Optimisation Assignment

Edward Baleni, BLNEDW003, Wayne Jiang, JNGWEN002

2023-05-28

```
require(Rglpk)
```

```
## Loading required package: Rglpk
```

```
## Loading required package: slam
```

```
## Using the GLPK callable library version 5.0
```

```
require(foreach)
```

```
## Loading required package: foreach
```

Data

```
X <- read.table("Data3.txt")
```

Linear Program

```
#LP
# Objective is to minimise total cost
my_obj1 <- X$Price

# Constraints
my_mat = rbind(X$pH,X$pH,X$Abrasiveness,
               X$Hardness,X$Dryness,X$Dryness,
               X$Bitterness,X$H,X$H,
               c(1,1,1,1,0,0,0),
               c(0,0,0,0,1,1,1))

# Bounds
my_dir=c(">","<",">","<",">","<",">",">","<","==","<=")

# RHS
my_rhs=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5)

# Data types
my_types<-c("C","C","C","C","I","I","I")

# Run MILP
LP=Rglpk_solve_LP(obj=my_obj1,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=F)
```

Simulated Annealing

```
#Simulated Annealing
#set seed to make result reproducible
set.seed(1)
#create a matrix to store all solution by different setting of Simulated Annealing
solu_SA=matrix(0,nrow=7,ncol=5)

#evaluation function
evaluate_x<-function(x){
  # Objective
  my_obj <- X$Price
  # Calculate objective
  eva=x%*%my_obj

  # If constraints are not met then give a large number (since we're minimizing)
  if(x%*%X$pH<3.52|x%*%X$pH>3.55|x%*%X$Abrasiveness<8
     |x%*%X$Hardness>10|x%*%X$Dryness<35|x%*%X$Dryness>38
     |x%*%X$Bitterness<7.7|x%*%X$H<17|x%*%X$H>18|sum(x[5:7])>5)
    {
    eva=1000
    }
  # Return output
  return(eva)
}

#function to give a feasible initial solution in order to find optimal
get_initial_x<-function(){
  # Get a sequence between 0 and 1
  x=seq(0,1,length.out=100)
  # initialise current solution
  cur_x=c(0.25,0.25,0.25,0.25,0,0,0)
  found=F

  # First find a feasible solution that meets all the constraints, might not be the optimal solution
  while(found==F)
  {
    sam=sample(1:4,4,replace = F)
    for(i in 1:100)
    {
      for(j in 1:100)
      {
        for(k in 1:100)
        {
          cur_x[sam[1]]=x[i]
          cur_x[sam[2]]=x[j]
          cur_x[sam[3]]=x[k]
          cur_x[sam[4]]=1-cur_x[sam[1]]-cur_x[sam[2]]-cur_x[sam[3]]
          if(cur_x%*%X$pH>3.52&cur_x%*%X$pH<3.55&cur_x%*%X$Abrasiveness>8
             &cur_x%*%X$Hardness<10&cur_x%*%X$Dryness>35&cur_x%*%X$Dryness<38
             &cur_x%*%X$Bitterness>7.7&cur_x%*%X$H>17&cur_x%*%X$H<18
             &cur_x[sam[4]]>=0)
          {
            cur_xt=cur_x
            found=T
```

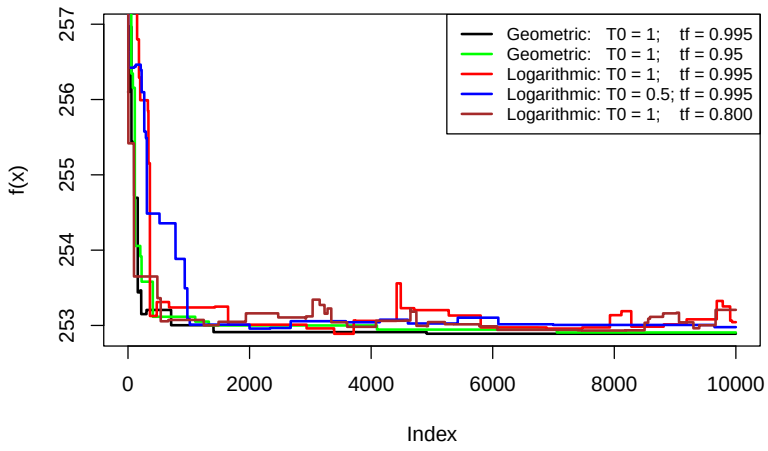
```
          }
        }
      }
    }
  }
  return(cur_xt)
}

# function to change some value of the solution vector
# either wine proportion or number of food additive will be change
perturb_x <- function(cur_x){
  # Choose a number between 1 and 2
  sam=sample(1:2,1,replace = F)
  # If 1 is sampled then change the continuous variables
  if(sam==1){
    sam=sample(1:4,2,replace = F)
    value=runif(1,-min(cur_x[sam[1]],cur_x[sam[2]])/2,min(cur_x[sam[1]],cur_x[sam[2]])/2)
    cur_x[sam[1]]=cur_x[sam[1]]+value
    cur_x[sam[2]]=cur_x[sam[2]]-value
  }
  else{
    # if 2 is sampled then change the integer variables
    sam=sample(5:7,1,replace = F)
    cur_x[sam]=sample(c(max(cur_x[sam]-1,0),cur_x[sam]+1),1,replace = F)
  }
  return(cur_x)
}

#Geometric with temperature factor=0.995
start_temp <- 1
temp_factor <- 0.995
all_fx=c()
all_x=c()

# Get initial solutioin
initx=get_initial_x()
cur_x=initx
cur_fx=evaluate_x(cur_x)

# Perform SA
for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp * temp_factor ^ i
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
```

```
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_G1=all_fx
solu_SA[,1]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_fx))[1]*(7))]

# Repeat for different temperature
#Geometric with temperature factor =0.95
start_temp <- 0.1
temp_factor <- 0.95
all_fx=c()
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp * temp_factor ^ i
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_G5=all_fx
solu_SA[,2]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_fx))[1]*(7))]

# Repeat for different cooling schedule
#Logarithmic with starting temp =1 temp_factor=0.995
start_temp <- 1
temp_factor <- 0.995
all_fx=c()
```

```
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp /(1+temp_factor*log(1+i))
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_L1=all_fx
solu_SA[,3]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_fx))[1]*(7))]

# Repeat for different starting temp
#Logarithmic with starting temp =0.5,temp_factor=0.995
start_temp <- 0.5
temp_factor <- 0.995
all_fx=c()
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp /(1+temp_factor*log(1+i))
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
```

```
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}
all_fx_L5=all_fx
solu_SA[,4]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_fx))[1]*(7))]

# Repeat for starting temp factor
#Logarithmic with starting temp =1 tf=0.8
start_temp <- 1
temp_factor <- 0.8
all_fx=c()
all_x=c()
cur_x=initx
cur_fx=evaluate_x(cur_x)

for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp /(1+temp_factor*log(1+i))
  accept_prob <- exp(-(prop_fx - cur_fx) / anneal_temp)
  # accept or reject the candidate
  if(prop_fx < cur_fx){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }
  else{ if(runif(1) < accept_prob){
    cur_x <- prop_x
    cur_fx <- prop_fx
  }}

  # store all results
  all_fx <- c(all_fx, cur_fx)
  all_x <- c(all_x,cur_x)
}

all_fx_L18=all_fx
solu_SA[,5]=all_x[((which(all_fx==min(all_fx))[1]-1)*(7)+1):(which(all_fx==min(all_fx))[1]*(7))]

# Plot final solution
plot(all_fx_G1,type="l", ylab = "f(x)",lwd=2)
points(all_fx_G5,type="l",col="green",lwd=2)
points(all_fx_L1,type="l",col="red",lwd=2)
points(all_fx_L5,type="l",col="blue",lwd=2)
points(all_fx_L18,type="l",col="brown",lwd=2)
legend("topright", legend=c("Geometric:   T0 = 1;    tf = 0.995","Geometric:   T0 = 1;    tf = 0.95",
```

```
                            "Logarithmic: T0 = 1;    tf = 0.995","Logarithmic: T0 = 0.5; tf = 0.995",
                            "Logarithmic: T0 = 1;    tf = 0.800"),
       col=c("black","green","red", "blue","brown"), lty=1, cex=0.8,lwd=2)
```



Genetic Algorithm

```
set.seed(6)
```

```
normal <- function(pop){
  # normalize to make sure that the proportions add up to 1
  if(!is.null(dim(pop)))
    pop/rowSums(pop)
  else
    pop/sum(pop)
}

init.pop <- function(n, p, X){
  # Obtain continuous
  Y <- normal(matrix(sample(1:100, n*(p-3), replace = T)/100, nrow = n, ncol = p-3))

    # Generate integer solutions if there are any
  test <- F
  hold <- matrix(NA, nrow = n, ncol = 3)
  for (i in 1:n) {
    while (test==F) {
      x <- sample(0:5, prob = c(50,10, 1, 0.5, 0.1 , 0.05), size = 3, replace = T)
      if(sum(x) <=5 )
        test <- T
      else
        test <- F
    }
    hold[i,] <- x
    test <- F
  }
```

```
  Y <- cbind(Y, hold)

  test <- FALSE
  count <-  1

  # Obtain a population that works for the solution
  #for (i in 1:n) {
  foreach(i=1:n) %do% {
    x <- Y[i,]
    while (test == F){
        # Calculate constraints
        ph      <- x %*% X$pH
        abras   <- x %*% X$Abrasiveness
        hard    <- x %*% X$Hardness
        dry     <- x %*% X$Dryness
        bitter  <- x %*% X$Bitterness
        hue     <- x %*% X$H

        # Check if solution is in feasible region
        test <- ifelse(ph < 3.52 | ph > 3.55, F,
          ifelse(abras < 8, F,
                 ifelse(hard > 10, F,
                        ifelse(dry < 35 | dry > 38, F,
                               ifelse(bitter < 7.7,  F,
                                      ifelse(hue < 17 | hue > 18, F,
                                             T))))))

        # Maybe it's the integers that are messing everything up
        if(test==F){
          x[5:7] <- sample(0:5, prob = c(50,10, 1, 0.5, 0.1 , 0.05), size = 3, replace = T)
          while (test == F & sum(x[5:7]) > 5) {
            x[5:7] <- sample(0:5, prob = c(50,10, 1, 0.5, 0.1 , 0.05), size = 3, replace = T)
          }
        }

        # Or maybe it's the continuous variables
        if (test==F){
          x <- c(normal(sample(1:100, p-3, replace = T)/100), x[5:7])
        }
        else{
          Y[i,] <- x
        }
    }
    test <- F
  }

  return(Y)
}
```

```
eval.pop <- function(pop, X){
  # Evaluate fitness
  n <- nrow(pop)
```

```
  # Initialize fitness
  fitness <- rep(0, n)
  # Calculate fitness
  fitness <- pop %*% X$Price
  ph <- pop %*% X$pH
  abras <- pop %*% X$Abrasiveness
  hard <- pop %*% X$Hardness
  dry <- pop %*% X$Dryness
  bitter <- pop %*% X$Bitterness
  hue <- pop %*% X$H

   # Handle constraints
   fitness <- ifelse(ph < 3.52 | ph > 3.55, 400,
          ifelse(abras < 8, 400,
                 ifelse(hard > 10, 400,
                        ifelse(dry < 35 | dry > 38, 400,
                               ifelse(bitter < 7.7,  400,
                                      ifelse(hue < 17 | hue > 18, 400,
                                             fitness))))))

  return(fitness)
}
```

```
# Rank based selection
select.rank <- function(pop, fit){
  n <- nrow(pop)
  p <- ncol(pop)
  # Rank population based on fitness
  new.pop <- matrix(NA, nrow = n, ncol = p)
  # Obtain order of fitness
  ord <- order(fit, decreasing = T)
  # Arrange according to order
  fit2 <- fit
  fit2 <- fit2[ord]
  fit2 <- cbind(fit2, 1:n )
  # Place back in regular order
  fit2[ord,] <- fit2[1:n,]

  # Sample based on rank
  rank.samp <- sample(1:n, prob = fit2[,2], replace = T)

  # Input these samples into the new population
  new.pop <- pop[rank.samp,]

  return(new.pop)
}

# Tournament Selection
select.tournament <- function(pop, fit, s.size){
  n <- nrow(pop)
  p <- ncol(pop)
  hold <- list()
  # Tournament based on fitness
  new.pop <- matrix(NA, nrow = n, ncol = p)
```

```
  # Tournament
  for (i in 1:n) {
    sub.samp <- sample(1:n, s.size, replace = T)
    hold[[i]] <- sub.samp[which.min(fit[sub.samp])]
  }
  pop[unlist(hold),]
}
```

```
# Uniform crossover
uni.cross <-  function(parents){
  # Georgina (2023)
  n <- nrow(parents)
  p <- ncol(parents)

  # Pick parents to mate
  parent_pairs <- matrix(sample(1:n), n/2, 2)

  # Initialise offspring
  offsprings <- matrix(NA, n, p)
  for(i in 1:n/2){
    # Get parents
    p1 <- parents[parent_pairs[i,1], ]
    p2 <- parents[parent_pairs[i,2], ]
    # Make kids
    c1 <- rep(NA, p)
    c2 <- rep(NA, p)
    # Apply uniform crossover to get kids
    for(j in 1:p){
      if(runif(1) <= 0.5){
        c1[j] <- p1[j]
        c2[j] <- p2[j]
      }else{
        c2[j] <- p1[j]
        c1[j] <- p2[j]
      }
    }
    # Store kids
    offsprings[2*i-1, ] <- c1
    offsprings[2*i, ] <- c2
  }
  return(offsprings)
}

# N-point crossover
n.cross <- function(parents){
    # N-point Crossover
  n <- nrow(parents)
  p <- ncol(parents)

  # Pick parents to mate
  pairs <- matrix(sample(1:n), n/2, 2)

  # Initialise offspring
```

```
  offsprings <- matrix(NA, n, p)

  # Perfom N-point
  for (i in 1:round(n/2)) {
    # Get parents
    p1 <- parents[pairs[i,1], ]
    p2 <- parents[pairs[i,2], ]

    # Make kids
    c1 <- rep(NA, p)
    c2 <- rep(NA, p)

    # Pick cross-point as 3
    c1 <- c(p1[1], p2[2], p1[3], p2[4], p1[5], p2[6], p1[7])
    c2 <- c(p2[1], p1[2], p2[3], p1[4], p2[5], p1[6], p2[7])


    # Store kids
    offsprings[2*i-1, ] <- c1
    offsprings[2*i, ] <- c2
  }
  return(offsprings)
}
```

```
scram.mut = function(cross, mutation_rate, check){
  # Georgina (2023)
  # Scramble mutation
  n <- nrow(cross)
  p <- ncol(cross)

  cross2 <- cross
  if (check){
    # Normalize if the proportions do not add up to between 0.9 and 1
    cross2 <- ifelse(matrix(rep(rowSums(cross), 4), n, p) < 0.9 |
                     matrix(rep(rowSums(cross), 4), n, p) > 1,
                     normal(cross),
                     cross)
  }

  # Initialise mutations
  mutations = matrix(NA, n,p)

  for(i in 1:n){
    persontomutate = cross2[i,]
    if(runif(1) <= mutation_rate){
      # Select two elements
      picks = sort(sample(1:p, 2, replace = FALSE))
      # Get sub-set
      temp = persontomutate[picks[1]:picks[2]]
      # Reshuffle
      temp = sample(temp, length(temp), replace = FALSE)
      # Add mutation
      persontomutate[picks[1]:picks[2]] = temp
      mutations[i,] = persontomutate
```

```
    }else{
      mutations[i,] = persontomutate
    }
  }
  return(mutations)
}

mut.func <- function(p){
  # Insert mutation function

  # Select two elements
  picks = sort(sample(1:p, 2, replace = FALSE))

  # Move second to first
  ord <- c(which(1:p <= picks[1]), picks[2], which(1:p > picks[1] & 1:p != picks[2]))

  # Return order
  return(ord)
}


insert.mut <- function(cross, mut.rate, check){
  # Insert mutation
  p <- ncol(cross)
  n <- nrow(cross)

  cross2 <- cross
  # Normalize if the proportions do not add up to between 0.9 and 1
  if (check){
    cross2 <- ifelse(matrix(rep(rowSums(cross), 4), n, p) < 0.9 |
                     matrix(rep(rowSums(cross), 4), n, p) > 1,
                     normal(cross),
                     cross)
  }
  # Initialise mutations
  mutations = matrix(NA, n, p)

  # Perform insert mutation
  rate <- replicate(n, runif(1))

  for (i in 1:n) {
    if(rate[i] <= mut.rate){
      mutations[i,] <- cross2[i,mut.func(p)]
    }
    else
      mutations[i,] <- cross2[i,]
  }
  return(mutations)
}
```

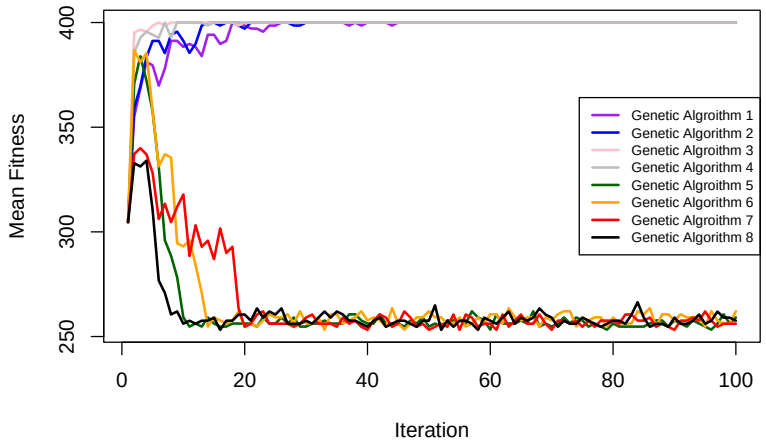
```
# Initialise matrix
pop.in1 <- pop.in2 <- pop.in3 <- pop.in4 <- pop.in5 <- pop.in6 <- pop.in7 <- pop.in8 <- init.pop(100,7,

# Initialise list to store fittest and mean fitness
```

```
fittest <- list()
fit_mean <- list()

# Number of generations to run GA
gen <- 100

# Perform GA
for (i in 1:gen) {
  # evaluate function
  evals1 <- eval.pop(pop.in1, X)
  evals2 <- eval.pop(pop.in2, X)
  evals3 <- eval.pop(pop.in3, X)
  evals4 <- eval.pop(pop.in4, X)
  evals5 <- eval.pop(pop.in5, X)
  evals6 <- eval.pop(pop.in6, X)
  evals7 <- eval.pop(pop.in7, X)
  evals8 <- eval.pop(pop.in8, X)

  # select by rank
  nxt.parent.rank1 <-  select.rank(pop.in1, evals1)
  nxt.parent.rank2 <-  select.rank(pop.in2, evals2)
  nxt.parent.rank3 <-  select.rank(pop.in3, evals3)
  nxt.parent.rank4 <-  select.rank(pop.in4, evals4)
  # select by tournament
  nxt.parent.tourn5 <- select.tournament(pop.in5, evals5, 5)
  nxt.parent.tourn6 <- select.tournament(pop.in6, evals6, 5)
  nxt.parent.tourn7 <- select.tournament(pop.in7, evals7, 5)
  nxt.parent.tourn8 <- select.tournament(pop.in8, evals8, 5)

  # Rank - Uni - Scram
    # cross by uni     # can do by rank or tournament
  offspring.cross.uni.rank <-  uni.cross(nxt.parent.rank1)
    # By Mutation by scramble
  offspring.mut.scram.uni.rank1 <- scram.mut(offspring.cross.uni.rank[,1:4], 0.05, T)
  offspring.mut.scram.uni.rank2 <- scram.mut(offspring.cross.uni.rank[,5:7], 0.05, F)
  offspring.mut.scram.uni.rank <- cbind(offspring.mut.scram.uni.rank1, offspring.mut.scram.uni.rank2)
    # Replace
  pop.in1 <- offspring.mut.scram.uni.rank

  # Rank - Uni - Insert
    # cross by uni     # can do by rank or tournament
  offspring.cross.uni.rank <-  uni.cross(nxt.parent.rank2)
    # By Mutation by insert
  offspring.mut.insert.uni.rank1 <- insert.mut(offspring.cross.uni.rank[,1:4], 0.05, T)
  offspring.mut.insert.uni.rank2 <- insert.mut(offspring.cross.uni.rank[,5:7], 0.05, F)
  offspring.mut.insert.uni.rank <- cbind(offspring.mut.insert.uni.rank1, offspring.mut.insert.uni.rank2)
    # Replace
  pop.in2 <- offspring.mut.insert.uni.rank

  # Rank - N - Insert
    # Cross by N
  offspring.cross.n.rank <-  n.cross(nxt.parent.rank3)
    # Mutation by insert
```

```
offspring.mut.insert.n.rank1 <- insert.mut(offspring.cross.n.rank[,1:4], 0.05, T)
offspring.mut.insert.n.rank2 <- insert.mut(offspring.cross.n.rank[,5:7], 0.05, F)
offspring.mut.insert.n.rank <- cbind(offspring.mut.insert.n.rank1, offspring.mut.insert.n.rank2)
  # Replace
pop.in3 <- offspring.mut.insert.n.rank

# Rank - N - Scramble
    # Cross by N
offspring.cross.n.rank <-  n.cross(nxt.parent.rank4)
  # Mutation by insert
offspring.mut.scram.n.rank1 <- scram.mut(offspring.cross.n.rank[,1:4], 0.05, T)
offspring.mut.scram.n.rank2 <- scram.mut(offspring.cross.n.rank[,5:7], 0.05, F)
offspring.mut.scram.n.rank <- cbind(offspring.mut.scram.n.rank1, offspring.mut.scram.n.rank2)
  # Replace
pop.in4 <- offspring.mut.scram.n.rank

# Tourn - N - Insert
  # Cross by N
offspring.cross.n.tourn <- n.cross(nxt.parent.tourn5)
  # Mutation by insert
offspring.mut.insert.n.tourn1 <- insert.mut(offspring.cross.n.tourn[,1:4], 0.05, T)
offspring.mut.insert.n.tourn2 <- insert.mut(offspring.cross.n.tourn[,5:7], 0.05, F)
offspring.mut.insert.n.tourn <- cbind(offspring.mut.insert.n.tourn1, offspring.mut.insert.n.tourn2)
  # Replace
pop.in5 <- offspring.mut.insert.n.tourn

# Tourn - N - Scram
  # Cross by N
offspring.cross.n.tourn <- n.cross(nxt.parent.tourn6)
  # Mutation by Scram
offspring.mut.scram.n.tourn1 <- scram.mut(offspring.cross.n.tourn[,1:4], 0.05, T)
offspring.mut.scram.n.tourn2 <- scram.mut(offspring.cross.n.tourn[,5:7], 0.05, F)
offspring.mut.scram.n.tourn <- cbind(offspring.mut.scram.n.tourn1, offspring.mut.scram.n.tourn2)
  # Replace
pop.in6 <- offspring.mut.scram.n.tourn

# Tourn - Uni - Insert
  # Cross by Uni
offspring.cross.uni.tourn <- uni.cross(nxt.parent.tourn7)
  # Mutation by Insert
offspring.mut.insert.uni.tourn1 <- insert.mut(offspring.cross.uni.tourn[,1:4], 0.05, T)
offspring.mut.insert.uni.tourn2 <- insert.mut(offspring.cross.uni.tourn[,5:7], 0.05, F)
offspring.mut.insert.uni.tourn <- cbind(offspring.mut.insert.uni.tourn1, offspring.mut.insert.uni.tou
  # Replace
pop.in7 <- offspring.mut.insert.uni.tourn

# Tourn - Uni - Scram
  # Cross by uni
offspring.cross.uni.tourn <- uni.cross(nxt.parent.tourn8)
  # Mutation by Scram
offspring.mut.scram.uni.tourn1 <- scram.mut(offspring.cross.uni.tourn[,1:4], 0.05, T)
offspring.mut.scram.uni.tourn2 <- scram.mut(offspring.cross.uni.tourn[,5:7], 0.05, F)
offspring.mut.scram.uni.tourn <- cbind(offspring.mut.scram.uni.tourn1, offspring.mut.scram.uni.tourn2
```

```
    # Replace
  pop.in8 <- offspring.mut.scram.uni.tourn

  # Store
  fittest[[i]] <- cbind(min(evals1), min(evals2), min(evals3), min(evals4), min(evals5), min(evals6), mi
  fit_mean[[i]] <- cbind(mean(evals1), mean(evals2), mean(evals3), mean(evals4), mean(evals5), mean(eval
}
```

```
# Plot convergence
fit <- matrix(unlist(fit_mean), nrow = gen, ncol = 8, byrow = T)
matplot(fit, col = c("purple", "blue", "pink", "grey", "darkgreen", "orange", "red", "black"), lwd = "2
legend("right", c("Genetic Algroithm 1", "Genetic Algorithm 2","Genetic Algroithm 3", "Genetic Algorithm
```



Multi-Objective Goal Programming

```
#MOGP
#max each goal to find range of value
my_obj1 <- X$Price
my_obj2 <- X$Alcohol
my_obj3 <- X$C
my_obj4 <- X$Sugar
my_obj5 <- X$Tannins
my_obj6 <- X$Anthocyanins

s1=Rglpk_solve_LP(obj=my_obj1,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s2=Rglpk_solve_LP(obj=my_obj2,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s3=Rglpk_solve_LP(obj=my_obj3,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s4=Rglpk_solve_LP(obj=my_obj4,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s5=Rglpk_solve_LP(obj=my_obj5,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)
s6=Rglpk_solve_LP(obj=my_obj6,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=T)

ss=rbind(s1$solution,s2$solution)
ss=rbind(ss,s3$solution)
ss=rbind(ss,s4$solution)
```

```
ss=rbind(ss,s5$solution)
ss=rbind(ss,s6$solution)
#the matrix can be use calculate range of value for each variables
mmm=ss%*%cbind(X$Price,X$Alcohol,X$C,X$Sugar,X$Tannins,X$Anthocyanins)

#use calculated weight to perform MOGP by Rglpk function
#Archimedean approach
my_obj1 <- c(rep(0,7),0.08,1.94,1.94,62.5,62.5,0.42,0.42,0.02,0.02,2.80,2.80)

my_mat = matrix(c(c(X$pH,rep(0,11)),
                  c(X$pH,rep(0,11)),
                  c(X$Abrasiveness,rep(0,11)),
                  c(X$Hardness,rep(0,11)),
                  c(X$Dryness,rep(0,11)),
                  c(X$Dryness,rep(0,11)),
                  c(X$Bitterness,rep(0,11)),
                  c(X$H,rep(0,11)),
                  c(X$H,rep(0,11)),
                  c(1,1,1,1,0,0,0,rep(0,11)),
                  c(0,0,0,0,1,1,1,rep(0,11)),
                  c(X$Price,-1,rep(0,10)),
                  c(X$Alcohol,0,-1,1,rep(0,8)),
                  c(X$C,rep(0,3),-1,1,rep(0,6)),
                  c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                  c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                  c(X$Anthocyanins,rep(0,9),-1,1)),ncol=18,byrow=T)

my_dir=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==","==")
my_rhs=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91)


my_types<-c("C","C","C","C","I","I","I","C","C","C","C","C","C","C","C","C","C","C")

ss1=Rglpk_solve_LP(obj=my_obj1,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=F)
round(ss1$solution,2)
```

```
##  [1]   0.33   0.01   0.41   0.25   0.00   0.00   0.00 253.95   0.00   0.83
## [11]   0.00   1.81 343.40   0.00   0.00 579.55   0.00   0.00
```

```
#Preemptive approach
#first step
my_obj2 <- c(rep(0,7),0.08,rep(0,10))
my_mat2 = matrix(c(c(X$pH,rep(0,11)),
                   c(X$pH,rep(0,11)),
                   c(X$Abrasiveness,rep(0,11)),
                   c(X$Hardness,rep(0,11)),
                   c(X$Dryness,rep(0,11)),
                   c(X$Dryness,rep(0,11)),
                   c(X$Bitterness,rep(0,11)),
                   c(X$H,rep(0,11)),
                   c(X$H,rep(0,11)),
                   c(1,1,1,1,0,0,0,rep(0,11)),
                   c(0,0,0,0,1,1,1,rep(0,11)),
                   c(X$Price,-1,rep(0,10)),
```

```
                    c(X$Alcohol,0,-1,1,rep(0,8)),
                    c(X$C,rep(0,3),-1,1,rep(0,6)),
                    c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                    c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                    c(X$Anthocyanins,rep(0,9),-1,1)),ncol=18,byrow=T)

ss2=Rglpk_solve_LP(obj=my_obj2,mat=my_mat,dir=my_dir,rhs=my_rhs,types=my_types,max=F)

#second step
#use optimal value from last step to constrain
my_obj3 <- c(rep(0,10),62.5,62.5,0,0,0,0,2.80,2.80)
my_mat3 = matrix(c(c(X$pH,rep(0,11)),
                    c(X$pH,rep(0,11)),
                    c(X$Abrasiveness,rep(0,11)),
                    c(X$Hardness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Bitterness,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(1,1,1,1,0,0,0,rep(0,11)),
                    c(0,0,0,0,1,1,1,rep(0,11)),
                    c(X$Price,-1,rep(0,10)),
                    c(X$Alcohol,0,-1,1,rep(0,8)),
                    c(X$C,rep(0,3),-1,1,rep(0,6)),
                    c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                    c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                    c(X$Anthocyanins,rep(0,9),-1,1),
                    c(rep(0,7),0.08,rep(0,10))),ncol=18,byrow=T)
my_dir3=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==","==","<=")
my_rhs3=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91,ss2$optimum)
ss3=Rglpk_solve_LP(obj=my_obj3,mat=my_mat3,dir=my_dir3,rhs=my_rhs3,types=my_types,max=F)

#third step
#use optimal value from previous step to constrain
my_obj4 <- c(rep(0,12),0.42,0.42,rep(0,4))
my_mat4 = matrix(c(c(X$pH,rep(0,11)),
                    c(X$pH,rep(0,11)),
                    c(X$Abrasiveness,rep(0,11)),
                    c(X$Hardness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Dryness,rep(0,11)),
                    c(X$Bitterness,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(X$H,rep(0,11)),
                    c(1,1,1,1,0,0,0,rep(0,11)),
                    c(0,0,0,0,1,1,1,rep(0,11)),
                    c(X$Price,-1,rep(0,10)),
                    c(X$Alcohol,0,-1,1,rep(0,8)),
                    c(X$C,rep(0,3),-1,1,rep(0,6)),
                    c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                    c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                    c(X$Anthocyanins,rep(0,9),-1,1),
```

```
                  c(rep(0,7),0.08,rep(0,10)),
                  c(rep(0,10),62.5,62.5,0,0,0,0,2.80,2.80)),ncol=18,byrow=T)
my_dir4=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==","==","<=","<=")
my_rhs4=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91,ss2$optimum,ss3$optimum)
ss4=Rglpk_solve_LP(obj=my_obj4,mat=my_mat4,dir=my_dir4,rhs=my_rhs4,types=my_types,max=F)

#last step
#use optimal value from previous step to constrain
my_obj5 <- c(rep(0,8),1.94,1.94,62.5,62.5,rep(0,6))
my_mat5 = matrix(c(c(X$pH,rep(0,11)),
                  c(X$pH,rep(0,11)),
                  c(X$Abrasiveness,rep(0,11)),
                  c(X$Hardness,rep(0,11)),
                  c(X$Dryness,rep(0,11)),
                  c(X$Dryness,rep(0,11)),
                  c(X$Bitterness,rep(0,11)),
                  c(X$H,rep(0,11)),
                  c(X$H,rep(0,11)),
                  c(1,1,1,1,0,0,0,rep(0,11)),
                  c(0,0,0,0,1,1,1,rep(0,11)),
                  c(X$Price,-1,rep(0,10)),
                  c(X$Alcohol,0,-1,1,rep(0,8)),
                  c(X$C,rep(0,3),-1,1,rep(0,6)),
                  c(X$Sugar,rep(0,5),-1,1,rep(0,4)),
                  c(X$Tannins,rep(0,7),-1,1,rep(0,2)),
                  c(X$Anthocyanins,rep(0,9),-1,1),
                  c(rep(0,7),0.08,rep(0,10)),
                  c(rep(0,10),62.5,62.5,0,0,0,0,2.80,2.80),
                  c(rep(0,12),0.42,0.42,rep(0,4))),ncol=18,byrow=T)
my_dir5=c(">","<",">","<",">","<",">",">","<","==","<=","<=","==","==","==","==","==","<=","<=","<=")
my_rhs5=c(3.52,3.55,8,10,35,38,7.7,17,18,1,5,0,15,59,2300,2458.85,414.91,ss2$optimum,ss3$optimum,ss4$op
ss5=Rglpk_solve_LP(obj=my_obj5,mat=my_mat5,dir=my_dir5,rhs=my_rhs5,types=my_types,max=F)
round(ss1$solution,2)
```

```
##  [1]   0.33   0.01   0.41   0.25   0.00   0.00   0.00 253.95   0.00   0.83
## [11]   0.00   1.81 343.40   0.00   0.00 579.55   0.00   0.00
```