SOLVING COMPLEX PROBLEMS WITH EVOLUTIONARY COMPUTATION

C. Gondro and B. P. Kinghorn

The Institute for Genetics and Bioinformatics, University of New England, Armidale, NSW 2351

SUMMARY
Evolutionary Computation encompasses optimization methods loosely based on biological evolutionary processes. These methods are efficient to find near-optimal solutions in large, complex non-linear search spaces. This paper sketches a roadmap to their adoption in animal genetics.

INTRODUCTION
Animal genetics is treading down the same path as most of the other biological sciences. It is heavily dependant on data, high-throughput techniques and on the development of computational tools and analytical methods that allow the interpretation and extraction of new knowledge from these vast amounts of data. There is a clear shift towards an information-centred science. The downside is that we still do not have widespread resources capable of optimizing large systems or solving complex problems purely through exhaustive search, nor deterministic algorithms guaranteed to derive optimal solutions for every conceivable problem. The alternative is to rely on intelligent computational methods that are capable of reducing the search space that must be covered and can guide the search to the most feasible areas. These methods are not guaranteed to always yield the optimal result but at least they offer good approximations that can be further tuned. A blooming field of research for such methods is Evolutionary Computation (EC).

Evolutionary Computation is a broad field of research in optimization methods loosely based on biological evolutionary processes such as mutation, crossover and selection. Biological evolution is a common source of inspiration for tackling difficult computational problems since, in rather simple terms it can be viewed as a search method for the survivability of a species over a huge solution space under a dynamic environment. The central idea behind EC is to create populations of candidate solutions of a problem and evolve these populations by selection based on an objective function which emulates natural selection (Fogel 1999; Bäck *et al*. 2000a, 2000b; Bäck 2003).

In the following sections we briefly review some of the main flavours of EC and dot-point some applications developed within our group. A review of EC would easily take up an entire book; here the focus is on a roadmap of which methods could be more suitable for a given problem and some practical pointers on how to implement them.

A SIMPLE RECIPE FOR SOLVING COMPLEX PROBLEMS
1. Write an objective function: This should be able to return a single value (criterion or fitness value) that represents the value of a single solution; albeit this value can consist of weighted multiple components directed at different sub-objectives (for multi objective optimization problems). 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).

2. If needed, write an algorithm to produce the input variables or states from a vector of real numbers. An example is given by Kinghorn and Shepherd (1999) who convert such a vector into a pattern of mating and selection. This algorithm should ideally produce only legal solutions to the problem.
3. Choose an optimization engine. For optimizing a vector of real numbers, our group has used Differential Evolution (Storn and Price 1997) quite widely. The optimization engine is quite simple 'on the outside'. It generates vectors of numbers and seeks the vector that gives the highest fitness.

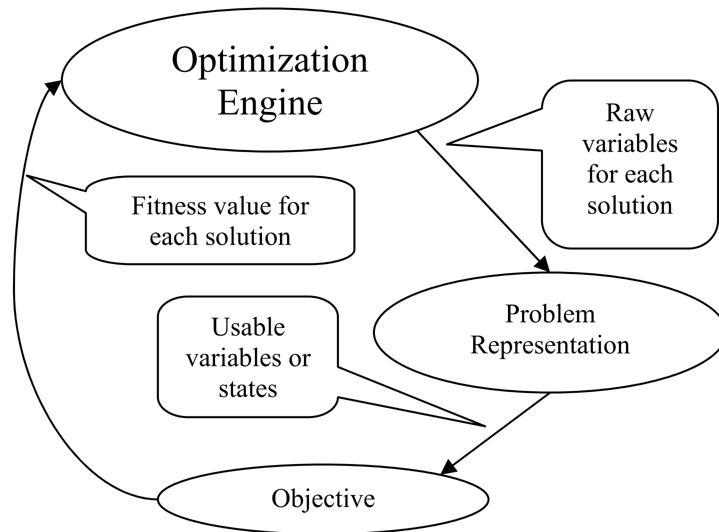

Figure 1. Optimization pathway.

OPTIMIZATION ENGINES: EVOLUTIONARY COMPUTATION

In a nutshell, EC tries to mimic the mechanisms of biological evolution to solve complex problems (Mitchell and Taylor 1999). EC methods are commonly referred to as Evolutionary Algorithms (EAs) and all have in common the use of populations of candidate solutions which reproduce, compete, and are subjected to selective pressures and random variation (Atmar 1994). These candidates are evaluated as to their adaptiveness which determines their capacity of generating descendants, thus propagating better fit solutions into the future generations. Implementations vary significantly and algorithms are not constrained to using only biological mechanisms, but there still are some common features which are shared by the different EC methods (Mitchell and Taylor 1999; Bäck *et al*. 2000a):

Population. A number *(n)* of candidate solutions (representations of the problem) compete against each other to remain in the population and generate offspring. Since EAs use populations, they can be seen as a parallelized search of the solution space. Population structures can be either steady-state or generational. Steady-state uses an overlapping generation approach in which parents and offspring simultaneously compete in the population. 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 is more pronounced and can lead to the loss of variability. To counteract this effect, larger populations should be used in steady-state systems (De Jong and Sarma 1993).

Fitness. Candidates 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 algorithm. A good fitness function should allow for a range of intermediary values which can be explored by the EA, the more continuous/smoother the fitness function, the higher the probability that the EA will converge on an adequate solution. In multi-objective optimization this issue 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. 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. A common approach to multi-objective optimization is to use a weighting scheme for the different objectives (Zitzler *et al.* 2000; Van Veldhuizen and Lamont 2000) which ranges from a fully self adaptive approach (the scheme evolves alongside the candidate solutions) 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 – the best direction to take depends on how far can be gone in each direction (Kinghorn *et al*. 2002).

Selection. 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. Note that since the process is stochastic, the best solutions are not necessarily always selected. This allows inferior solutions to be selected over better ones with a low probability and helps preserve the diversity of the population and also avoids a premature convergence on local optima. The main methods are proportionate, rank-based, Boltzmann and tournament. Proportionate selection assigns a probability of generating offspring based on the relative fitness of the solution. The simplest form of proportionate selection is roulette wheel; where each solution is assigned an area in the wheel proportional to its fitness – fitter solutions 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. In tournament selection a given number of candidates compete and the ones with the lowest fitness are replaced by new solutions, 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 roulette wheel 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. But, tournament selection can rapidly lead to a loss of diversity. To counterbalance this effect small tournaments are preferred in association with slightly higher mutation rates (Bäck *et al*. 2000a).

Search operators. These provide the variability necessary for the EC population to explore different areas of the solution space. The two main sources of variability are mutations, which are randomly generated new sources of variability, and crossover. Crossover is a search operator that does not generate new sources of variability in the populations albeit introducing new variation, meaning that it can only generate new combinations from the available diversity in the population. It operates by combining parts from two or more parents to generate one or more offspring. The drive behind

crossover is to manipulate the component sources of variation to explore new combinations which might be better solutions to the problem. The choice of a specific crossover method (Bäck *et al.* 2000a) will depend on the EA and the representation of the problem (eg. binary strings, real-valued vectors, finite-state machines or parse trees). A simple one-point crossover is depicted in figure 2.

Mutation (Figure 2) generates new 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.* 2000a). Mutation is an important operator to generate new sources of variability and expose new areas of the solution landscape whilst crossover can only shuffle and reveal existing variability. Mutation and crossover are the main search operators used in EC. 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 insufficient 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 selective pressure and population size, which are the four main parameters in an EA (Banzhaf *et al.* 1998). A generic EA combines the above 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 EA:

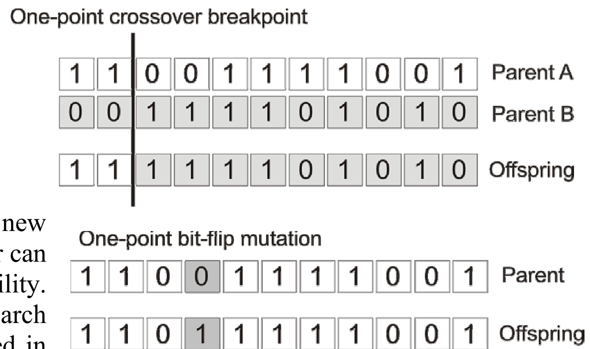

Figure 2. One-point crossover – a breakpoint is randomly selected and the two chromosomes swap bitstrings after the breakpoint. Point-mutation bit-flip – new offspring are produced by a random change to the parent.

1. Create an initial population – randomly or based on prior information;
2. Assign a fitness value to all solutions;
3. Select solutions 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 solutions 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.

MAIN TYPES OF EVOLUTIONARY ALGORITHMS

Evolutionary Programming (EP). The 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 μ_i until a predefined number λ_i 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. 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 solution. In this sense the genotype is of little importance and focus is on optimization of the phenotype. EP traditionally uses continuous-valued variables instead of the discrete representation common in Genetic Algorithms. Current versions of EP are self-adaptive, with the mutation parameters (variance, covariance) adapting to the current state of the population (Bäck 1996; Fogel 1999).

Evolution Strategies (ES). ES were initially developed to solve technical optimization problems. There are two main general notations for the strategy: $(\mu + \lambda)$ where the ES generates λ offspring from a parental population μ and selects the best μ from all $\mu+\lambda$ 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 $\mu:\lambda$ 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. In a typical ES mutation scheme each solution, alongside 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 $(\mu + 1)$ with a single replacement per iteration loop. 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 solution (Schwefel and Rudolph 1995).

Genetic Algorithms (GAs). The most widely disseminated EC branch, GAs date back to Holland's (1975) seminal work. Traditional GA solutions are represented as linear bitstrings which are referred to as chromosomes. 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 chromosome maps to a phenotypic expression, such as a parameter to be optimized. 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. 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). 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 (Bäck *et al*. 2000a).

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 *et al.* 2003). Original implementations of GP used tree-structured representations implemented in LISP (rarely used nowadays). 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 (Heywood and Zincir-Heywood 2000; Banzhaf *et al.* 1998). GP uses crossover (Figure 3) and mutation in a similar fashion as GAs.

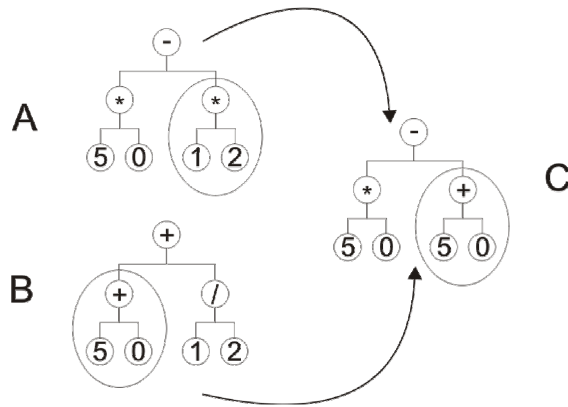

Figure 3. Crossover in a tree GP. Crossover between parents A and B generate offspring C, the function set is {*,+,/,-} and the terminal set is {2,0,1,5}. The trees code for A=(5*0)-(1*2); B=(5+0)+(1/2); C=(5*0)-(5+0).

EVOLUTIONARY ALGORITHMS APPLIED TO ANIMAL GENETICS

Our group has used EAs for many applications, including:

- Allocation of animals to treatments.
- Selective genotyping for mapping experiments.
- Allocating individuals to groups for DNA pooling in multi-trait association studies.
- Decisions on which animals to genotype for markers known to be of value.
- Selecting markers into panels for genotyping.
- Allocating DNA sequences to multiplex groups.
- Fitting complex non-linear growth models.
- Essentially the full range of issues impacting on breeding programs as implemented via mating and selection decisions.
- Optimising a wide range of management issues to maximize economic/logistical/sustainable efficiency in animal production systems.
- Optimising the number of harvesting sessions, truncation points or proportions to harvest at each session, and the timing of these harvestings, in order to maximize profit in the face of price grid(s) for the trait(s) concerned.
- Matching current and projected seedstock inventory to orders made by multiple customers for seedstock with specific customer requirements for genetic merit, marker status, and other animal

attributes, and specified dates of delivery, while also accommodating the logistics/welfare of assembling groups of animals into orders, and efficient use of animal facilities.

- Allocating families to common-environment untagged groups and simultaneously allocating genetic markers to these groups for subsequent use to derive familial contributions to groups and mean merit by family.
- Optimization of microarray experimental designs.
- Reconstruction and parameterization of genetic networks.

PRACTICAL CONSIDERATIONS

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 (Banzhaf *et al*. 1998). 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. Our group has largely favoured Differential Evolution in many applications because of these aspects.

Complex problems with convoluted constraints can be handled in two ways: (1) test conformance at the time of criterion evaluation, and allocate a criterion value to non-conforming solutions that is sufficiently low to exclude them from contention. The advantage here is simpler coding, but there can be big speed penalties, with sometimes almost all evolutionary pressure used to maintain conformance to constraints or (2) develop a filtering algorithm that will convert the vector of parameters being optimised into a solution that must always conform to constraints. This is the preferred solution if it can be achieved. The speed cost of implementing such a filtering algorithm is most likely to be small compared to the speed gains achieved from removing constraints as an issue to be handled in the optimisation part. So a filtering algorithm should be targeted to handle constraints. This is an issue of problem representation – how to present the problem to the optimisation engine (see Figure 1).

An appropriate choice of representation for the populations is crucial for 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. FORTRAN and C/C++ are good candidates, with the former having the upper hand in terms of speed. 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 EAs are easily parallelizable this problem is becoming less significant (Alba and Tomassini 2002).

In summary, real-world problems are complex and demand flexible tools that can be adapted to the nature of the problem and not tools that force the problem to adapt itself to them. EC provides a powerful framework for solving these problems.

ACKNOWLEDGMENTS
This research was partially funded by the Cooperative Research Centre for Cattle and Beef Quality and the University of New England.

REFERENCES

Alba, E. and Tomassini, M. (2002) *IEEE Trans Evol Comput* **6**:443.

Atmar, W. (1994) *IEEE Trans Neural Networks* **5**:130.

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

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

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

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

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

De Jong, K. and Sarma, J. (1993) In "Foundations of Genetic Algorithms 2", p.19, editor L. D. Whitley, Morgan Kaufmann, San Mateo.

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

Heywood, M.I. and Zincir-Heywood, A.N. (2000) 2000 IEEE Int. Conf. Syst, Man and Cybernetics.

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

Kinghorn, B.P. and Shepherd, R.K. 1999. Assoc. Advmt. Anim. Breed. Genet. **13**:130.

Kinghorn, B.P., Meszaros S.A. and Vagg, R.D. (2002) 7th WCGALP. **33**:179.

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

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

Mitchell, M. and Taylor, C.E. (1999) *Annu Rev Ecol Syst* **30**:593.

Schwefel, H.P. and Rudolph, G. (1995) In "Advances in Artificial Life, 3rd Int Conf ALIFE", **929**, p.893, editors F. Moran, A. Moreno, J.J. Merelo and P. Chacon, Springer-Verlag, Berlin.

Storn, R. and Price, K. (1997) *Journ Global Optim* **11**:341.

Van Veldhuizen, D.A. and Lamont, G.B. (2000) *Evol Comput* **8**:125.

Whitley, D. (2001) *Inf and Soft Techn* **43**:817.

Zitzler, E., Deb, K. and Thiele, L. (2000) *Evol Comput* **8**:173.