GAER

Sigurt Bladt Dinesen
sidi@itu.dk

March 30, 2016

1.1 Introduction

This is the lab-report for the first lab for the course `Artificial Life & Evolutionary Robotics: Theory, Methods and Art`. It is written as a literate Haskell program, which means the code will appear intertwined with the text.

1.2 Genetic algorithm for solving the Travelling Salesman Problem

First the Haskell library functions used in this lab:

```
import Data.Array (Array, accumArray, indices, (!))
import Data.Maybe (fromJust)
import Data.List (intersect, union, find, sortOn, maximum)
import System.Random (newStdGen, randoms)
```

There are four main points to define a genetic algorithm. Here are the definitions used in this lab:

Genotype and phenotype The genotype is just an ordered list of vertexes (city ids). The interpretation of the genotype (the phenotype) is a path through the graph representing our instance of TSP, progressing through the vertices in the genome, returning the first vertex in the end.

the first vertex in the list, and progress by traveling

```
type Genome = [Vertex]
```

Mutation The step that takes one generation of genomes, and creates a new generation of genomes. In this report, mutation is done by sexual reproduction between two genomes, by random crossover, producing only one child.

I experimented with random mutation as well, but did not achieve any better results.

```
crossOver :: Genome → Genome → Int → Genome
crossOver g h split = take split g ++ drop split h

mutate :: [(Int, Int)] → Genome → [Genome]
mutate (r : rs) genome = replace r genome : mutate rs genome
  where replace (i, j) g = map (λx → if x ≡ i then j else x) g
```

Fitness Determines the *evolutionary* value of a genome, for use in `selection`. The obvious choice for a (fitness) function here is the travel-cost of the phenotype. Neither mutation nor the definition of the genome ensures the *validity* of a genome in this solution, i.e. a genome could look like $[1,1,1,1,1,1,1,1,1,1]$, which is not just a *bad* solution, but not a solution at all. So the fitness function must steer the selection away from such genomes. To achieve this, another term is included in the fitness function: An exponential function (for smoothness) of the Jaccard distance between the genome, and the list of all vertexes ($[1,2,3,4,5,6,7,8,9,10]$). The final fitness function of a genome g is defined as

$$2^{J(g) \cdot T} + pathlength(g)$$

Where J is one minus the Jaccard distance between g and the set of all vertices, and T is a large constant term to ensure that non-zero Jaccard distances are weighed heavier than any path length.

```
pathCost :: Genome → Graph Int → Double
pathCost genome graph = f (genome ++ [head genome]) graph
  where f [x] _ = 0
    f (x : y : xs) graph = getEdgeCost x y graph + f (y : xs) graph
```

```
jaccardCost :: Genome → Graph Int → Double
jaccardCost genome graph =
  let allVertices = indices graph
    lenDouble = fromIntegral ∘ length
    unionSize :: Double; unionSize = lenDouble $ union genome allVertices
    intersectSize :: Double; intersectSize = lenDouble $ intersect genome allVertices
  in 1 − intersectSize / unionSize
```

```
cost :: Genome → Graph Int → Double
cost genome graph =
  let jaccard = jaccardCost genome graph
    path = pathCost genome graph
  in 2 ** (jaccard * (10 ↑ 2 + 20)) + path
```

Selection The step that takes one generation of genomes, and decides who (or whose children) gets to be part of the next generation of genomes. In this project, selection is done by taking the best half of the population, and letting it, and their children, be the new population. This does not preserve diversity very well, but seems to produce good results nonetheless.

```
evolve :: [Genome] → Graph Int → [Int] → [IO ()]
evolve pop g rInts = evolve_ pop g rInts 1 30
evolve_ _ _ _ _ 0 = [return ()]
evolve_ pop g rInts gen genLeft =
  let (winners, losers) = splitAt 250 $ sortOn (`cost`g) pop
    bestCrossed = map (λ(g, h, r) → crossOver g h r) $ zip3 winners (reverse winners) rInts
    newPop = bestCrossed ++ winners
    output = "Gen " ++ show gen ++ " " ++
      "Best: " ++
      show (cost (head winners) g) ++ " -- " ++
      show (head winners) ++ " " ++
      "Worst: " ++ show (cost (last winners) g) ++ " -- " ++
      show (last winners)
  in putStrLn output : evolve_ newPop g (drop 500 rInts) (gen + 1) (genLeft − 1)
```