some stage. In most cases it is not critical to find the exact best solution – one that is pretty close to that will be sufficient – a "satisficing solution". "Satisficing is a decision-making strategy which attempts to meet criteria for adequacy, rather than to identify an optimal solution" http://en.wikipedia.org/wiki/Satisficing.

Criteria for stopping

There are many ways of deciding when there has been sufficient convergence. This chapter suggests a small number of approaches for accepting convergence. Some or all of these can be applied simultaneously in practice. They are listed in the order that they appear in the demonstration program that will be used for illustration:

Criterion 1: The solution must exceed a specified percentage of the current predicted asymptotic maximum solution. This is described below.

Criterion 2: No improvement over the last p_{nc} percent of n_{last} generations, where n_{last} is the last generation in which the best solution improved on the best solution in the previous generation, and p_{nc} is a percentage. It can be >100%.

It is sensible to make p_{nc} a function of n_{last}. For example, if we fixed p_{nc} at 20%, then we would say "Stop!" at generation 12 if the last improvement was at generation 10. This is not sensible, whereas we would say "Stop!" at generation 24,000 if the last improvement was at generation 20,000. This is more sensible. Here is a suggestion that 'tunes' p_{nc} to 20,000 generations:

$$p_{nc}(\text{corrected}) = p_{nc} * \text{sqrt}\left(\frac{20{,}000}{\text{n}_{\text{last}}}\right)$$

If we use this, then we would say "Stop!" at generation 100 if the last improvement was at generation 10 – more sensible.

Criterion 3: No improvement for a specified fixed number of generations. This is an extra safeguard against premature stopping.

Criterion 4: No fewer than n_{min} generations in total.

Criterion 5: No more than n_{max} generations in total.

The actual decision to stop can depend on meeting various combinations of these criteria, as described later.

A predicted asymptotic maximum solution

The concept is quite simple here. Fit a non-linear regression of best solution in each improved generation against generation number, and use this to predict the asymptotic maximum solution. An 'improved generation' here is one in which the best solution is better that that in the previous generation. However, in practice we should include the most recent generation, even if it does not give an improvement.

The non-linear fit is a bit of an art, because we never have a "correct" model for the form of the non-linear function to be used. You might find a better recipe – but here is one that has worked well for me:

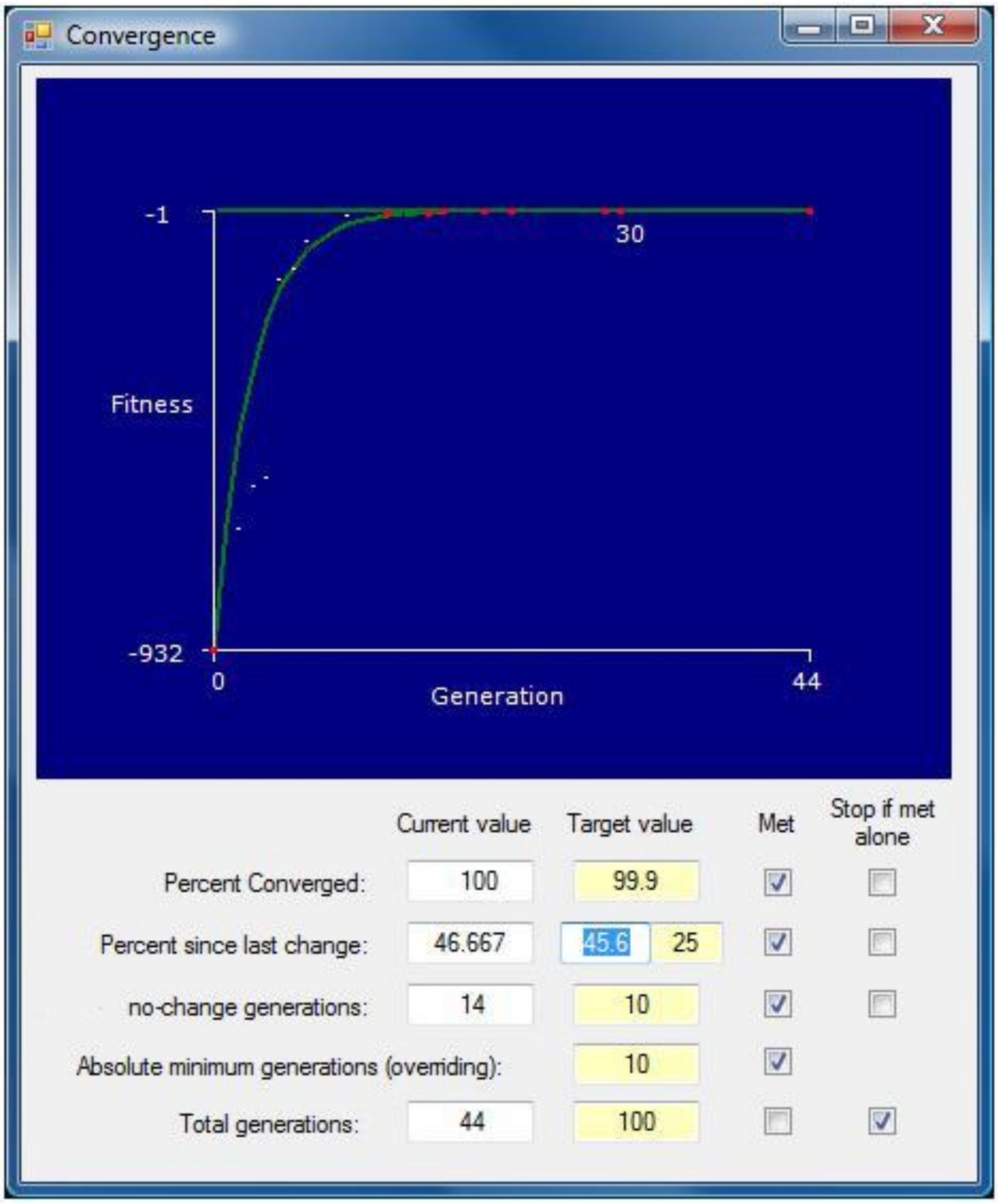
$$S_g = S_{max} * \left(1 - \mathrm{e}^{-k.g^b}\right) + \text{error}$$

where S_g is the best solution at generation g, S_{max} is the asymptotic maximum solution, k is a rate constant, and b is a bender of the exponential function, which I treat as a fixed constant. I find that b=½ or 1 has been reasonable – choose a value that seems to give a good fit for your own scenario, easily seen if you have graphic output, as below. S_{max} and k are to be found to give the best fit to the data (g and S_g), probably by minimizing $\sum(S_g - \hat{S}_g)^2$. The fit is facilitated by forcing the regression to pass through both the origin and the last point. This defines S_{max} for a given k, such that we only have one degree of freedom to adjust (k) to minimize $\sum(S_g - \hat{S}_g)^2$. (See appendix for some code).

Because the fit of the curve is most critical towards the asymptote, I only use (or 'track') the last nTrackJumps = 8 number of improvements (or 'jumps'), plus the origin, in the exponential fit. The most recent generation tested can be an exception here – it may not constitute a jump, and as generation number increases with no further jump in fitness, the predicted asymptote reduces to approach this stable fitness value.

Example

The next diagram shows an example of the convergence module that is attached to the DE_Demo program. In this example run, convergence is deemed to have been achieved in generation 44, and generation 30 is the last generation in which an improvement in fitness occurred.



	Current value	Target value		Met	Stop if met alone
Percent Converged:	100	99.9		✓	☐
Percent since last change:	46.667	45.6	25	✓	☐
no-change generations:	14	10		✓	☐
Absolute minimum generations (overriding):		10		✓	
Total generations:	44	100		☐	✓

Note that there are nTrackJumps = 8 red points in addition to the red point at the origin. These 9 points have been used to make the exponential fit (the green curve) and hence the asymptotic prediction of maximum fitness (the green line at Fitness = -1).

Note also that four component criteria have been met – four ticks under "Met". Looking at the last column of tick boxes, you can see that "Total generations" is the only individual criterion that on its own will stop the program (when Generation = n_{max} = 100 is reached). However, in this case n_{max} is not reached, because all other criteria have been met. In the case of "Percent since last change", the target value has been scaled to:

$$p_{nc}(\text{corrected}) = p_{nc} * \text{sqrt}\left(\frac{\text{n\_max}}{\text{n}_{\text{last}}}\right) = 25 * \text{sqrt}\left(\frac{100}{30}\right) = 45.6\%$$

"Percent converged" is rounded up to 100% from a value >= 99.9995%. Such figures can give false confidence, which is why a mix of criteria is important.

The first three criteria are volatile, in that they can be met, and then not met as evolution progresses, because of the finding of new better solutions. This is why scaling of the "Percent since last change" criterion is important, to help avoid premature stopping.

NB: When changing the objective function ("Changing the goal posts") on the fly, be sure to reset the generation counter for the purposes of diagnosing convergence.

No guarantee!!

There is no guarantee that convergence has been truly met, unless you know the maximum from an algebraic approach or an exhaustive search. What can help is to observe the best solution achieved, and realize (if true) that it has all the properties that you think it should have – that you are satisfied with it. That it is a "Satisficing Solution".

Appendix

Here is a version of the exponential fit in Fortran. VB.NET is available within the DE_Demo program code.

```
module commonbits
       Integer      :: Asymptote_npoints, Asymptote_npointer, Asymptote_maxpoints
       Real(Kind=4), Allocatable  :: Asymptote_data(:,:), PredOpt
End module commonbits


…
Allocate (Asymptote_data(2, 0:Asymptote_maxpoints))
Asymptote_data(1, 0) = <Fitness in first generation >
Asymptote_data(1, >=1) = <Fitnesses as deviations from Asymptote_data(1, 0)>
Asymptote_data(2, >=1) = <Generation numbers>
Call PredictOptimum(GenerationPointer(nTrackJumps),Pred_k, Pred_A, Pred_ExpBender)
PercentConverged=100.*(Fitness-Asymptote_data(1, 0))/(PredOpt-Asymptote_data(1, 0))
…

subroutine PredictOptimum(FromCount,k,A,ExpBender)
       USE commonbits
       implicit none
       integer              ::  i, j, FromCount
       real(Kind=8)         ::  k, kStart, A, Pred, SSE, SSEold, Jump, ExpBender

       kStart = -10*log(1-.5)/1000 ! Not critical. .9 converged after 100 NewGens
       k=0
       Jump=1
       ExpBender=0.5
       SSE = 1.0E+29
       SSEold=SSE+1
       do j=1,50
              if(SSE>SSEold) Jump = -1.*Jump/4.
              SSEold=SSE
              k=k + kStart*Jump
              if(k<0.001)k=0.001
              ! Find A that uses this k but fixes the current point to be hit...

              A = Asymptote_data(1, Asymptote_npointer)/&
                  (1-exp(-k*Asymptote_data(2, Asymptote_npointer)**ExpBender))
              SSE=0
              do i = 1,Asymptote_npoints
                     if(Asymptote_data(2, i)>=FromCount) then
                            Pred = A*(1-exp(-k*Asymptote_data(2, i)**ExpBender))
                            SSE=SSE + (Pred - Asymptote_data(1, i))**2
                     endif
              enddo
       !      Print'(i3,10f15.4)', j, k, A, SSE, Jump
       enddo
!      Print*, k, A, Asymptote_npoints
       PredOpt=A+Asymptote_data(1, 0)
end subroutine PredictOptimum
```

Chapter 11: Applications in bioinformatics, systems biology and Artificial Life

Cedric Gondro

Up there the skies are blue

Introduction

Evolutionary Computation is being widely employed to solve bioinformatics problems. And this is not particularly surprising; bioinformatics problems are complex, noisy and non-linear – the perfect setting for Evolutionary Computation to thrive. Some of the current efforts include sequence reconstruction from shotgun sequencing data, multiple-sequence alignment of protein or DNA sequences, tertiary protein folding inference, identification of coding regions in DNA sequences, microarray data clustering and reconstruction of metabolic and genetic pathways. Fogel and Corne (2003) provide a comprehensive review of the current research topics in EC applied to Bioinformatics.

In this chapter we will focus on mentioning some applications and discuss the implementation approach used and some tips (the ones that worked for us!). The idea is to give you a taste of what can be done and how to do it.

Bioinformatics – multiple sequence alignment

Multiple sequence alignment (MSA) plays an important role in molecular sequence analysis. An alignment is the arrangement of two (pairwise alignment) or more (multiple alignment) sequences of ‘residues’ (nucleotides or amino acids) that maximizes the similarities between them. Algorithmically, the problem consists of opening and extending gaps in the sequences to maximize an objective function (measurement of similarity).

A simple genetic algorithm works fine here (Gondro and Kinghorn 2007). Genetic algorithms are well suited for problems of this nature since residues and gaps are discrete units.

An evolutionary algorithm cannot compete in terms of speed with progressive alignment methods which are the most common method used for sequence alignment, but it has the advantage of being able to correct for initially misaligned sequences; which is not possible with the progressive method.

EAs can have an important role for MSA because the alignment scoring functions still constitute an open field of research, Since there is a clear distinction between objective function and EAs, they make extending and/or replacing objective functions a trivial task.

Population Initialization and structure

A group of sequences to be aligned consist of *n* sequences of DNA of different lengths. An alignment is represented as a matrix with *n* rows in which each row represents a sequence. Each position in the array is occupied by a symbol from the alphabet {*A*,*T*,*C*,*G*,*–*} in the case of nucleotides. Gaps are represented by the symbol '–'. Evidently, the order of the nucleotides in the sequences has to be preserved and is only interspaced with gaps.

Each organism in the GA consists of a candidate alignment. The organisms of the initial population are generated from pairwise alignments of all the sequences. Initially, all global pairwise alignments between the sequences are computed with dynamic programming using the Needleman-Wunsch algorithm (seeded population). For each sequence one of the pairwise alignments corresponding to that sequence is randomly selected to form the organism. At the beginning of the sequence, a randomly defined number of gaps is placed to allow for some expansion.

Even accounting for the overhead to calculate the pairwise alignments, an initial population seeded from pairwise alignments is overall faster and greatly improves the scores with reduced convergence times when compared to randomly generated ones. With this approach the initial population starts with a high mean fitness. An alternative approach is to include a pre-alignment which is inserted into the initial population. This can lead to stagnation at a local optimum but can be used to fine tune alignments obtained through progressive methods.

The GA uses steady-state generations and selection is elitist with tournament selection. The winner of the tournament remains in the population and the loser(s) are replaced by its (their) offspring. Crossover uses the tournament winner and each of the losers to generate an offspring which will replace the respective loser in the population.

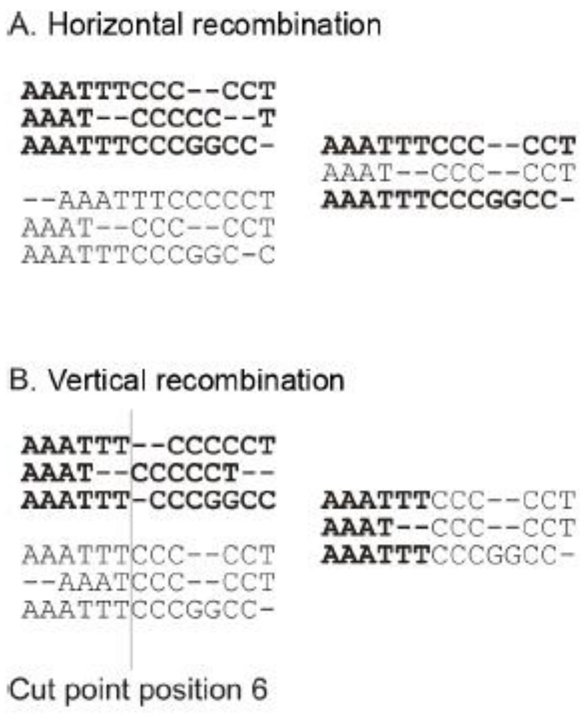
Search operators

An MSA is defined by the position and size of the gaps in the sequences. From an EC perspective it can be viewed as a "gap-shuffling" operation. Search operators can be: recombination between parents to produce offspring alignments and gap mutations.

Crossover can be:

1. Horizontal, which builds an offspring by randomly selecting each sequence from one of the parents.

2. Vertical, which randomly defines a cut point in the sequence and the offspring is built by copying the sequence from position *1* up to the cut point from one parent and from the cut point to the end of the sequence from the other parent. With vertical recombination the positions of gaps have to be accounted for to ensure integrity of the structure of the sequences.



Horizontal (A) and vertical (B) recombination in the MSA genetic algorithm. (A) Offspring are generated by randomly selecting entire sequences from either of the parents. (B) A randomly defined cut point splits the sequences of the parents in two; offspring are generated by selecting one substring from each parent.

Mutation operators can only act on gaps: open a new gap, close an existing gap, extend gap size or reduce gap size. We used three mutation operators to manipulate gaps. To open a new gap a block mutation operator was used - a position in a sequence is randomly selected and a block of gaps of variable size is inserted into the sequence. For gap extension, a block of gaps is randomly selected and an extra gap position is added. The third mutation operator is gap reduction, a block of gaps is randomly selected and a gap position is removed; the probability of a gap position being removed is an inverse function of the size of the gap, meaning that the smaller the number of gap positions the higher the probability that a gap position will be removed. If the selected gap block consists of a single position, it will always be removed, and the gap will be closed.

Bioinformatics – optimization of cDNA microarray experimental designs

The cDNA microarray is an important tool for generating large datasets of gene expression measurements. An efficient design is critical to ensure that the experiment will be able to address relevant biological questions.

Microarray experimental design can be treated as a multicriteria optimization problem. For this class of problems evolutionary algorithms (EAs) are well suited, as they can search the solution space and evolve a design that optimizes the parameters of interest based on their relative value to the researcher under a given set of constraints. We used EAs for optimization of experimental designs of spotted microarrays using a weighted objective function (Gondro and Kinghorn 2007).

Even though the application here is microarray design, the concepts can easily be extended into other design problems.

Design problem

Since a cDNA microarray is essentially a comparison between two samples, how these are paired in an experiment affects which comparisons can be made. Comparisons of interest should be closely connected in the design, preferably on the same array, thus removing the variability between slides which is greater than the variability within slides. Typically the correlation of measured intensities between duplicated spots on the same slide is around 95%; dropping to between 60% and 80% on different slides. An efficient design will ensure an unbiased dataset with the effects of interest (sample × gene interactions) not confounded with other sources of variation. In EA terms this is a combinatorial problem. The problem can be treated as an assignment problem with three optimization parameters: (1) number of arrays (slides), (2) allocation of hybridization pairs to the slides and (3) dye allocation for the variety pairs on the slides.

There is no single optimal design. The experimental design should balance three basic principles:

1. balance among the factors – particularly dyes
2. use approximately the same sampling of varieties
3. reduce the distances between pairs of varieties – especially the ones of interest which should preferably be hybridized on the same array allowing for direct comparisons.

Population (design) representation

Each candidate design in the EA population is represented as a numeric array of index $[n, m]$ where n=2 (block size) corresponds to each one of the channels in the microarray, thus defining the labeling dye of a sample. Index m is a constraint on the maximum number of hybridizations allowed. The experimental samples (varieties) are assigned a unique numeric identifier s_i in the array. In this simple manner complex designs can be easily represented with the hybridization pairs defined by position m and the dye colors by n, as depicted in the following figure. An additional dimension is added to the array, corresponding to the population size of the EA.

To allow for variable design sizes (different number of slides) a vector is used to control the effective experimental size. The effective size (S) is an integer denoted as $S \in [s-1, m]$, where s is the number of samples in the study and m is the maximum allowed number of arrays. The minimum number of slides is defined as $s-1$ since this is the minimal criterion for connectivity. S is an evolvable parameter included in the EA.

Green Channel (Cy3) - n_0	s_0	s_1	s_2	s_3	s_4
Red Channel (Cy5) - n_1	s_1	s_2	s_3	s_4	s_0
Slide	m_0	m_1	m_2	m_3	m_4

A candidate design in the EA population with 5 varieties (numbered s_0 – s_4) used to represent a loop design. Dimension *n* is used for dye assignment and dimension *m* represents the number of arrays in the design.

Selection

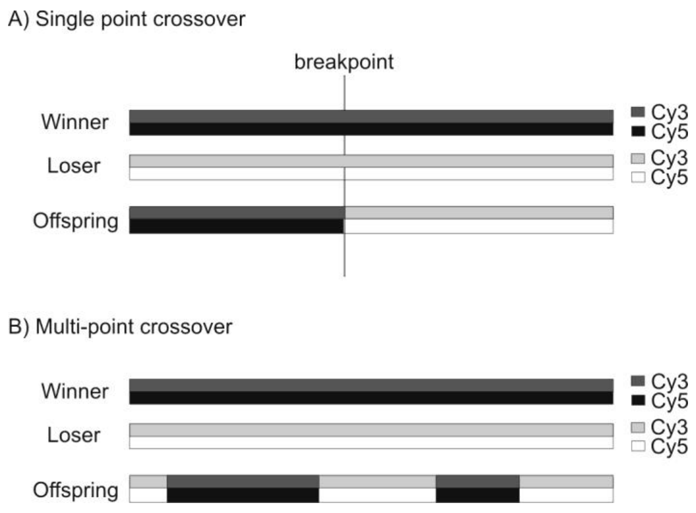
The EA uses steady-state generations and tournament selection with elitism. The elitist approach ensures that the best solution is always retained in the population. In each selection round, *t* (tournament size) candidates are randomly selected from the population. The candidate with the best fitness is the winner of the tournament and remains in the population, whilst the loser(s) is (are) replaced by new candidate solutions (offspring). This means that for a tournament of size *t*, *t*–1 new offspring will be created at each selection round.

Crossover

Two operators were used:

1. Single-point crossover selects a random uniform breakpoint between zero and the maximum length of the design (along the slides – array dimension *m*). With an equal probability, an offspring is generated by selecting one parent between the winner and each of the tournament losers in turn; the selected parent is copied into the offspring from position zero in the array up to the breakpoint. The remainder of the offspring is built by copying the other parent from the breakpoint until the end of the array.

2. Multi-point crossover, the entire tournament loser is copied into the offspring and, with an equal probability *(1/3)*, a number of blocks between *1* and *3* of variable sizes are selected from the tournament winner and grafted into the offspring in the same position they held in the tournament winner. The size of each block is chosen from a random uniform value between a minimum of *1* and a maximum of *1/3* of the length of the array. In terms of the design, each block consists of a variable number of slides with their respective hybridization pairs and channel assignments.



Crossover operators. A) ***Single point crossover*** **– offspring is generated by copying one of the parents from the start of the array up to the breakpoint and copying the other parent from the breakpoint to the end of the array. The parent used as a starting point is randomly selected with equal probabilities. In the illustration, the tournament winner was selected as the starting point. B*) Multi-point crossover*** **with 2 blocks – the tournament loser is copied into the offspring and blocks of variable size from the winner are grafted into the offspring.**

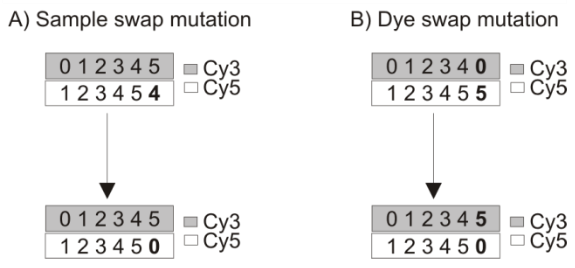
Mutation

We used three mutation operators:

1. Sample swap – each position (each sample s_i) in the new candidate, along both dimensions (n – channels and m – number of slides), is tested for a user-defined uniform probability of the mutation occurring. In each position selected for mutation the current sample is replaced with a different one which is randomly selected from the available sample pool with an equal probability for each sample. This operator ensures that the entire solution space is accessible for exploration.

2. Dye swap – each position along dimension m (slides) is tested for a dye swap mutation event; for those positions in which the mutation occurs the two samples are swapped across dimension n (channels) in the array; that is, the same samples are still on the same slide but the dye assigned to each sample has been inverted. This operator is important to explore designs that are evenly balanced.

3. Effective size mutation – assigns a random number of slides as the effective size of the design. The mutated S only replaces the current one if it improves the fitness of the offspring. This demands three fitness function calls, two with the original effective sizes from each of the parents and one with the new value. The S that yields the highest fitness is assigned to the offspring. Designs are very sensitive to changes in the number of slides, for this reason the overhead of making additional fitness calls for each offspring is justified.



Mutation operators. A) *Sample swap mutation* – a sample in the array (in bold) is replaced by a new sample, with this single change the design becomes a loop design. B) *Dye Swap mutation* – the dye channels of a hybridization pair are swapped (in bold), the change turned the unbalanced design into a balanced one.

Systems biology – model reconstruction and parameterization

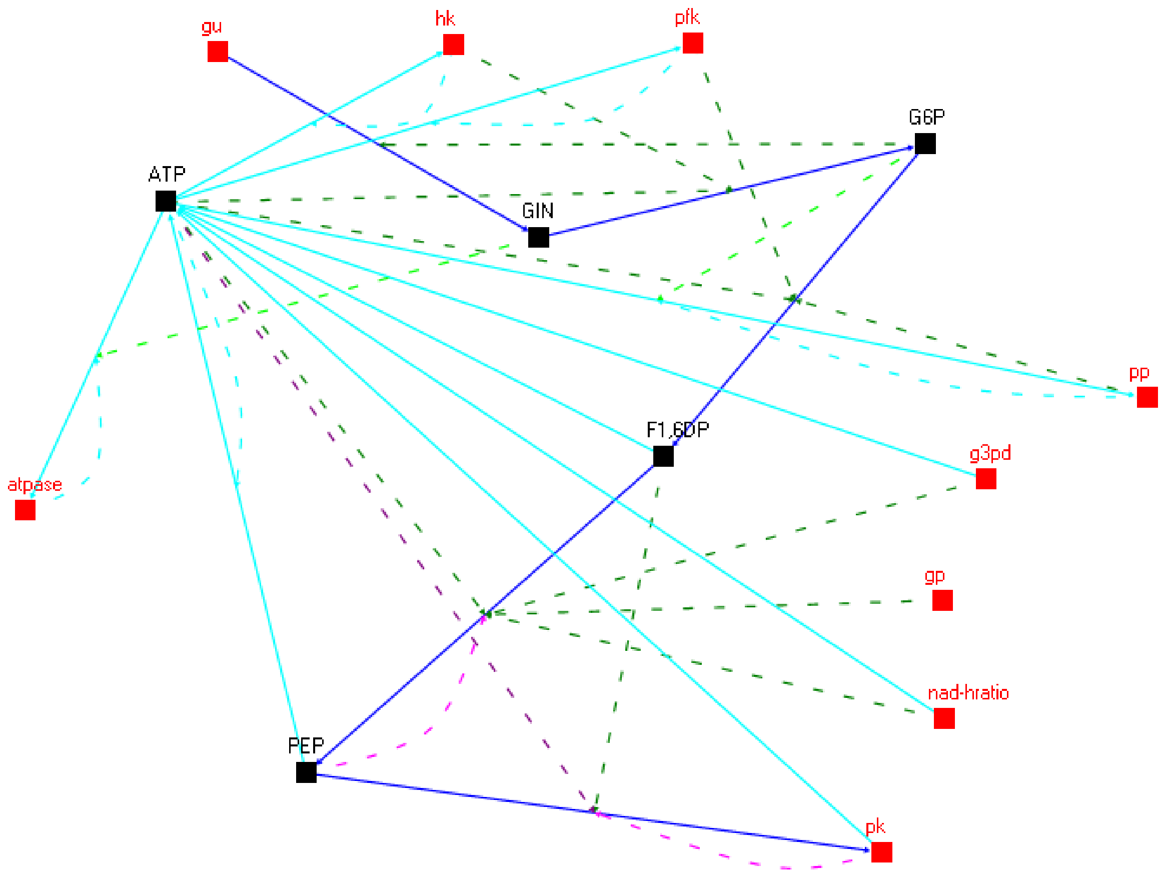
Systems Biology is the exponent representative of the shift that life-science research is undergoing. Focus is changing from a reductionist approach centered at identifying and understanding the function of individual components to a holistic approach geared towards an integrative understanding of biological systems. Notably a system-oriented approach is not viable without relying on knowledge derived from reductionist studies, so the approaches should not be seen as conflicting but rather as complementary. The need for an integrative approach is clear from the fact that single levels of information cannot fully explain the dynamics of biological processes. To understand the whole one must study the whole.

Modeling of biological processes has become an important research topic to understand processes from a systems point of view. Driven by the ever-growing availability of data – with gene expression data a major source – genetic and biochemical models try to explain how the components and their interactions affect the behavior of the entire system. The most common approach to modeling is through differential equations; which include S-Systems (Voit 2000).

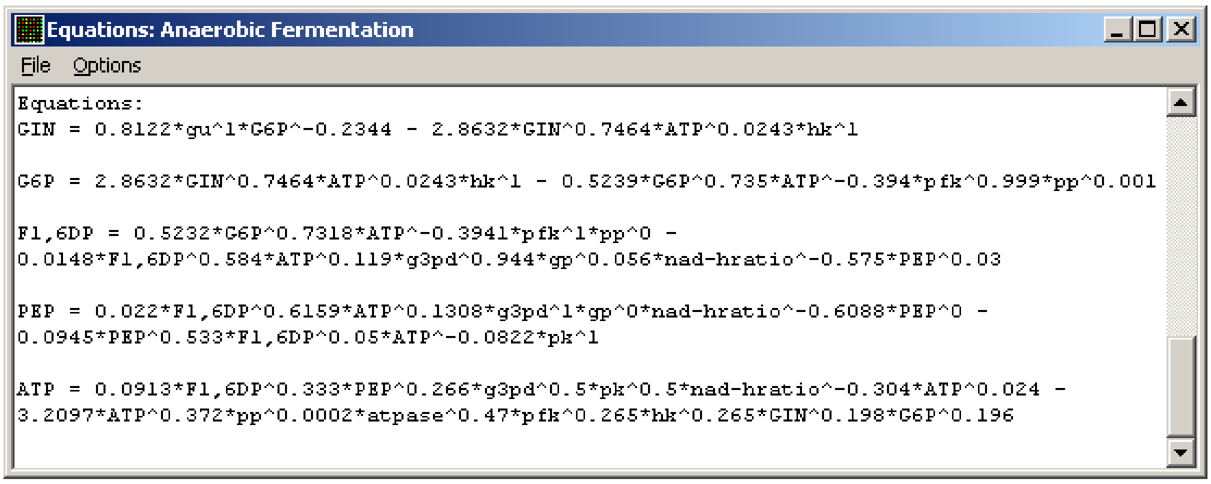
Parameterization of S-systems in yeast

A major difficulty with S-Systems is identifying an appropriate set of parameter values for a model. We used Differential Evolution to optimize model parameters (rate constants and kinetic orders). Results with time course simulated data of fermentation in *Saccharomyces cerevisae* show that a full parameter set evolved that fits well four out of five of the time-course data points and adequately models the dynamics of the system.

To exemplify, the yeast model is a system of 9 independent variables and 5 dependent variables with various interactions as shown below.

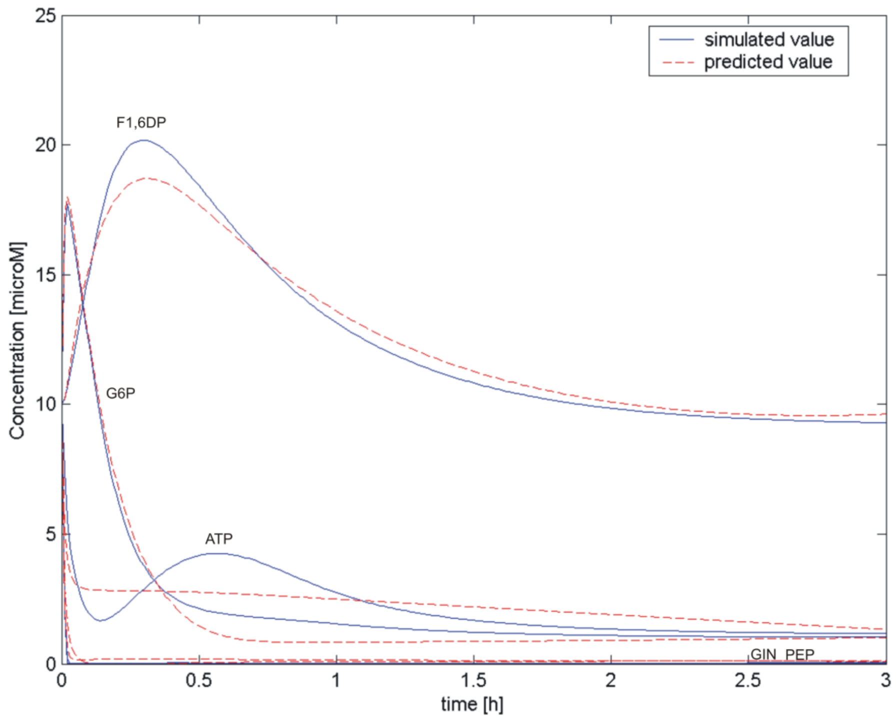


The underlying equations are of the form:



```
Equations:
GIN = 0.8122*gu^1*G6P^-0.2344 - 2.8632*GIN^0.7464*ATP^0.0243*hk^1

G6P = 2.8632*GIN^0.7464*ATP^0.0243*hk^1 - 0.5239*G6P^0.735*ATP^-0.394*pfk^0.999*pp^0.001

F1,6DP = 0.5232*G6P^0.7318*ATP^-0.3941*pfk^1*pp^0 -
0.0148*F1,6DP^0.584*ATP^0.119*g3pd^0.944*gp^0.056*nad-hratio^-0.575*PEP^0.03

PEP = 0.022*F1,6DP^0.6159*ATP^0.1308*g3pd^1*gp^0*nad-hratio^-0.6088*PEP^0 -
0.0945*PEP^0.533*F1,6DP^0.05*ATP^-0.0822*pk^1

ATP = 0.0913*F1,6DP^0.333*PEP^0.266*g3pd^0.5*pk^0.5*nad-hratio^-0.304*ATP^0.024 -
3.2097*ATP^0.372*pp^0.0002*atpase^0.47*pfk^0.265*hk^0.265*GIN^0.198*G6P^0.196
```

We used DE to parameterize the model for a time series data set. The full parameter set for the yeast model consists of 55 parameters. For such a complex model the evolved parameter set fits well to the original data and adequately reflects the dynamics of four out of the five dependent variables, as shown in the figure below. All the evolved parameters are within the usual parameter values of S-systems and the model is stable. The dynamics of *ATP* were not adequately modelled with the evolved equation being essentially a linearization of the data points. This is still acceptable since the *ATP* equation is particularly complex with 15 parameters. The other four equations are a good fit to the data with a slight overshoot in *F1,6DP* and an undershoot in *G6PD*. For simpler models a virtually perfect fit can be obtained. For complex models the use of structural constraints could improve the optimization results.

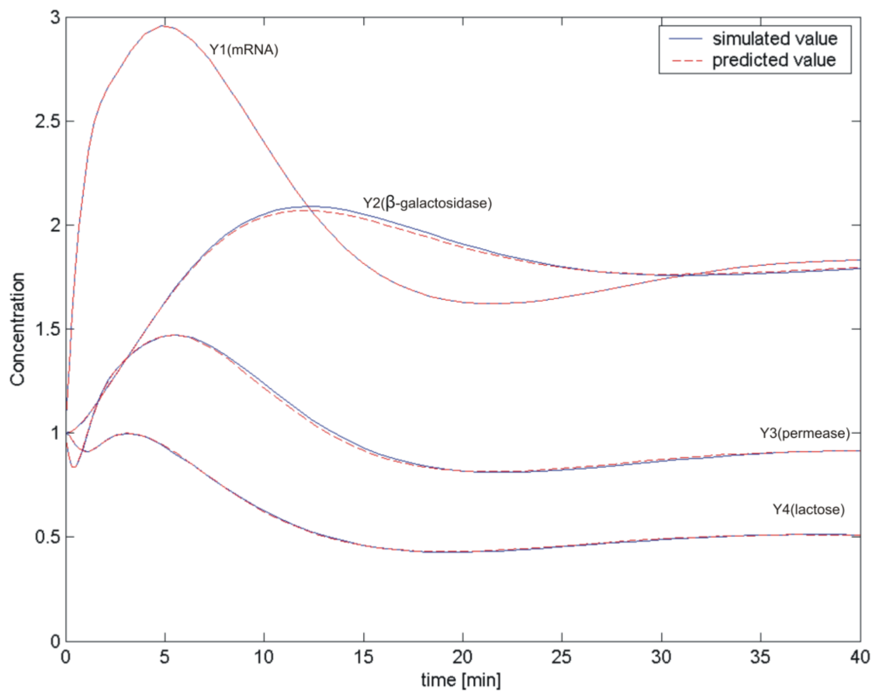


Model reconstruction and parameterization of the lac operon in *E. coli*

Of course the ultimate modeling method will allow construction of entire models, fully parameterized from biological datasets. This goal is still out of reach, but some steps can be taken to advance the research. The hybrid EA we discussed in chapter 9 was used to evolve models of biological processes as systems of differential equations and simultaneously co-evolve a set of parameters for these models from time-series data. Recall that the hybrid algorithm uses Gene Expression Programming for model inference with an embedded Differential Evolution for model parameterization.

We looked at two models for the lac operon and attempted to reconstruct both the parameters and the underlying model from a simulated time series dataset.

For the simpler model the predicted data and the simulated data points are virtually indistinguishable with an almost perfect fitness value.



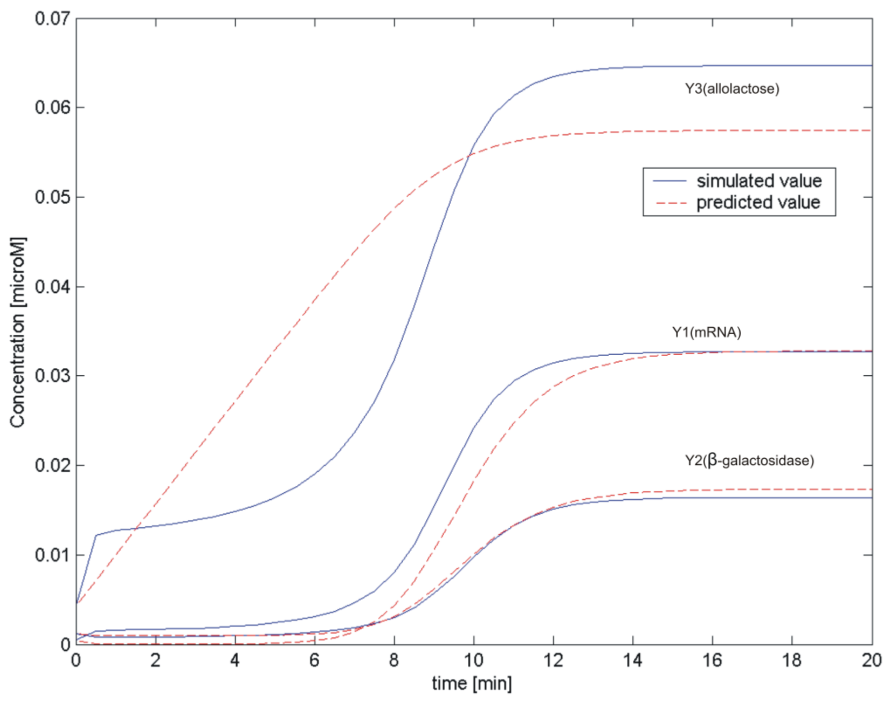
The simplified and rearranged form of the run is shown below. Where *y1* is the concentration of mRNA, *y2* is permease, *y3* is β-galactosidase and *y4* is lactose. The original equations are on the right hand side to facilitate comparisons.

$$y1 = \frac{4.9987 y_4^{t-0.64} y_4^{t-0.64} + 1}{y_4^{t-0.64} y_4^{t-0.64} + 1} - y_1$$
$$y2 = y_1 - (y_2 + y_2)$$
$$y3 = y_1 / 10.1023 - y_3 / 10.1023$$
$$y4 = y_2 - y_3 y_4$$

$$\frac{dM}{dt} = \frac{1 + k_1 y_4^{\rho}}{1 + y_4^{\rho}} - b_1 y_1$$
$$\frac{dP}{dt} = y_1 - b_2 y_2$$
$$\frac{dB}{dt} = r_3 y_1 - b_3 y_3$$
$$\frac{dL}{dt} = S y_2 - y_3 y_4$$

The evolved system of differential equations preserves the structure of the original model with the correct production and degradation components and the relationships between the elements. Out of the ten available variables for optimization only three appear in the final model. These do not necessarily mimic the original parameters but rather are adapted to the evolved equations. An appropriate time delay was discovered even though it is not a perfect match to the original value (0.86) which is the main cause of deviation between the simulated and predicted data.

For the more complex model we did not get such a good fit:



The evolved equations after simplification and rearrangement are shown below. The original equations are on the right hand side to facilitate comparisons (*y1* is the concentration of mRNA, *y2* is β-galactosidase and *y3* is allolactose). Likewise to the previous model, out of the 20 available variables only 8 were used in the equations.

.

$$y1 = 0.123003^{8.1379-\left(\frac{y3}{y1+y1+8.7614}\right)} - y1$$
$$y2 = y1 - y1^{t-0.53614}$$
$$y3 = 0.12303 - (y2^{0.516968})$$

$$\frac{dM}{dt} = \alpha_M \frac{1+k_1(e^{-\mu\tau_M}A_{\tau_M})^n}{k+k_1(e^{-\mu\tau_M}A_{\tau_M})^n} - \tilde{Y}_M M$$
$$\frac{dB}{dt} = \alpha_B e^{-\mu\tau_B} M_{\tau_B} - \tilde{Y}_B B$$
$$\frac{dA}{dt} = \alpha_A B \frac{L}{K_L+L} - \beta_A B \frac{A}{K_A+A} - \tilde{Y}_A A$$

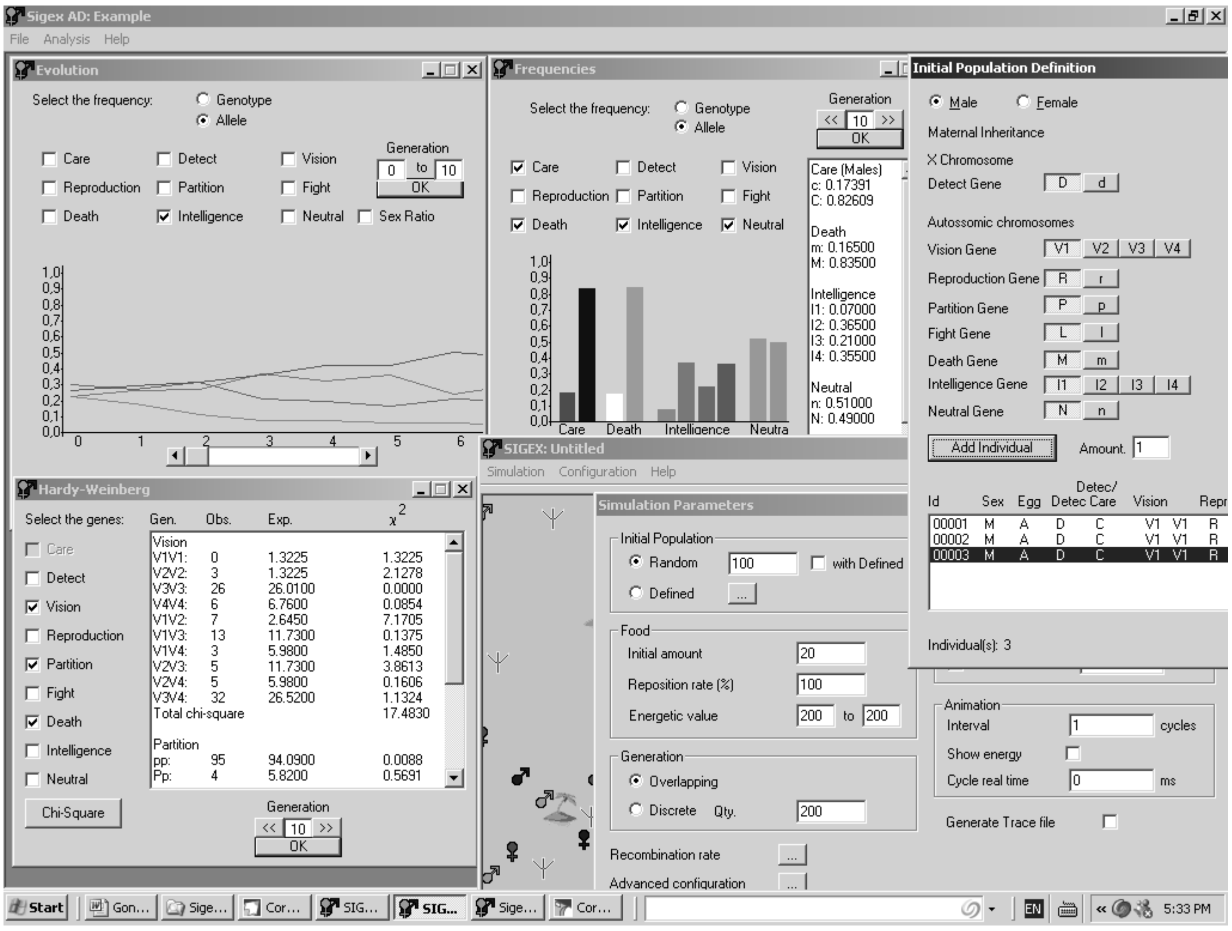
The fit of the predicted values to the simulated data points is worse than for the other model particularly for allolactose, but still a reasonable fit (R^2 0.987 – *y1*, 0.999 – *y2* and 0.836 – *y3*) for such a complex model. Changes of the EA parameters may improve convergence to a better fit. Of more concern are the equations which do not always reflect the true relationships between the different components of the system as these are of key importance to understand a biochemical pathway or a genetic network.

Artificial life – population genetics dynamics

EC has stolen concepts from biology to develop optimization methods. We can steal them back and use them to better understand biology. EAs do not necessarily need an objective function for a problem, we can simply select organisms from a purely Darwinian approach – survival of the fittest.

Artificial agents that model natural populations can use a classic canonic GA structure for their inheritance model (Gondro and Magalhaes 2005). The value in each position of the bitstring is an allele (0 or 1) and the position itself is a gene or locus. The combination of values (alleles) in the bitstring (chromosome) maps to a phenotypic expression. So, the GA operates at two structural levels: a genotypic and a phenotypic one. Selection operates on the overall genomic value (phenotype) while search operators act on the genotype, modifying the chromosomes which may or may not change the phenotypic expression.

In our work these virtual organisms are an abstraction of Mendelian populations, meaning that they are a single species of freely interbreeding diploid organisms with two sexes on an XY system. There are two genes in the sex chromosomes and seven genes distributed in a variable number of autosomes (between 1 and 7). Each gene has between two and four allelic variants with user-defined phenotypic expressions within a certain interval limit. The genes through their phenotypic expressions express characteristics that intimately relate to the universe ensuring a rapid evolution of the population. For example the gene for *vision* determines the line of sight of the organism which is an important trait for searching for food in the environment and finding a partner for reproduction. The genes not only relate to the environment but they also relate to other organisms, as for instance the *fight* gene which defines the level of aggressiveness of an organism.



Evolution — Select the frequency: Genotype [✓ Allele]; Care, Detect, Vision, Reproduction, Partition, Fight, Death, [✓ Intelligence], Neutral, Sex Ratio; Generation 0 to 10, OK

Frequencies — Select the frequency: Genotype [✓ Allele]; Generation << 10 >>, OK; [✓ Care], Detect, Vision, Reproduction, Partition, Fight, [✓ Death], [✓ Intelligence], [✓ Neutral]

```
Care (Males)
c: 0.17391
C: 0.82609

Death
m: 0.16500
M: 0.83500

Intelligence
I1: 0.07000
I2: 0.36500
I3: 0.21000
I4: 0.35500

Neutral
n: 0.51000
N: 0.49000
```

Hardy-Weinberg — Select the genes: Care, Detect, [✓ Vision], Reproduction, [✓ Partition], Fight, [✓ Death], Intelligence, Neutral; Generation << 10 >>, OK

```
Gen.      Obs.   Exp.          χ2
Vision
V1V1:     0      1.3225        1.3225
V2V2:     3      1.3225        2.1278
V3V3:     26     26.0100       0.0000
V4V4:     6      6.7600        0.0854
V1V2:     7      2.6450        7.1705
V1V3:     13     11.7300       0.1375
V1V4:     3      5.9800        1.4850
V2V3:     5      11.7300       3.8613
V2V4:     5      5.9800        0.1606
V3V4:     32     26.5200       1.1324
Total chi-square               17.4830

Partition
pp:       95     94.0900       0.0088
Pp:       4      5.8200        0.5691
```

Initial Population Definition — [✓ Male] Female

Maternal Inheritance
X Chromosome
Detect Gene: D d
Autossomic chromosomes
Vision Gene: V1 V2 V3 V4
Reproduction Gene: R r
Partition Gene: P p
Fight Gene: L l
Death Gene: M m
Intelligence Gene: I1 I2 I3 I4
Neutral Gene: N n
Amount. 1

```
                       Detec/
Id     Sex  Egg  Detec Care    Vision    Repr
00001  M    A    D     C       V1  V1    R
00002  M    A    D     C       V1  V1    R
00003  M    A    D     C       V1  V1    R
```

Individual(s): 3

Simulation Parameters

Initial Population: [✓ Random] 100, with Defined; Defined ...
Food: Initial amount 20; Reposition rate (%) 100; Energetic value 200 to 200
Generation: [✓ Overlapping]; Discrete Qty. 200
Recombination rate ...
Advanced configuration ...
Animation: Interval 1 cycles; Show energy; Cycle real time 0 ms
Generate Trace file

Sigex – an educational package for studies of population genetics and evolution. The program consists of four modules: a simulator of virtual organisms, a genotype editor, a data analysis tool and a manual/tutorial of population genetics and evolution.

This type of work can be used in education applications to help students understand the dynamics of populations and the concepts of population genetics and can also be a useful tool for research in population genetics. Even though computational simulations cannot accurately depict the dynamics of natural populations, for theoretical studies and as an initial approach to test a new model *artificial* data can be used prior to obtaining experimental data. This approach not only allows testing on rapidly obtainable, controlled data but can also help in determining which experimental data is relevant, assisting in the design of the experiment.
I got involved in EC because I could not fathom having to count even one single more *Drosophila*!

Conclusion

Evolutionary Algorithms are efficient for addressing complex biological problems. Many biological questions are being studied using EAs; and with the ever increasing volume of biological data, it can be expected that the use of EAs will only keep on growing. Evolutionary Computation has come full circle; originally inspired by biological processes, it has found its way back into biology to help investigate complex, challenging and relevant problems.

References

Differential Evolution Homepage (recommended)
http://www.icsi.berkeley.edu/~storn/code.html

Atmar, W. (1994). "Notes on the Simulation of Evolution." IEEE Transactions on Neural Networks 5(1): 130-147.

Bäck, T. (1996). Evolutionary Algorithms in theory and practice. New York, Oxford University Press.

Bäck, T. (2000). Introduction to Evolutionary Algorithms. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 59-63.

Bäck, T., D. B. Fogel and T. Michalewicz, Eds. (2000a). Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing.

Bäck, T., D. B. Fogel and T. Michalewicz, Eds. (2000b). Evolutionary Computation 2: Advanced Algorithms and Operators. Bristol, Institute of Physics Publishing.

Bäck, T., D. B. Fogel, L. D. Whitley and P. J. Angeline (2000c). Mutation Operators. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 237-255.

Bäck, T., Ed. (2003). Handbook of Evolutionary Computation. Bristol, Institute of Physics Publishing.

Banzhaf, W., P. Nordin, R. E. Keller and F. D. Francone (1998). Genetic Programming - An Introduction. San Mateo, Morgan Kaufmann.

Booker, L. B., D. B. Fogel, L. D. Whitley, P. J. Angeline and A. E. Eiben (2000). Recombination. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 256-307.

Coello Coello C. A., G. B. Lamont and D. A. van Veldhuizen (2007). Evolutionary Algorithms for Multi-Objective Problems. Berlin, Springer.

De Jong, K., D. B. Fogel and H. P. Schwefel (2000). A history of evolutionary computation. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 40-58.

De Jong, K. A. (2006). Evolutionary Computation. Cambridge, MIT Press.

Eshelman, L. J. (2000). Genetic Algorithms. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 64-80.

Ferreira, C. (2001). "Gene Expression Programming: A new adaptive algorithm for solving problems." Complex Systems 13(2): 87-129.

Ferreira, C. (2002). Gene Expression Programming. Mathematical Modelling by an Artificial Intelligence. Angra do Heroismo, GepSoft.

Fogel, D. B. (1999). Evolutionary Computation: Toward a New Philosophy of Machine Intelligence. Piscataway, Wiley-IEEE.

Fogel, D. B. (2000a). Introduction to Evolutionary Computation. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 1-3.

Fogel, D. B. (2000b). Principles of Evolutionary Computation. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 23-26.

Fogel, D. B. and D. W. Corne, Eds. (2003). Evolutionary Computation in Bioinformatics. San Mateo, Morgan Kaufmann.

Forrest, S. (1993). "Genetic Algorithms - Principles of Natural-Selection Applied to Computation." Science 261(5123): 872-878.

Goldberg, D. E. (1987). Simple genetic algorithms and the minimal, deceptive problem. Genetic Algorithms and Simulated Annealing. L. Davis. San Mateo, Morgan Kaufmann: 74-88.

Gondro, C. and B. P. Kinghorn (2007). "Solving complex problems with evolutionary computation." Proceedings of the 17th Conference of the Australian Association for the Advancement of Animal Breeding and Genetics. 17: 272-279.

Gondro, C. and B. P. Kinghorn (2007). “A simple genetic algorithm for multiple sequence alignment.” Genetics and Molecular Research 6(4): 964-982.

Gondro, C. and B. P. Kinghorn (2008). “Optimization of cDNA microarray experimental designs using an Evolutionary Algorithm.” IEEE Transactions on Computational Biology and Bioinformatics, (in press, doi:10.1109/TCBB.2007.70222).

Gondro, C. and J. C. M. Magalhaes (2005). “A simple genetic algorithm for studies of mendelian populations.” Recent Advances in Artificial Life. H. A. Abbass, T. Bossamaier and J. Wiles. London, World Scientific Publishing: 85-98.

Hancock, P. J. B. (2000). A comparison of selection mechanisms. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 212-227.

Heywood, M. I. and A. N. Zincir-Heywood (2000). Page-based linear genetic programming. Systems, Man, and Cybernetics, 2000 IEEE International Conference, IEEE Press.

Holland, J. H. (1975). Adaptation in natural and artificial systems. Ann Arbor, University of Michigan Press.

Kantschik, W. and W. Banzhaf (2001). "Linear-tree GP and its comparison with other GP structures." Genetic Programming, Proceedings 2038: 302-312.

Kinghorn, B. P. and R. K. Shepherd (1999). "Mate selection for the tactical implementation of breeding programs." Assoc. Advmt. Anim. Breed. Genet. 13:130-133.

Koza, J. R. (1989). Hierarchical genetic algorithms operating on populations of computer programs. Proceedings of the 11th International Joint Conference on Artificial Intelligence, San Mateo, Morgan Kaufmann.

Koza, J. R. (1992). Genetic Programming: On the Programming of Computers by Means of Natural Selection. Cambridge, MIT Press.

Koza, J. R. (1994). Genetic Programming II: Automatic Discovery of Reusable Programs. Cambridge, MIT Press.

Koza, J. R., F. H. Bennett III, D. Andre and M. A. Keane (1999). Genetic Programming III: Darwinian Invention and Problem Solving. San Francisco, Morgan Kaufmann.

Koza, J. R., M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu and G. Lanza (2003). Genetic Programming IV: Routine Human-Competitive Machine Intelligence. Boston, Kluwer Academic Publishers.

Langdon, W. B. and R. Poli (2002). Foundations of genetic programming. Heidelberg, Springer-Verlag.

Lobo, F. G., C. F. Lima and Z. Michalewicz, Eds. (2007). Parameter Setting in Evolutionary Algorithms. Berlin, Springer.

Mayer, D.G., Kinghorn, B.P. and Archer, A.A. 2005. Differential evolution – an easy and efficient evolutionary algorithm for model optimisation. *Agricultural Systems* 83:315-328

Michalewicz, T. and D. B. Fogel (2000). How to solve it: modern heuristics. Heidelberg, Springer-Verlag.

Mitchell, M. and C. E. Taylor (1999). "Evolutionary computation: An overview." Annual Review of Ecology and Systematics 30: 593-616.

Nedjah, N., E. Alba and L. de Macedo Mourelle, Eds. (2006). Parallel Evolutionary Computations. Berlin, Springer.

Nordin, J. P. (1994). A Compiling Genetic Programming System that Directly Manipulates the Machine code. Cambridge, MIT Press.

Palshikar, G. K. (2001). "Simulated annealing: A heuristic optimization algorithm." Dr Dobbs Journal 26(9): 121-124.

Porto, V. M. (2000). Evolutionary Programming. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 89-102.

Price, K. and R. Storn. (1997). Differential evolution. Dr. Dobb's Journal 264: 18-24.

Rudolph, G. (2000). Evolution Strategies. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 81-88.

Schewefel, H. P., I. Wegener and K. Weinert, Eds. (2003). Advances in computational intelligence. Berlin, Springer-Verlag.

Schwefel, H. P. (2000). Advantages (and disadvantages) of evolutionary computation over other approaches. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 20-22.

Schwefel, H. P. and G. Rudolph (1995). Contemporary Evolution Strategies. Advances in Artificial Life. Third International Conference on Artificial Life. F. Moran, A. Moreno, J. J. Merelo and P. Chacon. Berlin, Springer-Verlag. 929: 893-907.

Smith, R. E. (2000). Learning classifier systems. Evolutionary Computation 1: Basic Algorithms and Operators. Bristol, Institute of Physics Publishing: 114-123.

Storn, R. and K. Price (1997). "Differential evolution - A simple and efficient heuristic for global optimization over continuous spaces." Journal of Global Optimization 11(4): 341-359.

Van Veldhuizen, D. A. and G. B. Lamont (2000). "Multiobjective evolutionary algorithms: analyzing the state-of-the-art." Evol Comput 8(2): 125-47.

Vlachos, C., R. Gregory, R. C. Paton, J. R. Saunders and Q. H. Wu (2004). "Individual-based modelling of bacterial ecologies and evolution." Comparative and Functional Genomics 5(1): 100-104.

Voit, E. O. (2000). Computational Analysis of Biochemical Systems: A Practical Guide for Biochemists and Molecular Biologists. Cambridge, Cambridge University Press.

Whitley, D. (2001). "An overview of evolutionary algorithms: practical issues and common pitfalls." Information and Software Technology 43(14): 817-831.

Wilson, S. W. (1994). "ZCS: a zeroth level classifier system." Evolutionary Computation 2: 1-18.

Wolpert, D. H. and W. G. Macready (1996). No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute.

Zitzler, E., K. Deb and L. Thiele (2000). "Comparison of multiobjective evolutionary algorithms: empirical results." Evol Comput 8(2): 173-95.

Zitzler, E., M. Laumanns and S. Bleuler. (2004) "A tutorial on evolutionary multiobjective optimization." Metaheuristics for Multiobjective Optimisation. Germany, Springer: 3-38.