A genetic algorithm for the Knapsack problem

The aim of this document is to show how works a genetic algorithm (GA) for the knaspack problem. It is not a good alternative to solve these problems, but it can help to show how GAs work.

Defining test instances

To test the algorithm, we define first a function that generates instances of the knapsack problem. It is based in recommendations of

Pisinger, D. (2005). Where are the hard knapsack problems?. Computers & Operations Research, 32(9), 2271-2284.

The utility of each instance is quite close to its weight, so it can be somewhat hard to solve:

```
KPInstance01 <- function(R, size, h, seed=1212){
  set.seed(1212)
  w <- sample(1:R, size, replace=TRUE)
  u <- sapply(1:size, function(x) sample((w[x] + R/10 - R/500):(w[x] + R/10 + R/500), 1))
  W <- round(h/(101)*sum(w))
  return(list(w=w,u=u,W=W))
}
```

Every instance is encoded as a list containing the weights `w`, utilities `u` and knapsack capacity `W`.

Defining the GA

Crossover and mutation operators

Then we will define the building blocks of the genetic algorithm. First we define the **fitness function**. It is equal to total utility for feasible solutions, and equal to zero to non-feasible solutions:

```
FitnessKP <- function(dades, sol){
  if(sum(dades$w*sol) <=dades$W) fit <- sum(dades$u*sol)
    else fit <- 0
    return(fit)
}
```

The following function defines a 1-point **crossover operator** for the KP. It takes for granted that we encode the solution as a binary string of length equal to the number of items `n`.

```
CrossoverKP1point <- function(sol1, sol2){
  n <- length(sol1)
  point <- sample(1:(n-1), 1)
  son <- c(sol1[1:point], sol2[(point +1):n])

  return(son)
}
```

To test the operator, here are several crossovers with a all-zeroes and a all-ones strings:

```
set.seed(1213)
a <- rep(1, 10)
b <- rep(0, 10)
for(i in 1:5) print(CrossoverKP1point(a,b))

##  [1] 1 1 0 0 0 0 0 0 0 0
##  [1] 1 1 0 0 0 0 0 0 0 0
##  [1] 1 1 1 1 1 1 1 1 0 0
##  [1] 1 1 1 1 0 0 0 0 0 0
##  [1] 1 0 0 0 0 0 0 0 0 0
```

This function defines a three-point **mutation operator**. It changes the value of three consecutive positions of a string:

```
MutationKP3points <- function(sol){
  n <- length(sol)
  point <- sample(1:(n-2), 1)
  sol <- as.logical(sol)
  sol[point:(point+2)] <- !sol[point:(point+2)]
  sol <- as.numeric(sol)

  return(sol)
}
```

Let's run five times the mutation operator for a all-ones string:

```
ex <- rep(1, 10)
for(i in 1:5) print(MutationKP3points(ex))

##  [1] 0 0 0 1 1 1 1 1 1 1
##  [1] 0 0 0 1 1 1 1 1 1 1
##  [1] 1 1 1 1 1 1 1 0 0 0
##  [1] 1 1 1 1 1 1 0 0 0 1
##  [1] 1 1 0 0 0 1 1 1 1 1
```

The genetic algorithm function

Here is the function that runs the GA for an instance of KP:

```
GeneticAlgorithmKP <- function(dades, npop, iterations, pmut, elitist =TRUE,
verbose=FALSE){
```

```
#problem size
n <- length(dades$w)

#initial population
rel.size <- min(dades$W/sum(dades$w),1)
population.parent <- replicate(npop, sample(0:1, n, replace = TRUE, prob=c(1-rel.size,
rel.size)), simplify=FALSE)

#initializing next generation
population.son <- replicate(npop, numeric(n), simplify=FALSE)

  best.obj <- 0
best.sol <- numeric(n)

count <- 1

while(count <= iterations){

  #assess fitness of population parent

  fitness <- sapply(population.parent, function(x) FitnessKP(dades, x))

  #finding best solution
  iter.obj <- max(fitness)
  pos.max <- which(fitness == max(fitness))[1]
  iter.sol <- population.parent[[pos.max]]

  #update best solution
  if(iter.obj > best.obj){
    best.sol <- iter.sol
    best.obj <- iter.obj
  }

  #elitist strategy: replace worst solution by best solution so far
  if(elitist){
    pos.min <- which(fitness == min(fitness))[1]
    population.son[[pos.min]] <- best.sol
    fitness[pos.min] <- best.obj
  }

  #verbose printing
  if(verbose & (count%%10==0 | count==1)) {
    print(paste("population", count))
    sapply(1:npop, function(x) print(c(population.parent[[x]], fitness[x]), digits=0))
  }

  #computing probability of selection
  prob <- fitness^2
  prob <- prob/sum(prob)
  #creating son population
  for(i in 1:npop){
    #crossover
    parents <- sample(1:npop, 2, prob = prob)
    v1 <- population.parent[[parents[1]]]
    v2 <- population.parent[[parents[2]]]
    population.son[[i]] <- CrossoverKP1point(v1, v2)
    #mutation
    if(pmut > runif(1)) population.son[[i]] <- MutationKP3points(population.son[[i]])
  }


  #population son becomes parent
  population.parent <- population.son

  count <- count + 1
 }

 return(list(sol=best.sol, obj=best.obj))

}
```

Some comments to this function:

- The function parameters are: instance data `dades`, population size `npop`, number of `iterations` to be performed and probability of mutation `pmut`. If the logical `verbose` is set to `TRUE`, the function prints the population and fitness of each element every ten runs.
- Selection probability is proportional to the square of the fitness function of each instance. Then, all unfeasible solutions will have selection probability equal to zero.
- The GA may be stuck if all solutions of a population are unfeasible. To avoid this, I have included in each element of the starting population a number of items proportional to the ratio weight of knapsack vs. total weight of items.
- If the flag `elitist` is true, in each generation one of the worst solutions is replaced by the best solution obtained so far.

Generating instances

To test the algorithm, I have generated three instances:

```
InstanceKP10 <- KPInstance01(1000, 10, 40)
InstanceKP50 <- KPInstance01(1000, 50, 40, 2424)
InstanceKP100  <- KPInstance01(1000, 100, 40, 3333)
```

In the case of the KP, we can obtain the optimal solution using linear programming. This function returns the results of linear programming in the same format of the GA function:

```
LPKP <- function(dades){
  library(Rglpk)
  n <- length(dades$u)
  obj <- dades$u
  mat <- matrix(dades$w, nrow=1)
  types <- rep("B", n)
  sol.lp <- Rglpk_solve_LP(obj=obj, mat=mat,  dir="<=", rhs=dades$W, types=types,
max = TRUE)
  return(list(sol = sol.lp$solution, obj = sol.lp$optimum))
}
```

Then we proceed to solve instances of size 10 and 50 with linear programming:

```
solPL.KP10 <- LPKP(InstanceKP10)

## Warning: package 'slam' was built under R version 3.3.2

solPL.KP50 <- LPKP(InstanceKP50)
```

GA in action

Let's run the genetic algorithm for `InstanceKP10` and probability of mutation equal to zero:

```
set.seed(4444)
solGA.KP10 <- GeneticAlgorithmKP(InstanceKP10, 10, 50, 0, elitist=FALSE,
verbose=TRUE)

## [1] "population 1"
##  [1]    1    0    0    0    0    0    1    1    1    0 1514
##  [1] 1 1 1 0 1 0 0 0 1 1 0
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    0    0    0    1    1    1    0    0    0    0 1323
##  [1]    0    0    0    1    1    0    1    1    0    0 1532
##  [1]    1    1    0    0    1    0    0    0    0    0 1312
##  [1] 0 0 1 0 1 0 0 1 0 0 0
##  [1] 0 1 1 1 0 1 0 0 0 1 0
##  [1]   1   0   0   0   0   1   0   0   0   0 499
##  [1]   0   0   0   0   0   0   0   1   0   0 158
## [1] "population 10"
##  [1]   1   0   0   1   0   1   0   0   0   0 952
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    0    1    0    1    0    0    0 1286
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1] 1 0 0 0 1 1 0 0 1 0 0
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]   0   0   0   0   1   0   1   0   0   0 921
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
## [1] "population 20"
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
## [1] "population 30"
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
## [1] "population 40"
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
## [1] "population 50"
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
```

The algorithm gets stuck in the best feasible solution of the initial population. Let's run the algorithm with nonzero probability of mutation and elitist strategy:

```
set.seed(4444)
solGA.KP10 <- GeneticAlgorithmKP(InstanceKP10, 10, 100, 0.2, elitist=TRUE,
verbose=TRUE)
```

```
## [1] "population 1"
##  [1]    1    0    0    0    0    0    1    1    1    0 1514
##  [1]    1    1    1    0    1    0    0    0    1    1 1758
##  [1]    1    0    0    1    0    1    0    0    1    0 1758
##  [1]    0    0    0    1    1    1    0    0    0    0 1323
##  [1]    0    0    0    1    1    0    1    1    0    0 1532
##  [1]    1    1    0    0    1    0    0    0    0    0 1312
##  [1] 0 0 1 0 1 0 0 1 0 0
##  [1] 0 1 1 1 0 1 0 0 0 1 0
##  [1]   1   0   0   0   0   1   0   0   0   0 499
##  [1]   0   0   0   0   0   0   0   1   0   0 158
## [1] "population 10"
##  [1]    0    0    0    1    0    0    0    0    1    0 1259
##  [1]    1    0    0    1    0    0    0    0    1    0 1624
##  [1]    1    1    1    1    0    1    0    1    0    0 1846
##  [1] 0 0 0 1 1 0 0 0 1 0 0
##  [1]    1    0    0    1    1    0    0    0    0    0 1554
##  [1]    0    0    0    1    0    1    0    0    1    0 1393
##  [1] 1 0 0 1 1 0 0 0 1 0 0
##  [1]   0   0   0   1   0   0   0   0   0   0 453
##  [1]    0    0    0    1    0    0    1    1    1    0 1602
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
## [1] "population 20"
##  [1]    0    0    0    1    0    0    0    0    1    1 1600
##  [1]    0    0    0    1    0    0    0    0    1    1 1600
##  [1]    0    0    0    1    0    0    0    0    1    1 1600
##  [1]    1    0    0    1    0    1    0    0    1    1 1846
##  [1]    0    0    0    1    0    1    0    0    1    0 1393
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
##  [1] 0 1 1 0 0 0 0 1 1 0
##  [1]    0    0    0    1    0    0    0    0    1    1 1600
##  [1] 0 0 0 1 1 0 1 0 1 0 0
## [1] "population 30"
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
##  [1]    0    0    0    1    0    1    0    1    1    1 1846
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
##  [1] 0 0 1 0 1 1 1 1 0 1 0
##  [1]    0    0    0    1    0    1    0    0    1    1 1734
##  [1]   0   0   0   1   0   0   0   0   0   1 794
##  [1]    0    0    0    1    0    0    0    0    1    1 1600
##  [1] 0 0 0 1 0 1 0 1 1 1 0
## [1] "population 40"
##  [1]    0    0    0    0    1    1    0    1    1    1 1846
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1] 0 0 1 0 1 0 0 1 1 1 0
##  [1] 1 1 1 1 0 0 0 1 1 1 0
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1] 1 1 1 1 0 0 0 0 1 1 0
##  [1] 1 1 1 1 0 0 0 1 1 1 0
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]   0   0   0   1   0   0   1   0   0   1 979
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
## [1] "population 50"
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    1    1    1    1    0    0    0    1    1    1 1846
##  [1] 0 0 0 1 1 1 1 1 1 1 0
##  [1]    0    0    1    1    0    0    0    0    0    0 1518
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
## [1] "population 60"
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    1    1    1    1    0    0    0    1    1    1 1846
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1] 0 0 0 1 0 1 1 0 1 1 0
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
## [1] "population 70"
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]   0   0   0   1   0   0   0   0   0   0 453
##  [1]    0    0    0    0    1    1    0    1    1    1 1846
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
## [1] "population 80"
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    1    1    0    1    1 1846
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]   0   0   0   1   0   0   0   0   0   0 453
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1] 0 0 1 1 0 0 0 1 1 1 0
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
## [1] "population 90"
##  [1]    1    1    1    1    0    0    0    1    1    1 1846
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]   0   0   0   1   0   0   1   0   0   1 979
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1] 1 1 1 1 0 0 0 1 1 1 0
##  [1] 1 1 1 1 0 0 0 1 1 1 0
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1] 0 1 1 0 0 0 1 0 0 1 0
##  [1] 1 1 1 1 0 0 0 1 1 1 0
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
## [1] "population 100"
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    1    1    1    1    1 1846
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
##  [1]    0    0    0    1    0    0    0    1    1    1 1758
```

Finally, we will run the algorithm with 1000 iterations and :

```
set.seed(4444)
solGA.KP10 <- GeneticAlgorithmKP(InstanceKP10, 10, 1000, 0.2, elitist=TRUE,
verbose=FALSE)
solGA.KP10

## $sol
##  [1] 1 1 0 0 1 1 0 1 0 1
##
## $obj
## [1] 1945

solPL.KP10

## $sol
##  [1] 0 1 0 1 0 1 1 1 1 0
##
## $obj
## [1] 1947
```

For the instances of size 50 and 100 we have:

```r
set.seed(4444)
solGA.KP50 <- GeneticAlgorithmKP(InstanceKP50, 10, 1000, 0.2, elitist=TRUE,
verbose=FALSE)
solGA.KP50

## $sol
##  [1] 1 0 0 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 0 1 0 1 1 0 0 0 1 0 1 0 1 0 1
## [36] 1 0 0 0 1 1 1 1 0 1 0 0 0 0 0
##
## $obj
## [1] 12071

solPL.KP50

## $sol
##  [1] 1 1 0 1 0 1 1 1 0 1 0 0 1 1 0 1 1 1 0 1 1 1 1 1 0 1 0 0 1 0 1 1 1 0 1
## [36] 1 0 0 0 1 1 1 0 1 1 1 1 0 0 0
##
## $obj
## [1] 12477

set.seed(4444)
solGA.KP100 <- GeneticAlgorithmKP(InstanceKP100, 10, 1000, 0.2, elitist=TRUE,
verbose=FALSE)
solGA.KP100

## $sol
##   [1] 0 1 0 0 1 1 0 1 1 0 1 0 0 0 0 0 1 1 1 0 0 0 1 1 0 0 1 0 1 0 1 0 1 0 1
##  [36] 1 0 1 1 1 0 0 1 1 0 0 1 0 1 0 1 1 0 0 1 1 0 0 1 1 1 1 0 0 0 0 1 0 0 0
##  [71] 0 0 1 1 0 0 1 1 0 1 1 1 1 0 0 0 0 1 0 1 1 0 0 1 0 0 0 0 1 1
##
## $obj
## [1] 23002
```

The optimal solution of the instance with 100 items has objective value equal to 24643. So the genetic algorithm work well solving small instances of the knapsack problem.