Laboratory of Population-Based Optimisation Methods

Adriano De Marino
a.demarino@campus.unimib.it

Department of Informatics, Systems and Communication
University of Milano-Bicocca, Milano, Italy

1 Problem

Question 1

Build a system which simulate a drone aimed to clean an entire discrete space at minimum time cost.

This is the same as

Question 2

Build a system which solves the Traveling Salesman Problem (TSP) without loops.

Since questions 1 and 2 are equivalent, we answer to question 2.

2 Method

We solve the aforementioned problem using a Genetic Algorithm (GA). In the context of our (modified) TSP problem we can define the following
Elements:

- **gene**: a cell (represented as (x, y) coordinates) of the discrete space
- **individual** (chromosome): a single route which visits each cell exactly one time.
- **population**: a collection of possible routes (i.e., collection of individuals)

Moves:

- **initialize()**: Randomly select the initial population whose elements range in the feasible space
- **fitness()**: Compute the fitness function for each element. In our case the fitness function of a path is the cost of that path.
- **select()**: Select a subset of the current population according to the fitness values. In our case we select the top n-elements having the lowest fitness function where n is a tuning parameter.
- **generate()**: Generate a new population of solutions from those selected through a genetic operator (e.g. crossover). We use the cyclic crossover operator explained in Section 2.1
- **mutate()**: Mutate the current offspring using a genetic operator. In our case, with (user) specified probability, two genes (cells) will swap places in an individual (route).

- **ga(pop, g, n)**: Given an initial population pop, $ga()$ loops the **fitness()**, **select()**, **generate()** and **mutate()** moves until the maximum number of generations g is reached. Then $ga()$ returns the best n individuals (routes) of the last offspring having the minimum fitness() value.

The proposed GA algorithm iteratively apply $ga()$ function until the optimum is reached or the maximum number of iterations is reached. We interrupt the iteration if the improvement in the cost is lower than a pre-defined ε. The corresponding pseudocode is provided in Algorithm 1. Since the distance between two cells is defined as the corresponding euclidean distance, the optimal cost for a grid of size N (namely with N cells) is equal to $N - 1$. We implement the proposed algorithm in R (see Section 4).

Algorithm 1: `GA algorithm`

Input:
G: a discrete grid of N cells
D: the pairwise distances between cells
max_{IT}: maximum n. of iterations
Result: a (sub-)optimal route r

$offspring \leftarrow initialize()$;
while $n_{IT} < max_{IT}$ & *cost > N - 1* & $eps > \varepsilon$ **do**
  $offspring \leftarrow GA(offspring)$;
  $res \leftarrow minCostRoute(offspring)$;
  $oldCost \leftarrow cost$;
  $cost \leftarrow fitness(res)$;
  $eps \leftarrow |cost - oldCost|$;
  $n_{IT} < n_{IT} + 1$;
end
return res ;

2.1 Crossover

We explain cyclic crossover with the following example. Let P1 and P2 be two parents
P1 : 2 8 0 1 3 4 5 7 9 6
P2 : 1 0 5 4 6 8 9 7 2 3
Select the first city of P1 make it as the first city of offspring1(01)
01: 2 - - - - - - - - -
To find the next city of offspring 01 search current city, which is selected from P1 in P2. Find the location of city in P2 and select the city which is in the same location in P1.
01: 2 - - - - - - - 9 -
Continue the same procedure, we will get 01 as
01: 2 8 0 1 - 4 5 - 9 -
In the next step we will get the city 2 which is already present in O1 and then stop the procedure. Copy the cities from parent P2 in the corresponding locations
01: 2 8 0 1 6 4 5 7 9 3
For the generation offspring 02 the initial selection is from the parent P2, and repeat the procedure with P1. Therefore we obtain
02: 1 0 5 4 3 8 9 7 2 6
If the initial population contain N parents it will generate N(N-1) offsprings.

3 Results

As shown in Figure 1, after $n_{IT} = 1$ iterations we reached an optimum path for both the 3×3 cells and 3×4 cells cases. Furthermore, after $n_{IT} = 4$ iterations we reached an optimum path for the 4×4 cells case. After $n_{IT} = 6$ iterations we reached a sub-optimal path for the 4×5 cells case with cost equal to 20.65028 (the optimal one has a cost equal to 19). After $n_{IT} = 9$ iterations we reached a sub-optimal path for the 5×5 cells case with cost equal to 28.26241 (the optimal one has a cost equal to 24). After $n_{IT} = 10$

iterations we reached a sub-optimal path for the 5×6 cells case with cost equal to 36.36506 (the optimal one has a cost equal to 29). Finally, after $n_{IT} = 12$ iterations we reached a sub-optimal path for the 6×6 cells case with cost equal to 44.77927 (the optimal one has a cost equal to 36). In each case we use a number $n_{parents}$ of parents equal to 4 for both the initial population and each output from $ga()$.

Not surprisingly, both the distance from the optimum and the number of iteration needed to obtain a solution increase with the dimension of the grid. We also set the number of maximum iterations max_{It} to 30 and the tolerance ε to 0.01.

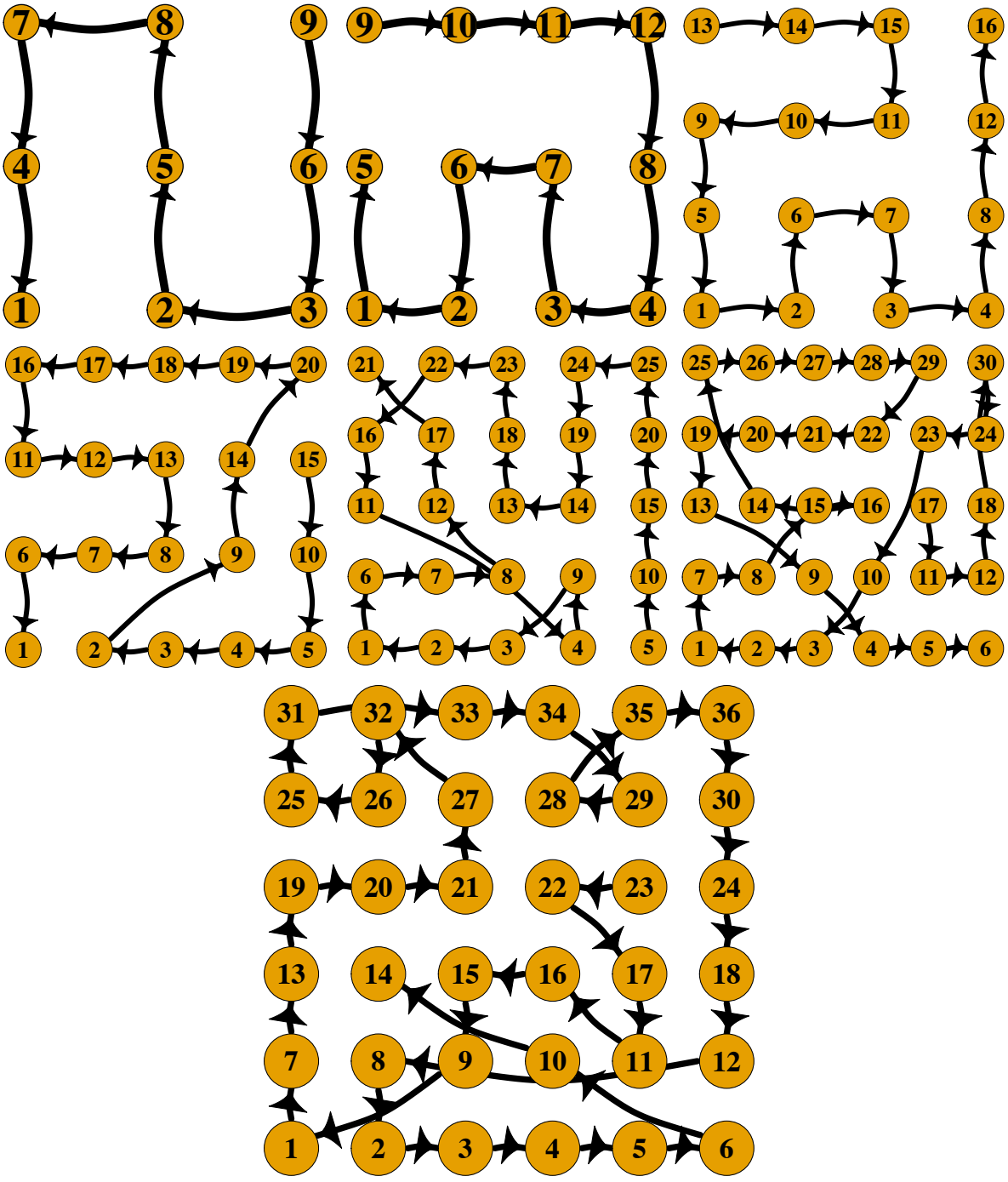

Figure 1: Final route obtained for each instance. Each node is a cell and each (directed) edge is a route from one cell to another. The position of each cell depends on its euclidean (x, y) coordinates.

4 Implementation

[main.R]

```
### define the grid
nRows <- 6
nCols <- 6
nCells <- nRows*nCols
cellMapping <- matrix(data = NA, nrow = nCells, ncol = 2)
colnames(cellMapping) <- c("x","y")
cellMapping[,1] <- rep(1:(nRows), each = nCols)
cellMapping[,2] <- rep(1:(nCols), times = nRows)
rownames(cellMapping) <- 1:(nCells)

### define the corresponding costs
TSP <- matrix(data = NA, nrow = nCells, ncol = nCells)
rownames(TSP) <- 1:nCells
colnames(TSP) <- 1:nCells
for (rr in 1:nrow(TSP)) {
  for (cc in 1:ncol(TSP)) {
    TSP[rr,cc] <- norm2(cellMapping[rr,],cellMapping[cc,])
  }
}
diag(TSP) <- 9999 #+Inf

minCost <- + Inf
maxIt <- 30
nParents <- 4
nGen <- 4

### initialization of a population
nTot <- 10^4
offspring <- matrix(data = NA, nrow = nTot, ncol = nCells)
for (nPar in 1:nTot) {
  offspring[nPar,] <- sample(x = 1:nCells, nCells, replace = FALSE)
}
nPropSel <- nParents
offspring <- propSelection(population = offspring, n = nPropSel)
# apply(offspring, 1, computeRouteCost)

### GA algorithm
nIt <- 0
EPS <- .01
eps <- +Inf
oldCost <- minCost
while (nIt < maxIt & eps > EPS & oldCost > (nCells - 1)) {
  offspring <- ga(nGen, nParents, offspring, +Inf, .5, .1)
  currCost <- computeRouteCost(offspring[1,])
  eps <- abs(oldCost - currCost)
  oldCost <- currCost
  message("\t", currCost)
  nIt <- nIt + 1
}
res <- offspring[1,]
minCost <- computeRouteCost(res)

minCost - (nCells - 1) ## distance from the optimum
```

[functions.R]

```
ga <- function(nGen, nParents, offspring, minCost, selFrac, mutRate){

  gen <- 1

  while (gen <= nGen) { ## loop over generations

    message(''gen: '', gen, ''\tminCost ='', minCost)
    pa <- offspring
    offspring <- cyclicCrossOver(pa) ## compute new offspring

    dupOff <- offspring[duplicated(apply(offspring, 1, computeRouteCost)),]
    if(nrow(matrix(dupOff, ncol = nCells)) > 0){
      ## apply mutation
      offspring[duplicated(apply(offspring, 1, computeRouteCost)),]
        <- t(apply(matrix(dupOff, ncol = nCells), 1,
                   FUN = function(x){mutation(x, mutationRate = mutRate)}))
    }

    paAndOff <- rbind(pa, offspring)
    costs <- apply(paAndOff, 1, computeRouteCost)

    currMinCost <- min(costs)
    if(currMinCost <= minCost){
      res <- paAndOff[which(costs == currMinCost)[1],]
      minCost <-  currMinCost
    }

    nPropSel <- round(nrow(offspring)*selFrac)
    ## apply selection
    offspring <- propSelection(population = offspring, n = nPropSel)

    gen <- gen + 1

  }
  ## return last offspring (only best nParents elements)
  return(propSelection(population = offspring, n = nParents))
 }


norm2 <- function(x,y){
  return(sqrt(sum((x - y)^2))) ## euclidean norm
}

propSelection <- function(population, n){
  fitness <- apply(population, 1, computeRouteCost)
  names(fitness) <- 1:length(fitness)
  subPop <-
        population[head(as.numeric(names(sort(fitness, decreasing = FALSE))), n), ]

  return(subPop)
}
```

```
mutation <- function(x, mutationRate){
  nCells <- length(x)

  for (i in 1:nCells) {
    if(runif(1,0,1) < mutationRate){
      j <- sample(x = setdiff(1:nCells, i), size = 1, replace = FALSE)

      temp <- x
      temp[i] <- temp[j]
      temp[j] <- x[i]
      x <- temp
    }
  }
  return(x)
}

cyclicCrossOver <- function(pa){
  nParents <- nrow(pa)
  nCells <- ncol(pa)
  offspring <- matrix(data = NA, nrow = nParents*(nParents - 1), ncol = nCells)

  nOff <- 1
  for (i in 1:nParents) {
    for (j in i:nParents) {
      if(j != i){
        pi <- pa[i,]
        pj <- pa[j,]
        offspring[nOff,] <- crossOver(A = pi, B = pj)
        offspring[nOff + nParents*(nParents - 1)/2,] <- crossOver(A = pj, B = pi)

        nOff <- nOff + 1
      }
    }
  }

  return(offspring)
}

crossOver <- function(A,B){
  os <- rep(NA, length(A))
  terminated <- FALSE

  currCol <- 1
  os[1] <- curr <- A[currCol]

  terminated <- A[which(B == curr)] == os[1]
  while (!terminated) {
    curr <- os[which(B == curr)] <- A[which(B == curr)]
    terminated <- A[which(B == curr)] == os[1]
  }
  if(sum(is.na(os)) > 0){
    os[is.na(os)] <- B[is.na(os)]
  }

  return(os)
}
```

```
computeRouteCost <- function(route){
  nCells <- length(route)

  cost <- 0
  for(z in 1:nCells){
    if(z < nCells){
      row1 <- route[z]
      col1 <- route[z+1]
      cost <- cost + TSP[row1,col1]
    }
  }

  return(cost)
}
```

[plots.R]

```
library(igraph)
TSP_res <- TSP
TSP_res[,] <- NA

for(z in 1:nCells){
  if(z < nCells){
    row1 <- res[z]
    col1 <- res[z+1]
    TSP_res[row1,col1] <- TSP[row1,col1]
  }
}

g <- graph_from_adjacency_matrix(adjmatrix = TSP_res, mode = "directed")
g <- simplify(g, remove.loops = TRUE)
g <- set_edge_attr(g, "curved", value = .2)
plot(g, layout = layout_on_grid, cex = 0, curved = TRUE,
     edge.width = 7, vertex.size = 25, edge.arrow.width = 3,
     edge.arrow.size = 1, edge.color = "black",
     vertex.label.font = 2, vertex.label.cex = 2,
     vertex.label.color = "black")
```