Scientific Programming: Assignment 3

University of Cambridge

Henrik Åhl

January 14, 2017

Preface

This is an assignment report in connection to the *Scientific Programming* module in the Computational Biology MPhil progeamme at the University of Cambridge, Michaelmas term 2016. All related code is as of January 14, 2017 available per request by contacting the author at hpa22@cam.ac.uk.

1 The Travelling Salesman Problem

The Travelling Salesman Problem (TSP) is an NP-hard problem where a fictionary salesman wishes to travel past every city in a network such that every city is visited exactly once, and then return from the city of origin. The salesman also wishes to do this the fastest way possible, and thus seeks to find the shortest path such that the conditions are fulfilled.

In our version of the TSP, we utilise the problems found at the Ruprecht-Karls-Universität Heidelberg TSPLIB database [1], and choose to seek an optimal path in all the two-dimensional euclidian problems with known, mathematically proven optimal solutions. Our individuals are defined as `R`*list* objects, containing a tour and its corresponding fitness (cost).

We measure fitness in all of our implementations by considering the *cost* of a solution – in our case the total euclidian distance between the nodes in our given networks. As our implementations by design are not dependent on the cost aside from a greater than–lower than perspective, we do not normalise our measure (see below). The general encoding is defined as a vector containing integers corresponding to node indices, i.e. one index per node. All parameters were chosen through manual optimisation, i.e. a testing-correcting procedure.

1.1 Genetic Algorithm Design

Mutation operator We consider multiple mutation operators in this assignment. Key is that we initially mainly are interested in producing solutions widely different from our own, and later on solutions that swap or move longer connected circuits. We consider three separate operators, partly inspired by approaches mentioned by Larrañaga et al. [2] as well as by Schneider [3]:

Chunk mutation Two integers in the range of the tour length are drawn randomly. The bases in between of these indices are selected and moved to an interval determined by a new random number, which signifies the start position of the chunk.

Swap mutation For every base in the sequence a random number is drawn. If this number is lower than a defined *mutation probability*, the number is added to a pool. All the numbers into this pool are then shuffled and reinserted. For our mutation probability, we use the rate $1/2C$, where C is the number of cities in the map.

Reverse mutation Two indices are again drawn randomly. The sequence is then inverted and inserted, e.g. the subsequence (red) 3-5-2-4-1 being selected would return the sequence 3-4-2-5-1.

Testing various approaches shows that using several different operators is too computationally intensive for making the approach comparable to simulated annealing. We therefore in our latter analyses restrict ourselves to using only our *reverse mutation* operator. The settings then used consist of a population size of 14, where 90 % (12) of these undergo mutations every iteration. Similarly, 20 % (2) undergo crossover. Which individuals are selected for these operations is like the survival based on a tournament selection procedure (see below).

Crossover operator For our crossover operator we are interested in finding a way to preserve loops, but

otherwise retain the current structure. We therefore implement our version of what we denote as *pillar crossover*. In this process, we take two individuals, draw two random indices where the bases in between are kept. The lacking bases are taken from the other individual in the sequence they are found such that no already included bases will appear. For the other offspring, the inverse is applied, where all the bases aside from those in the prior region are kept, and bases from the first offspring then inserted into the gap. This process is inspired by Jacobson [4].

Selection operator We apply coupled elitistic ($n = 1$) and pairwise tournament selection, i.e. for every generation, we keep the most fit individual, and draw from the remaining subset iteratively two random individuals where we then choose the subject for survival from a binomial distribution ($p = 1$) until a full population is produced. That is, we always keep the best solution of the whole population, as well as the best individuals in the pairs.

1.2 Simulated Annealing Design

State alteration New states are produced through the same procedure as in the genetic algorithm *reverse mutation*, though also the . *chunk mutation* operator is tested; simulations do however give better results with the prior approach.

Temperature Initial temperature is set to 1000° Å[1]. Temperature is decreased exponentially, with a rate factor of $k = 0.9999$ per iteration.

1.3 Comparison

We simulate both of our algorithms on euclidian two-dimensional tours for 30 minutes, as both methods have in general converged at that point. Both approaches do a fairly good job with finding good solutions to the problem. However, none of the algorithms are particularly good at finding the global optimum, even though they typically get fairly close. Simulated annealing tends to have a much faster convergence towards a near-optimal state with respect to runtime, whereas the genetic algorithm converges more slowly, due to the higher computational effort needed.

Figure 5 shows the end results for the algorithms after 30 minutes runtime, As the result suggest, both methods produce very similar results in most cases, aside from the one tour with contains a very large number of nodes. Nevertheless, simulated annealing consistently performs slighlty better than the genetic algorithm, although it is likely that these differences mainly lie with respect to the design choices made, and not the intrinsic nature of the algorithms themselves; a 1-sized genetic algorithm with fitness-based selection should be comparable to simulated annealing. Also, reducing the selective pressure for the genetic algorithm should introduce more diversity, and possibly better results. However, this introduces another variable to configure, and would likely require a significantly larger population size, which would decrease the optimisation speed.

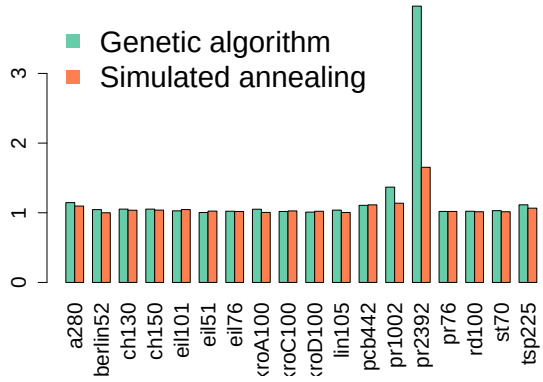

Figure 1: Performance comparison between genetic algorithms and simulated annealing, having run the simulations for 30 minutes. The fraction shown is the fractional distance travelled as opposed to the optimal tour. Only one simulation manages to completely solve the problem, namely the simulated annealing method in the *berlin52* setting.

We should also note that while the scaling of error measure used here does not affect the genetic algorithm design in any direct way, as our selection is deterministic in the comparative step. However, as soon as a probabilistic approach is introduced, we would have to account for this. In the case of simulated annealing the error measure already affects the parameter setup, i.e. the temperature and the cooling schedule.

2 The Kohonen Self-Organising Feature Map

2.1 Weight Distribution in Output Space

The Kohonen SOFM is designed utlising a rectuangular grid and linear learning factor decay. Being

[1] Degrees Åhl

aware of the benefits in choosing an appropriate initial distribution of our weight nodes, we disregard this in favor of mimicking the setup used by Beale and Jackson [5], i.e. a random initial distribution centered within weight-space. In this initial exercise, we generate one data point per iteration, and modify our network accordingly.

As can be seen in figure fig. 2, the ability of the grid to arrange itself according to the input space is heavily dependent on the parameters used. In the first row, the SOFM is able to neatly align itself according to its uniform two-dimensional input, whereas in the other cases, we are less successful. As all input is already on the same scale and of the same type, we do not bother with vector normalisation.

We can infer from our selection of simulations that the initial distribution is undoubtedly important. In particular, we can see how the neighbourhood is plays a significant role in breaking the initial distribution of nodes. When this does not happen, nodes can get trapped with the wrong setup relative to each other, as is particularly evident in fig. 2 rows 2 and 4. In contrast, row 3 shows a distribution tending towards the correct one, but is unable to break loose of the clusters due to the size of the neighbourhood. In principle they ultimately will as long as their relative position is correct with respect to the input distribution, but the time required for this depends largely on the learning rate.

2.2 SOFMs for Image Compression

2.2.1 SOFM Design and Test Images

For image compression, we apply our approach to two separate images: the common *Lena* image, featuring Swedish playmate Lena Söderberg, as well as an image of Italian actor and fashion model Fabio Lanzoni. These both consist, after modification, of 512×512 pixel lossless images (.tiff and .webp respectively), which are translated into their RGB colour channels in Rusing the `pixmap` and `webp` packages.

The images are encoded in the SOFM using either 4×4 or 8×8 bases large flattened input vectors, corresponding to same-sized pixel blocks in the original images. For training of the network, we employ two strategies:

Uniform input All input is looped over, passed into the SOFM one block at a time, giving an overall equal representation of all the blocks in the image.

Random input Random blocks in the original image is passed as input.

In theory, it ought to be reasonable to consider a third approach, where the images vectors passed as input are done so corresponding to how prevalent they are in the original image, i.e. a frequently occurring block would need to be passed as input less often. However, due to the need to computationally assess the similarity between different blocks, we do not consider this approach practically, although it is likely that it would produce better results due to being able to account for details to a greater degree.

Also, as opposed to Amerijckx et al. [6], who also used a DCT filter and a differential coding scheme, we have in our method only applied the SOFM directly. Introducing those extra treatments are likely to further enhance the quality of the compression. It is also likely that the SOFM could be improved upond through the implementation of a neighborhood function which would allow for proportional adjustment of the nodes given their respective distance to the winning one in the selection step, as it possibly would prevent the occurence of nodes being locked "out of place", as we have seen can happen in fig. 2.

2.2.2 Comparison to Single-Value Decomposition

In order to compare our method to SVD, we utilise the implementation done by Myles [7]. As we can see visually in fig. 5, both methods do a comparable job in compression when using a 30-worth SVD and a 4×4 block SOFM with 100 nodes. Still, it should be noted that the use of so many nodes was somewhat superfluous; similar results for the Lena image were attained using a mere 25 nodes.

The quality of the compression is notably degraded when using random inputs, which seems to reinforce the bias towards the more prevalent blocks, or at least establishes further noise with respect to the ability to pick up details in more rarer chunks. Compared to SVD, the SOFM produces grainier images related to the overall amount of detail in the image, i.e. a blockwise graininess. The SVD method on the other hand produces graininess with respect to visual features, such as in the transition between the curtain and the arm in the Fabio image, which are both of similar hue. Having the ability of self-organisation, the Kohonen approach is in that sense more capable of accounting for point-wise details by clustering more nodes in the region of the image weight-space which is more variable.

As we can note in fig. 3, the Kohonen net and the SVD approach are somewhat different in their outcome with respect to file size. Even though the results have here been ported through PNG compres-

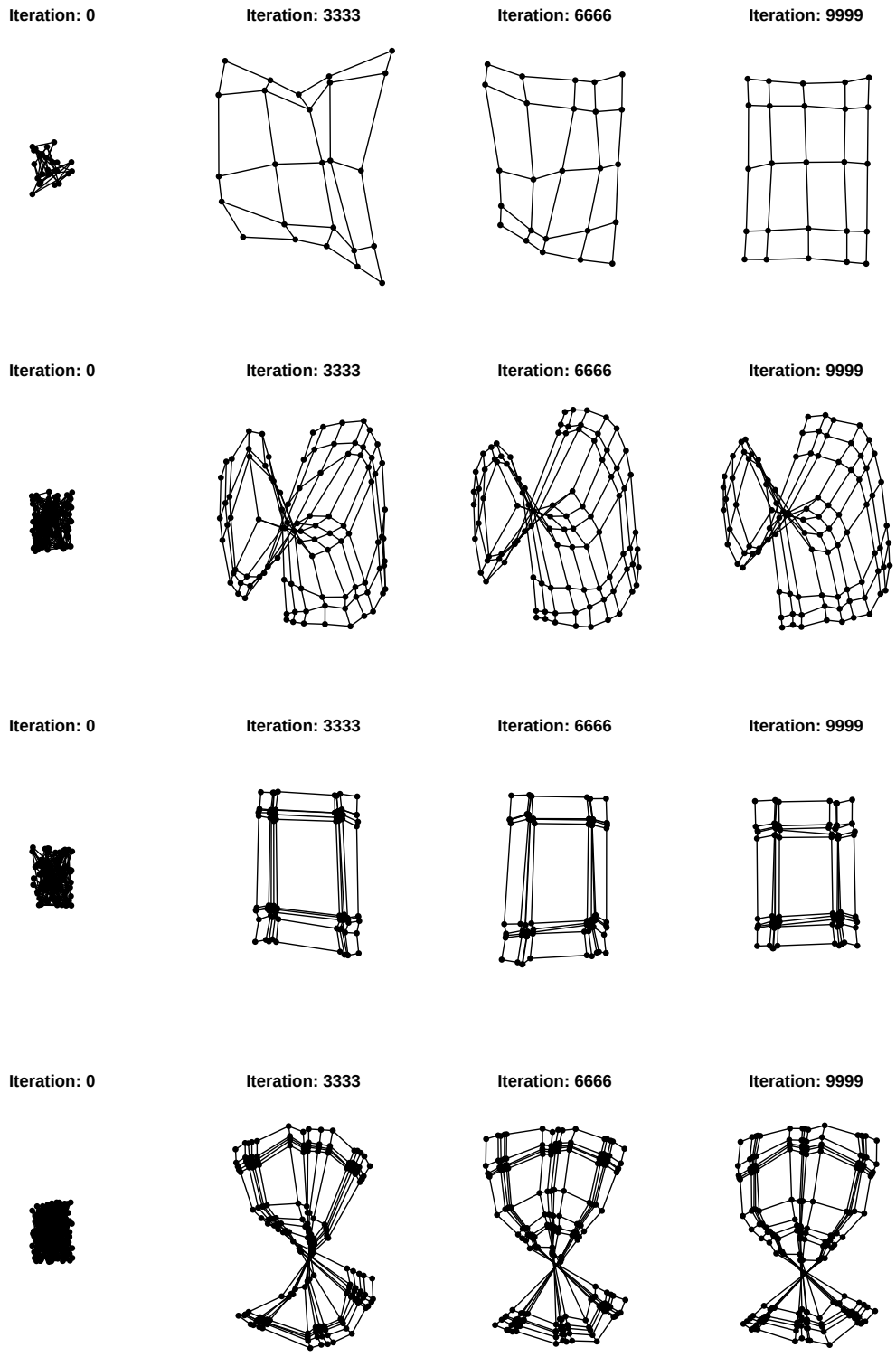

Figure 2: Shapes in weight-space produced by the Kohonen SOFM. The rows show various configurations of parameters, namely $\{0.5, 0.05, 0.03, 0.01\}$ for the learning rate (with a final rate of 0.01), $\{25, 100, 100, 225\}$ for the number of nodes (square grid), and $\{2, 3, 5, 5\}$ for the initial neighbourhood, in order. In all figures, the input consists of uniform, randomly generated 2D data.

sion, we can see traits of the respective encodings. In the SVD case, we achieve a linear or logarithmic trend for file size with increasing decomposition values, whereas the Kohonen net is significantly more static. We can also see the probabilistic nature of the SOFM, alternating in file size independently of the number of nodes (and also without loss of visual quality; image not shown). Furthermore, while it can be said that the SVD compression is affiliated with higher visual quality of the images the higher the decomposition value, the same cannot the said of the SOFM. However, when using larger blocks sizes, e.g. 8×8 chunks, the quality quickly deteriorates.

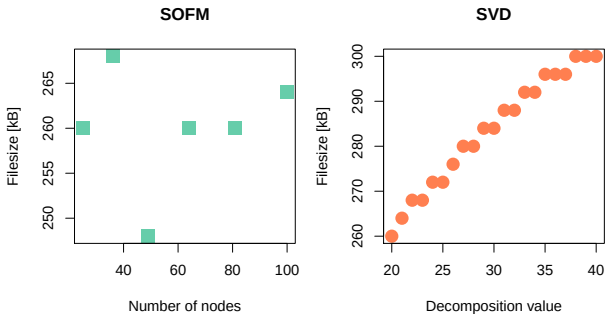

Figure 3: File sizes for images ported through PNG compression after their respective treatments. Note the differing scales.

Finally, we can consider the amount of information needed to store our data in fig. 5, where the visual quality is overall fairly comparable. In the Kohonen case we require $no.\ nodes \times block\ size + no.\ blocks = 17984$ pieces of information, whereas we for the SVD require $DV \times image\ width + DV^2 + DV \times image\ height = 31620$ pieces, compared to the 262144 pixels in the uncompressed image.

References

[1] *TSPLIB*. 2017. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/ (visited on 01/13/2017).

[2] P. Larrañaga et al. "Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators". In: *Artificial Intelligence Review* 13.2 (1999), pp. 129–170. ISSN: 1573-7462. DOI: 10.1023/A:1006529012972. URL: http://dx.doi.org/10.1023/A:1006529012972.

[3] Todd Schneider. *The Traveling Salesman with Simulated Annealing, R, and Shiny*. 2017. URL: http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/ (visited on 01/12/2017).

[4] Lee Jacobson. *Applying a genetic algorithm to the traveling salesman problem*. 2017. URL: http://www.theprojectspot.com/tutorial-post/applying-a-genetic-algorithm-to-the-travelling-salesman-problem/5 (visited on 01/14/2017).

[5] Russell Beale and Tom Jackson. *Neural Computing-an introduction*. CRC Press, 1990.

[6] Christophe Amerijckx et al. "Image compression by self-organized Kohonen map". In: *IEEE Transactions on neural networks* 9.3 (1998), pp. 503–507.

[7] John Myles White. *Image Compression with the SVD in R*. 2017. URL: http://www.johnmyleswhite.com/notebook/2009/12/17/image-compression-with-the-svd-in-r/ (visited on 01/12/2017).

Figure 4: Colour channels for the Lena and Fabio images. Lena is affiliated with more light and details, whereas Fabio has both a smoother background and more uniform lines in large.

Figure 5: Compression comparison for the different methods. The first column shows the uncompressed variants, whereas the second and third show compression using a 4×4 block SOFM (100 nodes), and compression using a 30-type SVD.

A Code

```
source("http://www.bioconductor.org/biocLite.R")
require(methods)
args     = commandArgs(trailingOnly=TRUE)
HOMEDIR = "/home/henrik/spa3/"
TSPDIR  = "tsp_data/"
setwd(HOMEDIR)
OPTFILE = ifelse(length(args) == 0, "tsp_data/a280.opt.tour", args[1])
FILE = gsub("\\.opt\\.tour", "\\.tsp", OPTFILE)
trim = function (x) gsub("^\\s+|\\s+$", "", x) # Trim whitespaces
source("aux.R")

# Read in data
file       = paste0(HOMEDIR, TSPDIR, FILE)
cities     = read.coord(FILE)
cities     = as.matrix(cities)
class(cities) = "numeric"

# Define organism class
org = function(tour, fit) list(tour=tour, fit=fit)

# Define error measure and fitness
# Note: We could've just precalculated this
tour.length = function(ind){
  sum(
    sqrt((cities[ind[1:(no.cities-1)], 2] - cities[ind[2:(no.cities)], 2])**2 +
         (cities[ind[1:(no.cities-1)], 3] - cities[ind[2:(no.cities)], 3])**2)) +
    sqrt((cities[ind[no.cities], 2]       - cities[ind[1], 2])**2 +
         (cities[ind[no.cities], 3]       - cities[ind[1], 3])**2)
}

# fitness = function(ind){1 / tour.length(ind)}
fitness = function(ind){tour.length(ind)}
random  = function(){tour = sample(1:no.cities, no.cities); return(org(tour=tour, fit=fitness
    (tour)))}

# Setting
generations     = 5000000
population.size = 14
no.cities       = nrow(cities)
population      = replicate(population.size, random(), simplify = FALSE) # Initialise
    population
```

```
mutation.rate   = 1.0 / no.cities
crossover.fract = 0.2
mutation.fract  = 0.9
# selection.prob  = 0.9 # For binomial
best.fit        = min(sapply(population, function(x) x$fit))

#####################################
### Selection
#####################################
tournament.selection = function(n = population.size)
{
  # Keep the best one
  newpop      = vector(mode="list", length = n)
  best.index  = which(sapply(population, function(x) x$fit) == best.fit)[1]
  newpop[[1]] = population[[best.index]]
  population  = population[-best.index]

  if(n < 2)
    return(newpop)

  # Take random pair, pick best
  for(count in 2:n)
  {
    rands = sample(1:length(population), 2, replace = FALSE) # Pick random pair
    index = ifelse(population[[rands[1]]]$fit < population[[rands[2]]]$fit, rands[1], rands
        [2]) # Sort
    newpop[[count]] = population[[index]] # Take chosen individual and add to new population
    population      = population[-index]  # Remove from population
  }
  return(newpop)
}
#####################################
### Crossover
#####################################
pillar.crossover = function(ind1, ind2)
{
  rands   = sample(no.cities, 2)
  rands   = sort(rands)
  newind1 = vector(mode = "integer", length = no.cities)
  newind2 = vector(mode = "integer", length = no.cities)

  # Swap first chunk and add in remnants
  newind1[seq(rands[1], rands[2])] = ind1[seq(rands[1], rands[2])]
  newind2[!1:no.cities %in% seq(rands[1], rands[2])] = ind1[!1:no.cities %in% seq(rands[1],
      rands[2])]
  newind1[-seq(rands[1], rands[2])] = ind2[!ind2 %in% newind1[seq(rands[1],  rands[2])]]
  newind2[seq(rands[1],  rands[2])] = ind2[!ind2 %in% newind2[-seq(rands[1], rands[2])]]
  return(list(org(tour=newind1, fit = fitness(newind1)), org(tour=newind2, fit=fitness(
      newind2))))
}

#####################################
### Mutations
#####################################
swap.mutation = function(ind, base){
  rand = sample(1:no.cities, length(base))
  replace(ind, c(base, rand), ind[c(rand, base)])
}

mutate.randomise = function(ind){
  mutate.offers = which(runif(no.cities) < mutation.rate * 2) # Which bases are to be swapped
      ?
  if(length(mutate.offers) < 2) return(org(tour=ind$tour, fit=ind$fit))
  ind.seq = ind$tour
  ind.seq[mutate.offers] = sample(ind.seq[mutate.offers], length(mutate.offers))
  return(org(tour=ind.seq, fit=fitness(ind.seq)))
}
```

```
103
104 mutate.chunk = function(ind)
105 {
106   # Get random intervals
107   interval = sample(no.cities, 2)
108   interval = sort(interval)
109   width    = interval[2] - interval[1]
110   pos      = sample(1:(no.cities-width), 1)
111
112   # Get regions for exchange
113   first.seq       = seq(pos, pos + width)
114   second.seq      = seq(interval[1], interval[2])
115   diff            = intersect(first.seq, second.seq)
116   first.seq       = first.seq[!first.seq   %in% diff]
117   second.seq      = second.seq[!second.seq %in% diff]
118
119   # Swap chunks
120   newind             = ind$tour
121   first.batch        = newind[first.seq]
122   second.batch       = newind[second.seq]
123   newind[first.seq]  = second.batch
124   newind[second.seq] = first.batch
125
126   return(org(tour=newind,fit=fitness(newind)))
127 }
128
129 # Randomly select two indices, and reverse the sequence in between
130 mutate.reverse = function(ind){
131   newind  = ind$tour
132   randseq = sample(no.cities, 2)
133   randseq = randseq[1]:randseq[2]
134   newind[randseq] = rev(newind[randseq])
135   return(org(tour=newind,fit=fitness(newind)))
136 }
137
138 ###############################################
139 ## MAIN METHOD
140 ###############################################
141 start.time =  proc.time()
142 for(iter in 1:generations){
143 #while((proc.time() - start.time)["elapsed"] < 1800){ # Run for 30 min
144   # Mutate
145   # population = append(population, lapply(population, mutate.randomise))
146   # population = append(population, lapply(population, mutate.chunk))
147   mutation.pop = tournament.selection(ceiling(mutation.fract * population.size))
148
149   # Apply crossover
150   cross.pop     = tournament.selection(2 * trunc(crossover.fract * population.size/2.0))
151   cross.pop.ind = sample(length(cross.pop), length(cross.pop))
152   cross.pop.ind = matrix(cross.pop.ind, 2)
153
154   # Append to population
155   population    = append(population, lapply(mutation.pop, mutate.reverse))
156   population    = append(population, apply(cross.pop.ind, 2,
157                       function(x) pillar.crossover(cross.pop[[x[1]]]$tour,
158                                                    cross.pop[[x[2]]]$tour))[[1]])
159
160   # Apply selection
161   population = tournament.selection(population.size)
162
163   # Update / print best fitness
164   best.fit = min(sapply(population, function(x) x$fit))
165    if (iter %% 1000 == 0){
166      cat(iter, "\t", best.fit, "\n")
167       best = population[which(sapply(population, function(x) x$fit) == best.fit)][[1]]
168      solution = rbind(cities[best$tour, ], cities[best$tour[1], ])
169       plot(cities[best$tour, 2], cities[best$tour, 3], type='o', pch = 16)
```

```
  }
}

# Print best solution
best = population[which(sapply(population, function(x) x$fit) == best.fit)][[1]]
write(c(gsub("\\.tsp", "", FILE), best$fit, tour.length(read.opt(OPTFILE))), ncolumns=3, sep
    = "\t", file = paste0(gsub("\\.tsp", "", FILE), "_ga_solution.txt"))
```

ga.R

```
#!/usr/bin/env Rscript
require(methods)
HOMEDIR = "/home/henrik/spa3/"
TSPDIR  = "tsp_data/"
setwd(HOMEDIR)
OPTFILE = ifelse(length(args) == 0, "tsp_data/a280.opt.tour", args[1])
FILE = gsub("\\.opt\\.tour", "\\.tsp", OPTFILE)
trim = function (x) gsub("^\\s+|\\s+$", "", x) # Trim whitespaces
source("aux.R")

# Read in data
file           = paste0(HOMEDIR, TSPDIR, FILE)
cities         = read.coord(FILE)
cities         = as.matrix(cities)
class(cities) = "numeric"

# Define organism class
org = function(tour, fit) list(tour=tour, fit=fit)

# Define error measure and fitness
tour.length = function(ind){
  sum(
    sqrt((cities[ind[1:(no.cities-1)], 2] - cities[ind[2:(no.cities)], 2])**2 +
           (cities[ind[1:(no.cities-1)], 3] - cities[ind[2:(no.cities)], 3])**2)) +
    sqrt((cities[ind[no.cities], 2]       - cities[ind[1], 2])**2 +
           (cities[ind[no.cities], 3]       - cities[ind[1], 3])**2)
}
fitness = function(ind){tour.length(ind)}
random  = function(){tour = sample(1:no.cities, no.cities); return(org(tour=tour, fit=fitness
    (tour)))}

# Parameters
generations = 5e6
temp        = 1e3
temp.rate   = 1 - 1e-5
state       = random()

# Same as our good old mutate.reverse
state.neighbour = function(state){
  newtour = state$tour
  randseq = sample(no.cities, 2)
  randseq = randseq[1]:randseq[2]
  newtour[randseq] = rev(newtour[randseq])
  return(org(tour=newtour,fit=fitness(newtour)))
}

start.time =  proc.time()
for(iter in 1:generations){
  # Suggest update
  proposed.state = state.neighbour(state)

  # Is it better?
  if(proposed.state$fit <= state$fit)
    state = proposed.state
  # Should we still go there?
  else if (runif(1) < exp((state$fit - proposed.state$fit)/temp)){
    state <<- proposed.state
  }
```

```
58
59    # Change temperature
60    temp = temp * temp.rate
61
62    # Print current result
63    if (iter %% 1000 == 0)
64    {
65      cat(iter, "\t", temp, "\t", state$fit, "\n")
66      #solution = cities[state$tour, ]
67      #plot(solution[, 2], solution[, 3], type='b')
68    }
69  }
70
71  # Print bezt
72  best = population[which(sapply(population, function(x) x$fit) == best.fit)][[1]]
73  write(c(gsub("\\.tsp", "", FILE), best$fit, tour.length(read.opt(OPTFILE))), ncolumns=3, sep
        = "\t", file = paste0(gsub("\\.tsp", "", FILE), "_simann_solution.txt"))
```

simann.R

```
1  require("stats")
2  grid.width  = 5
3  grid.height = grid.width
4  initial.rad = 3
5  radii       = rep(initial.rad, grid.width * grid.height)
6  iterations  = 10000
7  data        = matrix(runif(iterations*2),  iterations, 2)
8  input.dim   = ncol(data)
9  rad.deg     = 0.9999
10 alpha       = seq(0.05, 0.01, length.out = iterations)
11 no.images   = 3
12
13 # Init
14 # weights     = expand.grid(1:grid.width, 1:grid.height) / grid.width
15 weights   = matrix(runif(grid.width**2*input.dim,min=.4,max=.6),grid.width*grid.height, input
       .dim)
16 nodes     = expand.grid(1:grid.width, 1:grid.height)
17 distances = as.matrix(stats::dist(nodes, method = "maximum")) # Distances between nodes (grid
       -space)
18
19 # Plot grid
20 plot.grid = function(main="", sub=""){
21   plot(weights,xlim=c(0,1),ylim=c(0,1),xlab="",ylab="", xaxt = "n",yaxt="n", bty="n", pch =
         16)
22   for (ii in 1:grid.width)
23     for (jj in 1:grid.height) {
24       if (ii < grid.width)
25         lines(rbind(weights[which(nodes[, 1] == ii & nodes[, 2] == jj), ], weights[which(
               nodes[, 1] == (ii + 1) & nodes[, 2] == jj),]))
26       if (jj < grid.height)
27         lines(rbind(weights[which(nodes[, 1] == ii & nodes[, 2] == jj), ], weights[which(
               nodes[, 1] == ii & nodes[, 2] == (jj + 1)),]))
28     }
29   title(main=main, line=-1)
30 }
31
32 # GO!
33 plot.grid(main=paste0("Iteration: ",0))
34 for (iter in 1:iterations){
35   if(iter %% trunc(iterations / no.images) == 0) plot.grid(main=paste0("Iteration: ",iter))
36
37   # Get input-weights distance
38   dists     = apply(weights, 1, function(x) sum((data[iter,] - x)**2))
39   minimum   = which(dists == min(dists))[1]
40
41   # Find the neighbours and update them accordingly
42   neighbours = which(distances[minimum,] < radii[minimum])
```

```
  datum       = matrix(rep(data[iter, ],length(neighbours)),ncol=ncol(weights),byrow=TRUE)#rep
      (train.data[iter, ], length(neighbours)), ncol=ncol(weights), byrow=TRUE)
  weights[neighbours, ] = weights[neighbours, ] + alpha[iter] * (datum - weights[neighbours,
      ])

  # Update radius for the winning node
  radii[minimum] = rad.deg * radii[minimum]

  # Print progress
  if(iter %% 1000 == 0)
    print(iter)
}
```

koh.R

```
#!/usr/bin/env Rscript
setwd("~/spa3")
library('pixmap')
library('webp')
library('grid')
library('gridExtra')

# Take in data
lena.file      = "lena512color.tiff"
fabio.file     = "fab_fabio.webp"
lena.img.name = strsplit(lena.file, "\\.")[[1]]
lena.img.name = ifelse(length(lena.img.name) > 1, lena.img.name[-length(lena.img.name)], lena
    .img.name)
lena.img.name = paste0(lena.img.name, ".ppm")
aesthet        = system(paste0("convert ", lena.file, " ", lena.img.name))
lena.image     = read.pnm(lena.img.name, cellres=1)
fab.file       = read_webp("fab_fabio.webp")

# Retrieve matrix specific data
lena.red.matrix    = matrix(lena.image@red,   nrow = ncol(lena.image@red),
                            ncol = nrow(lena.image@red))
lena.green.matrix  = matrix(lena.image@green, nrow = ncol(lena.image@red),
                            ncol = nrow(lena.image@red))
lena.blue.matrix   = matrix(lena.image@blue,  nrow = ncol(lena.image@red),
                            ncol = nrow(lena.image@red))
fabio.red.matrix   = matrix(fab.file[,,1], nrow = ncol(fab.file[,,1]),
                            ncol = nrow(fab.file[,,1]))
fabio.green.matrix = matrix(fab.file[,,2], nrow = ncol(fab.file[,,2]),
                            ncol = nrow(fab.file[,,1]))
fabio.blue.matrix  = matrix(fab.file[,,3], nrow = ncol(fab.file[,,3]),
                            ncol = nrow(fab.file[,,1]))

# Define parameters
iterations  = 10000
grid.width  = 10
grid.height = 10
rad.start   = 2
rad.deg     = 0.9
block.side  = 4
no.nodes    = grid.width*grid.height
nodes       = expand.grid(1:grid.width, 1:grid.height)
nodes       = nodes[, c(2,1)] # Loop-intuitive order

# Define node distances
nodes.dist = function(x) as.matrix(stats::dist(x, method = "euclidian"))
distances  = nodes.dist(nodes)

# Retrieve Kohonen interpreted matrix
get.matrix = function(mat, method = "uniform"){
  cat("\nInitialising new Kohonen compression\n")
  # Initialise
  alpha   = seq(0.2, 0.01, length.out = iterations)
  radii   = rep(rad.start, grid.width * grid.height)
```

```
  weights = matrix(runif(grid.height*grid.width*block.side**2, max = .1),
                   grid.width * grid.height, block.side**2)

  # Have a guess
  get.random.block = function(){
    ii = sample(ncol(mat) / block.size, 1)
    jj = sample(nrow(mat) / block.size, 1)
    x.ind = ((ii-1)*block.side + 1):((ii)*block.side)
    y.ind = ((jj-1)*block.side + 1):((jj)*block.side)
    as.vector(mat[x.ind, y.ind])
  }

  # Create all the input arrays
  data = list()
  for(ii in 1:(ncol(mat)/block.side)) {
    for(jj in 1:(nrow(mat)/block.side)) {
      x.ind = ((ii-1)*block.side + 1):((ii)*block.side)
      y.ind = ((jj-1)*block.side + 1):((jj)*block.side)
      if(method == "uniform")
        data = append(data, list(as.vector(mat[x.ind, y.ind])))
      else if (method == "random")
        data = append(data, list(get.random.block()))
    }
  }

  # GO!
  for (iter in 1:iterations){
    # Get input-weights distance
    dists     = apply(weights, 1, function(x) sum((data[[((iter - 1) %% (length(data)))+1]] -
        x)**2))
    minimum   = which(dists == min(dists))[1]

    # Find the neighbours and update them accordingly
    neighbours = which(distances[minimum,] < radii[minimum])
    datum      = matrix(rep(data[[((iter - 1) %% (length(data)))+1]],
                            length(neighbours)), ncol=ncol(weights), byrow=TRUE)
    weights[neighbours, ] = weights[neighbours, ] + alpha[iter] * (datum - weights[neighbours
        , ])

    # Update radius for the winning node
    radii[minimum] = rad.deg * radii[minimum]

    # Print progress
    if(iter %% 1000 == 0)
      print(iter)
  }

  # Convert back to complete matrix
  # Note: In practice we store the (minimum) indices and the net.
  new.image = matrix(0, ncol(mat), nrow(mat))
  sapply(1:(ncol(mat) / block.side),
    function(ii) sapply(1:(nrow(mat) / block.side),
      function(jj) {
        x.ind = ((ii - 1) * block.side + 1):((ii) * block.side)
        y.ind = ((jj - 1) * block.side + 1):((jj) * block.side)
        input = as.vector(mat[x.ind, y.ind])
        dists = apply(weights, 1, function(x) sum((input - x) ** 2))
        minimum = which(dists == min(dists))[1]
        new.image[x.ind, y.ind] <<- weights[minimum, ]
      }))
  return(new.image)
}

# Take out the images
lena.image.red   = get.matrix(lena.red.matrix,   method = "uniform")
lena.image.blue  = get.matrix(lena.blue.matrix,  method = "uniform")
lena.image.green = get.matrix(lena.green.matrix, method = "uniform")
```

```
fabio.image.red   = get.matrix(fabio.red.matrix,   method = "uniform")
fabio.image.blue  = get.matrix(fabio.blue.matrix,  method = "uniform")
fabio.image.green = get.matrix(fabio.green.matrix, method = "uniform")

# Plot images
par(mfrow=c(2,3), mar = c(1,1,1,1))
image(t(apply(lena.red.matrix,2,rev)),xaxt="n",yaxt="n",main="Red")
image(t(apply(lena.blue.matrix,2,rev)),xaxt="n",yaxt="n",main="Green")
image(t(apply(lena.green.matrix,2,rev)),xaxt="n",yaxt="n",main="Blue")
image(t(apply(fabio.red.matrix,2,rev)),xaxt="n",yaxt="n")
image(t(apply(fabio.blue.matrix,2,rev)),xaxt="n",yaxt="n")
image(t(apply(fabio.green.matrix,2,rev)),xaxt="n",yaxt="n")
image(t(apply(lena.image.red,2,rev)),xaxt="n",yaxt="n",main="Red")
image(t(apply(lena.image.blue,2,rev)),xaxt="n",yaxt="n",main="Green")
image(t(apply(lena.image.green,2,rev)),xaxt="n",yaxt="n",main="Blue")
image(t(apply(fabio.image.red,2,rev)),xaxt="n",yaxt="n")
image(t(apply(fabio.image.blue,2,rev)),xaxt="n",yaxt="n")
image(t(apply(fabio.image.green,2,rev)),xaxt="n",yaxt="n")

# Merge channels
lena.all.three       = rgb(lena.image.red,  lena.image.green,  lena.image.blue)
fabio.all.three      = rgb(fabio.image.red, fabio.image.green, fabio.image.blue)
dim(lena.all.three)  = dim(lena.image.red)
dim(fabio.all.three) = dim(fabio.image.red)

# Prepare to plot
grob.size  = .95
ll         = rgb(lena.image@red, lena.image@green, lena.image@blue)
rr         = rgb(fab.file[,,1], fab.file[,,2], fab.file[,,3])
dim(ll)    = dim(lena.image.red)
dim(rr)    = dim(fabio.image.red)
l.grob     = rasterGrob(ll, interpolate = FALSE, width = grob.size)
f.grob     = rasterGrob(rr, interpolate = FALSE, width = grob.size)
lena.grob  = rasterGrob(lena.all.three,  interpolate=FALSE, width=grob.size)
fabio.grob = rasterGrob(fabio.all.three, interpolate=FALSE, width=grob.size)

###################################
# Compare to SVD
###################################
do.svd = function(mat, value){
  mat.svd = svd(mat)
  d = mat.svd$d; u = mat.svd$u; v = mat.svd$v
  mat.reconstruction = u #latex# diag(d) #latex# t(v)
  mat.compressed = u[,1:value] #latex# diag(d[1:value]) #latex# t(v[,1:value])
}

# Do SVD stuff, account for overshooting
lena.svd.red    = do.svd(lena.red.matrix,    30)
lena.svd.green  = do.svd(lena.green.matrix,  30)
lena.svd.blue   = do.svd(lena.blue.matrix,   30)
fabio.svd.red   = do.svd(fabio.red.matrix,   30)
fabio.svd.green = do.svd(fabio.green.matrix, 30)
fabio.svd.blue  = do.svd(fabio.blue.matrix,  30)
lena.svd.red[which(lena.svd.red       < 0)] = 0
lena.svd.green[which(lena.svd.green   < 0)] = 0
lena.svd.blue[which(lena.svd.blue     < 0)] = 0
lena.svd.red[which(lena.svd.red       > 1)] = 1
lena.svd.green[which(lena.svd.green   > 1)] = 1
lena.svd.blue[which(lena.svd.blue     > 1)] = 1
fabio.svd.red[which(fabio.svd.red     < 0)] = 0
fabio.svd.green[which(fabio.svd.green < 0)] = 0
fabio.svd.blue[which(fabio.svd.blue   < 0)] = 0
fabio.svd.red[which(fabio.svd.red     > 1)] = 1
fabio.svd.green[which(fabio.svd.green > 1)] = 1
fabio.svd.blue[which(fabio.svd.blue   > 1)] = 1
lena.all.three.svd       = rgb(lena.svd.red,  lena.svd.green,  lena.svd.blue)
fabio.all.three.svd      = rgb(fabio.svd.red, fabio.svd.green, fabio.svd.blue)
```

```r
dim(lena.all.three.svd)  = dim(lena.svd.red)
dim(fabio.all.three.svd) = dim(fabio.svd.red)

# Plot stuff
par(mfrow=c(1,2)); grid.newpage();
lena.svd.grob  = rasterGrob(lena.all.three.svd,  interpolate=FALSE, width=grob.size)
fabio.svd.grob = rasterGrob(fabio.all.three.svd, interpolate=FALSE, width=grob.size)
grid.arrange(grobs=list(l.grob, lena.grob, lena.svd.grob, f.grob,
                        fabio.grob,  fabio.svd.grob), ncol = 3, nrow=2)
```

comp.R