A Generic Functional Framework for Developing Genetic Algorithm

John Hawkins and Ali Abdallah

Centre of Applied Formal Methods
South Bank University
102 Borough Road, London, UK
{John.Hawkins,A.Abdallah}@sbu.ac.uk

Abstract. Almost all genetic algorithms are different instances of a single skeleton which is neatly expressed as a higher order function in Haskell. This paper is intended to give a brief overview of evolutionary computation and to present a general functional framework for developing genetic algorithms (GAs). This is mainly centered around a generic functional GA skeleton and a rich library for instantiating its components. The usefulness of this framework is demonstrated by considering four different computationally complex problems and developing GA solutions to these problems within the functional framework.

1 Introduction

The term *Genetic Algorithm* [?,?,?], or Evolutionary Algorithm, refers to a class of approximate search and optimisation techniques that mimic evolutionary behaviour found in nature. They are often used to solve problems that are very complex, or for which an exact algorithm is not yet known, but for which an optimal solution is not necessarily required. Genetic Algorithms have been studied widely in both academic and industrial contexts, and as such there exists an enormous quantity of differing methods and implementations. Despite this diversity almost all GAs will share some common fundamental concepts; notions such as chromosomes, population, fitness, selection and reproduction. It is therefore not unreasonable to consider whether it might be possible to specify a generic scheme, a framework which captures the functionality of the entire class of genetic algorithms, or at least a good portion thereof.

Generic Programming [?] has been applied to many other classes of algorithms. The notion is a simple, yet powerful one. The commonalities between algorithms in a particular class are specified in a block of code parameterised by those sections where the algorithm's behaviour is more problem specific. Terms such as *skeletons* and *patterns* are often coined to describe these generic programming constructs. The merits of generic programming go beyond the obvious reusability of code and speed of development. Through generic programming we aim to get a deeper understanding of the algorithm, often allowing us to realise that algorithms we previously thought were entirely distinct are in fact instances of the same scheme. Exactly this is achieved within this work, and towards the end is a short discussion on how other quite different AI search techniques can in fact be viewed as instances of our GA scheme.

High level functional programming languages such as *Haskell* [?] (used throughout this work) provide an excellent framework for generic programming through higher order functions, polymorphism and so on. Although genetic algorithms have traditionally been almost exclusively within the domain of the imperative programmer, we hope to prove that, for the above reasons, a language such as Haskell does in fact provide a very good environment for GA development.

It is therefore the purpose of this paper to present a generic, functional genetic algorithm. This comprises a single, higher order function, which encompasses the functionality of the majority of genetic algorithms commonly used. It is parameterised by components familiar to the GA developer, and a library of these has also been implemented to speed development further as many are re-usable between different problems. The use of this framework is demonstrated with an assortment of non-trivial case studies, which illustrate both its flexibility and simplicity.

The rest of this work is divided as follows. Section 2 gives an overview of the components that go to make up a standard GA, and how they are modelled within the system. Section 3 describes the genetic algorithm itself. Section 4 looks at four non-trivial case studies. Section 5 examines some related work, and finally Section 6 concludes.

2 Genetic Algorithm Components

Genetic algorithms operate by maintaining populations of potential solutions. We refer to these potential solutions throughout this work as *chromosomes*. The usual scheme is to start by generating a random population of chromosomes. We then require some means of evaluating these chromosomes, which is achieved with a fitness function. This assigns some metric to each chromosome which can be thought of as indicating how close it is to being an acceptable solution, and allows us to compare one chromosome against another. It is unlikely (but not impossible) that an acceptable solution to our problem will exist in our initial population, so we will need to create some new chromosomes from those already there. We want the quality of our population to improve over time, so we employ selection, in the same way we might find in nature, whereby fitter chromosomes are more likely to have the chance to reproduce. Once chromosomes have been selected for reproduction, we can apply operators to them that make small changes to the characteristics of single chromosomes (mutation), or that combine the characteristics of several chromosomes (crossover). We now have a group of new chromosomes, which we should evaluate and integrate into our existing population, after which the cycle repeats. Eventually it is hoped the population will contain a chromosome that represents an acceptable solution to our problem.

2.1 Chromosomes

The term *chromosome*, as mentioned above, is used to refer to a potential solution to the problem in hand. One of the main aims of this framework is to have as wide applicability as possible, so we make no restriction whatsoever on the typing of chromosomes.

Often in GAs, chromosomes are represented as fixed length lists of numerical values, all within the same range. Each of these values is termed a *gene*, and each of the values it can take is termed an *allele*. Given g genes and a possible alleles we have g^a possible combinations. This representation is demonstrated in the Bin Packing problem, in Section **??**.

Also fairly common are permutation problems. Here we have a list of n values, and are attempting to find a particular ordering of these values. Thus we have $n!$ possible combinations. We must take care with these representations to choose operators that preserve their structure, i.e. do not allow values to be repeated or left out, and so on. This type of representation is demonstrated in the Travelling Salesperson (Section **??**) and N Queens problems (Section **??**).

As noted, we make no restriction on the type of chromosomes. One example is a binary tree structure, as used in the Layout problem (Section **??**). Again, we need to take care here to choose operators that respect the structures used.

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. An evaluated chromosome is represented as a tuple. The second element is the actual chromosome, and the first element is the corresponding fitness value.

```
type Evaluated f c = (f,c)
```

We can read this as, an evaluated chromosome, in a problem where fitness is represented by a value of type `f`, and a chromosome is represented as a value of type `c`, is a simple pairing of these types.

2.3 Comparing Chromosomes

Note we make no class restrictions in the above definition of a fitness value. Neither are we forced to have a fitness value that is numerical nor ordinal, although in almost all cases they will probably be both. So that these values can be compared, we must supply a function of type `CompareFitness`.

```
type CompareFitness = f -> f -> Bool
```

This should take in two fitness values, and return `True` if the first is fitter, or `False` otherwise. This system also allows us to model both minimisation and maximisation problems, simply by changing this parameter.

2.4 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 f c = [Evaluated f c]
```

2.5 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 f c = c -> f
```

We may find the following function useful, this takes a fitness function and an unevaluated chromosome, and returns an evaluated chromosome.

```
evaluate :: Fitness f c -> c -> Evaluated f c
evaluate fitness c = (fitness c, c)
```

2.6 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
```

2.7 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, however mostly this will adhere to one of the two schemes below. The first will create an unsorted population.

```
type MakePopulation f c = Size -> Fitness f c -> RandomChromosome c
                          -> Seed -> Population f c
```

With the additional ability to compare fitness values, we can produce a sorted population.

```
type MakeSortedPopulation f c = CompareFitness f -> MakePopulation f c
```

An example function with a wide applicability for creating sorted populations is given below. The function `rands` is passed a seed and used to generate an infinite list of random numbers. We take the first `popSize` of these, each being passed to the random chromosome generator `randChrom`. We now have an unordered list of random (unevaluated) chromosomes. We give each of these a fitness value, and finally sort them, passing `isFitter` as a parameter to the function `popSort`, which sorts a population given an order implied by the `CompareFitness` parameter passed in.

```
makeSortedPop :: MakeSortedPopulation f c
makeSortedPop isFitter popSize fitness randChrom
 = popSort isFitter . map (evaluate fitness) .
   map randChrom . take popSize . rands
```

2.8 Reproduction

We cannot 'evolve' our population until we have some mechanism for creating new chromosomes from our existing ones. Often in GA implementations we employ techniques called *mutation* and *crossover*. Although these are usually handled quite separately, they are essentially two instances of the same scheme, which we shall call *reproduction*. This is modelled by a function of type `Reproduce`, shown below. Here we take in a list of unevaluated chromosomes (the parents), as well as a seed, and output another list of unevaluated chromosomes (the offspring).

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

The term mutation refers to asexual reproduction - there is just a single parent and (usually) just a single offspring. Crossover involves two (or more) parents, and will result in one or more offspring. To solve the potential problem of knowing how many parents to supply to a particular reproduce function, we pass an infinite list of parents. Lazy evaluation will of course ensure that only those parents needed for reproduction are generated.

Below we give two example reproduce functions. The first is a mutation - it takes a single parent and returns a single offspring. The second is a crossover - it takes two parents, and in this particular case returns two offspring.

Allele Swap A widely applicable 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. One possible implementation for this is given below. This is not necessarily the most efficient possible implementation, but highlights the required functionality. The type `NrGenes` is simply a more readable synonym for the type `Int`.

```
alleleSwap :: NrGenes -> Reproduce [a]
alleleSwap nrGenes (c:_) seed
 = [ setAt p1 (c!!p2) ( setAt p2 (c!!p1) c ) ]
 where p1 = r1 `mod` nrGenes
       p2 = r2 `mod` nrGenes
       (r1:r2:rs) = rands seed
```

The function `setAt` here takes a position, an item and a list and returns a list matching that passed in, but with the item at the specified position changed to the item given. Its definition is given in the appendix.

Note the use of the (_) character to identify the unused remainder of the infinite list of parents passed in.

In a real world implementation of this function, we might want to add additional code to guard against `p1` and `p2` being equal, as this would cause a clone of the parent chromosome to be created.

One point Crossover Given two parent chromosomes each of length n, a random value k is chosen in the range $[0..(n-1)]$ as a crossover point. The first new chromosome then comprises the first k genes of the first parent, and the last $n-k$ genes of the other parent. The second new chromosome is the inverse of this. This can be written quite concisely using `take` and `drop`.

```
onePoint2p2c :: NrGenes -> Reproduce [a]
onePoint2p2c n (m:f:_) seed
  = [ take k m ++ drop k f, take k f ++ drop k m ]
   where k = r1 `mod` n
         (r1:rs) = rands seed
```

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.

2.9 Choices

A common GA will use more than one reproduce operator. Often we may have one mutation operator and one crossover operator, with an associated parameter giving the probability of one being used over the other. In some cases we may have just a single operator (mutation or crossover) that is always used. Alternatively, we may want to give a selection of operators, each with an attached probability. All of these schemes can be modelled through the `Choices` type. This essentially is a list of values paired with their probabilities of being chosen..

```
type Probability = Double
type Choices a = [(Probability,a)]
```

Assuming the probabilities of these choices sum to 1, we can employ the function `choose` to pick one, respecting the probabilities given.

```
choose :: Choices a -> Seed -> a
```

2.10 Selection

Within our GA we require some scheme to dictate which members of the population are to be chosen for reproduction. This is achieved by the *selection* function. This essentially takes in the population and returns a list of evaluated chromosomes that have been selected, with a higher chance of fitter chromosomes being selected than the less fit ones. Most common selection functions will return just a single chromosome. The list return type is used here to allow scope for techniques like heterogeneous selection - where we select parents that are as dissimilar as possible to give a good mixup of gene information. The selection function is also passed the size of the population (for convenience), and a seed to allow some degree of randomness.

```
type Select f c = Population f c -> Size -> Seed -> [Evaluated f 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 conceivable problem. An example selection function is given below.

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 value. This comparison of fitness values requires a `CompareFitness` function to be passed in. This selection method is both very effective in maintaining population diversity, and very simple to implement.

```
tournamentSelect :: CompareFitness f -> Size -> Select f c
tournamentSelect isFitter n pop popSize
  = mkSingleton . chooseBest isFitter . map (pop!!) .
    map (`mod` popSize) . take n . rands
```

The definitions of `mkSingleton` and `chooseBest` are given in the appendix. Again, there are a variety of other selection functions we could implement, including roulette wheel (or fitness proportionate) and rank based.

2.11 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 f c
 = Population f c -> Population f c -> Seed -> Population f c
```

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 in the existing population.

```
stableMerge :: CompareFitness f -> Merge f c
stableMerge isFitter old new sd
 = popInsert isFitter (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. This requires a CompareFitness function to dictate the chromosome's ordering. Again, the definition of popInsert can be found in the appendices.

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.

2.12 End Condition

One obvious and important part of any search technique is knowing when to stop - so we must supply our genetic algorithm with an *end condition*.

```
type EndCondition f c = Population f c -> Bool
```

This simply takes in the current population, and returns a boolean. If true, the population is of, or at least contains a chromosome of, satisfactory quality. Otherwise evolution should continue.

3 The Genetic Algorithm

3.1 Parameters

Given the components as previously described, we are now able to construct our generic genetic algorithm. This will have the following type.

```
gga :: Population f c -> GAVars -> Count -> EndCondition f c ->
       Fitness f c -> Choices (Reproduce c) -> Select f c ->
       Merge f c -> Population f c
```

The first parameter (of type Population f c) is the initial population. Next there is a GAVars structure, giving all the numeric and boolean parameters the function will need, with the exception of the number of generations to evolve for, which is given as a separate parameter of type Count.

```
data GAVars = GAVars
               {populationSize :: Int,
                populationSizeVaries :: Bool,
                reproductionsPerGeneration :: Int,
                offspringPerGeneration :: Int,
                countReproductions :: Bool}
```

The populationSize value gives the size of the population, if it remains constant throughout evolution. For some problems this may not be the case, so we also have a boolean value, populationSizeVaries. If this is true the size of the population will be recalculated every generation. The next three values relate to how many offspring will be created in each generation. If countReproductions is true we will perform a number of reproductions equal to reproductionsPerGeneration. This is useful if we have reproduction operators which produce an unpredictable number of offspring, but we want to ensure that all of them are at least considered for inclusion in the new population. In the alternative scheme, where countReproductions is false, we instead want to produce a fixed number of offspring every generation - performing as many reproductions as is necessary to produce the given number (`offspringPerGeneration`) of offspring. This scheme is useful if we have two or more reproduction operators that produce differing numbers of offspring but we want the number created each generation to be constant.

Returning to the parameters to `gga`, next we have the end condition and fitness function for our problem. After this we specify the reproduce functions used, contained in the `Choices` type, described earlier. As the two final parameters we have our chosen selection and merging functions.

3.2 Implementation

The full implementation of the `gga` function is included in the appendices. Here a high level description of its general functionality is given.

- **Testing**
  If the current population satisfies the end condition, or we have reached the maximum number of generations allowable, we can stop evolving and return the current population. Otherwise we must create a new population, as described below.
- **Selection**
  Select parents for reproduction using the given selection function.
- **Reproduction**
  Create 'litters' of offspring. Each litter is created by randomly choosing a reproduce operator from those passed in, and passing to it a list of parents, created above. The number of offspring created depends on the values set in the `GAVars` parameter.
- **Evaluation**
  Each new offspring is assigned a fitness value by applying the given fitness function.
- **Merging**
  A new population is created by merging the now evaluated offspring into the existing population, using the given merge function.

4 Examples

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. This problem is a member of the more general class known as *knapsack problems* - see [**?**].

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

An example configuration is given in Figure ??. Here we have 8 bins and 20 weights. The weights are randomly generated, taking values between 1 and 50. The configuration depicted is the fittest found in a randomly generated initial population of 10. It has a fitness of 105 - the lightest bin totals 63 and the heaviest totals 168. This is represented by the following chromosome:

[0,1,3,3,1,2,0,1,0,0,2,4,3,2,4,0,3,2,3,0]

Given the following weights (randomly generated):

[49,48,17,27,5,19,40,50,21,8,25,43,50,22,20,1,48,13,3,49]

So, the first weight (49) goes to bin 0, the second (48) goes to bin 1, the third (17) goes to bin 3, and so on.

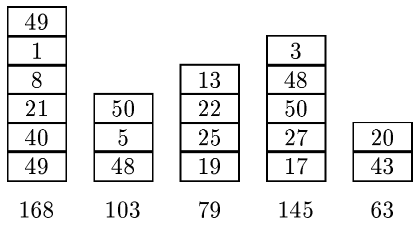

Fig. 1. Example Bin Packing Configuration

Fitness A fitness function for this problem can be implemented as follows:

```
binFitness :: Fitness BinPackFitness BinPackChrom
binFitness = weightDiff . map fst . foldr addWeight [] . zip weights
```

We first pair each bin assignment in the chromosome with its corresponding weight value. Next the function `addWeight` is used to accumulate the total weight held in each bin.

```
addWeight (nw,nb) [] = [(nw,nb)]
addWeight (nw,nb) ((w,b):ws)
 = if b==nb then (nw+w,b):ws
   else (w,b) : addWeight (nw,nb) ws
```

Finally, `weightDiff` takes in the total weights of all the bins, finds the heaviest and lightest bin, and returns the difference between the two.

```
weightDiff xs = maximum xs - minimum xs
```

In this problem we are attempting to minimise the weight difference, so we have the following:

```
binIsFitter :: CompareFitness BinPackFitness
binIsFitter = (<)
```

Initial Population A random chromosome can be created by generating w integers between 0 and $b-1$.

```
binRandChrom :: RandomChromosome BinPackChrom
binRandChrom = map (`mod` nrBins) . take nrWeights . rands
```

In order to create an initial population, we of course need to know how large it should be.

```
binPopSize :: Int
binPopSize = 10
```

We now have all the ingredients needed to create an initial population, which is done using the function `makeSortedPop`.

```
binInitPop :: Seed -> Population BinPackFitness BinPackChrom
binInitPop
 = makeSortedPop binIsFitter binPopSize binFitness binRandChrom
```

Reproduction Next is the issue of reproduction. We shall use one mutation and one crossover function. Both shall produce a single offspring. Firstly, the function `binMutate`, an instance of the library function `changeMutate`. This picks out a gene at random and selects a new allele for it, again at random, from the list of all possible alleles passed in (in this case `[0..nrBins-1]`). We could, of course, have used the allele swap function defined earlier as an alternative.

```
binMutate :: Reproduce BinPackChrom
binMutate = changeMutate [0..nrBins-1] nrBins nrWeights
```

For crossover we are using `onePoint2p1c`, a one offspring version of the one point crossover function defined earlier (Section **??**).

```
binCrossover :: Reproduce BinPackChrom
binCrossover = onePoint2p1c nrWeights
```

We shall give a slight bias towards crossover.

```
binReproduce :: Choices (Reproduce BinPackChrom)
binReproduce = bias 0.6 binCrossover binMutate
```

The function `bias` is defined in the appendices.

Selection and Merging We shall use tournament selection, with a tournament size of 2.

```
binSelect :: Select BinPackFitness BinPackChrom
binSelect = tournamentSelect binIsFitter 2
```

Next we need to choose a merge function. For simplicity's sake, we shall choose the stable state merge scheme, defined earlier.

```
binMerge :: Merge BinPackFitness BinPackChrom
binMerge = stableMerge binIsFitter
```

A: 0 generations, fitness 105.					B: 20 generations, fitness 74.				
49								3	
1			3					13	
8		13	48		49		50	1	
21	50	22	50		20	48	19	43	25
40	5	25	27	20	17	22	5	21	8
49	48	19	17	43	48	50	49	40	27
168	103	79	145	63	134	120	123	121	60

B: 40 generations, fitness 53.					D: 60 generations, fitness 25.				
			3					3	25
49		50	13	25			50	13	8
20	48	19	1	8	49	48	19	1	21
17	22	5	43	21	20	22	5	43	27
48	50	49	40	27	48	50	49	40	17
134	120	123	100	81	117	120	123	100	98

E: 80 generations, fitness 25.					F: 100 generations, fitness 23.				
								3	
			3	25				13	25
		50	13	8				1	8
49	48	19	1	21	49	48	50	43	21
20	22	5	43	27	20	22	19	40	27
48	50	49	40	17	48	50	49	5	17
117	120	123	100	98	117	120	118	105	98

Fig. 2. Progression of Bin Packing GA

Other Parameters We do not necessarily know an optimal fitness value for this problem in advance, so there isn't an immediately obvious end condition. In this case it might be more convenient to only end when the number of generations specified is reached, thus we can use the following end condition.

```
binEndCondition :: EndCondition BinPackFitness BinPackChrom
binEndCondition population = False
```

This leaves us with the assorted numeric and boolean parameters to set. We shall have a fixed sized population of size 20 (defined earlier). We are using a stable state merge, so we only want to create a single offspring every generation. Both of our reproduce operators produce just one offspring so we can either specify a number of reproductions or a number of offspring; the choice is arbitrary.

```
binVars = GAVars
           {populationSize = binPopSize,
            populationSizeVaries = False,
            reproductionsPerGeneration = 1,
            countReproductions = True}
```

Solution Our final task is to use all of the components now defined to specify a solution to the bin packing problem as an instance of our general scheme.

```
binGA nrGens seed1 seed2
 = gga (binInitPop seed1) binVars nrGens binEndCondition binFitness
       binReproduce binSelect binMerge seed2
```

Results Figure ?? demonstrates an example run of the GA for 100 generations, showing the best chromosome found, along with its fitness, at each 20 generation interval.

4.2 Travelling Salesperson

One of the most common GA solved problems [?,?], originally proposed by a German travelling salesman in 1932 - see [?]. 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. Figure **??** gives an example route for a 10 city problem.

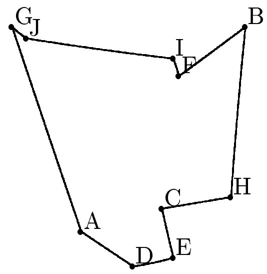

Fig. 3. A Typical Travelling Salesperson Route

Representation This is essentially a permutation problem, which lends itself naturally to a simple list representation. We give each city a label, in this case an integer in the range $1..n$ (where n is, of course, the number of cities). A route is then defined by the ordering of these labels within the list.

```
type City = Int
type TspChrom = [City]
```

These labels can be used to index an array giving the corresponding position of the city.

```
type Position = (Int,Int)
type CityPos = Array City Position
```

Fitness A function `orderToEdges` can be used to convert a city ordering into a list of edges (adjacent cities), including the implied edge from the last to the first city.

```
orderToEdges :: [a] -> [(a,a)]
```

The length of a single edge can be determined by simple Euclidean distance, using the function `edgeLength`.

```
edgeLength :: (City,City) -> Double
```

As individual edge lengths are of type `Double`, so to will the overall route length be:

```
type TspFitness = Double
```

Our fitness function can now take the following form:

```
tspFitness :: Fitness TspFitness TspChrom
tspFitness = sum . map edgeLength . orderToEdges
```

We are attempting to minimise route length, so:

```
tspIsFitter :: CompareFitness TspFitness
tspIsFitter = (<)
```

Initial Population A random chromosome can be generated with the function `mixup` (defined in the appendices) which takes in a list and a seed and returns a random permutation of that list.

```
tspRandChrom :: RandomChromosome TspChrom
tspRandChrom = mixup cities
```

We can now construct an initial population, in the usual way.

```
tspInitPop :: Seed -> Population TspFitness TspChrom
tspInitPop = makeSortedPop tspIsFitter tspPopSize
             tspFitness tspRandChrom
```

Reproduction Let us use the `orderedSwap` function defined earlier (Section **??**) for mutation. We must give this the length of the list it is to mutate, which will be equal to the number of cities.

```
tspMutate :: Reproduce TspChrom
tspMutate = alleleSwap nrCities
```

As a crossover function, we are using a slightly non standard operator - edge based crossover. This is because, in TSP, we are more interested in the edges between individual cities than the overall ordering - it is not so significant that A comes somewhere before B in the route as it is that an edge exists directly between A and B. Edge based crossover will combine the edges of the two parents to make a new route using only those edges; other order based crossover operators may well introduce new edges not existing in either parent. The first parameter to `edgeXover2p1c` is a pair containing the minimum and maximum node (city) labels.

```
tspCrossover :: Reproduce TspChrom
tspCrossover = edgeXover2p1c (1,nrCities)
```

We shall give these operators an equal chance of being used:

```
tspReproduce :: Choices (Reproduce TspChrom)
tspReproduce = fair [tspCrossover,tspMutate]
```

The function `fair` is defined in the appendices.

Selection and Merging As before we shall use tournament selection, with a tournament size of 2.

```
tspSelect :: Select TspFitness TspChrom
tspSelect = tournamentSelect tspIsFitter 2
```

Next we need to choose a merge function. Again, for simplicity's sake, we shall choose the stable state merge scheme, defined earlier.

```
tspMerge :: Merge TspFitness TspChrom
tspMerge = stableMerge tspIsFitter
```

Other Parameters As with the previous problem, we do not know an optimal fitness value in advance, so we should simply proceed until the specified number of generations expires.

```
tspEndCondition :: EndCondition TspFitness TspChrom
tspEndCondition population = False
```

The remaining numeric parameters will also be very similar to those used in the previous problem.

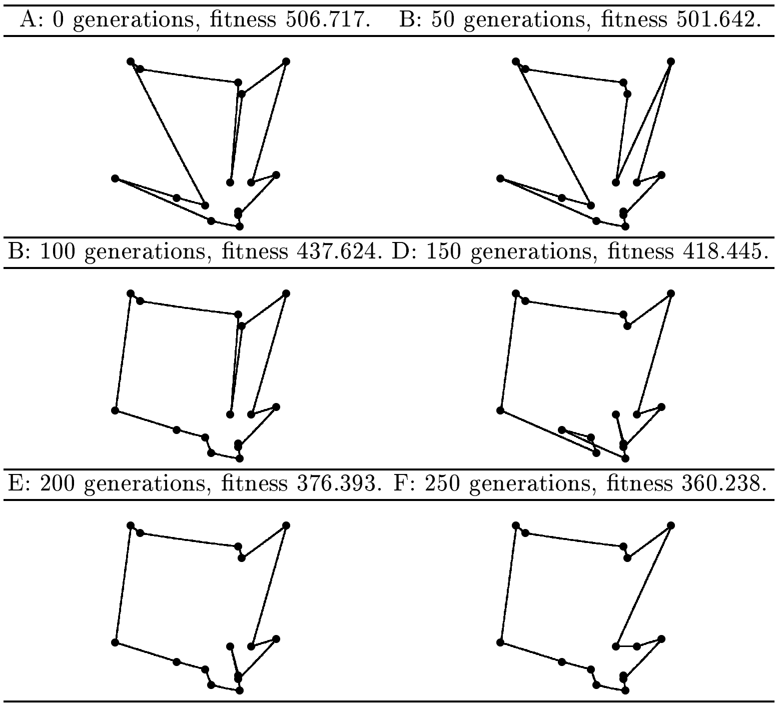

Fig. 4. Progression of Travelling Salesperson GA

Solution We are now ready to specify our solution to the travelling salesperson problem:

```
tspGA :: Count -> Seed -> Seed -> Population TspFitness TspChrom
tspGA nrGens s1 s2
 = gga (tspInitPop s1) tspVars nrGens tspEndCondition tspFitness
       tspReproduce tspSelect tspMerge s2
```

Results Figure **??** demonstrates an example run of the GA for 250 generations, with 15 cities and a population size of 30, showing the best chromosome found, along with its fitness, at each 50 generation interval.

4.3 N Queens

The N Queens [**?**] problem is a particularly common one, however, perhaps not quite so common is the use of approximate methods to solve it. For those (if any!)

unfamiliar with this problem, the task is to place n queens on an $n \times n$ chess board, such that no queen can take another. This is effectively an optimisation problem - we are attempting to minimise the number of queens that can take each other, and so an approximate method such as a genetic algorithm might not be such an unintuitive strategy after all.

Representation We shall use a representation based on that found in [**?**]. Here we assign a queen to each column and only allow the queens to move within that column, also stipulating that no row may be shared either. Thus a potential solution is simply a list of length n, containing some permutation of the numbers $1..n$. If the kth element of this list has value m we can deduce that there is a queen on the board in column k, row m.

```
type QueenPos = Int
type QueensChrom = [QueenPos]
```

Fitness For a fitness function we need to count the number of queens that can take another.

```
type QueensFitness = Int
```

As the representation has already eliminated the possibility of two queens occupying the same row or column, we need only check the diagonals. The function `check` tells us if two queens occupy the same diagonal line:

```
check :: (Int,Int) -> (Int,Int) -> Bool
check (i,j) (m,n) = (i+j==m+n) || (i-j==m-n)
```

We can then use `checks` to sum these for a list of queens.

```
checks :: Int -> QueensChrom -> Int
checks c [] = 0
checks c (r:rs)
 = length [ True | (x,y)<-zip [c+1..] rs, check (c,r) (x,y) ] +
   checks (c+1) rs
```

This gives us the following type and definition for our fitness function:

```
queensFitness :: Fitness QueensFitness QueensChrom
queensFitness chrom = checks 1 chrom
```

We are trying to minimise this fitness value. A value of 0 gives us a valid solution, so we have a known minimum. Additionally we have a known maximum - the worst possible solution places all the queens on the same diagonal line so that all can take each other. This will give us a fitness value equal to the sum from 1 to $n - 1$.

```
queensIsFitter :: CompareFitness QueensFitness
queensIsFitter = (<)
```

Initial Population A random chromosome can be generated in exactly the same way we did for the previous problem - a random permutation created by the function `mixup`.

```
queensRandChrom :: RandomChromosome QueensChrom
queensRandChrom = mixup queens
```

Where, for an n queens problem, `queens` is the list `[1..n]`. The same tried and tested method can now be used to construct our initial population.

```
queensInitPop :: Seed -> Population QueensFitness QueensChrom
queensInitPop
 = makeSortedPop queensIsFitter queensPopSize
   queensFitness queensRandChrom
```

Reproduction More re-use can be employed here, as `alleleSwap`, used previously, again makes a simple but effective mutation operator.

```
queensMutate :: Reproduce QueensChrom
queensMutate = alleleSwap nrQueens
```

For crossover a standard function can be used - Partial Match Crossover. See [**?**,**?**] for more details. This particular implementation produces just a single offspring, although the library also contains the more standard two offspring version.

```
queensCrossover :: Reproduce QueensChrom
queensCrossover = partialMatchXover2p1c nrQueens
```

As before, we shall give these two operators an equal chance of being chosen.

```
queensReproduce :: Choices (Reproduce QueensChrom)
queensReproduce = fair [queensCrossover, queensMutate]
```

Selection and Merging We shall use the standard tournament selection and stable state merging arrangement. These are given as local definitions to the `queensGA` function below.

Solution We are now ready to specify our solution. For once we shall give an end condition, as noted previously we have an optimal fitness that is known in advance - zero fitness means we have found a valid solution to the problem. Thus when the best chromosome in the population (which will always be at the head) has a fitness of zero, we can stop searching .

```
queensGA :: Count -> Seed -> Seed ->
            Population QueensFitness QueensChrom
queensGA nrGens s1 s2
 = gga (queensInitPop s1) queensVars nrGens endCondition
       queensFitness queensReproduce select merge s2
   where endCondition ((bf,bc):pop) = bf==0
         select = tournamentSelect queensIsFitter 2
         merge = stableMerge queensIsFitter
```

Results Figure **??** demonstrates an example run of the GA for 250 generations, with 20 queens and a population size of 20, showing the best chromosome found, along with its fitness, at each 50 generation interval.

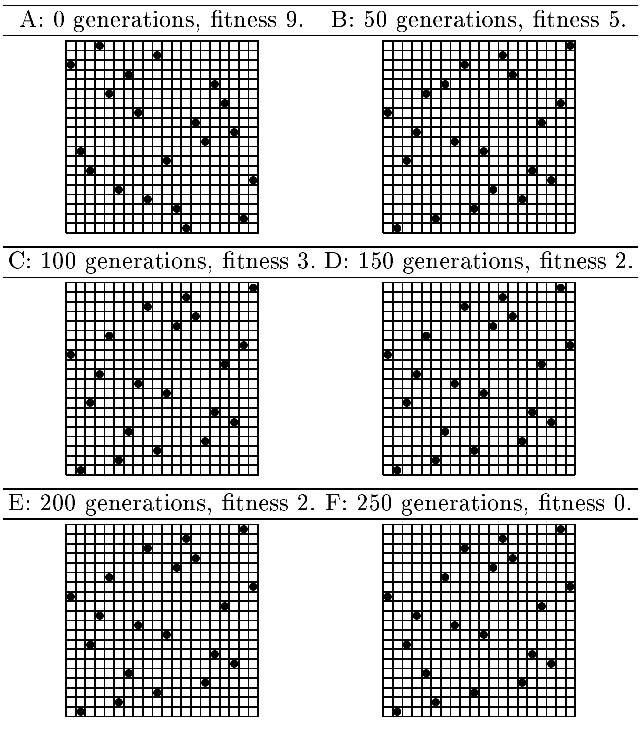

Fig. 5. Progression of N-Queens GA

4.4 Layout

So far we have looked at problems with relatively simple representations. One basic list of values, and two permutation problems. To demonstrate the flexibility of the framework, let us now instead examine a problem requiring a rather different representation. In the layout problem (see [**?**]), we are given a number of rectangular objects, each with differing dimensions. It is our task to arrange them in a non-overlapping configuration such that the resulting bounding rectangle occupies the minimum area possible.

Representation Such a configuration can be represented by a Slicing Tree Structure (see [**?**]). Essentially, this is a binary tree where leaf nodes are the shapes themselves, and branch nodes are operators specifying how their two sub-trees are positioned relative to each other - either side by side, or one above the other. The shapes are represented by characters, which are used as an index into an array which gives their respective dimensions.

```
type Shape = Char
type Dimensions = (Int,Int)
type ShapeTable = Array Shape Dimensions
```

A slicing tree structure can be represented as follows:

```
data SlicingTree a = Above (SlicingTree a) (SlicingTree a) |
                     Beside (SlicingTree a) (SlicingTree a) |
                     Item a
```

Thus our chromosome can be defined as follows:

```
type LayoutChrom = SlicingTree Shape
```

Fitness A fairly simple function can be constructed to find the dimensions of the bounding rectangle for a given chromosome:

```
sizeTree :: LayoutChrom -> Dimensions
sizeTree (Item s) = getShapeDims s
sizeTree (Above t1 t2) = addAbove (sizeTree t1) (sizeTree t2)
 where addAbove (a,b) (c,d) = (max a c, b+d)
sizeTree (Beside t1 t2) = addBeside (sizeTree t1) (sizeTree t2)
 where addBeside (a,b) (c,d) = (a+c, max b d)
```

For a fitness value, the logical approach would be to use the area of the given chromosome's bounding rectangle, its width multiplied by its height, to give a single integer value. To demonstrate the point made earlier about the flexibility of fitness values, let us instead use the dimensions themselves as a fitness value.

```
type LayoutFitness = Dimensions
```

If we are to compare two sets of dimensions, the fitter be that with a lower area. Note however, that keeping both dimensions as a fitness value would also allow us to, for example, compare the aspect ratio of two chromosomes, and give preferences to one or the other on this basis also.

```
layoutIsFitter :: CompareFitness LayoutFitness
layoutIsFitter a b = area a < area b
```

Initial Population A random chromosome can be created by first generating a random permutation of the shapes (using `mixup`), then recursively splitting this list into two randomly sized sections to form two sub trees of a node with a randomly chosen operator.

We can now construct an initial population, in the usual way.

```
layoutInitPop :: Seed -> Population LayoutFitness LayoutChrom
layoutInitPop = makeSortedPop layoutIsFitter layoutPopSize
                layoutFitness layoutRandChrom
```

Reproduction There are a number of different crossover and mutation techniques we could use for this kind of representation. Those used in this example are briefly described below:

- **Shape Swap**: Choose two shapes at random and swap their positions.
- **Operator Change**: Choose an operator at random and switch it - above becomes beside and vice verse.
- **Branch Swap**: Choose a branch at random and swap its two subtrees.
- **Shape Order**: Impose the shape ordering from one tree onto the structure of another tree.
- **Operator Order**: As above but with the operators.

Assuming we have a function to perform each of these tasks, we can bind them together in a `Choices` structure, and assign equal probability of each being used with the function `fair`.

```
layoutReproduce :: Choices (Reproduce LayoutChrom)
layoutReproduce = fair [shapeSwap,operatorChange,branchSwap,
                        shapeOrder,operatorOrder]
```

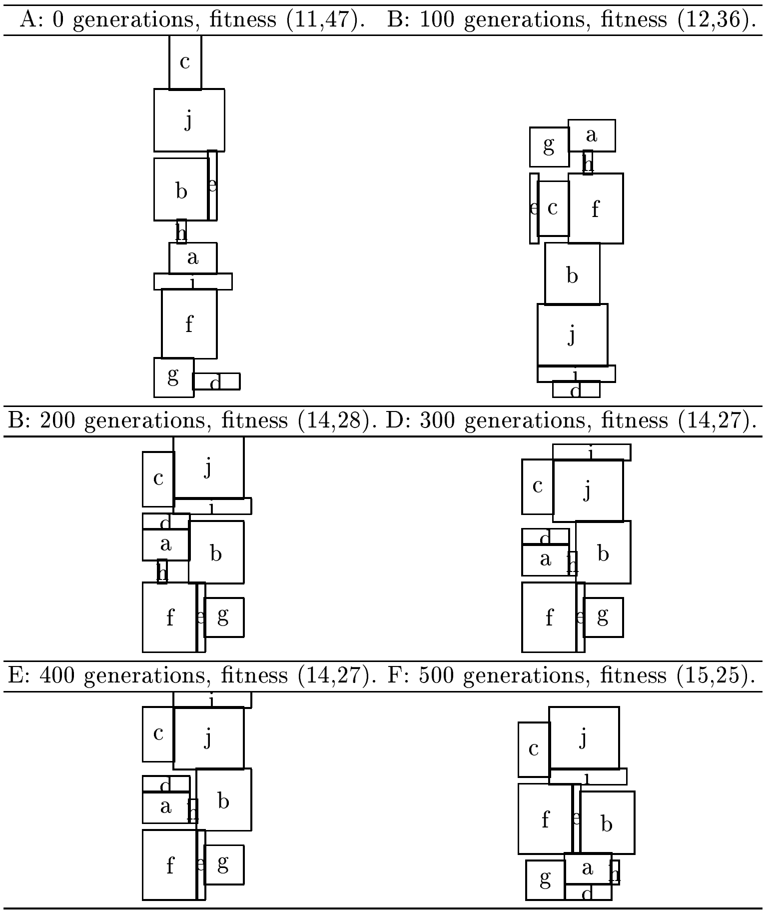

Fig. 6. Progression of Layout GA

Selection and Merging Despite this problem's irregular representation requiring us to write custom reproduce functions, we can as always use standard selection and merging operators. Here for simplicity we shall stick to tournament selection and a stable state merging scheme.

Solution We can now construct our GA solution to the problem:

```
layoutGA :: Count -> Seed -> Seed ->
            Population LayoutFitness LayoutChrom
layoutGA nrGens s1 s2
 = gga (layoutInitPop s1) layoutVars nrGens endCondition
       layoutFitness layoutReproduce select merge s2
       where endCondition = const False
             select = tournamentSelect layoutIsFitter 2
             merge = stableMerge layoutIsFitter
```

Results Figure **??** demonstrates an example run of the GA for 500 generations, with 10 shapes and a population size of 30, showing the best chromosome found, along with its fitness, at each 100 generation interval.

5 Related Work

The approach adopted in this work has been used in part (or parts) in several other areas. Mainly these appear to be in the field of genetic programming (genetic algorithms used to develop programs), one particular subset of the overall genetic algorithm field. Finnegan, as used in [?], is a system implemented in SML/NJ and makes use of its support for parameterized modules to allow quick interchange of GA (more specifically GP) components, a similar method to that used here. SML/NJ is also used for the ADATE system [?], another evolution based genetic programming platform. PolyGP [?] is a genetic programming system also written in Haskell. FGP [?] is a genetic programming technique based on FP. Moving to the broader area of optmisation, in [?] a generic functional algorithm is given for sequential decision processes, a particular class of optimisation algorithms, which is then used to solve some problems including two (coincidentally) similar to two of those looked at here.

6 Conclusion

We have given a functional overview of the field of evolutionary computation. The main emphasis has been on a single higher order function which encapsulates the functionality of a wide class of genetic algorithms. We have also provided a library of useful functions for instantiating the parameters of the GA skeleton making the construction of new GA algorithms as easy as possible. We have demonstrated the usefulness of this approach by developing practical GA solutions to four computationally hard problems. All these solutions are different instances of the GA skeleton.

There are several advantages for adopting this approach. Firstly, ease and speed of construction are effortlessly achieved by combining reusable components. Secondly, possibilities of experimenting with variations of a given GA are easily achieved by using different instances of the GA parameters from the GA library of components.

Future development of this work will focus on three main aspects: parallelisation of the GA skeletons, integrating visualization features with the functional GA tool, and the evaluation of strategies for developing genetic algorithms based on a theory of probabilistic reasoning.

Acknowledgements

The authors would like to thank Dave Corne for his comments which have significantly improved the paper.

References

1. R. Backhouse, P. Jansson, J. Jeuring and L. Meertens, Generic Programming - An Introduction, In Advanced Functional Programming, LNCS 1608, Springer 1998.
2. R. S. Bird, and P. Wadler, Introduction to Functional Programming, Prentice-Hall, 1988.
3. R. S. Bird, Introduction to Functional Programming Using Haskell, Prentice-Hall, 1998.
4. D. Corne, Applied Evolutionary Computation, Springer-Verlag, 1999.
5. O. Dahl, E. W. Dijkstra, and C. A. R. Hoare, Structured Programming, Academic Press, London, 1972.
6. D. E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, MA, 1989.

7. L. Huelsbergen, Toward Simulated Evolution of Machine-Language Iteration, In Proceedings of the Conference on Genetic Programming, pp 315-320, 1996.
8. E. L. Lawer, J. K. Lenstra, A. H. G Rinnooy Kan and D. B. Shmoys, The Traveling Salesman Problem, John Wiley, Chichester, UK, 1985.
9. S. Martello and P. Toth, Knapsack Problems, Wiley Interscience, 1990.
10. Z. Michaelewicz, Genetic Algorithms + Data Structures = Evolution Programs, Third Edition, Springer, 1996.
11. O. de Moor, A Generic Program for Sequential Decision Processes. PLILP 95 pages 1-23, 1995.
12. O. de Moor and G. Sittampalam, Generic Program Transformation, In Advanced Functional Programming, LNCS 1608, Springer 1998.
13. J. R. Olsson, Population management for automatic design of algorithms through evolution, International Conference on Evolutionary Computation, IEEE Press, 1998.
14. R. H. J. M. Otten. Automatic floorplan design, In Proceedings of the 19th ACM-IEEE Design Automation Conference, pages 261-267, 1982.
15. F. Rabhi and G. Lapalme, *Algorithms: A functional programming approach*, Addison-Wesley, 1999.
16. 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.
17. T. Yu and C. Clack, PolyGP: A Polymorphic Genetic Programming System in Haskell, In Proceedings of the Third Annual Genetic Programming Conference, 1998.

A Auxilary Functions

A.1 Fitness and Populations

```
fitnessOf :: Evaluated f c -> f
fitnessOf (f,c) = f


chromosomeOf :: Evaluated f c -> c
chromosomeOf (f,c) = c


chooseBetter :: CompareFitness f -> Evaluated f c ->
                Evaluated f c -> Evaluated f c
chooseBetter isFitterThan c1 c2
 = if fitnessOf c1 `isFitterThan` fitnessOf c2 then c1 else c2


chooseBest :: CompareFitness f -> [Evaluated f c] -> Evaluated f c
chooseBest isFitter = foldr1 (chooseBetter isFitter)


popInsert :: CompareFitness f -> Evaluated f c ->
             Population f c -> Population f c
popInsert isFitterThan x [] = [x]
popInsert isFitterThan x (y:ys)
 = if fitnessOf x `isFitterThan` fitnessOf y
   then x:y:ys
   else y : popInsert isFitterThan x ys


popSort :: PopulationSort f c
popSort isFitter = foldr (popInsert isFitter) []
```

A.2 List Processing

```
mkSingleton :: a -> [a]
mkSingleton x = [x]


setAt :: Integral b => b -> a -> [a] -> [a]
setAt 0 i (x:xs) = i:xs
setAt (n+1) i (x:xs) = x : setAt n i xs


insertAt :: Int -> a -> [a] -> [a]
insertAt n a [] = [a]
insertAt 0 a (x:xs) = a:x:xs
insertAt (n+1) a (x:xs) = x:(insertAt n a xs)
```

A.3 Random

```
mixup :: [a] -> Seed -> [a]
mixup xs sd = snd (foldr f ((rands sd,1),[]) xs)
 where f y ((r:rs,sz),ys)
        = ((rs,sz+1),insertAt (r `mod` sz) y ys)
```

A.4 Choices

```
always :: a -> Choices a
always a = [(1.0,a)]


bias :: Probability -> a -> a -> Choices a
bias p a b = [(p,a),(1.0-p,b)]


fair :: [a] -> Choices a
fair as = map f as
 where p = 1.0 / (fromInt n)
       n = length as
       f a = (p,a)
```

B The Genetic Algorithm

```
gga population gaVars nrGens endCondition fitness
    reproducers select merge seed
 | nrGens==0 = population
 | endCondition population = population
 | otherwise = nextGeneration seed
 where nextGeneration = (\(r1:r2:rs) -> gga (newPopulation r1)
                         gaVars (nrGens-1) endCondition fitness
                         reproducers select merge r2) . rands
       newPopulation  = (\(r1:r2:rs) -> merge
                         population (offspring r1) r2) . rands
       countScheme    = if countReproductions gaVars
                        then (concat . take nrReps)
                        else (take nrOff . concat)
       offspring      = map (evaluate fitness). countScheme .
```

```
                  map litter . rands
litter          = (\(r1:r2:r3:rs) -> (choose reproducers r1)
                   (parents r2) r3) . rands
parents         = map snd . concat .
                  map (select population popSize) . rands
popSize         = if populationSizeVaries gaVars
                  then length population
                  else populationSize gaVars
nrReps          = reproductionsPerGeneration gaVars
nrOff           = offspringPerGeneration gaVars
```