Week 9 Homework

Paul Isaacson

07/18/2017

We'll define several functions for you and then you'll use the optimization tools introduced in the presentation to try to find (approximately) optimal solutions to the problems detailed below. You should include your solutions right in this document and "knit" it into a Word document (just like you did for DS705 assignments). When you're done, submit both the Word document and this .rmd file to the D2L dropbox.

## Problem 1 - Redistricting with a Genetic Algorithm

Solve problem 13.10-6. Install the 'gramEvol' package to get access to a genetic algorithm that uses integer encoding called GeneticAlg.int(). Use that algorithm to solve this problem. You'll have to read the documentation to figure out how to use the algorithm.

In the block below add your code to use the genetic algorithm. Either experiment with different random number seeds or use a for loop to conduct the optimization many times to find the best solution you can.

result = GeneticAlg.int(genomeLen = 18, codonMin = 1, codonMax = 10, evalFunc = demrepFit, allowrepeat = TRUE, iterations = 1000)  
(distAssign(result$best$genome))

## Dem Rep Tot Win  
## 1 0 0 0 FALSE  
## 2 0 0 0 FALSE  
## 3 883 922 1805 TRUE  
## 4 0 0 0 FALSE  
## 5 0 0 0 FALSE  
## 6 0 0 0 FALSE  
## 7 0 0 0 FALSE  
## 8 436 399 835 FALSE  
## 9 0 0 0 FALSE  
## 10 0 0 0 FALSE

Make sure you print out your best solution. How many districts do Republicans win with your solution? (I don't think Republicans can win all 10 in this example, but they can get close.)

Republicans won 8 districts with my solution.

## Problem 2 - TSP with a Genetic Algorithm

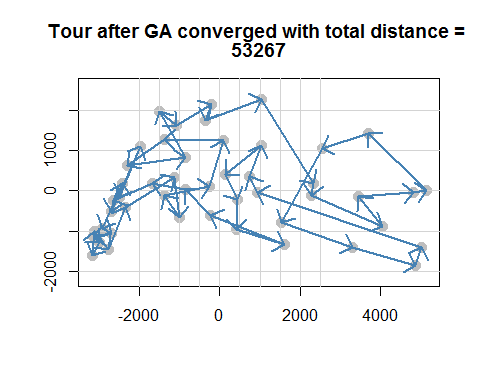
Use the ga() function with permutation encoding from the 'GA' package to approximate a solution to this 48 city TSP problem. Try different random number seeds and report the best result you can find. Copy the code from saTSP.R to make a graph of the tour. Your fitness function is tspFitness().

## +-----------------------------------+  
## | Genetic Algorithm |  
## +-----------------------------------+  
##   
## GA settings:   
## Type = permutation   
## Population size = 50   
## Number of generations = 100   
## Elitism = 2   
## Crossover probability = 0.8   
## Mutation probability = 0.1   
##   
## GA results:   
## Iterations = 100   
## Fitness function value = 1.877335e-05   
## Solution =   
## x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19  
## [1,] 40 8 1 16 22 41 5 48 10 42 26 4 2 29 39 24 45 35 25  
## x20 ... x48  
## [1,] 14 31

## +-----------------------------------+  
## | Genetic Algorithm |  
## +-----------------------------------+  
##   
## GA settings:   
## Type = permutation   
## Population size = 50   
## Number of generations = 100   
## Elitism = 2   
## Crossover probability = 0.8   
## Mutation probability = 0.1   
##   
## GA results:   
## Iterations = 100   
## Fitness function value = 1.817125e-05   
## Solution =   
## x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19  
## [1,] 46 33 12 25 13 3 16 40 31 9 15 47 39 2 26 4 35 45 10  
## x20 ... x48  
## [1,] 48 20

## +-----------------------------------+  
## | Genetic Algorithm |  
## +-----------------------------------+  
##   
## GA settings:   
## Type = permutation   
## Population size = 50   
## Number of generations = 100   
## Elitism = 2   
## Crossover probability = 0.8   
## Mutation probability = 0.1   
##   
## GA results:   
## Iterations = 100   
## Fitness function value = 1.98519e-05   
## Solutions =   
## x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19  
## [1,] 9 33 11 47 20 15 40 12 14 4 26 35 45 24 42 29 2 5 41  
## [2,] 36 7 28 43 37 27 17 19 6 30 18 46 1 8 16 22 3 23 21  
## [3,] 28 43 37 27 17 19 6 30 18 46 1 8 16 22 3 23 21 13 10  
## [4,] 38 31 44 36 7 28 43 37 27 17 19 6 30 18 46 1 8 16 22  
## x20 ... x48  
## [1,] 34 38  
## [2,] 13 44  
## [3,] 48 7  
## [4,] 3 9

## +-----------------------------------+  
## | Genetic Algorithm |  
## +-----------------------------------+  
##   
## GA settings:   
## Type = permutation   
## Population size = 50   
## Number of generations = 100   
## Elitism = 2   
## Crossover probability = 0.8   
## Mutation probability = 0.1   
##   
## GA results:   
## Iterations = 100   
## Fitness function value = 1.924335e-05   
## Solution =   
## x1 x2 x3 x4 x5 x6 x7 x8 x9 x10 x11 x12 x13 x14 x15 x16 x17 x18 x19  
## [1,] 29 34 13 14 38 8 41 16 22 1 9 20 12 15 40 23 25 10 24  
## x20 ... x48  
## [1,] 45 2



## Problem 3 - TSP with Simulated Annealing

Modify the code in saTSP.R and include it below to approximate an optimal tour for the 48 city TSP problem in Problem 2. Include a graph of the best tour you are able to find.

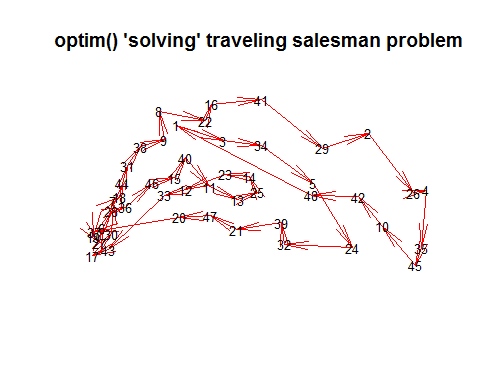
distance <- function(sq) { # Target function  
 sq2 <- embed(sq, 2)  
 sum(D[cbind(sq2[,2], sq2[,1])])  
}  
  
  
genseq <- function(sq) { # Generate new candidate sequence  
 idx <- seq(2, NROW(D)-1)  
 changepoints <- sample(idx, size = 2, replace = FALSE)  
 tmp <- sq[changepoints[1]]  
 sq[changepoints[1]] <- sq[changepoints[2]]  
 sq[changepoints[2]] <- tmp  
 sq  
}  
  
sq <- c(1:nrow(D), 1) # Initial sequence: alphabetic  
  
set.seed(123)  
res <- optim(sq, distance, genseq, method = "SANN",  
 control = list(maxit = 30000, temp = 2000, trace = TRUE,  
 REPORT = 500))

## sann objective function values  
## initial value 157553.000000  
## iter 5000 value 50120.000000  
## iter 10000 value 48766.000000  
## iter 15000 value 46646.000000  
## iter 20000 value 45096.000000  
## iter 25000 value 44892.000000  
## iter 29999 value 44892.000000  
## final value 44892.000000  
## sann stopped after 29999 iterations

res

## $par  
## [1] 1 3 34 5 24 32 39 21 47 20 37 19 27 6 28 36 46 15 40 11 13 25 14  
## [24] 23 12 33 43 17 30 7 18 44 31 38 9 8 22 16 41 29 2 26 4 35 45 10  
## [47] 42 48 1  
##   
## $value  
## [1] 44892  
##   
## $counts  
## function gradient   
## 30000 NA   
##   
## $convergence  
## [1] 0  
##   
## $message  
## NULL

tspres <- loc[res$par,]  
plot(x, y, type = "n", asp = 1, xlab = "", ylab = "",  
 main = "optim() 'solving' traveling salesman problem", axes = FALSE)  
arrows(tspres[s,1], tspres[s,2], tspres[s+1,1], tspres[s+1,2],  
 angle = 10, col = "red")  
text(x, y, cex = 0.8)



How do the results compare to those achieved using a genetic algorithm? Is one algorithm significantly more efficient than the other? Compare the total tour distances produced by the two algorithms, does one algorithm consistently produce better results?

## Problem 4 - Comparing Algorithms for a 30 dimensional Rastrigin function

The 30 dimensional Rastrigin function is considered very difficult to optimize and is a test case for many optimization algorithms. We know that the global minimum value of 0 occurs at the origin. For this problem you should compare the performance of Naive Multistart, the Genetic Algorithm plus local search, and the Simulated Annealing algorithm GenSA() from the 'GenSA' package. If you can get it to work, then also try the mlsl() function in the 'nloptr' package as it should work considerably better than Naive Multistart.

This is a somewhat open problem, but at the very least you should try each algorithm multiple times (possibly in for loop) and report on which algorithms are most efficient (fewest function calls) and which are most reliable (able to consistently identify the global minimum). Experiment with the algorithm parameters (population size, number of iterations of local search, etc.) You'll likely have to increase population sizes and the maximum number of iterations to successfully solve the 30 dimensional problem. Look at the source code in the presentation .Rmd file included in the download packet for guidance in setting up your algorithms.

# your code goes in this block  
dimension = 30;  
lower = rep(-5.12,dimension); upper = rep(5.12,dimension);   
x0 = runif(dimension,-5.12,5.12)  
  
bestmin <- 100000;  
for (j in 1:10){  
 x0 <- as.vector(runif(dimension,min=-5.12,max=5.12));  
 result <- optim(x0, Rastrigin, method="L-BFGS-B", lower=lower, upper=upper)  
 if (result$value<bestmin){ bestmin = result$value; bestx = result$par}  
}  
bestmin

## [1] 174.117

bestx

## [1] 9.949586e-01 5.356616e-09 1.989912e+00 9.949586e-01 -5.620771e-08  
## [6] 9.949586e-01 1.989912e+00 -6.964051e-08 -2.984856e+00 -1.989912e+00  
## [11] -9.949586e-01 -3.979784e+00 3.979784e+00 4.974691e+00 1.989912e+00  
## [16] -2.984856e+00 -2.984856e+00 1.183478e-08 -9.949586e-01 2.737254e-09  
## [21] -3.979784e+00 2.984856e+00 9.949586e-01 -1.989912e+00 -1.989912e+00  
## [26] 9.949586e-01 3.979784e+00 2.984856e+00 -9.949586e-01 2.984856e+00

result = ga(type="real-valued",fitness=Rastrigin,min=lower,max=upper,maxiter=500, monitor=FALSE, seed=100)  
result@fitnessValue

## [1] 1063.729

result2 = ga(type="real-valued",fitness=Rastrigin,min=lower,max=upper,maxiter=500, monitor=FALSE, seed=99)  
result2@fitnessValue

## [1] 1100.196

outGenSA <- GenSA(x0, Rastrigin, lower, upper)  
outGenSA$value

## [1] 0

Discuss your algorithm comparison here: The simulated annealing produced the best result. It appears that the optim() function produces similar results