A Generic Functional Genetic Algorithm

John Hawkins and Ali Abdallah
School of Computing, Information Systems and Mathematics
South Bank University
London, SE1 0AA, UK
{John.Hawkins, A.Abdallah}@sbu.ac.uk

Abstract

We present a generic genetic algorithm expressed in the functional programming style, specifically the language Haskell. This is a single, higher order function that can encapsulate the functionailty of almost any Genetic Algorithm, and is accompanied by a library of standard GA components. We demonstrate its flexibility by expressing a number of differing case studies as specific instances of the generic solution.

1 Introduction

The use of Genetic Algorithms (GAs), or Evolutionary Computing, provides a method of creating approximate solutions to a wide class of complex and/or intractible problems. It has been widely studied and applied in both academic and industrial contexts [3].

Despite the wide variety of problems solved by the evolutionary method, it can be said that there are a large number of concepts similar between each and every implementation. The basic requirements of a GA are, firstly, to have some method of representing potential solutions to a problem within the algorithm. Next, we require a means of evaluating them so that the relative merits of different potential solutions can be compared. This is often referred to as a fitness function. In addition it is necessary to have some means of making small random changes to potential solutions to produce similar but new solutions, a process generally named mutation. Finally it is often also necessary to have some means of combining the attributes of two or more solutions together, again to produce new solutions. This is usually termed crossover.

It may be said, therefore, that the similarities that lie among GAs might lead us to look towards some kind of generic library for evolutionary computation, with which new problems can be tackled with a minimum of effort.

There are already a number of libraries and systems that perform this role. However, what isn't perhaps so common is the use of functional programming for the implementation of Genetic Algorithms. Functional programming has, interestingly, been used in some depth in the context of Genetic *Programming* (see [5, 8, 9]), the application of Genetic Algorithms for the evolution of programs. Thi is one specific area in the general field of evolutionary computing. We are interested here, however, in the more general case.

Functional programming facilitates the generic approach through higher order functions. Functions in such languages are treated as first class values, they can be passed as parameters or returned as results in exactly the same way more basic types can. Thus we can specify a generic framework, parameterised by the bits of functionality that are instance specific. Additionally we have polymorphism, whereby a single function can be given that will operate over a number of types.

The generic programming approach has been used in many other areas. One particular common example is for the divide and conquer paradigm. Here a single generic higher order function can capture the functionality of any divide and conquer algorithm. One possible implementation of this generic function is defined in Figure 1. Given this we can define merge sort, for example, in a very concise manner:

```
mergesort = dc (id) ((<=1) .  length)
               (msplit) (fold merge)
```

Given a function `msplit`, which takes in a list and returns a list of lists, containing together the contents of the input list; and `merge`, a function which takes in two sorted lists and merges them to form a single sorted list.

The advantages of this generic approach to programming are numerous. There is of course straightforward code reuse - a classic 'Holy Grail' of software development. As a side effect of this, if the generic framework is known to be correct when written, we can assert that it is also correct every time instantiated, so any problems found can be

0-7695-1165-1/01 $10.00 © 2001 IEEE

```
dc trivialCase isTrivial split combine input
  = if isTrivial input then trvialCase input
    else combine (map dc' (split input))
    where dc' = dc trivialCase isTrivial split combine
```

Figure 1. A Generic Divide and Conquer Algorithm.

attributed to the instance-specific components used. In addition we are presented with a clear design strategy - to, in essence,'fill in the blanks'. Often we may find that certain components used are actually applicable to more than one problem, taking reusability further still. Generic programming can also provide a good mechanism for testing different components and how they effect the overall operation of the algorithm. Finally, such a view of an algorithm gives us a clear perception of the underlying functionality, which can be a great aid to our understanding of it. Let us not forget that this kind of understanding is not just of academic merit- a poorly understood algorithm is one which will almost certainly not be implemented correctly.

In this work we attempt to demonstrate the merits of a similar generic, functional approach in the area of genetic algorithms, specifically with the non-strict, purely functional language Haskell (see [2]).

The rest of this paper continues as follows. In Section 2 we look at the standard components that go to make up a genetic algorithm, along with some specific examples. In Section 3 we introduce the actual generic genetic algorithm. Following this, in Section 4, we present three example problems - Bin Packing; Travelling Salesperson and N-Queens. The paper is then concluded in Section 5.

2 Genetic Algorithm Components

2.1 Chromosomes

A chromosome is a problem specific representation of a possible solution to that problem. Polymorphism, one of the merits of using a functional language, allows us to specify this as being of any type.

Perhaps the most common representation is a simple list of items. Each of these items is termed a gene. These genes can take any one of some finite number of values, or alleles. Thus there are a^g possible combinations, where g is the number of genes, and a the nunber of alleles. This type of chromosome can often be modelled as a straightforward list of integers.

Often we are more interested in the ordering of the items, for example with the Travelling Sales Person problem (see Section 4.2). In this case certain combinations of the chromosome may not be valid (for example, the same value appearing twice), so we must be careful to choose functions that preserve this structure. These are termed order-based chromosomes.

The framework does not limit us to lists, however, any value valid in a functional language may be used to represent a chromosome. Recursive datatypes (such as trees) and even functions are all potential candidates.

2.2 Evaluated Chromosomes

Vital to the functioning of a GA is the ability to compare the relative merits of one chromosome against another. For this we give chromosomes a fitness value. Usually (at least for all the case studies presented here) the lower the fitness the better.

An evaluated chromosome is represented as a tuple. The second element is the actual chromosome, and the first element is the corresponding fitness value. We nominally use the Haskell type `Int` here to represent fitness values. In a more flexible framework we may want to have this user-definable.

```
type FitnessValue = Int
type Evaluated c = (FitnessValue,c)
```

2.3 Populations

A population is a collection of chromosomes. Here a list representation is used. It is often convenient to keep the population in fitness order, thus we use evaluated chromosomes.

```
type Population c = [Evaluated c]
```

2.4 Fitness

A fitness function is one that takes a chromosome and calculates its fitness value. This is perhaps the most problem specific of all the functions described here, a unique fitness function will probably be required for every problem tackled. The type is quite trivial:

```
type Fitness c = c -> FitnessValue
```

2.5 Random Chromosomes

One necessary task in genetic algorithms is to construct 'random' chromosomes. Such a function need only take in a seed from which to generate random numbers, and should output a chromosome (unevaluated).

```
type RandomChromosome c = Seed -> c
```

The type `Seed`, used here and throughout this work, is defined as follows:

```
type Seed = Int
```

2.6 Initial Population

Given those components already defined, we can now create an initial population. The exact way this population is constructed may vary from problem to problem, but one fairly typical method is encapsulated by the function `makepop`. For this we require a random chromosome function, a fitness function, the size of the population, and finally a seed.

```
makepop ::  Size -> Fitness a ->
            RandomChromosome a ->
            Seed -> Population a
```

The actual implementation is relatively straightforward and requires just a couple of lines of code. One method might be to generate a list of n (the value of the `Size` parameter) random numbers from the seed given, and pass each separately to the random chromosome generating function to generate n chromosomes. We then have to apply the fitness function to each and order them on their fitness values.

2.7 Mutation

Of course we cannot evolve our population until we have some mechanism for making changes to exisiting chromosomes. This is where mutation and crossover come in. As mutation is often the simpler of these two, we shall deal with it first. A mutation function takes in a chromosome and a seed. It should then return a chromosome based on that passed in but with a small 'random' change.

```
type Mutate c = c -> Seed -> c
```

2.7.1 Allele Swap

A widely applicable mutation function, as it can be used for order-based chromosomes (such as used in the Travelling SalesPerson and N-Queens problems looked at later) as well as those with a more standard representation, is the allele swap. Here two genes are chosen at random, and their alleles are swapped.

2.8 Crossover

Crossover is the process by which attributes of two or more chromosomes are merged together in a random fashion to produce a new chromosome. Most commonly only two parent chromosomes are employed, however to maintain generality we shall provide this function with a list of chromosomes. As usual we also pass in a seed to allow a degree of randomness. Crossover functions therefore have the following type.

```
type Xover c = [c] -> Seed -> c
```

As with mutation, crossover functions may also be shared between different problems.

2.8.1 One point Crossover

This is explained in more depth in [3]. An implementation can be given quite concisely using `take` and `drop`. Taking two parent chromosomes each of length n, a random value k is chosen in the range $0..(n-1)$ as a crossover point. The new chromosome then comprises the first k genes of one parent (`take pa m`), and the last $n-k$ genes of the other parent (`drop (nrgenes-pa) f`).

```
onepoint2p ::  NrGenes -> Xover a
onepoint2p nrgenes [m,f] sd
  = take pa m ++ drop (nrgenes-pa) f
   where pa = (rnums!!0) `mod` nrgenes
         rnums = rands (newSeed sd)
```

There are a variety of other crossover operators we could implement along similar lines to the above, including some common ones such as two (or generalised to n) point and uniform crossover. Again, see [3] for further details.

2.9 Selection

Obviously, we do not always wish to use the same parent chromosomes for mutation and crossover, and so we must employ a selection function. The selection function is passed the population and, as always, a seed, and based on some heuristic (which should involve some randomness) should return a single chromosome. For convenience the size of the population is also passed in.

```
type Select c = Population c -> Seed ->
                Size -> Evaluated c
```

Selection functions will be highly generic as they will almost always only examine the fitness value of a chromosome rather than the chromosome itself. Thus we can create a library of these to apply to almost any concievable problem.

2.9.1 Tournament Selection

With tournament selection, we choose n (2 for example) chromosomes at random from the population, compare their fitnesses and return the chromosome with the best (usually this means the lowest) value. This is both very effective in maintaining population diversity, and very simple to implement.

```
tsel ::  Size -> Select a
tsel n pop popsize size
  = fstminimum .  map (pop!!)   .
    map (`mod` popsize) .
    (take n) .  rands .  newSeed
```

Here the full stop is the functional composition operator. The output of the function on the right of this is used as the input to that on the left. So, reading from right to left. We first generate a new seed, then create an infinite list of random numbers, take just the first n of them, then map a modulus function to each to fix them in the range 0 up to $popsize - 1$. These numbers can then be used as indices to the population, and we map the index operator (`!!`) over these to retrieve chromosomes from the population at the corresponding position. Finally, the function `fstminimum` takes in a list of pairs and returns the pair with the lowest first item, ignoring the value of the second item.

Again, there are a variety of other selection functions we could implement, including roulette wheel (or fitness proportionate) and rank based.

2.10 Merging

The behaviour of a genetic algorithm is largely characterised by how the new chromosomes (created by mutation and crossover) are integrated into the existing population. This task is undertaken by the merge function. This function effectively takes in two populations, the existing one and that formed by the new chromosomes. In addition a seed as always, in the event that some randomness is required in the merging process. It returns the new population, which will contain some, all or none of the chromosomes from either population.

```
type Merge c = Population c ->
               Population c ->
               Seed ->
               Population c
```

2.10.1 Stable State

In a stable state GA we generate a small number of new chromosomes (say 1) in each generation, which then replace the worst chromosomes from the previous population.

```
stablemerge ::  Merge a
stablemerge old new sd
 = popinsert (head new) (init old)
```

The function `popinsert` simply inserts the new chromosome in the population at the right position, assuming the population is sorted by fitness. The functions `head` and `init` retrieve the first, and elements but the first from the list respectively.

Another common type of merging is generational, where we replace the entire population at each generation. Further types of merging are possible. For example, in an attempt to maintain population diversity, we may avoid inserting new chromosomes into the population that have similar fitnesses or are in themselves very similar to existing ones. We may also envisage a stable state merge where several new chromosomes are produced at each step, and then the best is chosen to be inserted into the population.

3 The Genetic Algorithm

First let us define two simple synonyms as an aid to readability:

```
type NrParents = Int
type NrOffspring = Int
```

Given the higher order nature of functional languages, we can now create a single generic function, taking in all of the above components, that performs the actual evolution.

```
gga :: Population a -> Count -> Seed ->
       Size -> NrParents ->
       NrOffspring -> Probability ->
       Fitness a -> Mutate a ->
       Xover a -> Select a ->
       Merge a -> Population a
```

As can be seen, this takes in a number of functions as already described. Note here, if it isn't obvious, that the type variable a denotes the type of the chromosome for the problem being solved. In addition there is one `Count` parameter (number of generations to evolve for), a `Seed` parameter, a `Size` parameter (the population size), a `NrParents` (per

crossover) parameter, a `NrOffspring` (created per generation) parameter, and a `Probability` parameter. The `Probability` parameter is the crossover rate, a value between 0 and 1 which gives the probability of each offspring created being generated by crossover or by mutation.

The implementation shall not be given here in full for space reasons. The algorithm is broadly an iterative procedure, along the following lines. At each step, we check the number of generations to evolve for, and if it is zero, we return the current population. Otherwise we choose a list of parent chromosomes from the current population using the selection operator. We then use the crossover and mutation functions to generate a group of new offspring from these. The decision as to which function to use is based on the crossover rate. The merge function is then applied to integrate the new offspring into the existing population, the number of generations to evolve for is decreased by one, and the process repeats.

4 Case Studies

4.1 Bin Packing

In this problem, we are given b bins and w weights, weighing differing amounts. The object is to assign all the weights to bins, to give as equal a distribution as possible. This can be gauged by totalling the weight held in each bin, and examining the difference between the heaviest bin and the lightest bin. The smaller this difference, the better. We can represent a potential solution to this problem as a list of integers. Each of these integers (genes) corresponds to a particular weight, and its value (allele) denotes which bin this weight resides in. Thus we have b^w potential combinations.

```
type BinPackChrom = [Int]
type Weights = [Int]
```

An example configuration is given in Figure 2. Here we have 8 bins and 20 weights. This is in fact the best chromosome found for this problem in a typical run. It has a fitness of 13 and was found after about 500 generations.

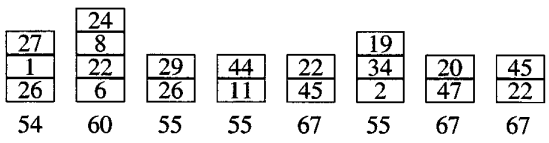

Figure 2. Best Bin Packing Configuration Found

Given the above definition, a fitness function for this problem is relatively easy to implement, and requires only about 7 or 8 lines of Haskell code. A random chromosome can be produced simply by generating w random numbers between 0 and $b-1$.

As regards mutation, we have a choice of several of the standard mutation functions. For now we will use single allele change. A similar choice applies to crossover, selection and merging. We will use two point crossover, tournament selection (size 2) and stable state merging. We can then concisely define the solution to our problem, as one particular instance of our generic genetic algorithm.

```
binpackga :: Seed -> Count ->
             Population BinPackChrom
binpackga sd n
 = gga (initpop sd) n (newSeed sd)
       popsize 2 1 xRate (binfitness)
       (binmutate) (binxover)
       (tsel 2) (stablemerge)
```

Here `binmutate` and `binxover` are basically `changemutate` and `twopoint2p` respectively, functions taken from the library, with the relevant parameters specified. The function `initpop` used here is written in terms of our generic function `makepop`, described earlier. Also, `popsize` and `xRate` are constants which can be defined elsewhere.

The above hopefully demonstrates how compact and generic the system is. In order to provide a solution to the binpacking problem, we have only really had to write one custom function (to calculate the fitness), after which we've been able to get everything else we need 'off the shelf'.

4.2 Travelling SalesPerson

Perhaps the most commonly looked at GA problem. Here we have n cities, all of which must be visited once only, and our job is to find an order in which to visit them that gives the shortest overall route length. We assume here that direct paths exist between every pair of cities, and the length of these can be determined simply by euclidean distance between the two. An example of such a route, for 15 cities in a 100 by 100 area is given in Figure 3.

For simplicity's sake we shall represent the route as a list of the co- ordinates of the cities. Thus it should be a trivial task to write a fitness function; one that simply adds the distances between each adjacent set of co- ordinates (remembering also to add the distance between the first and the last). For mutation we can use the standard allele swap operator, as described in Section 2.7.1.

Crossover is a slightly more complicated issue. In TSP, the attributes we are trying to preserve are essentially the edges between cities, rather than the actual ordering. For example, it is likely to be of far more consequence that A is adjacent to B, than the fact that A is somewhere before C

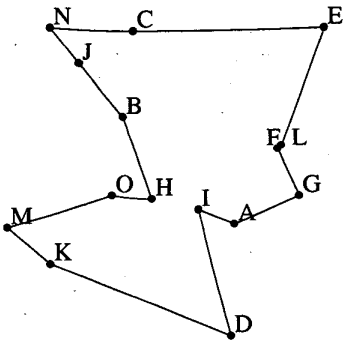

Figure 3. Best TSP route found for 15 cities

in the ordering. Thus a commonly used TSP operator takes in the two orderings, extracts from them all their edges, and forms a new route from a mixture of these edges.

As usual, for selection and merging we can use entirely standard functions. Thus again we can specify a solution as an instance of the gga function.

```
tspga :: Seed -> Count ->
         Population TspChrom
tspga sd n
 = gga (initpop sd) n (newSeed sd)
       popsize 2 1 xRate (tspfitness)
       (orderedSwap nrcities)
       (orderedXover2p)
       (tsel 2) (stablemerge)
```

There is a formula to calculate an expected minimum for a TSP route given the area and the number of cities:

$$0.765 \times \sqrt{(n+1) \times w \times h}$$

For the problem tested, the expected minimum is therefore 306.0. In practice we have often found this formula to underestimate, thus we can be relatively assured that the chromosome found after 500 generations, with a fitness of 360 (pictured in Figure 3) is fairly near optimal.

4.3 N-Queens

Perhaps not such a conventional GA problem is the N-Queens problem. Here we have the task of placing n queens on an $n \times n$ board such that none are able to take each other. In theory there is a vast search space for this problem. We have:

$$\frac{(n^2)!}{(n^2-n)!}$$

total possible combinations of configurations. However, given a more sensible representation, we can limit this somewhat. The above would correspond to storing the co-ordinate of every queen, and only stipulating that no two queens can occupy the same square. We can of course derive a representation whereby no two queens can occupy the same row or colum either, if we represent the board as a permutation of the queens row positions.

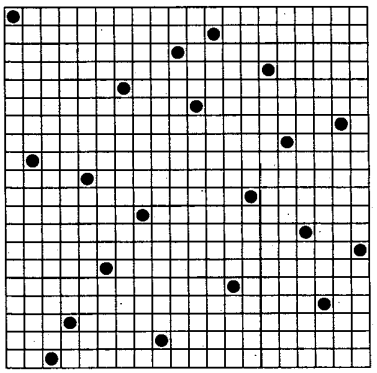

Figure 4. A Solution to the Queens Problem With 20 Queens.

For a fitness function, we can count the number of queens that are able to take other queens, using a modified version of the function given in [1]. We will then attempt to minimise this value, given that a fitness of 0 would give us a valid solution. For mutation we can use the same allele swap used previously. Crossover is again a little more complicated, as we are (similar to TSP) more interested in the relative positionings than the overall order. However, as always, we can use standard functions for selection and merging.

```
nqueensga :: Seed -> Count ->
             Population QueensChrom
nqueensga sd n
 = gga (initpop sd) n (newSeed sd)
       popsize 2 1 xRate
       (queensfitness)
       (orderedSwap nrqueens)
       (orderedXover2p)
       (tsel 2) (stablemerge)
```

The first chromosome found with a fitness of zero in a typical for the 20 queens scenario was as follows:

[20,12,1,3,11,6,16,9,2,18,15,19,5,10,17,13,8,4,14,7]

This configuration can be seen in Figure 4, and took about 600 generations to reach.

5 Conclusion

We have presented a single higher order function which can encapsulate the functionality of almost any genetic algorithm. This is accompanied by a library of standard GA functions, which, as shown, can be used in a variety of different situations. We have demonstrated this by using it to solve a collection of differing problems, and have received real usable results. In doing this we have also demonstrated the speed and ease with which such solutions can be constructed, due to the large scale code reusability and the design methodology inherant to the system. Hopefully this approach also gives a good overview understanding to those not familiar with the field of evolutionary computation. This approach also provides scope for parallelisation of these algorithms, a subject which will form the focus of future work.

There are a variety of other problems that have been solved using this system. A good example is the layout problem, covered in detail in [4]. Here we need to arrange a number of rectangular shapes to fit into a minimal area. The representation for such a configuration is a tree structure, and so quite different to any of the problems looked at here. This requires us to write specific mutation and crossover functions, however, selection, merging and everything else can still be used 'off the shelf' - further proof of the wide applicability of this approach.

References

[1] R. S. Bird, and P. Wadler, Introduction to Functional Programming, Prentice-Hall, 1988.

[2] R. S. Bird, Introduction to Functional Programming Using Haskell, Prentice-Hall, 1998.

[3] D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989.

[4] J. Hawkins, and A. E. Abdallah, A Generic Functional Genetic Algorithm, in Draft Proceedings of the 1st Scottish Functional Programming Workshop, 1999.

[5] L. Huelsbergen, Toward Simulated Evolution of Machine-Language Iteration, In Proceedings of the Conference on Genetic Programming, pp 315-320, 1996.

[6] J. R. Olsson, Population management for automatic design of algorithms through evolution, International Conference on Evolutionary Computation, IEEE Press, 1998.

[7] R. H. J. M. Otten. Automatic floorplan design, In Proceedings of the 19th ACM-IEEE Design Automation Conference, pages 261-267, 1982.

[8] P. J. Walsh, A Functional Style and Fitness Evaluation Scheme for Inducting High Level Programs, In Proceedings of the Genetic and Evolutionary Computation Conference W. Banzhaf et al, eds., Morgan Kaufmann, 1999.

[9] T. Yu and C. Clack, PolyGP: A Polymorphic Genetic Programming System in Haskell, In Proceedings of the Third Annual Genetic Programming Conference, 1998.