Stochastic Models and Optimization: Problem Set 1

Roger Garriga Calleja, José Fernando Moreno Gutiérrez, David Rosenfeld, Katrina Walker

March 3, 2017

Problem 5. TSP Computational Assignment:
Visit the website: http://www.math.uwaterloo.ca/tsp/world/countries.html. Solve the Traveling Salesman Problem for Uruguay based on the dataset provided. You can use your favorite programming language and solution method for the TSP. Provide a printout of your code with detailed documentation, and compare the optimal solution you obtain to the one available at the website.

The code has been done in R. We used 3 heuristic approaches to find approximate the problem: The nearest neighbor, the greedy algorithm and the simulation anneling. We can see that the best approach (anneling) is above the optimal solution by 12%, however comparing to the second best it just 1% below. Furthermore, this 1% represented an important loose in terms of efficiency. In the following table you can see some important results:

	optimal	nearest neighbor	greedy	anneling
distance	79114.00	100056.45	89559.29	88985.51
distance/optimal		1.26	1.13	1.12
run time (min)		0.19	2.55	11.69

```
library(fields)
library(dplyr)

# Read data and estimate distances between cities
data_uy734       ← read.csv("/home/chpmoreno/Dropbox/Documents/BGSE/Second_Term/SMO/Problemsets/PS2/uy734.csv")[, -1]
cities_distances ← rdist(data_uy734) # euclidean distance estimation

# ///////////////////////////////
# nearest Neighbor approach ####
# ///////////////////////////////
city_path_nearest_neighbor ← function(cities_distances, city = round(runif(1, 1, nrow(cities_distances)))) {
  # Create an auxiliar distance matrix for eliminating selected cities
  cities_distances_aux ← cities_distances
  # Impose big distances for 0 diagonal values of distance matrix. If we do not do this the diagonal will be
  # the minimum distance for each city.
  cities_distances_aux[cities_distances_aux == 0] ← 1000000000
  n_cities ← nrow(cities_distances_aux) # number of cities

  city_path ← city # initial city (by default usually random)

  # nearest neighbor O(n^2) algorithm:
```

```
  # 1. Select a random city.
  # 2. Find the nearest unvisited city and go there.
  # 3. Are there any unvisitied cities left? If yes, repeat step 2.
  # 4. Return to the first city.
  i = 1
  while(length(city_path) < (n_cities + 1)) {
    current_city_distances  ← cities_distances_aux[, city_path[i]] # current city
    nearest_city_to_current ← which.min(current_city_distances) # find the minimum available distance
    city_path ← c(city_path, nearest_city_to_current) # add the nearest city to the path
    cities_distances_aux[city_path, city_path[i + 1]] ← 1000000000 # eliminate the new current city distance
    i = i + 1
  }
  city_path ← c(city_path, city_path[1]) # return to the first city

  # Calculate the total distance of the path
  total_distance ← 0
  for(i in 1:(length(city_path) - 1)){
    total_distance ← total_distance + cities_distances[city_path[i], city_path[i + 1]]
  }

  # return the path and its distance
  return(list(path = city_path, distance = total_distance))
}

# Compute the best nearest Neighbor path from all the cities as initial ones
best_path_nearest_neighbor ← function(cities_distances) {
  nearest_neighbor_paths ← NULL
  nearest_neighbor_distances ← NULL
  for(i in 1:nrow(cities_distances)) {
    estimator_aux ← city_path_nearest_neighbor(cities_distances, i)
    nearest_neighbor_paths     ← cbind(nearest_neighbor_paths, estimator_aux$path)
    nearest_neighbor_distances ← c(nearest_neighbor_distances, estimator_aux$distance)
  }

  return(list(best_path = nearest_neighbor_paths[, which.min(nearest_neighbor_distances)],
              distance = min(nearest_neighbor_distances)))
}

# ////////////////////////////////
# Greedy Algorithm approach ####
# ////////////////////////////////
city_path_greedy ← function(cities_distances) {
  n_cities ← nrow(cities_distances)
  # Take all the edges and weights from distance matrix
  edges_and_weights_matrix ← NULL
  for(i in 1:n_cities) {
```

```
      city_distance_vector      <- cities_distances[i:n_cities ,i][-1]
      if(length(city_distance_vector) > 0)
        edges_and_weights_matrix <- rbind(edges_and_weights_matrix, cbind(rep(i, length(city_distance_vector)),
                                                                           seq(i+1, n_cities),
                                                                           city_distance_vector))
    }
    # Order the edges by weights
    edges_and_weights_df          <- as.data.frame(edges_and_weights_matrix)
    edges_and_weights_ordered_df <- arrange(edges_and_weights_df, city_distance_vector)

    # greedy O(n2log_2(n)) algorithm:
    # Constrains: gradually constructs the by
    # repeatedly selecting the shortest edge and adding it to
    # the path as long as it does not create a cycle with less
    # than N edges, or increases the degree of any node to
    # more than 2. We must not add the same edge twice. Then:
    # 1. Sort all edges.
    # 2. Select the shortest edge and add it to our
    # path if it does not violate any of the constraints.
    # 3. Do we have N edges in our tour? If no, repeat
    # step 2.
    city_path <- edges_and_weights_ordered_df[1, 1:2]
    total_distance <- 0
    for(i in 2:nrow(edges_and_weights_ordered_df)) {
      # Constrains
      if((sum(city_path ==  edges_and_weights_ordered_df[i, 1]) < 2 &
           sum(city_path ==  edges_and_weights_ordered_df[i, 2]) < 2) &
          sum((city_path[edges_and_weights_ordered_df[i, 1] == city_path[, 1], 2] ==
               city_path[edges_and_weights_ordered_df[i, 2] == city_path[, 2], 1])) == 0) {
        # path fill
        city_path <- rbind(city_path, edges_and_weights_ordered_df[i, 1:2])
        # compute the distance
        total_distance <- total_distance +  edges_and_weights_ordered_df[i, 3]
      }
    }
    return(list(best_path = city_path, distance = total_distance))
}

# ////////////////////////////////
# Simulated annealing approach ####
# ////////////////////////////////

# This approach is based on Todd W. Schneider code and his blog post, availables
```

```
on:
# * http://toddwschneider.com/posts/traveling-salesman-with-simulated-annealing-r-and-shiny/
# * https://github.com/toddwschneider/shiny-salesman

# Calculate the path distance
calculate_path_distance = function(path, distance_matrix) {
  sum(distance_matrix[embed(c(path, path[1]), 2)])
}

# Compute the current temperature
current_temperature = function(iter, s_curve_amplitude, s_curve_center, s_curve_width) {
  s_curve_amplitude * s_curve(iter, s_curve_center, s_curve_width)
}

s_curve = function(x, center, width) {
  1 / (1 + exp((x - center) / width))
}

# simulation anneling O() algorithm:
# 1. Start with a random path through the selected cities.
# 2. Pick a new candidate path at random from all neighbors of the existing path.
# This candidate path might be better or worse compared to the existing one.
# 3. If the candidate path is better than the existing path, accept it as the new path. If the candidate
# path is worse than the existing tour, still maybe accept it, according to some probability. The probability
# of accepting an inferior tour is a function of how much longer the candidate is compared to the current tour,
# and the temperature of the annealing process. A higher temperature makes you more likely to accept an inferior
# path.
# 4. Go back to step 2 and repeat as many times as you want or can.
city_path_annealing_process = function(distance_matrix, path, path_distance, best_path = c(), best_distance = Inf,
                                       starting_iteration = 0, number_of_iterations = 10000000,
                                       s_curve_amplitude = 400000, s_curve_center = 0, s_curve_width = 300000) {

  n_cities = nrow(distance_matrix) # number of cities

  for(i in 1:number_of_iterations) {
    iter = starting_iteration + i
    # computation of temperature
    temp = current_temperature(iter, s_curve_amplitude, s_curve_center, s_curve_width)

    candidate_path = path # initial path
    swap = sample(n_cities, 2) # new path
    candidate_path[swap[1]:swap[2]] = rev(candidate_path[swap[1]:swap[2]])
    candidate_dist = calculate_path_distance(candidate_path, distance_matrix) #
```

```
compute the distance for new path

    # ratio indicator
    if (temp > 0) {
      ratio = exp((path_distance - candidate_dist) / temp)
    } else {
      ratio = as.numeric(candidate_dist < path_distance)
    }
    # probabilistic decision
    if (runif(1) < ratio) {
      path = candidate_path
      path_distance = candidate_dist
      # best path and best distance
      if (path_distance < best_distance) {
        best_path = path
        best_distance = path_distance
      }
    }
  }
  return(list(path=path, path_distance=path_distance,
              best_path=best_path, distance=best_distance))
}

# ///////////////////////
# Code execution #######
# //////////////////////
# Optimal solution given by http://www.math.uwaterloo.ca/tsp/world/uytour.html
optimal = 79114
# nearest Neighbor
nearest_neighbor_time ← Sys.time()
nearest_neighbor_distance ← best_path_nearest_neighbor(cities_distances)$distance
nearest_neighbor_time ← Sys.time() - nearest_neighbor_time
# Greedy
greedy_time ← Sys.time()
greedy_distance ← city_path_greedy(cities_distances)$distance
greedy_time ← Sys.time() - greedy_time
# Anneling
distance_matrix = cities_distances
path = sample(nrow(distance_matrix))
path_distance = calculate_path_distance(path, distance_matrix)
anneling_time ← Sys.time()
anneling_distance ← city_path_annealing_process(distance_matrix = distance_matrix,
                                                 path = path,
                                                 path_distance = path_distance)$distance
anneling_time ← Sys.time() - anneling_time
# Comparison table
comparison_table ← rbind(c(optimal, nearest_neighbor_distance, greedy_distance, anneling_distance),
                          c(NA, nearest_neighbor_distance / optimal, greedy_distance / optimal,
                            anneling_distance / optimal),
```

```
                               c(NA, nearest_neighbor_time / 60, greedy_time, anneling_time))
comparison_table ← round(as.data.frame(comparison_table), 2)
colnames(comparison_table) ← c("optimal", "nearest_neighbor", "greedy", "anneling")
rownames(comparison_table) ← c("distance", "distance/optimal", "run time (min)")
```