Week 14: The travelling salesperson problem (TSP)

Robin T. Bye

25th April 2017

Time	Topic	Reading
09.15	Overview and status update	
09.30	Lecture: Introduction to TSP & alternative chromosome encoding (no slides)	
10.15	Workshop/tutorial: The travelling salesperson problem (TSP)	*

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

Date for workshop: **Tuesday 25 April**

Data files for two TSPs:

- wi29info.txt
- wi29.txt
- dj38info.txt
- dj38.txt

This PDF document is available in an HTML version at `http://www.hg.schaathun.net/FPIA/week14.html`.

Contents

1 Tutorial: GA for TSP

In this tutorial, we will implement an integer-based GA for TSP based on the continuous GA that we implemented previously. A lot of the continuous GA code can be reused as is or just slightly modified. Therefore, this tutorial will necessarily contain much of the same information as the tutorial on the continuous GA.

The purpose of this GA is solve problems that are formulated as TSPs. That is, given a set of cities in the Euclidean 2D space, starting at any city, the GA should find the shortest connected path visiting all cities exactly once before returning the original city. A chromosome should represent a candidate solution, that is, a sample permutation (sequence of cities). We will assume that n cities are labeled as natural numbers, $1, 2, \ldots, n$. For example, for $n = 7$ cities, a candidate solution could be given by $[4, 6, 2, 7, 3, 1, 5]$. The cost function is the total distance travelled when visiting the cities of such a permutation.

However, because a TSP tour cannot include repeated cities, ordinary crossover and mutation will not work, because these genetic operations may introduce duplicates in offspring. There are several ways to fix this problem. For example, we can write specialised crossover and mutation methods that checks if duplicates will occur and avoids it (e.g., the PMX method). Here, we will use the method from a paper by Üçoluk (2002), where a one-to-one relationship between a permutation and its inverse is presented. The inverse is allowed to have duplicates. Therefore, we can encode our chromosomes as the inverse of the permutations they represent, and use our ordinary crossover and mutation methods as before. To evalute the total distance travelled (cost), we must decode chromosomes to permutations and use a distance function on the resulting permutation.

1.1 Test problems

We will implement a GA for TSP using the alternative chromosome encoding proposed in Üçoluk (2002) and try to solve the following TSPs (taken from www.math.uwaterloo.ca/tsp/world/countries.html):

- 29 cities in Western Sahara:

  - wi29info.txt: Information about problem and list of cities with (x,y) coordinates.
  - wi29.txt Stripped file with list of cities with (x,y) coordinates only.
- 38 cities in Djibouti:
  - dj38info.txt: Information about problem and list of cities with (x,y) coordinates.
  - dj38.txt Stripped file with list of cities with (x,y) coordinates only.

For motivation, we will run an informal competition to see who manages to get the shortest TSP distance..!

You should place both the stripped files in the same directory as your Haskell module.

1.2 Test module

To show how you can read in the test data, we first provide a test module that can be used later when you have finished the implementation of your TSP GA:

```haskell
module TestTSPGA where

import TSPGA
import Data.List (sort)
import System.Random (newStdGen)

main :: IO ()
main = do
        contents <- readFile "wi29.txt"
        let s = map words (lines contents)
        let cities = map stringsToCity s
```

This test module makes use of a helper function:

```haskell
stringsToCity :: [String] -> City
stringsToCity [c, x, y] = (read c, (read x, read y))
```

The function `stringsToCity` converts a list of three strings of the format `[cityid, xpos, ypos]` to a `City`.

1.3 A continuous GA for the TSP

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

```haskell
module TSPGA where
```

1.4 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 (2004) and define the following parameters with some default settings:

```
numPop = 100    :: Int     -- Population size (number of chromosomes)
xRate = 0.5     :: Double  -- Selection rate
mutRate = 0.5   :: Double  -- Mutation rate
numElite = 20   :: Int     -- Number of elite chromosomes
itMax = 500     :: Int     -- Max number of iterations
```

Note that these settings may be good or bad, and you will likely need to experiment to obtain good solutions.

1.5 Genes, chromosomes, and population

In the following, we will assume that a gene is an integer that encodes a single number in an inverse sequence corresponding to a permutation, and that a chromosome is a list of genes that encodes the complete sequence.

The total number of (encoding) variables, `numVar`, is equal to the number of cities in t test problem we will try to solve:

```
numVar = 29 :: Int -- replace with length of a list of cities
```

This number tells us how many genes are required in each chromosome. Here, we have hardcoded the number to suit the Djibouti problem with 29 cities, and you must change it to 38 if solving the Western Sahara problem.

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 = 100` and `xRate = 0.5`, we get `numKeep = 50`. 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:

```
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:

```
type CityID     = Int                  -- city id
type Position   = (Double, Double)     -- city position (x, y)
type City       = (CityID, Position)   -- (city id, x-coordinate, y-coordinate)
type Gene       = CityID               -- integer
type Chromosome = [Gene]               -- list of genes
type Population = [Chromosome]         -- list of chromosomes
```

A `CityID` is an integer representing a city, whilst `Position` is a tuple consisting of the x and y coordinates of a city in a Euclidean space. A `City` is a tuple consisting of a `CityID` and the `Position` of that city. We simply let a `Gene` be a `CityID`, a `Chromosome` a list of `Gene`, and a `Population` a list of `Chromosome`.

The variables (city IDs) are constrained as

```
pLo = 1      :: CityID
pHi = numVar :: CityID
```

whilst we let the genes have the same constraints.

```
gLo = pLo :: CityID
gHi = pHi :: CityID
```

The reason we have the same constraints on the genes is that the inverse sequence (an encoded chromosome) of a permutation will contain only (some of) the same numbers as those of the permutation. Note that if we planned to use some other encoding scheme, we would have to change `gLo` and `gHi` accordingly.

1.6 Encoding functions

1.6.1 The `encodeCity` function

We need a function `encodeCity` to encode a `City` as a `Gene`:

```
encodeCity :: City -> Gene
encodeCity (cityID, cityPos) = cityID
```

1.6.2 Direct encoding using the `encodeTour'` function

If we encode a TSP tour (list of cities) directly to a chromosome, we could use the function `encodeTour'`:

```haskell
encodeTour' :: [City] -> Chromosome
encodeTour' = map encodeCity
```

However, as we have discussed already, this would cause problems with repeated cities when performing ordinary crossover and mutation.

1.6.3 The `inverseSequence` function

Instead, we implement a function `inverseSequence` that creates a sequence of numbers representing a permutation:

```haskell
-- Inverse sequence representing a permutation
-- This sequence allows repetitive values and hence is robust under ordinary
-- (n-point) crossover. There is a one-to-one mapping from ordinary permutation
-- representation to the inversion sequence representation.
-- Source: http://www.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf
inverseSequence :: [CityID] -> [Int]
inverseSequence cs = inverseSequence' cs ns
    where ns = [1..length cs]

inverseSequence' :: [CityID] -> [Int] -> [Int]
inverseSequence' cs [j] = [length $ filter (> j) $ takeWhile (/= j) cs]
inverseSequence' cs (j:js) = inverseSequence' cs [j] ++ inverseSequence' cs js
```

This sequence is allowed to have duplicates. Later, we implement a `fromInverse` function to get back to a permutation from such an inverse sequence.

1.6.4 The `encodeTour` function

To encode a TSP tour (permuation of cities) as an inverse sequence in the chromosome, we compose a function consisting of `inverseSequence` and the direct encoding function `encodeTour'`:

```haskell
-- Encode a tour (permutation of cities) as an inverse sequence
encodeTour :: [City] -> Chromosome
encodeTour = inverseSequence . encodeTour'
```

1.6.5 The `encodeTours` function

Finally, the `encodeTours` function encodes a list of tours as a list of inverse sequences:

```haskell
encodeTours :: [[City]] -> Population
encodeTours = map encodeTour
```

1.7 Decoding functions

In addition to encoding functions, we also need decoding functions able to convert back from inverse sequences contained in chromosomes back to TSP tours (permutations).

1.7.1 The `fromInverse` function

We define a `fromInverse` function that converts an inverse sequence to a permutation:

```haskell
-- Convert inverse sequences generated with inverseSequence to permutations
-- Source: http://www.ceng.metu.edu.tr/~ucoluk/research/publications/tspnew.pdf
fromInverse :: [Int] -> [Int]
fromInverse inv = map snd $ filter ((>0) . fst) $ sort $ zip permpos [0..]
    where permpos = updatePosMany inv0 pos0 i n
          inv0 = 0 : inv
          pos0 = take (length inv0 + 1) $ repeat 0
          i = length inv
          n = i
```

It is slightly hard work to implement the `fromInverse` function based on the second iterative algorithm in Üçoluk (2002), and a number of helper functions are required (you may be able to do this in a simpler way yourself):

```haskell
-- Helper functions to fromInverse
updatePosMany :: [Int] -> [Int] -> Int -> Int -> [Int]
updatePosMany inv pos i n
            | i < 1 = pos
            | otherwise = updatePosMany inv (updatePos inv pos i n) (i - 1) n

updatePos :: [Int] -> [Int] -> Int -> Int -> [Int]
updatePos inv pos i n = updatePosi inv (updatePosmMany inv pos (i + 1) n i) i

updatePosi :: [Int] -> [Int] -> Int -> [Int]
updatePosi inv pos i = replaceAtIndex i (inv!!i + 1) pos

updatePosmMany :: [Int] -> [Int] -> Int -> Int -> Int -> [Int]
updatePosmMany inv pos m n i
        | m == n     = updatePosm inv pos m i
        | m > n      = updatePosmMany inv (updatePosm inv pos m i) (m - 1) n i
        | otherwise = updatePosmMany inv (updatePosm inv pos m i) (m + 1) n i

updatePosm :: [Int] -> [Int] -> Int -> Int -> [Int]
updatePosm inv pos m i
            | pos!!m >= inv!!i + 1 = replaceAtIndex m (pos!!m + 1) pos
            | otherwise = pos
```

1.8 Testing the `inverseSequence` and `fromInverse` functions

We can test the `inverseSequence` and `fromInverse` functions on two permutations p_1 and p_2 with corresponding inverse sequences i_1 and i_2, respectively, given in Üçoluk (2002):

$$p_1 = [5, 7, 1, 3, 6, 4, 2] \leftrightarrow i_1 = [1, 5, 2, 3, 0, 1, 0] \quad (1)$$

$$p_2 = [4, 6, 2, 7, 3, 1, 5] \leftrightarrow i_2 = [5, 2, 3, 0, 2, 0, 0] \quad (2)$$

Example usage in `ghci`:

```
*TSPGA> let perm1 = [5,7,1,3,6,4,2]
*TSPGA> let perm2 = [4,6,2,7,3,1,5]
*TSPGA> let invseq1 = inverseSequence perm1
*TSPGA> let invseq2 = inverseSequence perm2
*TSPGA> invseq1
[2,5,2,3,0,1,0]
*TSPGA> invseq2
[5,2,3,0,2,0,0]
*TSPGA> let perm1' = fromInverse invseq1
*TSPGA> let perm2' = fromInverse invseq2
*TSPGA> perm1'
[5,7,1,3,6,4,2]
*TSPGA> perm2'
[4,6,2,7,3,1,5]
```

1.8.1 The `decodeGene` function

There is no way to decode a gene by itself, since a gene is a number in an inverse sequence in which the elements depend on each other. We need to decode an entire chromosome to find the permutation it represent; we cannot decode separate genes. Also, there is no need for a `decodeGene` function. Hence, this function is not implemented.

1.8.2 The `decodeChromosome` function

To decode a `Chromosome`, we just call the `fromInverse` function:

```haskell
decodeChromosome :: Chromosome -> [CityID]
decodeChromosome = fromInverse
```

1.8.3 The `decodePopulation` function

To decode an entire population (list of chromosomes) of chromosomes, we just map the `decodeChromosome` function over the population:

```haskell
decodePopulation :: Population -> [[CityID]]
decodePopulation = map decodeChromosome
```

1.9 Testing of encoding/decoding

Example functionality in `ghci`:

```
*TSPGA> let c1 = (1,(0,0)) :: City
*TSPGA> let c2 = (2,(1,1)) :: City
*TSPGA> let c3 = (3,(1,2)) :: City
*TSPGA> let c4 = (4,(2,1)) :: City
*TSPGA> let tour1 = [c1,c2,c3,c4]
*TSPGA> let tour2 = [c2,c3,c1,c4]
*TSPGA> let tour3 = [c3,c1,c2,c4]
*TSPGA> let tour4 = [c4,c1,c3,c2]
*TSPGA> let tours = [tour1,tour2,tour3,tour4]
*TSPGA> let pop = encodeTours tours
*TSPGA> pop
[[0,0,0,0],[2,0,0,0],[1,1,0,0],[1,2,1,0]]
*TSPGA> decodePopulation pop
[[1,2,3,4],[2,3,1,4],[3,1,2,4],[4,1,3,2]]
```

1.10 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`:

```haskell
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.

1.10.1 The `randGene` function

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

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

1.10.2 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
```

1.10.3 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
```

1.10.4 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`.

1.10.5 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
```

1.10.6 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
```

1.11 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.

1.11.1 The `distance` function

We begin with the `distance` function, which calculates the Euclidean distance between two cities in Euclidean 2D space:

```haskell
distance :: City -> City -> Double
distance (c1, (x1, y1)) (c2, (x2, y2)) = sqrt $ (x2 - x1)^2 + (y2 - y1)^2
```

1.11.2 The `distanceTour` function

The `distanceTour` function calculates the total distance of a TSP tour. We first connect the end city back to the starting city, and then use a helper function `distanceTour'` that adds up all the intercity distances:

```haskell
-- Distance of a TSP tour
distanceTour :: [City] -> Double
distanceTour cs = distanceTour' cs'
    where cs' = cs ++ [head cs]

-- Distance of a tour where the end city is not connected back to the beginning
distanceTour' :: [City] -> Double
distanceTour' [] = 0
distanceTour' [c1,c2] = distance c1 c2
distanceTour' (c1:c2:cs) = distance c1 c2 + distanceTour' (c2:cs)
```

1.11.3 The `cityIDToCity` function

The cityIDToCity function looks up a CityID in a list of cities [City] and returns the city if it exists, otherwise it returns an error:

```haskell
cityIDToCity :: [City] -> CityID -> City
cityIDToCity cities cityID = (cityID, stripMaybe cityPos)
    where cityPos = lookup (cityID) cities
          stripMaybe (Just cityPos) = cityPos
          stripMaybe (Nothing) = error $ "City does not exist!"
```

1.11.4 The `chromToTour` function

The `chromToTour` function uses the `cityIDToCity` function to look up each of the cities in a decoded chromosome to find the TSP tour (list of cities) that the chromosome corresponds to:

```haskell
chromToTours :: [City] -> Chromosome -> [City]
chromToTours cities = map (cityIDToCity cities) . decodeChromosome
```

1.11.5 The `popToTours` function

The `popToTours` function maps the `chromToTour` function over each of the chromosomes in a population to construct a list of TSP tours:

```haskell
popToTours :: [City] -> Population -> [[City]]
popToTours cities = map (chromToTours cities)
```

1.11.6 The `chromCost` function

We are now ready to find the cost of a chromosome, which is the total distance of the TSP tour that the chromosome encodes:

```haskell
chromCost :: [City] -> Chromosome -> Double
chromCost cities = distanceTour . chromToTours cities
```

1.11.7 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

```haskell
type ChromCost = (Double, Chromosome)
```

Next, we define a `chromCostPair` function to construct such tuples from a list of cities and the chromosome we want to evaluate:

```haskell
chromCostPair :: [City] -> Chromosome -> ChromCost
chromCostPair cities c = (chromCost cities c, c)
```

1.12 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`.

1.12.1 The `evalPop` function

The `evalPop` function evaluates all of the chromosomes in a population by mapping the partial function `chromCostPair cities` over the population, where `cities` is a list of all the cities for the TSP and their `Position`. 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 :: [City] -> Population -> [ChromCost]
evalPop cities = map (chromCostPair cities)
```

1.12.2 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 = sort
```

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.

1.12.3 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:

```haskell
selection :: [ChromCost] -> [ChromCost]
selection = take (numKeep - numElite) . drop numElite
```

1.12.4 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.

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

1.12.5 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`:

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

1.13 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.

Note that because we are using inverse sequences for our encoded chromosomes, we can use the ordinary single-point crossover function and the mutation function defined previously for the continuous GA.

1.13.1 The `crossover` function

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

```haskell
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`.

1.13.2 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:

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

The implementation of `matePairwise` is given below:

```haskell
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.

1.14 Mutation functions

Mutation involves changing a gene value to a new random value limited to g_{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.

1.14.1 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:

```haskell
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]
```

1.14.2 The `mutateChrom` function

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

```haskell
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.

1.14.3 The `mutateChromInPop` function

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

```haskell
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)
```

1.14.4 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`:

```haskell
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:

```haskell
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:

```haskell
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 n pop
```

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.

1.15 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.

1.15.1 The `evolvePopOnce` function

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

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

Its input is a list of cities and their positions named `cities` and 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 by finding the total distance of the TSP tour they each encode, and then sorted.

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`.

1.15.2 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.

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

1.16 Alternative encoding/decoding scheme

As pointed out by student Rune Valle (thank you!), an alternative encoding/decoding scheme for TSP problems can be found here: frcatel.fri.uniza.sk/users/pesko/publ/DeTSP.pdf

Please feel free to use this scheme if you like.

Below is a module with the necessary code:

```haskell
-- Alternative encoding/decoding scheme for TSP taken from
-- https://frcatel.fri.uniza.sk/users/pesko/publ/DeTSP.pdf

module GAtsmLehman where

-- Test data
perm1, perm2, invseq1, invseq2 :: [Int]
perm1 = [5,7,1,3,6,4,2]
perm2 = [4,6,2,7,3,1,5]
invseq1 = [1,5,2,3,0,1,0]
invseq2 = [5,2,3,0,2,0,0]

type Permutation = [Int]
type Lehman = [Int]

encodePermutation :: Permutation -> Lehman
encodePermutation [] = []
encodePermutation (ah:at) = (length $ filter (<ah) at):(encodePermutation at)

popItem :: Int -> [a] -> (a, [a])
popItem 0 (ah:at) = (ah, at)
popItem i (ah:at) = (output, ah:list)
    where
    (output, list) = popItem (i-1) at

identityPermutation :: Int -> Permutation
identityPermutation a = [1..a]

decodePermutation :: Lehman -> Permutation
decodePermutation a = decodePermutation' (identityPermutation (length a)) a
    where
    decodePermutation' :: Permutation -> Lehman -> Permutation
    decodePermutation' [] [] = []
    decodePermutation' n (lh:lt) = res:(decodePermutation' nrest lt)
        where
        (res, nrest) = popItem lh n
```

1.17 Final remarks

Congratulations! You should now have a working GA contained in your `TSPGA` module. Even if you are generous with comments and line shifts (as I am), your module should be around 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 TSP GA works properly on a set of TSP data such as `wi29.txt` and `dj38.txt`. Create a test module called `TestTSPGA.hs` to hold your code:

```haskell
module TestTSPGA where
```

The module should import the TSPGA module:

```haskell
import TSPGA
```

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

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

For example, you could do something like the following:

```haskell
        -- read in TSP test data and obtain a list cities with positions
        -- obtain a PRNG g
        -- initialise a random population
        -- print the decoded population (list of candidate tours)
        -- evolve the population for itMax iterations
        -- print the decoded evolved population (list of evolved tours)
        -- print the decoded best chromosome found and its cost,
        -- that is, the tour and its distance
        -- compare the distance of the tour with the optimal distance
        -- check that the solution is valid
```

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

```haskell
main :: IO ()
main = do
        contents <- readFile "wi29.txt"
        let s = map words (lines contents)
        let cities = map stringsToCity s
        g <- newStdGen
        let (initPop, g') = randPop g
        putStr "Initial population decoded to cities:\n"
        print $ decodePopulation initPop
        let newPopEvolved = fst $ evolvePop g' itMax cities initPop
        putStr "newPopEvolved:\n"
        print newPopEvolved
        putStr "Valid tours using encoded chromosomes:\n"
        print $ map validTour newPopEvolved
        putStr "Final population decoded to cities:\n"
        print $ decodePopulation newPopEvolved
        putStr "Valid tours using decoded chromosomes:\n"
        print $ map validTour (decodePopulation newPopEvolved)
        let bestSolution = (decodePopulation newPopEvolved) !! 0
        putStr "Best solution:\n"
        print bestSolution
        putStr "Cost of solution:\n"
        print $ chromCost cities (newPopEvolved !! 0)
        let bestTour = chromToTour cities (newPopEvolved !! 0)
        putStr "Best solution as a list of cities with positions:\n"
        print bestTour
        putStr "Length of tour:\n"
        print $ distanceTour bestTour
        putStr "Length of optimal tour: \n"
        putStrLn "27603 (wi29) and 6656 (dj38)"
        putStrLn " Deviation from optimal:"
        let deviation = ((distanceTour bestTour) - 27603) / 27603
        print deviation -- for wi29
        putStr "Valid solution:\n"
        print $ validTour' bestSolution (length cities)

stringsToCity :: [String] -> City
stringsToCity [c, x, y] = (read c, (read x, read y))

validTour :: [CityID] -> Bool
validTour tour = [1..length tour] == sort tour

validTour' :: [CityID] -> Int -> Bool
validTour' tour n = [1..n] == sort tour
```

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 a optimal TSP tour
- cost (distance) of the GA-generated tour (deviation from optimal)
- number of iterations needed
- total runtime

for the two TSP problems `wi29.txt` and `dj38.txt`.

References

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

Üçoluk, G. (2002). Genetic algorithm solution of the TSP avoiding special crossover and mutation. *Intelligent Automation & Soft Computing*, *8*(3), 265–272.