Week 13: The continuous GA

Robin T. Bye

25th April 2017

Time	Topic	Reading
09.15	Overview and status update	
09.30	Lecture: The continuous GA	
10.30	Workshop/tutorial: The continuous GA	*

* Please read and follow the instructions given in the workshop documents *before* coming to class!

Date for workshop: **Tuesday 18 April**

This PDF document is available in an HTML version at
`http://www.hg.schaathun.net/FPIA/week13.html`

Contents

1 Tutorial: Continuous GA

In this tutorial, we will implement a continuous GA based on the binary GA that we implemented in the previous works. A lot of the binary GA code can be reused as is or just slightly modified, and also some code related to the binary encoding can simply be removed. Therefore, this tutorial will necessarily contain much of the same information as the tutorial on the

binary GA.

To keep things simple, we will convert the binary GA that was able to learn strings into a continuous GA for doing the same thing. After having completed the tutorial, you should try to modify it so that it can be used to solve and optimise test functions such as those given in the appendix of Haupt & Haupt (2004), which is available on Fronter.

1.1 A continuous GA for string learning

The purpose of this GA is to discover, or learn, a target string. To learn the target string, the GA will contain a population of candidate solutions (chromosomes) that represent strings. Each chromosome is a guess and can be evaluated and assigned a cost, where the cost should relate to the difference between the string it represents and the target string.

The GA can then evolve the population over several generations until one or more of the individuals in the population become the target string. Such individuals will have a cost of zero.

Note that this example is adapted from (and is slightly easier than) the song learning example in Chapter 4 of Haupt & Haupt (2004), where a GA is used to learn the song "Mary had a little lamb."

We begin by creating a module `ContinuousGA.hs` for our code:

```
module ContinuousGA where
```

1.2 GA settings

Before we proceed, it is useful to add some GA parameter settings to the top of our file so that they can easily be found and tweaked when we will test the GA later. We adopt the terminology used in class and in Haupt & Haupt and define the following parameters with some default settings:

```
numPop = 40    :: Int    -- Population size (number of chromosomes)
xRate = 0.5    :: Double -- Selection rate
mutRate = 0.1  :: Double -- Mutation rate
numElite = 1   :: Int    -- Number of elite chromosomes
itMax = 1000   :: Int    -- Max number of iterations
```

1.3 Alphabet

The `alphabet` is the set of possible symbols, or characters, that can constitute a string. We store the `alphabet` as a string with 48 characters

```
alphabet' = "abcdefghijklmnopqrstuvwxyz0123456789.,;:?!_+-*/ " :: String
```

Please note that the final character `" "` in `alphabet'` is the space character and must not be removed. Because we are using a continuous (real-valued) GA, we do not have to worry about binary encoding and padding the alphabet to make it have a length that is a power of 2.

1.4 Target strings

For testing and debugging purposes, we define a set of test strings:

```
s1, s2, s3, s4, s5, s6 :: String
s1 = "h"
s2 = "abc"
s3 = "hello world!"
s4 = "abc123.,;"
s5 = "descartes: cogito ergo sum"
s6 = "2+2 is: 4, 2*2 is: 4; why is _/not/_ 2.2*2.2 equal to 4.4?!"
target = s2
```

1.5 Genes, chromosomes, and the population

In the following, we will assume that a gene is a continuous real-valued variable normalised to the range $[0, 1]$ and encodes a single character, and that a chromosome is a list of genes that encodes a string.

The number of characters in the target string is sometimes called the number of (encoding) variables, `numVar`:

```
numVar = length target :: Int -- Number of characters (genes) in target
```

This number tells us how many genes are required in each chromosome.

Next, we need to know the number of chromosomes `numKeep` to keep between generations. This number is a fraction (the selection rate `xRate`) of the population size `numPop` and should be rounded up to the nearest even number to simplify the mating procedure:

```
numKeep | numKeep'' > numPop = numPop - numPop `mod` 2
        | numKeep'' < 2 = 2
        | otherwise = numKeep''
        where numKeep'' = numKeep' + numKeep' `mod` 2
              numKeep' = ceiling $ xRate * fromIntegral numPop :: Int
```

For example, for `numPop = 40` and `xRate = 0.5`, we get `numKeep = 20`. Some cases are introduced for safety to ensure `numKeep` is always greater than or equal to 2, and always smaller than or equal to `numPop`.

We also need to determine the number of chromosomes to mutate, `numMut`, which is a fraction (the mutation rate `mutRate`) of the population size, and must be rounded up to nearest integer:

```haskell
numMut = ceiling $ mutRate * fromIntegral numPop :: Int
```

For `numPop = 40` and `mutRate = 0.1`, we get `numMut = 4`.

Finally, we define some type aliases for readability and debugging purposes:

```haskell
type Gene = Double
type Chromosome = [Gene]
type Population = [Chromosome]
```

First of all, note that we possibly could have used an integer type such as `Int` for `Gene`, however, we want our GA to be general purpose and able to handle other problems where the variables will not be characters that can be represented by integers but real-valued numbers, e.g., to optimise the test functions in the appendix in Haupt & Haupt (2004).

Exercise: Experiment with different alphabets and different values for `numPop`, `xRate`, and `mutRate` and verify that `numGene`, `numKeep`, and `numMut` are calculated correctly by testing in the `ghci`.

1.6 Normalisation and denormalisation

The plan is to normalise the genes values to $g_i \in [0, 1]$, however, the characters in the alphabet can be represented as integers that corresponds to their position (index) in the alphabet. Hence, we need a `normalise` function to convert variable p_i, which is an integer in the range $[0, \text{length alphabet} - 1]$, to a real-valued gene g_i in the range $[0, 1]$, and likewise, a `denormalise` function to convert back from a gene to a variable value.

We begin by defining the constraints on variable values and gene values:

```haskell
-- Variable constraints
pLo = 0.0 :: Double
pHi = fromIntegral $ length alphabet - 1 :: Double

-- Gene constraints
gLo = 0.0 :: Double
gHi = 1.0 :: Double
```

To normalise a variable p in the range $[p_{\text{low}}, p_{\text{high}}]$ to a range $[g_{\text{low}} = 0, g_{\text{high}} = 1]$, we can use the following formula:

$$p_{\text{norm}} = \left(p_{\text{low}} + \frac{p - p_{\text{low}}}{p_{\text{high}} - p_{\text{low}}} \right) (g_{\text{high}} - g_{\text{low}}) + g_{\text{low}} \tag{1}$$

which simplifies to

$$p_{\text{norm}} = p_{\text{low}} + \frac{p - p_{\text{low}}}{p_{\text{high}} - p_{\text{low}}} \tag{2}$$

for $g_{\text{low}} = 0$ and $g_{\text{high}} = 1$.

The `normalise` function becomes

```haskell
-- Normalise a variable to range [gLo, gHi] with safety checks
normalise :: Double -> Double
normalise p = p'
    where p' | normalise' p < gLo = gLo
             | normalise' p > gHi = gHi
             | otherwise = normalise' p

-- Normalise a variable to range [gLo, gHi], unsafe
normalise' :: Double -> Double
normalise' p = pnorm * (gHi - gLo) + gLo
    where pnorm = pLo + (p - pLo) / (pHi - pLo)
```

where we add some cases for safety to ensure that the normalised variable p_{norm} stays within the limits of $[g_{\text{low}} = 0, g_{\text{high}} = 1]$.

The `denormalise` function becomes

```haskell
denormalise :: Double -> Double
denormalise pNorm = pNorm * (pHi - pLo) + pLo
```

1.7 Encoding functions

We need a function `encodeString` to encode a string as a `Chromosome`. This means that if we have another function `encodeChar` that can encode a single character as a `Gene`, we can just map that function over a string, which is a list of characters, in order to obtain the encoded `Chromosome`. In addition, it will likely prove useful to have a function `encodeStringList` that can encode a list of strings as a `Population`.

1.7.1 The `encodeChar` function

Let us begin with defining the `encodechar` function:

```haskell
-- Encode a single character as a gene
encodeChar :: Char -> Gene
encodeChar c = normalise $ fromIntegral $ stripMaybe index
    where index = elemIndex c alphabet
          stripMaybe (Just index) = index
          stripMaybe (Nothing)    = error $ "elemIndex returned Nothing. " ++
                                            "A character is not in the alphabet!"
```

Given a character `c`, the function should return a normalised gene that corresponds to the character's index in `alphabet`.

A convenient function to determine the index of an element in a list is `elemIndex`, which is part of the `Data.List` library:

```haskell
import Data.List (elemIndex)
```

Its type signature is given by
`elemIndex :: Eq a => a -> [a] -> Maybe Int`
Therefore, we will need to "strip" the `Just` from the return value. which is done by the helper function `stripMaybe`.

We can test the `encodeChar` function in the interpreter:

```
*ContinuousGA> encodeChar 'a'
0.0
*ContinuousGA> encodeChar 'b'
2.127659574468085e-2
*ContinuousGA> encodeChar '/'
0.9787234042553191
```

1.7.2 The `encodeString` function

The `encodeString` function is simple enough:

```
encodeString :: String -> Chromosome
encodeString = map encodeChar
```

We just map `encodechar` over a list of characters (a string) to obtain a list of encoded genes, or a `Chromosome`.

We can test the function in `ghci`:

```
*ContinuousGA> encodeString "abc"
[0.0,2.127659574468085e-2,4.25531914893617e-2]
*ContinuousGA> encodeString "*/ "
[0.9574468085106383,0.9787234042553191,1.0]
```

1.7.3 The `encodeStringList` function

The `encodeStringList` function is just as simple:

```
encodeStringList :: [String] -> Population
encodeStringList = map encodeString
```

The output in `ghci` for a list of two test strings yields

```
*ContinuousGA> encodeStringList ["ab","*/"]
[[0.0,2.127659574468085e-2],[0.9574468085106383,0.9787234042553191]]
```

Exercise: What happens if one of the encoding functions above encounter a character not in `alphabet`?

1.8 Decoding functions

In addition to encoding functions, we also need decoding functions able to convert back from normalised genes to characters in `alphabet`.

1.8.1 The `decodeGene` function

we define a `decodeGene` function that converts a `Gene` to an index (a decimal number) and then accesses the character at the index position in `alphabet` using the `!!` function:

```
decodeGene :: Gene -> Char
decodeGene g = alphabet!!idx
    where idx = round $ denormalise g
```

Example usage in `ghci`:

```
*ContinuousGA> decodeGene 0.0
'a'
*ContinuousGA> decodeGene 0.01
'a'
*ContinuousGA> decodeGene 0.02
'b'
*ContinuousGA> decodeGene 0.98
'/'
*ContinuousGA> decodeGene 0.99
' '
*ContinuousGA> decodeGene 1.0
' '
```

Exercise: What happens if a gene is outside the range $[0, 1]$?

1.8.2 The `decodeChromosome` function

To decode a `Chromosome`, we just map the `decodeGene` function over it to obtain the corresponding string of characters:

```
decodeChromosome :: Chromosome -> String
decodeChromosome = map decodeGene
```

Here is an example in `ghci`:

```
*ContinuousGA> let band = encodeString "ac/dc"
*ContinuousGA> band
[0.0,4.25531914893617e-2,0.9787234042553191,6.382978723404255e-2,4.25531914893617e-2]
*ContinuousGA> decodeChromosome band
"ac/dc"
```

1.8.3 The `decodePopulation` function

It is also convenient to have a function able to decode an entire `Population` of chrosomoses to a list of decoded strings. For this, we just map the `decodeChromosome` function over the population list of chromosomes:

```
decodePopulation :: Population -> [String]
decodePopulation = map decodeChromosome
```

Example functionality in `ghci`:

```
*ContinuousGA> let pop = encodeStringList ["ac/dc","heavy","rock!"]
*ContinuousGA> pop
[[0.0,4.25531914893617e-2, ...],
 [0.14893617021276595, ...],
 [0.3617021276595745, ...]]
*ContinuousGA> decodePopulation pop
["ac/dc","heavy","rock!"]
```

1.9 Randomness functions

Randomness is vital for a GA, e.g., to create a random initial population, a random crossover point, a random mutation, and so on. We will use two functions from the `System.Random` library to implement the functions we need for randomness, namely `StdGen` and `randomR`:

```
import System.Random (StdGen, randomR)
```

It will be convenient to have functions for generating random genes, chromosomes, and populations; and also for generating a random index (e.g., a crossover point) in a list.

Note that the reason we also return a `StdGen` in many or most of the functions dealing with randomness is so that we can repeatedly apply them, e.g., in a genetic evolution. In such cases, we need to pass on a `StdGen` for the next function calls.

1.9.1 The `randGene` function

The `randGene` function generates a random number of type `Gene` in the range $[g_{\text{low}}, g_{\text{high}}]$:

```
randGene :: StdGen -> (Gene, StdGen)
randGene g = (value, g')
    where (value, g') = randomR (gLo, gHi) g
```

1.9.2 The `randGenes` function

The `randGenes` function uses the `randGene` function to generate a list of random genes with length `n`:

```haskell
-- Create a list with n random genes
randGenes :: Int -> StdGen -> ([Gene], StdGen)
randGenes 0 g = ([], g)
randGenes n g =
    let (value, g') = randGene g
        (restOfList, g'') = randGenes (n-1) g'
    in  (value:restOfList, g'')
```

For convenience, we also implement a version of this function, `randGene'`, that does not return `StdGen`:

```haskell
-- Create a list with n random genes, do not return StdGen
randGenes' :: Int -> StdGen -> [Gene]
randGenes' n g = take n $ randomRs (gLo, gHi) g
```

1.9.3 The `randChrom` function

The `randChrom` and `randChrom'` functions are identical to `randGenes` and `randGenes'`, respectively, except that they return a `Chromosome` instead of a list of genes `[Gene]`:

```haskell
-- Random chromosome with n genes
randChrom :: Int -> StdGen -> (Chromosome, StdGen)
randChrom n g = randGenes n g

-- Random chromosome with n genes, do not return StdGen
randChrom' :: Int -> StdGen -> Chromosome
randChrom' n g = randGenes' n g
```

1.9.4 The `randChroms` function

The `randChroms` function uses the `randChrom` function to generate a list of `cn` random chromosomes, each with length `gn`:

```haskell
-- Create a list with cn random chromosomes with gn genes
randChroms :: Int -> Int -> StdGen -> (Population, StdGen)
randChroms 0 _ g = ([], g)
randChroms cn gn g =
    let (value, g') = randChrom gn g
        (restOfList, g'') = randChroms (cn-1) gn g'
    in  (value:restOfList, g'')
```

Again, we can also define a `randChrom'` function that does the same thing as `randChrom` but does not return at `StdGen`:

```haskell
randChroms' :: Int -> Int -> StdGen -> Population
randChroms' cn gn g = chunksOf gn values
    where values = randGenes' (cn * gn) g
```

This function requires the use of the function `chunksOf` in the `Data.List.Split` library:

```haskell
import Data.List.Split (chunksOf)
```

That is, we first generate a large list of `cn` × `gn` genes, and then we split that list into a list of sublists, each of length `gn`.

1.9.5 The `randPop` function

The `randPop` function is just a special case of the `randChroms` function that generates a list of `cn = numPop` random chromosomes, each with `gn = numVar` genes:

```haskell
-- Random population with numPop chromosomes with numVar genes
randPop :: StdGen -> (Population, StdGen)
randPop = randChroms numPop numVar
```

And again, we can also define a `randPop'` function that does not return a `StdGen` if we want to:

```haskell
-- Random population with numPop chromosomes with numVar genes,
-- do not return StdGen
randPop' :: StdGen -> Population
randPop' = randChroms' numPop numVar
```

1.9.6 The `randIndex` function

Finally, the `randIndex` function returns a random index of a list `xs` of length `n` in the closed interval $[0, n-1]$:

```haskell
-- Random index in a chromosome
randIndex :: StdGen -> [a] -> (Int, StdGen)
randIndex g xs = (ind, g')
    where (ind, g') = randomR (0, length xs - 1) g
```

1.10 Cost functions

Before we proceed with implementing the core of the GA, namely operations such as selection, mating, mutation, and evolution, we need functions to evaluate the cost of chromosomes.

1.10.1 The `elemCost` function

We begin with the `elemCost` function, which has the type signature
`elemCost :: (Eq a) => a -> a -> Int`
The function compares two elements or items of comparable generic type `a` belonging to the typeclass `Eq` and returns a cost of zero if they are equal or a cost of one if they are unequal:

```haskell
-- Compare two inputs and return zero if equal, 1 otherwise
elemCost a b | a == b = 0
             | otherwise = 1
```

We can then use `elemCost` together with `zipWith` to find the accumulated cost of two lists of elements, e.g., a gene or a string.

Some examples of usage in `ghci` include:

```
*ContinuousGA> elemCost 5 13
1
*ContinuousGA> elemCost 1 1
0
*ContinuousGA> elemCost 'a' 'b'
1
*ContinuousGA> elemCost 'a' 'c'
1
*ContinuousGA> elemCost 'a' 'a'
0
```

1.10.2 The `stringCost` function

The `stringCost` function is implemented by zipping together two lists of characters (two strings) with the `elemCost` function and summing the result:

```haskell
-- Sum the number of unequal chars, char by char, of two strings
stringCost :: String -> String -> Int
stringCost s1 s2 = sum $ zipWith elemCost s1 s2
```

Example usage in `ghci`:

```
*ContinuousGA> let s1 = "ac/dc"
*ContinuousGA> let s2 = "ac*dc"
*ContinuousGA> stringCost s1 s2
1
```

1.10.3 The `chromCost` function

The `chromCost` function uses the `stringCost` function to compare a target string with the string decoded from a chromosome:

```haskell
-- Sum the number of incorrect characters in a chromosome
chromCost :: String -> Chromosome -> Int
chromCost target c = stringCost target (decodeChromosome c)
```

Example usage in `ghci`:

```
*ContinuousGA> let c = encodeString s2
*ContinuousGA> chromCost s1 c
1
```

1.10.4 The `chromCostPair` function

As we shall see later, it is convenient to store the evaluated cost of a chromosome together with the chromosome itself in a tuple. For example, if we have a population of such tuples, we can sort it in increasing order of cost (ranking), which is necessary for selection, where we typically want to select better chromosomes before worse ones.

We therefore define a new type

```
type ChromCost = (Int, Chromosome)
```

Next, we define a `chromCostPair` function to construct such tuples:

```
chromCostPair :: String -> Chromosome -> ChromCost
chromCostPair target c = (chromCost target c, c)
```

Note that we use the target string representation as an input to the function instead of its chromosome representation that we used when we defined the function for the binary GA.

Using the variables defined above, we can test the function in `ghci`:

```
*ContinuousGA> chromCostPair s1 c
(1,[0.0,4.25531914893617e-2,0.9574468085106383,6.382978723404255e-2,4.25531914893617e-2
```

The first part of the tuple is the cost of chromosome `c`, that is, the number of characters in the decoded string that `c` represents that differ from the target string `s1`. The second part of the tuple is the chromosome `c` itself.

1.11 Population functions

We now turn our attention to some functions dealing with an entire population of chromosomes. The functions below are used to evaluate the cost of each chromosome in a population (`evalPop`); to sort a population in increasing order of cost (`sortPop`); to select chromosomes apart from elite chromosomes to keep for next generation and to use for mating (`selection`); get a list of chromosomes to be used as parents for mating (`getParents`); and to convert a list of (cost, chromosome) pairs, `[ChromCost]`, to type `Population`.

1.11.1 The `evalPop` function

The `evalPop` function evaluates all of the chromosomes in a population by comparing the decoded chromosomes (strings) with a target string and returns list of `ChromCost`. This is accomplished by mapping the partial function `chromCostPair target` over the population. This is an example of partial function application, where we call a function with too few parameters and get back a partially applied function, that is, a function that takes as many parameters as we left out.

```haskell
evalPop :: String -> Population -> [ChromCost]
evalPop target = map (chromCostPair target)
```

Example usage in `ghci`:

```
*ContinuousGA> let target = "ac/dc"
*ContinuousGA> let s1 = "ac*dc"
*ContinuousGA> let s2 = "hello"
*ContinuousGA> let s3 = "a1234"
*ContinuousGA> let s4 = "ac/dc"
*ContinuousGA> let pop = encodeStringList [s1,s2,s3,s4]
*ContinuousGA> let popEvaluated = evalPop target pop
*ContinuousGA> popEvaluated
[(1,[0.0,4.25531914893617e-2, ...]),
 (5,[0.14893617021276595,8.51063829787234e-2, ...]),
 (4,[0.0,0.574468085106383,0.5957446808510638, ...]),
 (0,[0.0,4.25531914893617e-2,0.9787234042553191, ...])]
```

1.11.2 The `sortPop` function

To sort a population, we can use the `sort` function from the `Data.List` library. We therefore add `sort` to our import statement:

```haskell
import Data.List (elemIndex, sort)
```

The `sortPop` function is then implemented simply as

```haskell
sortPop :: [ChromCost] -> [ChromCost]
sortPop pop = sort pop
```

Note that this works because when given a list of tuples (`ChromCost` is a (cost, chromosome) pair, or tuple), `sort` sorts the list by the first element of the tuples, namely the cost.

Of course, we could have used `sort` directly, however, implementing `sortPop` ensures correct types so it adds some safety to our code.

Example usage in `ghci` using the evaluated population above:

```
*ContinuousGA> sortPop popEvaluated
[(0,[0.0,4.25531914893617e-2, ... ]),
 (1,[0.0,4.25531914893617e-2,0.9574468085106383, ...]),
 (4,[0.0,0.574468085106383,0.5957446808510638, ...]),
 (5,[0.14893617021276595,8.51063829787234e-2, ...])]
```

1.11.3 The `selection` function

In addition to `numElite` elite chromosomes, a number of other chromosomes must be kept in the population for the next generation and also serve the role as parents for mating. There

are several ways to select such chromosomes, including roulette wheel selection, tournament selection, random selection, etc. Here, we just take a fraction `xRate` (called the selection rate) of a sorted (ranked) population. The total number of elite chromosomes plus selected chromosomes should equal `numKeep`. A possible implementation of a `selection` function is given below:

```
selection :: [ChromCost] -> [ChromCost]
selection pop = take (numKeep - numElite) $ drop numElite pop
```

1.11.4 The `getParents` function

The `getParents` function returns the chromosomes to keep in the population for the next generation and to serve as parents. These chromosomes consists of the `numElite` best chromosomes in the population plus some chromosomes that are selected using the `selection` function above.

```
getParents :: [ChromCost] -> [ChromCost]
getParents pop = (take numElite pop) ++ selection pop
```

1.11.5 The `toPopulation` function

For some of the genetic operations dealing with the population, it may be convenient not to have to handle a list of (cost, chromosome) pairs but just a list of chromosomes. We therefore define a function `toPopulation` to perform a conversion of a list of `ChromCost` to `Population`:

```
toPopulation :: [ChromCost] -> Population
toPopulation [] = []
toPopulation ((cost,chrom) : pop) = chrom : toPopulation pop
```

1.11.6 Testing the population functions

We can test the population functions in `ghci`. First, we set the GA parameters in the module that will affect the functions:

```
numPop = 8     :: Int    -- Population size (number of chromosomes)
xRate = 0.5    :: Double -- Selection rate
numElite = 2   :: Int    -- Number of elite chromosomes
```

That is, we will play with a population of `numPop == 8` chromosomes, keep `numKeep == 4` chromosomes, of which `numElite == 2` shall be elite chromosomes.

In `ghci`, we do the following:

```
*ContinuousGA> let stringList = ["cde","klm","123","*/
","bbb","abb","aab","aah"]
*ContinuousGA> let targetString = "aah"
*ContinuousGA> let pop = encodeStringList stringList
*ContinuousGA> let sortedPop = sortPop $ evalPop targetString pop
*ContinuousGA> sortedPop
[(0,[0.0,0.0,0.14893617021276595]),
 (1,[0.0,0.0,2.127659574468085e-2]),
 (2,[0.0,2.127659574468085e-2,2.127659574468085e-2]),
 (3,[2.127659574468085e-2,2.127659574468085e-2,2.127659574468085e-2]),
 (3,[4.25531914893617e-2,6.382978723404255e-2,8.51063829787234e-2]),
 (3,[0.2127659574468085,0.23404255319148937,0.2553191489361702]),
 (3,[0.574468085106383,0.5957446808510638,0.6170212765957447]),
 (3,[0.9574468085106383,0.9787234042553191,1.0])]
*ContinuousGA> let selectedChroms = selection sortedPop
*ContinuousGA> selectedChroms
[(2,[0.0,2.127659574468085e-2,2.127659574468085e-2]),
 (3,[2.127659574468085e-2,2.127659574468085e-2,2.127659574468085e-2])]
```

That is, the `selectedChroms` are the third and fourth chromosome in the sorted population, because the `selection` function ignores the first `numElite == 2` chromosomes and takes the next two chromosomes so that the total is equal to `numKeep == 4`.

Next, we can use the `getParents` function to complete the selection process, leaving us with the top four chromosomes:

```
*ContinuousGA> let parents = getParents sortedPop
*ContinuousGA> parents
[(0,[0.0,0.0,0.14893617021276595]),
 (1,[0.0,0.0,2.127659574468085e-2]),
 (2,[0.0,2.127659574468085e-2,2.127659574468085e-2]),
 (3,[2.127659574468085e-2,2.127659574468085e-2,2.127659574468085e-2])]
```

If we are curious which strings these chromosomes correspond to, we can use the function `toPopulation` to convert from a [`ChromCost`] list to a `Population`, and then decode the population:

```
*ContinuousGA> let parentsAsPop = toPopulation parents
*ContinuousGA> decodePopulation parentsAsPop
["aah","aab","abb","bbb"]
```

1.12 Mating functions

We are finally ready to begin implementing the core components of the GA, namely mating, mutation, and evolution. We begin with two function required for mating, the single point `crossover` function and the `matePairwise` function.

1.12.1 The `crossover` function

The function `crossover` is used for mating with single point crossover:

```
crossover :: Int -> Chromosome -> Chromosome -> Population
crossover cp ma pa = [take cp ma ++ drop cp pa, take cp pa ++ drop cp ma]
```

Given a crossover point `cp`, a mother chromosome `ma`, and a father chromosome `pa`, it returns two offspring, where one consists of the genes in `ma` and `pa` before and after `cp`, respectively, and the other consists of the remaining genes from `ma` and `pa`.

Example usage in `ghci`:

```
*ContinuousGA> let s1 = "abcd"
*ContinuousGA> let s2 = "efgh"
*ContinuousGA> let c1 = encodeString s1
*ContinuousGA> let c2 = encodeString s2
*ContinuousGA> c1
[0.0,2.127659574468085e-2,4.25531914893617e-2,6.382978723404255e-2]
*ContinuousGA> c2
[8.51063829787234e-2,0.10638297872340426,0.1276595744680851,0.14893617021276595]
*ContinuousGA> crossover 0 c1 c2
[[8.51063829787234e-2,0.10638297872340426,0.1276595744680851,0.14893617021276595],
 [0.0,2.127659574468085e-2,4.25531914893617e-2,6.382978723404255e-2]]
*ContinuousGA> crossover 1 c1 c2
[[0.0,0.10638297872340426,0.1276595744680851,0.14893617021276595],
 [8.51063829787234e-2,2.127659574468085e-2,4.25531914893617e-2,6.382978723404255e-2]]
*ContinuousGA> crossover 2 c1 c2
[[0.0,2.127659574468085e-2,0.1276595744680851,0.14893617021276595],
 [8.51063829787234e-2,0.10638297872340426,4.25531914893617e-2,6.382978723404255e-2]]
*ContinuousGA> crossover 3 c1 c2
[[0.0,2.127659574468085e-2,4.25531914893617e-2,0.14893617021276595],
 [8.51063829787234e-2,0.10638297872340426,0.1276595744680851,6.382978723404255e-2]]
*ContinuousGA> crossover 4 c1 c2
[[0.0,2.127659574468085e-2,4.25531914893617e-2,6.382978723404255e-2],
 [8.51063829787234e-2,0.10638297872340426,0.1276595744680851,0.14893617021276595]]
```

We observe that for `cp == 0`, we just clone the parents, as also happens for `cp` greater than 4. Otherwise, for `cp` equal to 1, 2, or 3, the offspring is created by a crossover point after gene number 1, 2, or 3, respectively.

1.12.2 The `matePairwise` function

There are many ways to choose which chromosomes should mate to create offspring, e.g., we could randomly draw a mother and a father chromosome from a subpopulation consisting of the `numKeep` best chromosomes in a population. Here, we simply use pairwise mating in a recursive function called `matePairwise`, where chromosomes 1 and 2 mate, chromosomes 3 and 4 mate, and so forth. The resulting offspring is returned as a `Population` together

with a `StdGen`, both in a tuple. The function requires several standard PRNGs. For this, we add the `split` function that comes with the `System.Random` library in our import:

```
import System.Random (StdGen, randomR, split, mkStdGen)
```

The implementation of `matePairwise` is given below:

```
matePairwise :: StdGen -> Population -> (Population, StdGen)
matePairwise g [] = ([], g)
matePairwise g [ma] = ([ma], g)
matePairwise g (ma:pa:cs) = (offspring ++ fst (matePairwise g' cs), g'')
    where (g', g'') = split g
          (g''', _) = split g''
          cp = fst $ randIndex g''' ma
          offspring = crossover cp ma pa
```

The two base cases say that an empty population should just return the empty list, and if the population only have a single chromosome, we should just clone it. The general case generates a random crossover point `cp` using the `randIndex` function and then calls the single point `crossover` function with `cp` and the first two chromosomes in the population to create two offspring. It then recursively repeats the process on the remainder of the population.

Example usage in `ghci`:

```
*ContinuousGA> let stringList =
["abcd","efgh","ijkl","mnop","qrst","uvwx","yz12","3456"]
*ContinuousGA> let pop = encodeStringList stringList
*ContinuousGA> pop
[[0.0,2.127659574468085e-2,4.25531914893617e-2,6.382978723404255e-2],
 [8.51063829787234e-2,0.10638297872340426,0.1276595744680851,0.14893617021276595],
 [0.1702127659574468,0.19148936170212766,0.2127659574468085,0.23404255319148937],
 [0.2553191489361702,0.2765957446808511,0.2978723404255319,0.3191489361702128],
 [0.3404255319148936,0.3617021276595745,0.3829787234042553,0.4042553191489361 4],
 [0.425531914893617,0.44680851063829785,0.46808510638297873,0.48936170212765956],
 [0.5106382978723404,0.5319148936170213,0.574468085106383,0.5957446808510638],
 [0.6170212765957447,0.6382978723404256,0.6595744680851063,0.6808510638297872]]
*ContinuousGA> let (newPop, g') = matePairwise (mkStdGen 99) pop
*ContinuousGA> newPop
[[0.0,2.127659574468085e-2,4.25531914893617e-2,0.14893617021276595],
 [8.51063829787234e-2,0.10638297872340426,0.1276595744680851,6.382978723404255e-2],
 [0.2553191489361702,0.2765957446808511,0.2978723404255319,0.3191489361702128],
 [0.1702127659574468,0.19148936170212766,0.2127659574468085,0.23404255319148937],
 [0.3404255319148936,0.44680851063829785,0.46808510638297873,0.48936170212765956],
 [0.425531914893617,0.3617021276595745,0.3829787234042553,0.4042553191489361 4],
 [0.5106382978723404,0.5319148936170213,0.6595744680851063,0.6808510638297872],
 [0.6170212765957447,0.6382978723404256,0.574468085106383,0.5957446808510638]]
*ContinuousGA> decodePopulation newPop
["abch","efgd","mnop","ijkl","qvwx","urst","yz56","3412"
```

We observe that the crossover point for chromosomes 1 and 2 in `pop` was after the third gene; for chromosomes 3 and 4 it was after the fourth gene (cloning); for chromosomes 5 and 6 it was after the first gene; and for chromosomes 7 and 8 it was after the second gene.

Note that if we do not want to allow cloning, we could limit the `randIndex` function to always return a random index smaller than the lenght of the chromosome.

1.13 Mutation functions

Mutation involves changing a gene value to a new random value limited to $g_{\text{low}}$and g_{high}. The fraction (mutation rate) of chromosomes in the population that should mutate is called `mutRate`, and corresponds to the integer `numMut`. For example, for a population size of `numPop == 100` and a mutation rate of `mutRate == 0.1`, the number of chromosomes to mutate would be `numMut == 10`.

The necessary mutation functions are given below.

1.13.1 The `replaceAtIndex` function

If we have a gene variable, or a list of genes (a chromosome), we cannot just change (overwrite) it as we likely would do in an imperative language, since data variables in a purely functional language like Haskell are immutable (they cannot change once defined). Instead, we must copy the data we need and put it together in a new data structure or variable. We therefore implement a helper function that can replace an item in a list at a particular index, namely the `replaceAtIndex` function below:

```
replaceAtIndex :: Int -> a -> [a] -> [a]
replaceAtIndex n item ls = as ++ (item:bs) where (as, (b:bs)) = splitAt n ls
```

This function uses the `splitAt` function available in Prelude (so no need to import it) to split the list `ls` at the position `n` into two sublists `as` and `(b:bs)`, where `as` are the first `n` elements in `ls`, and `(b:bs)` are the remainding elements. The element `b` is then replaced with `item` and the pieces are put together again using the `++` function.

Example usage in `ghci`:

```
*ContinuousGA> replaceAtIndex 5 99 [0,1,2,3,4,5,6,7,8,9]
[0,1,2,3,4,99,6,7,8,9]
```

1.13.2 The `mutateChrom` function

The `mutateChrom` function mutates a randomly selected gene among its list of genes:

```
mutateChrom :: StdGen -> Chromosome -> (Chromosome, StdGen)
mutateChrom g chrom = (mutChrom, g2')
    where (g1, g2) = split g
          (nGene, g1') = randIndex g1 chrom
          (mutGene, g2') = randGene g2
          mutChrom = replaceAtIndex nGene mutGene chrom
```

An index `nGene` in the chromosome `chrom` is picked randomly. A new mutated gene `mutGene` at this index is then mutated using the `randGene` function. Finally, a new chromosome `mutChrom` that is identical to `chrom` except that the gene at index `nGene` has been mutated is returned.

Example usage in `ghci`:

```
*ContinuousGA> let s = "hello world!"
*ContinuousGA> let c = encodeString s
*ContinuousGA> let (cMut,g') = mutateChrom (mkStdGen 99) c
*ContinuousGA> decodeChromosome  cMut
"hello worldl"
*ContinuousGA> let (cMut,g') = mutateChrom (mkStdGen 98) c
*ContinuousGA> decodeChromosome  cMut
"hello/world!"
```

Using a PRNG obtained from `mkdStdGen 99`, the last gene is mutated, that is, a randomly generated gene corresponding to the character 'l' replaces the letter '!', whereas using a PRNG obtained from `mkdStdGen 98`, the sixth gene is mutated, that is, a randomly generated gene corresponding to the character '/' replaces the space character ' '.

1.13.3 The `mutateChromInPop` function

The function `mutateChromInPop` mutates a chromosome at index `n` in population `pop`:

```
mutateChromInPop :: StdGen -> Int -> Population -> (Population, StdGen)
mutateChromInPop g n pop = (replaceAtIndex n mutChrom pop, g')
    where (g', g'') = split g
          (mutChrom, _) = mutateChrom g'' (pop!!n)
```

1.13.4 The `mutIndices` and `mutatePop` functions

Finally, we need a function called `mutatePop` that given a list of indices, mutates all its chromosomes at those indices.

To randomly generate a list of indices, we create a function `mutIndices`:

```
mutIndices :: Population -> StdGen -> [Int]
mutIndices pop g = take numMut $ randomRs (numElite, length pop - 1) g
```

The function makes use of the `randomRs` function from the `System.Random` library, hence we add it to our import statement:

```
import System.Random (StdGen, randomR, randomRs, split, mkStdGen)
```

The `randomRs` function produces an infinite list of random indices limited to lower and upper bounds given by it first argument. Here, the lower bound is `numElite`, because we do not want to mutate any of the elite chromosomes, and the upper bound is the index of the

last chromosome in the population. Finally, we take only the first `numMut` indices from the infinite list.

Now that we have a a function to generate a list of random indices for the chromosomes that shall be mutated, we can implement a recursive `mutatePop` function:

```
mutatePop :: StdGen -> [Int] -> Population -> (Population, StdGen)
mutatePop g _ [] = ([], g)
mutatePop g [] pop = (pop, g)
mutatePop g (n:ns) pop = mutatePop g' ns pop'
    where (pop', g') = mutateChromInPop g
```

The first base case says to do nothing if the population is empty or the list of indices is empty. Given a list `(n:ns)` of indices, the recursive case uses the `mutateChromInPop` function defined above to mutate the chromosome at index `n` in the population before it recursively continues with the remaining `ns` indices. A list of random indices can be provided by the `mutIndices` function.

Example usage in `ghci`:

```
*ContinuousGA> decodePopulation pop
["abcd","efgh","ijkl","mnop","qrst","uvwx","yz12","3456"]
*ContinuousGA> numMut
4
*ContinuousGA> let mutIdx = mutIndices pop (mkStdGen 99)
*ContinuousGA> mutIdx
[7,5,6,3]
*ContinuousGA> let (mutPop, g') = mutatePop (mkStdGen 99) mutIdx pop
*ContinuousGA> decodePopulation mutPop
["abcd","efgh","ijkl","miop","qrst","u5wx","yz1c","o456"]
```

Because `numMut == 4`, four random chromosomes in `pop` are mutated. The indices of these four chromosomes is determined by calling the `mutIndices` function, which returns `mutIdx = [7,5,6,3]`. By comparing the mutated population `mutPop` with the original population `pop`, we observe that for the chromosome at index 3, the second gene was mutated; for the chromosome at index 5, the second gene was mutated; for the chromosome at index 6, the fourth gene was mutated; and for the chromosome at index 7, the first gene was mutated.

1.14 Evolution functions

Our GA is almost finished but the most important step is left: evolution. We need two functions, `evolvePopOnce` and `evolvePop`, to complete the GA.

1.14.1 The `evolvePopOnce` function

The `evolvePopOnce` function evolves a population from one generation to the next:

```
evolvePopOnce :: StdGen -> Population -> (Population, StdGen)
evolvePopOnce g pop = (newPopMutated, g4)
    where (g', g'') = split g
          ePop = evalPop target pop
          sPop = sortPop ePop
          parents = toPopulation $ getParents sPop
          (offspring, g3) = matePairwise g' parents
          newPop = parents ++ offspring
          mutIdx = mutIndices newPop g3
          (newPopMutated, g4) = mutatePop g'' mutIdx newPop
```

Its input is a population `pop` and the output is a new population `newPopMutated` that has been constructed through genetic operations. Each of the chromosomes in `pop` is evaluated with respect to the target string `target` (hardcoded at the top of the module) and sorted according to their associated cost. Parents to be kept for the next generation and for generating offspring are selected using the `getParents` function, and to convert from a list of (cost, chromosome) pairs, we use the `toParents` function. The result is assigned to `parents`. The function `matePairwise` then create `offspring` using single point crossover on the chromosomes in `parents`. The `parents` and the `offspring` collectively become the new population `newPop`. Finally, a number `numMut` of the chromosomes in `newPop` are mutated and the results is the next generation `newPopMutated`.

Here is an example usage in `ghci` for a `target=s2` string, where `s2="abc"`, and the following GA settings:

```
numPop = 8     :: Int    -- Population size (number of chromosomes)
xRate = 0.5    :: Double -- Selection rate
mutRate = 0.5  :: Double -- Mutation rate
numElite = 2   :: Int    -- Number of elite chromosomes
```

```
*ContinuousGA> let stringList =
["xxx","def","ghj","klm","nop","qrs","tuv","wxy"]
*ContinuousGA> let pop = encodeStringList stringList
*ContinuousGA> let (popEvolvedOnce,g') = evolvePopOnce (mkStdGen 99) pop
*ContinuousGA> decodePopulation popEvolvedOnce
["def","ghj","wum","nwp","9hj","gef","kop","nlm"]
```

1.14.2 The `evolvePop` function

Finally, we create the `evolvePop` function. This function evolves a population `n` times recursively, making use of the `evolvePopOnce` function just described.

```
evolvePop :: StdGen -> Int -> Population -> (Population, StdGen)
evolvePop g 0 pop = (newpop, g)
    where newpop = toPopulation $ sortPop $ evalPop target pop
evolvePop g n pop = evolvePop g' (n-1) newPop
    where (newPop, g') = evolvePopOnce g pop
```

Example usage in `ghci`:

```
*ContinuousGA> let (popEvolvedNTimes,g') = evolvePop (mkStdGen 99) 10 pop
*ContinuousGA> decodePopulation popEvolvedNTimes
["dcf","dcf","dcf","dcf","dc1","dc1","x_1","5cf"]
*ContinuousGA> let (popEvolvedNTimes,g') = evolvePop (mkStdGen 99) 20 pop
*ContinuousGA> decodePopulation popEvolvedNTimes
["dcf","dcf","dcf","dcf","dcf","d6z","d+f","ucf"]
*ContinuousGA> let (popEvolvedNTimes,g') = evolvePop (mkStdGen 99) 100 pop
*ContinuousGA> decodePopulation popEvolvedNTimes
["cbc","cbc","cbc","cbc","cbc","cbc","cb:","g2c"]
*ContinuousGA> let (popEvolvedNTimes,g') = evolvePop (mkStdGen 99) 200 pop
*ContinuousGA> decodePopulation popEvolvedNTimes
["abc","abc","abc","abc","ab.","apc","9bc",".bc"]
```

We observe that neither 10, 20, nor 100 generations were sufficient to find (learn) the target string, whereas for 200 generations, the population evolved and the best chromosome was identical to the target, decoded as `"abc"`. The binary GA studied in the previous tutorial, only needed 20 generations for the same GA settings.

1.15 Final remarks

Congratulations! You should now have a working GA contained in your `ContinuousGA` module. Even if you are generous with comments and line shifts (as I am), your module should be less than 300 lines.

What remains is to test the GA. We will investigate this in the next section.

2 Exercises

2.1 Testing

You should test that the continuous GA for string learning works properly on a set of test strings. Create a test module called `TestContinuousGA.hs` to hold your code:

```
module TestContinuousGA where
```

The module should import the GA module:

```
import ContinuousGA
```

You should then implement a `main` function with the necessary steps to verify proper functionality of the GA:

```
main :: IO ()
main = do
```

For example, you could do something like the following:

```
        -- select a target string
        -- obtain a PRNG g
        -- initialise a random population
        -- print the decoded population (list of candidate strings)
        -- evolve the population for itMax iterations
        -- print the decoded evolved population (list of evolved strings)
        -- print the decoded best chromosome found and its cost
```

To help you out, here is an example module implementing the steps listed above:

```
module TestContinuousGA where

import ContinuousGA
import System.Random (newStdGen)

main :: IO ()
main = do
        g <- newStdGen
        let (initPop, g') = randPop g
        putStr "Initial population decoded to strings:\n"
        print $ decodePopulation initPop
        let newPopEvolved = fst $ evolvePop g' itMax initPop
        putStr "Final population decoded to strings:\n"
        print $ decodePopulation newPopEvolved
        putStr "Target:\n"
        print target
        putStr "Best solution:\n"
        print $ (decodePopulation newPopEvolved) !! 0
        putStr "Cost of solution:\n"
        print $ chromCost target (newPopEvolved !! 0)
```

Experiment with algorithm settings such as

- population size (number of chromosomes)
- maximum number of iterations
- selection rate
- mutation rate

and discuss how your choice of GA settings affect the following:

- ability to find the correct target string
- cost of the GA-generated string (zero for finding the correct string)
- number of iterations needed
- total runtime

for a number of different test strings, e.g.

- `abc123.,;`

- `descartes: cogito ergo sum`
- `2+2 is: 4, 2*2 is: 4; why is _/not/_ 2.2*2.2 equal to 4.4?!`

If you modify your cost functions or implement other cost functions, examine the performance of the GA for each of these.

2.2 Solve your own problem

This exercise is about adapting the continuous GA to some other problem. Find a problem that a continuous GA can solve. A suitable problem can be a combinatorial problem such as string learning presented here, or the test functions given in the appendix of Haupt & Haupt (2004). You will need to think about how to encode the chromosomes, e.g., the sequence of genes could correspond to optimisation variables. You must also define new cost functions to evaluate the chromosomes. Much of the existing code can be reused.

References

Haupt, R. L., & Haupt, S. E. (2004). *Practical Genetic Algorithms*. Wiley, 2nd ed.