Technical Report 06-015

Reversibility in Evolutionary Algorithms

Kamran Karimi

21-May-2006

School of Computer Science

Reversibility in Evolutionary Algorithms

Kamran Karimi

School of Computer Science
University of Windsor
Windsor, Ontario
Canada N9B 3P4
kamran@uwindsor.ca

Absract. Evolutionary algorithms are an effective way of solving search problems. They usually operate in a forward temporal direction, where, as new members of the population are created, information about the previous members is lost. With reversible computing, no information is lost, and one can undo the effects of a computation, thus making it possible to retrieve the previous generations. Reversible computing is of importance because it theoretically allows us to reduce power consumption in an algorithm. In this paper we propose the use of evolutionary operators for which a reverse exists, so that, given the output of an operation, the input can be reconstructed by applying the reverse operator. The paper presents examples of reversible genetic operations of crossover and mutation. Achieving reversibility requires a number of modifications to the usual framework of evolutionary algorithms, and we show how these changes can be integrated into an evolutionary algorithm. We suggest performing a series of such reversible operations to evolve the population. To record the order in which the operators are executed, we push them down a stack. We can then undo the changes by retrieving the list in reverse from the stack and applying the reverse operators, ending up with the original population.

1. Introduction

In this paper we introduce the concept of reversible evolutionary algorithms, and propose a possible framework for implementing such reversible algorithms. We use examples from the domain of genetic algorithms. The aim of the paper is to investigate the emerging idea and potentials of reversibility, and incite more work on the topic.

A structure, such as an artificial genome, is usually represented as a sequence of symbols, which has to be interpreted (executed) for observing a tangible effect, called a behaviour. With randomly generated structures, we seldom find useful behaviour, so we need a method of evolving the structures. Determining a useful behaviour is domain dependent and sometimes not obvious [3]. There are usually an exponential number of possibilities for representing the structures and the corresponding behaviours. In a complex environment, it is hard to derive the mapping from the syntactical representation of a structure to its run-time behaviour by a static examination alone. This difficulty explains why programming is such a hard and error-prone activity, since the programmer has to come up with a syntactical form (the programme) that behaves in a desirable way when executed.

Genetic algorithms and other evolutionary methods [1, 5] are of interest because, following certain guidelines, they produce the structures, called genotypes in genetic algorithms, and evaluate the

usefulness of their behaviour, which is consequently used for possible changes in the structures in next generations.

While it is essential to be able to progress in a forward direction of time to achieve the evolution of a population from a subjectively useless form to one that shows a desired behaviour, we may be interested in following the same process backwards in time. Reversibility is an interesting phenomenon that provides us with the ability to run an operation backward, and determine the inputs from the outputs of the operation. Time-reversal has proven to be of benefit in domains such as rule-generation [4]. In physics, it is shown that a reversible operation (computation) can in theory be performed without generating any heat, and reversible computing is a requirement of any operation in Quantum Computing [6].

Evolution can happen in a reversible fashion when, given a current evolutionary structure and the operator that produced it, we can re-generate the previous evolutionary structure. In other words, no information is lost when the operator is applied.

In other words, in evolutionary algorithms a reversal effect is achieved by making sure that we can recover the members of a generation from the *succeeding* one. For this purpose we make sure that the effects of any evolutionary operator are reversible. In this paper we will show that reversibility and evolution are compatible by giving examples of crossover and mutation operators that are reversible.
Compared to the possibilities, behaviours of interest to us are usually rare. So we end up with the situation where many different starting populations, with undesirable behaviours, lead to one with a desirable one. Following such an evolution backwards provides us a number of advantages. On the practical side, with reversible evolution we can roll back an evolution scenario as many generations as we need, and have it start again, possibly in a new direction. This option is useful when we see a population stagnate, or does not reach a desired result.

This effect is possible to achieve in a non-reversible environment by taking snapshots of the population at certain intervals. A system can be rolled back by simply re-start from a desired snapshot. But a brute-force saving of the members of all the generations may not be practical. The information that needs to be saved to provide reversibility may require less space.

Of more importance is the characteristic that a reversible evolution can allow us to have the benefits of reversible computing, which include less heat dissipation. This decrease in turn results in less energy consumption during the execution of evolutionary operations. It is proved that reversible operators dissipates no heat [2]. To achieve this zero level of heat dissipation, the operators have to be executed very slowly. But even at higher speeds, the heat dissipation of a reversible operator is less than a non-reversible one.

The other advantage of reversibility is the possibility of applying Quantum Computing methods in evolutionary algorithms. Quantum Computing has the potential of exponentially speeding up function calculations [7], bringing us the possibility of improving the running time of fitness evaluation functions, for example.

We must make it clear that the problem considered in this paper concerns employing concepts from reversibility and Quantum Computing in developing evolutionary algorithms, and not in evolving reversible or quantum algorithms using genetic algorithms, as in [8].

The power of Quantum Computing comes from the fact that it can apply a function to a superposition of input values. For example, suppose that the fitness function $f()$ takes as input a characteristic of the

members, represented by an n-digit number. Then quantum methods allow all 2^n values to be computed by one call to the function $f()$. To be more exact, assume that this fitness function is represented by the unitary (reversible) operator Uf. We can evaluate Uf on all the input values of the population at once. We have thus evaluated an exponential number of $f()$ calls in one application of Uf. The catch is that though we have evaluated all the possibilities, the results are in a superposition of all the values, and a measurement will give the result of $f()$ applied to a single value. The system designers should devise a way to extract a single desired value.

In this paper we consider a classical genetic evolution system, in that we are concerned with standard operators such as crossover and mutation, and provide examples in which the genomes are represented as strings of numbers. In the proposed scheme the changes in the population are gradual, and the generations can overlap. The concept of overlapping different generations in the same population has been proposed by steady state evolutionary methods as in [9]. More generally, a system in which a whole generation is not replaced by the next one is called asynchronous.

The rest of the paper is organised as follows. In Section 2 we present the concept of reversible evolutionary algorithms. Section 3 elaborates on the problem and provides an algorithm for this method. Section 4 concludes the paper.

2. Reversible Evolutionary Algorithms

In a non-reversible evolutionary algorithm, such as a genetic algorithm, members of the current generation act as parents for the next generation. The current generation can later be discarded. The operations that are performed on the parent(s) usually do not preserve enough information to allow us to reconstitute the parents (input) from the children (output). For example, given two structures in generation n+1, there is usually no way to reconstruct the parents in generation n that were involved in producing the structures. To achieve a reversible evolvement of a population, we need to satisfy two requirements.

The first requirement is that the members of the population change their role from being "reproducing parents" to that of "growing up", meaning that during any operation, instead of creating a new generation. If this condition is not satisfied, then we would need to save each successive generation if we wanted to recreate them in a reverse order, which may not be practical or desired. Even then, this option would still not allow us to reconstruct only certain members of a previous generation.

The second requirement is that the operations that are applied to the population members must be reversible, thus enabling us to retrace the changes in the members in a time-reverse order. In such a case, when given the output of an operation, we can reproduce the input by performing the reverse operation. A reversible operator is thus one that does not lose any information when transforming the input to output. To satisfy this requirement we need operators that transform a number of input members into an equal number in the output. For example, suppose each member is represented by a sequence of 4 numbers between 0 and 9. The operator Crossover(1234, 5678) cuts the representation of each of its operands in half and cross-pastes them together, resulting in 1278 and 5634. The Operation Crossover(1278, 5634) restores the operands to their original states. In this case the Crossover() operator is its own reverse.

The obvious problem with this method is that, in addition to the operator, in each step we also have to know which members of the population were used as inputs. The solution is to assign a unique number

to the members of the population. A built-in argument determines which members will be acted upon. For example, $Crossover_{50,80}()$ would affect members number 50 and 80. It is easy to see that $Crossover_{50,80}()$ and $Crossover_{80,50}()$ are the same.

A single-argument operator, as in a mutation, is implemented by similar means. For example, $Flip_{55,3}()$ would subtract 9 from the 3rd bit of member number 55. In this case this operator is its own reverse too. In general, a reversible operator *RO* can be represented as $RO_{j,...,k}(Member_j,...., Member_k)$, and acts on the members specified by the arguments. The members are then replaced by the results. The reverse operator, represented by $R\text{-}RO_{j,...,k}(Member_j,...., Member_k)$, undoes the effects of *RO*(). As we have seen, *RO* and *R-RO* may be the same.

Going from one generation to the next takes the form of applying a number of operators to the population and keeping track of the order of the applications. When done, we end up with a new generation, which can be subjected to the process of fitness measurement as usual in genetic algorithm.

Usually we create a new generation while the previous one stays intact, because the parents and the children are kept separate. Here the population changes gradually as the operations are applied. For this reason the order of applying the operations is important, and changing the order would lead to different results. For example, the results of executing $Crossover_{10,30}()$, followed by $Flip_{10,3}()$ differ from that of executing $Flip_{10,3}()$. first, and then $Crossover_{10,30}()$.

As a result, with a reversible evolutionary algorithm the classic notion of having successive generations can be set aside, giving us a population that is gradually and continuously changing. Different generations can be simulated in such a reversible environment by considering arbitrary checkpoints in the application of operators. The checkpoints may appear after either a fixed or a varying number of operations. For example, applying every *N* operators can result is a new generation. In such a case, the previous generation is derived by applying the same number of reverse operators, and in the reverse order of their original application.

A time-reversal of such an algorithm can be achieved by pushing each operator (or its reverse) into a stack. To role the system backwards, and thus consume as little energy as possible, one can follow the procedure common in Quantum Computing: First the population is evolved as desired, then the best solutions are saved, and finally the evolution is undone by removing the operators from the stack and performing the reverse operations.

3. Towards a Reversible Evolutionary Algorithm

Selection of structures for evolvement is a an essential ingredient in evolutionary algorithms. A fitness function is often used for this purpose, but to make sure that the population is not stagnating, some members may be chosen randomly for change. So one point to consider is creating randomness in a reversible environment. This effect is achieved by way of creating operators on the fly that affect randomly-determined members of the population. From a general pattern called the $Flip_{x,y}()$ operator, for example, one can automatically generate $Flip_{330,1}()$ to affect the 1st bit of member number 330, or $Flip_{16,2}()$ to affect the 2nd bit of member 16.

If the members are to be affected not randomly, but according to a fitness criterion, then the first step would be to determine a ranking of the members of the population according to their fitness. Doing so may lead to a simulation of the notion of generations, where, as discussed before, at certain intervals

the algorithm stops applying evolutionary operators and computes the fitness of every member of the population in preparation for growing them into the next generation.

After that, as members are selected for evolution, appropriate operators are created to affect them by choosing the corresponding membership numbers. For example, if it is determined that members 2 and 3 are to mate, then a $\text{Crossover}_{2,3}()$ is generated. To make the implementation of such a reversible system easier, we can opt to design operators that are easy to generate in both temporal orders.

One side-effect of the proposed method is that the number of members in the population will remain constant, as after creating the original population, we only evolve them by a process of morphing, rather than reproduction and deletion.

Algorithm 1 presents the steps that are taken to make an evolutionary algorithm reversible. Here after applying *N* operators, a new generation is considered to have been created. Executing Step 5 of this algorithm ensures that we end up with the same original population and an empty stack.

Reversible Evolutionary Algorithm. Given: Maximum number of generations, number of operations per generation, fitness function(s).

0. Create an empty stack *r*.
1. Create the original population.
2. Sequentially number the members.
3. While (not done) {
    3.1 Apply a fitness operator to the members and rank them.
    3.2 If fitness objectives are achieved, then done.
    3.3 If maximum iteration number exceeded, then done.
    3.4 Do *N* times {
        3.4.1 Determine the member(s) selected for change.
        3.4.2 Create an operator that takes on the appropriate number of members as input.
        3.4.3 Push the operator down the stack *r*.
        3.4.4 Perform the operator.
  }
}
4. Save the results
5. To undo the effects of our computations, apply the operators in reverse.

Algorithm 1. A Reversible Evolutionary Algorithm

Algorithm 1 is similar to a non-reversible evolutionary algorithm, and has similar abilities. The main differences concern the choice of reversible operators, and also recording the order of their application.

4. Concluding Remarks and Future Work

We introduced the concept of a reversible evolutionary algorithm. Though such algorithms it is by definition possible to follow the evolution of the members of the population from the beginning to the end, with a reversible evolutionary algorithm we can do the reverse, going from an evolved population to one that does not show the desired behaviour. We saw that reversibility has the potential of reducing the power consumption of the system, and presented a simple algorithm to make the evolution of a

population reversible. Both the process of evolution (applying genetic operators) and the fitness function itself can be chosen to be reversible.

Though the examples in this paper have come from genetic algorithms, the same principles can be applied to related disciplines such as genetic programming. For example, in a tree representation of a genetic programme, the crossover operator can be implemented in such a way that the input programmes can be re-produced from the input trees. This mechanism would be very similar to the reversible Crossover() operator in this paper.

Though currently at a theoretical stage, the property of reversibility can be of benefit to evolutionary algorithms by way of reducing power consumption and speeding up the computation of a fitness function. A possible future work is implementing a reversible evolutionary system. Since reversible operators form a subset of all possible operators, another possible future work is investigating the effects of choosing only reversible operators on different aspects of the evolvability of a system, such as convergence speed.

References

1. Bäck, T., Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary programming, Genetic Algorithms, Oxford University Press, New York, 1996.
2. Bennett, C. H., Logical Reversibility of Computation, *IBM Journal of Research and Development 17*, November, 1973. pp. 525-532.
3. Karimi, K., Johnson, J.A., and Hamilton, H.J., A Proposal for Including Behavior in the Process of Object Similarity Assessment with Examples from Artificial Life, *The Second International Conference on Rough Sets and Current Trends in Computing (RSCTC'2000)*, October 2000. pp. 603-607.
4. Karimi, K., Discovery *of Causality and Acausality from Temporal Sequential Rules*, Ph.D. Thesis, Department of Computer Science, University of Regina, 2005.
5. Koza, J.R., Bennett, F.H., Andre, D., Keane, M.A., *Genetic Programming III: Darwinian Invention and Problem Solving*, Morgan Kaufmann, 1999.
6. Mielsen, M., and Chuang, I., *Quantum Computation and Quantum Information*, Cambridge University Press, 2000.
7. Pittenger, A.O., An Introduction to Quantum Computing Algorithms, Birkhäuser, 2001.
8. Surkan, A.J. and Khuskivadze, A., Evolution of Quantum Algorithms for Computer of Reversible Operators, *NASA/DoD Conference on Evolvable Hardware (EH'02)*, 2002. pp.186.
9. Whitley, D. and Kauth, J., GENITOR: A Different Genetic Algorithm, *Technical Report CS-88-101* Colorado State University, 1988.