University of Cape Town

STA5071Z

Simulation and Optimization

Optimization Assignment 2

Author:
Raisa Salie

Student Number:
SLXRAI001

January 4, 2022

Contents

1 Simulated Annealing

1.1 Sudoku Solver

1.2 Problem Formulation

We are tasked with solving a Sudoku puzzle which has a number of fixed sites. Simulated annealing will be used to attempt to solve this puzzle. We formulate the problem as a minimisation of the number of errors we obtain in a candidate solution. That is, we count the number of duplicates which occur by row, column, and block. This constitutes the evaluation function.

1.2.1 Base Case

We are given a Soduko solver which employs simulated annealing on the minimization of the number of errors in the puzzle. The initial solution is generated randomly with no accounting for duplicates in rows, columns, and blocks. The perturbation function randomly assigns a new number to the initial solution to generate a new candidate solution. The cooling schedule adopted is geometric.

The code provided produces a solution for the Sudoku puzzle with 23 errors (Base Case, Appendix 3.1). This is of course not the real solution, which has 0 errors. However, the value of the evaluation function has converged, giving the appearance of a global minima, when the algorithm has likely become stuck in a local minima. We are tasked with improving this solution by adjusting the cooling schedule, starting solution, and perturbation function.

1.2.2 Changing Cooling Schedule

The cooling schedule can be used to improve the solution. The temperature at each run determines the probability with which we accept worse candidate solutions. As temperature decreases, so does this probability. The aim of this component of the algorithm is to achieve a balance between exploration and localisation that will lead to the global minima. We will modify this component to understand which variants are best suited to the problem. The base case uses a geometric cooling schedule defined by

$$T_k = T_0 \alpha^k.$$

Alternatively, we may use the logarithmic cooling schedule given by

$$T_k = \frac{T_0}{(1 + \alpha \log(1 + k))}.$$

This was done in Case 2.1 (Appendix 3.1). However, this produced worse solutions, with more than 80 errors. The resulting plot of fitnesses is completely random, indicating that the cooling has resulted in an algorithm which is too exploratory, and fails to localise a solution adequately.

Alternatively, a quadratic multiplicative cooling schedule defined by

$$T_k = \frac{T_0}{1 + \alpha k^2}$$

may be used (Case 2.2, Appendix 3.1). This improved the solution to 14 errors. Hence, the quadratic and geometric cooling schedules appear to be the most fitting using the base settings.

1.2.3 Changing Starting Solution

The generation of the starting solution could be adjusted to enforce legality (i.e. no duplicates) across rows, columns, or blocks.

For example, we may change the initialisation function such that it ensures there are no duplicates in each row. When run with a geometric cooling schedule (Case 3.1, Appendix 3.1) additional runs were required to reach convergence. A marginal improvement was made, with 12 errors produced.

Alternatively, we can employ the same strategy in each block (Case 3.2, Appendix 3.1). That is, we enforce that the starting solution has legality within each block. This was trialled with a geometric cooling schedule and a solution of 10 errors was produced - an improvement.

1.2.4 Improving Perturbation Function

A possible explanation of the convergence to a local minima in the base case is the perturbation function. Since the role of this component of the algorithm is to escape local minima, it is reasonable to believe that herein lies the problem.

An alternative perturbation function was considered, where the adjustment of a random value is limited by the the fixed sites in the puzzle by row and column. That is, the new value in the free site is sampled from the ”legal” values given the fixed sites in its row and column (see Case 4.1, Appendix 3.1).

This was trialled with a geometric cooling schedule, and an initial solution derived from legality within blocks. The solution produced 10 errors. The same approach with a quadratic cooling schedule (Case 4.2, Appendix 3.1) required more runs to reach convergence, but produced a better solution with 8 errors. However, this configuration did not appear to explore the solution space. This coould be a problem. It appears that this may have occurred as a result of cooling occurring too quickly. Hence, it was decided to experiment with various cooling factors within this configuration. It was also decided to increase the starting temperature.

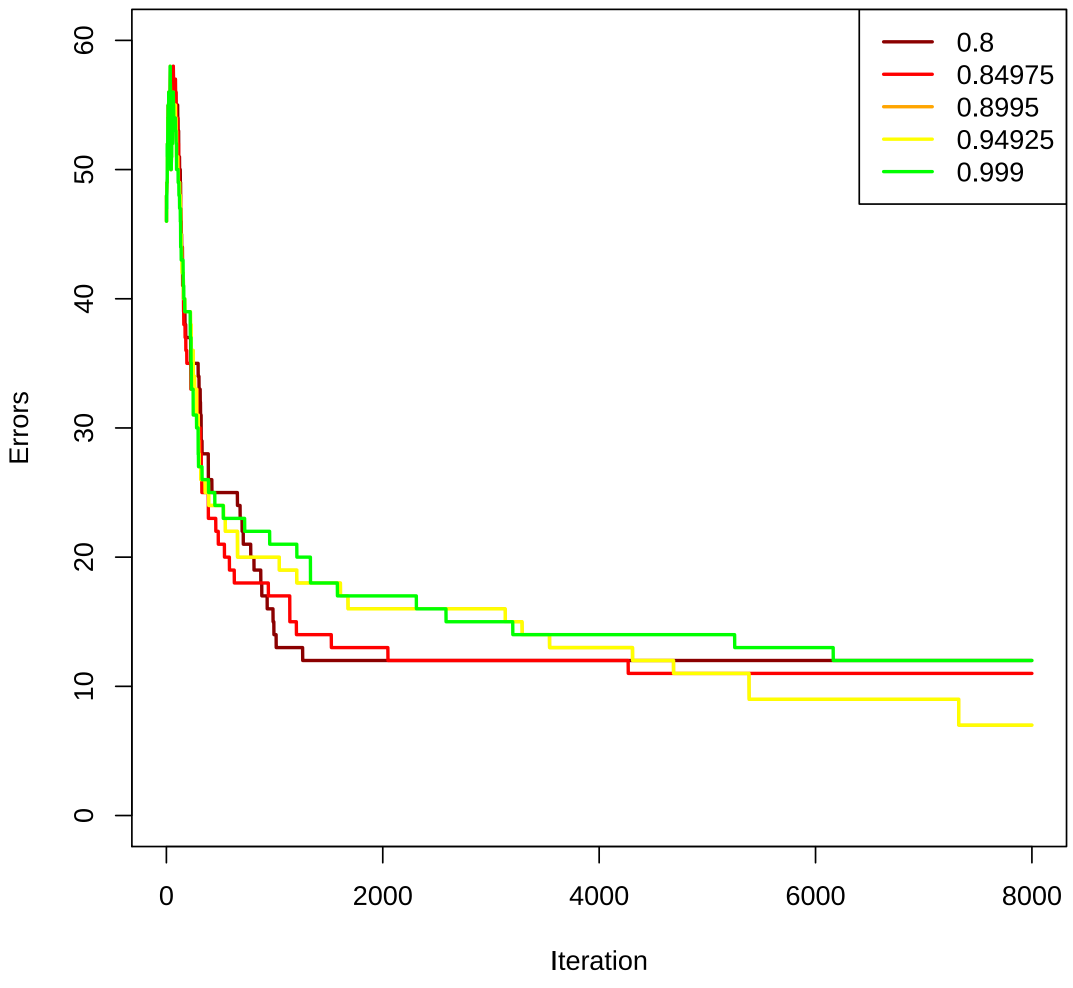

Figure 1: Sudoku solver with varying α (Case 4.3)

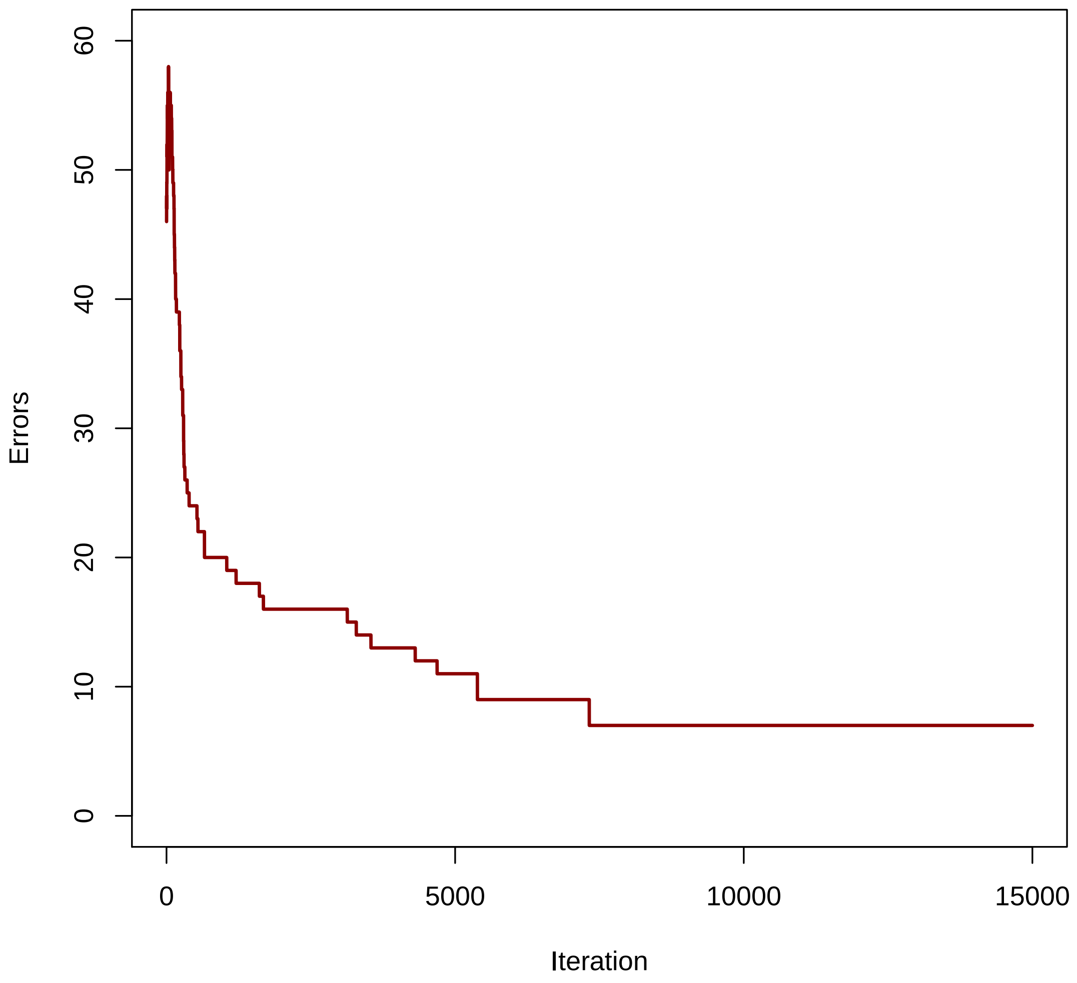

Figure 2: $\alpha = 0.94925$ run to convergence (Case 4.4)

1.2.5 Changing Cooling Rate

The best known configuration was run with 5 different cooling rates independently (Case 4.3, Appendix 3.1). This is illustrated in Figure 1. The best performing solution arises from $\alpha = 0.94925$, which seems to achieve an ideal balance between exploration and localisation. However, this chain does not appear to converge within the runs for this simulation. If explored until convergence (Figure 2), the chain produced a solution with 7 errors.

1.2.6 Brute Force Approach

After examining the cases above, it was decided to trial a brute force approach, where as many iterations as possible was used with the base case settings changing only the perturbation function to the second version (Case 5, Appendix 3.1). The results are illustrated in Figure 3. Although it is not clear that convergence is reached, the optimisation was time consuming to run as is. However, this approach led to the best solution so far with only 4 errors. Perhaps this approach would eventually lead to the global minima if given infinite number of iterations. However, this is not efficient.

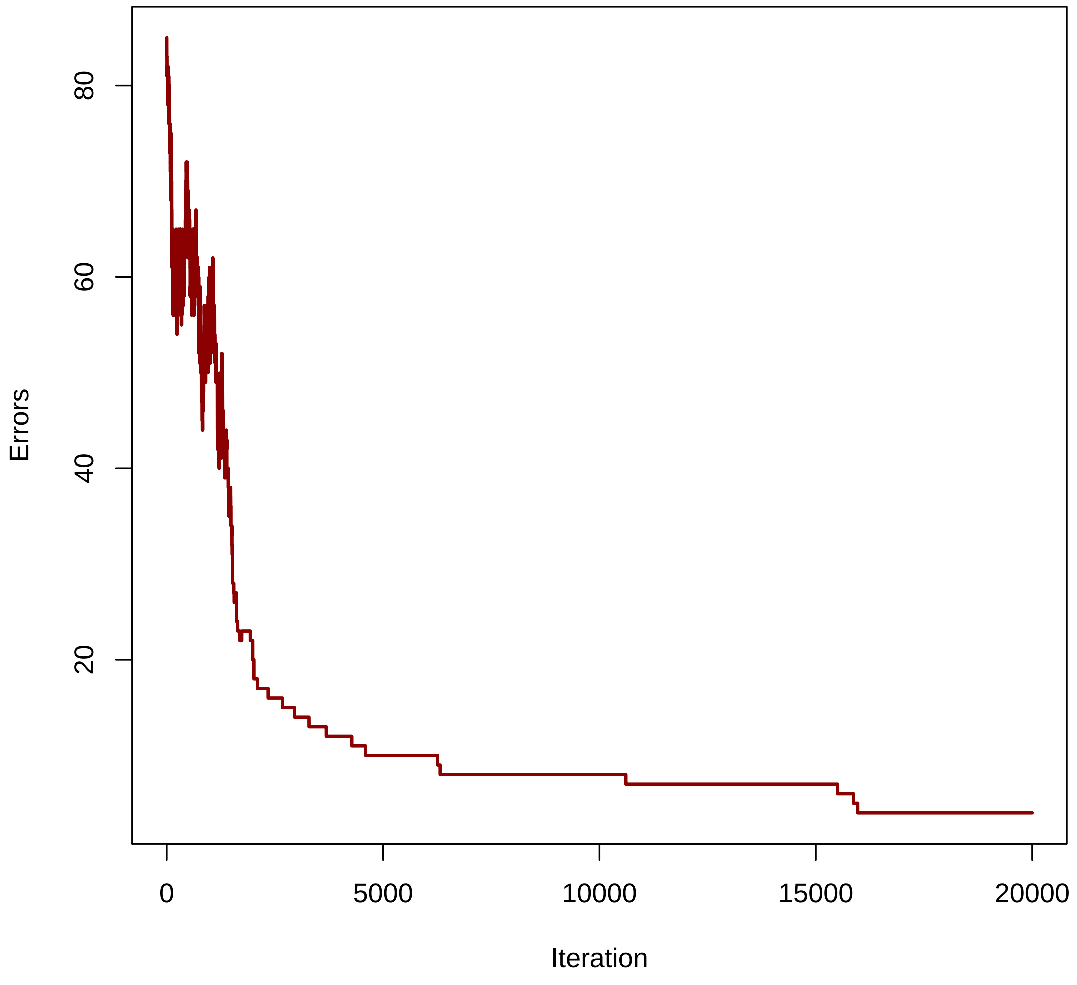

Figure 3: Brute force approach with base case settings and improved perturbation (Case 5)

1.2.7 Conclusion

The best solution obtained in the optimisations trialled had 4 errors, and arose from retaining all settings in the base case, aside from the perturbation function, which was improved. The algorithm did not converge timeously, which is a weakness.

It is postulated that simulated annealing is not the most efficient approach for solving a Sudoku puzzle using computing. Framing the problem as a nonlinear optimisation problem would guarantee a optimal solution and may have a shorter runtime. That is not to say that solving the puzzle using simulated annealing is impossible, however it was not achieved herein.

1.3 Travelling Salesman Problem

1.3.1 Problem Formulation

We are given a travelling salesman problem using the dataset `eurodist` in `R`. We are required to find the shortest tour in which all the cities are visited. This problem will be dealt with using simulated annealing. We aim to minimise the distance travelled subject to the condition that all cities are travelled.

The problem is formulated by firstly initialising a solution containing all cities. That is, a permutation containing all the numbers associated with each city exactly once, except for the first and last cities which are the same. The evaluation function calculates the total distance travelled. Two perturbation functions are considered. The first is the reversion function given by `perturb_x` in the code (Appendix 3.2). This randomly chooses two points in the permutation and reverses the string of numbers between them, including the numbers themselves. The second is the swap operator, which is given by `perturb_x2` in the code. This operator again randomly chooses two points in the string. Thereafter, these two points are swapped.

1.3.2 Base Case

We first trial the given settings of the algorithm, referred to as the base case. A staring temperature of 50 000 is used, with $\alpha = 0.98$. We use the reversion operator for perturbation. We ran this for 10 000 iterations. The base case (Case 1.1, Appendix 3.2) produces a distance of 12919. However, the solution converged quickly (see Figure 4), which means the solution space may not have been sufficiently explored, which may have resulted in the algorithm settling in a local minima. It was hence decided to increase the cooling factor, resulting in slower decrease in temperature. This would allow the algorithm to explore the solution space more before converging.

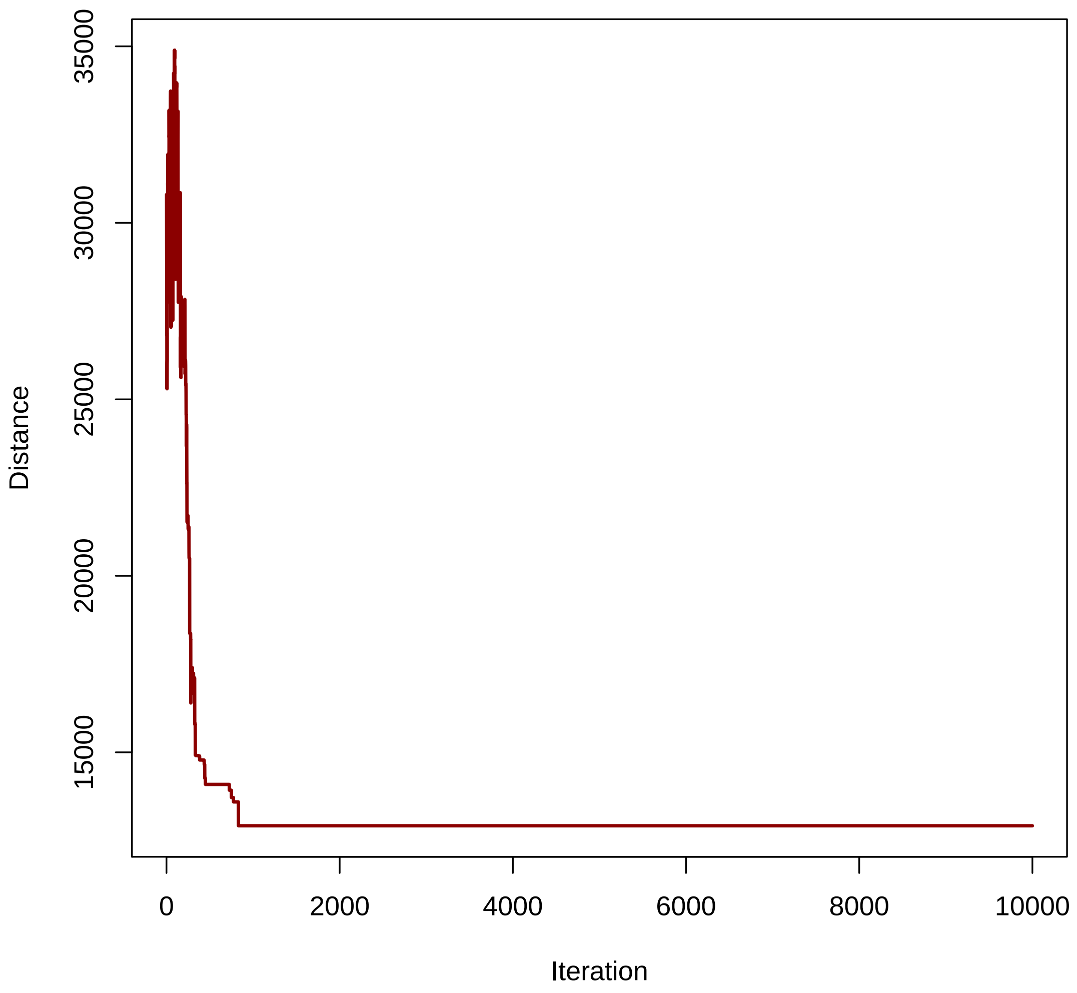

Figure 4: TSP Base Case (Case 1.1)

1.3.3 Increased Cooling Factor

For this run (Case 1.2, Appendix 3.2) we retained all the settings of the base case, and adjusted the cooling factor to $\alpha = 0.999$. This is illustrated in Figure 5. The solution space is explored more in this case compared to the base case. This produced a more convincing search of the solution space, and slightly improved the best distance to 12 842.

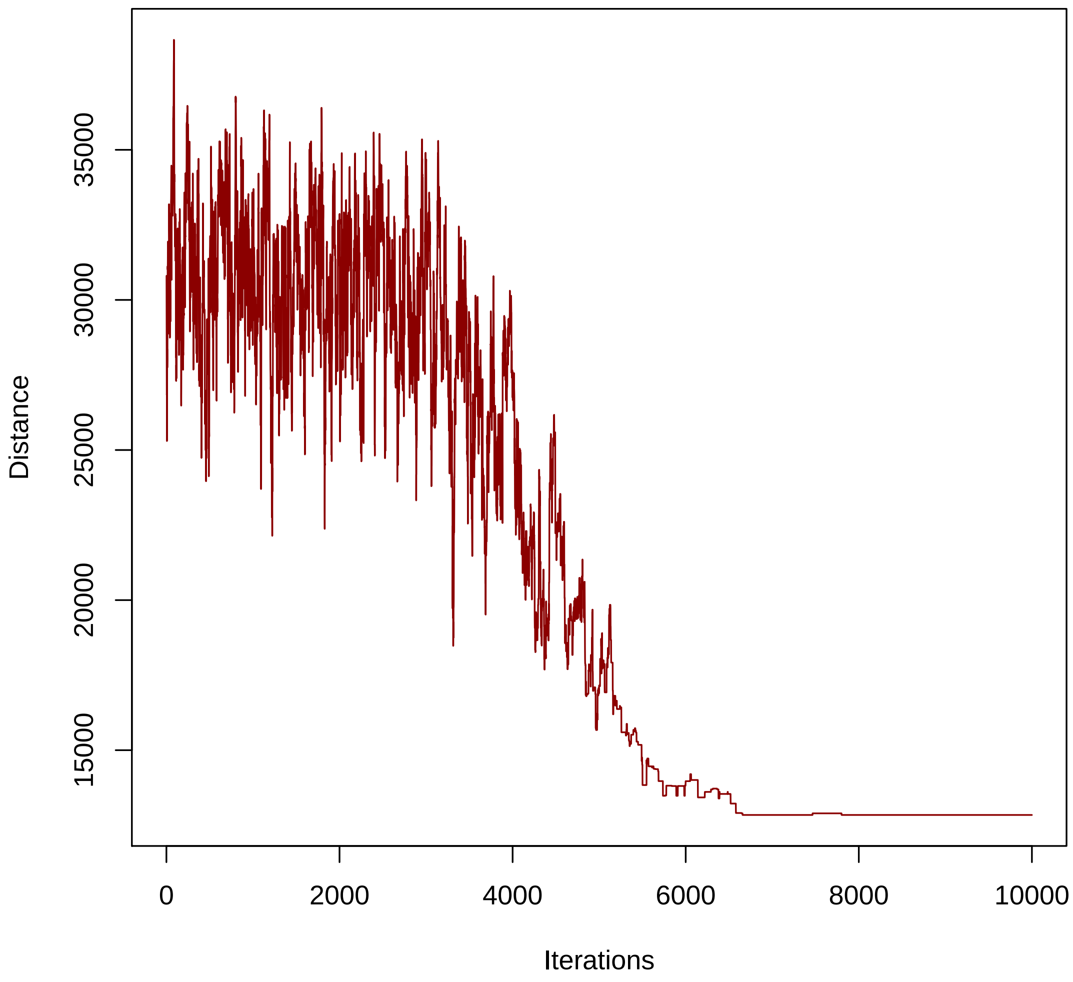

Figure 5: TSP with slower cooling (Case 1.2)

1.3.4 Other Adjustments

Other cases were examined, however none improved on the solution obtained in Case 1.2. The swap perturbation function was trialled with the same cooling (Case 2.1). However, this produced a worse solution of 13 363. An log cooling schedule was also trialled (Case 3.1) which led to a far worse distance of 20 955. Using a quadratic cooling schedule with these settings (Case 3.2) produced a distance of 12 984.

1.3.5 Conclusion

The best solution arose from the settings used in Case 1.2, corresponding to a distance of 12 842. A geometric cooling schedule was used with $\alpha = 0.999$. The starting temperature was set to 50 000. The slow cooling appeared to be the key to obtaining this minima.

2 Genetic Algorithm

2.1 Knapsack Problem

2.1.1 Problem Formulation

We are required to build a genetic algorithm to solve the knapsack problem. We will consider the 1-0 problem in which each item may be selected at most once. Suppose we have seven objects with weights and points described in Table 1. We have a weight limit of 20 kg in our knapsack. We wish to find the combination of objects for which we can get the highest points that will fit in this knapsack. Hence, the total points of a combination of objects will be considered the fitness of a solution.

Since each object may only be chosen once, we can frame possible solutions of this problem as binary strings of length 7, where 1 implies the object is taken, while 0 implies it is not. Hence, there are 2^7 possible solutions, some of which are invalid given the restriction on weight. To enforce this restriction, we will set the fitness of invalid solutions to 0, and thus circumnavigate them in the optimisation.

Table 1: Object weights and points

Object	1	2	3	4	5	6	7
Points	10	20	15	2	30	10	30
Weight (kg)	1	5	10	1	7	5	1

2.1.2 Selection Operators

Two selection operators were considered namely the proportional-to-fitness operator (`select1` in the code, Appendix 3.3), and a tournament selection variant (`select2`). The tournament selection operator selects $k = 10$ individuals randomly from the population to participate in the tournament. These individuals are then ranked according to fitness. The winner of the tournament is determined using probabilities. The individual with the best fitness wins with a probability of p, the second best with probability $p(1-p)$, the third with probability $p(1-p)^2$ and so on. The winner is added to the selection pool. This procedure is repeated until a sufficiently large pool of individuals is obtained.

2.1.3 Crossover Operators

The n-point (`crossover1`) and uniform crossover (`crossover2`) operators were considered.

2.1.4 Replacement Operators

Two replacement operators were considered. Firstly, the generational model (`crossover1`) where each population survives exactly one generation. Secondly, the steady state model (`crossover2`) where the best performing among all children and parents continue to the next generation.

2.1.5 Combining Operators

All combinations of these operators were trialled for 100 iterations. These are depicted in Figure 6. Interestingly, the best solution for 6 of the 8 chains was 102 points. The various genetic algorithm configurations shown in Figure 6 show how sensitive the chain of solutions are to the various operators used. The worst performing configurations occur in the plot in position (1,2), where tournament selection and n-point crossover were used. Perhaps these two operators are not well-suited to the problem.

2.1.6 Conclusion

The best solution included all objects except object 3. The total value of the knapsack was 102 points, with a weight of 20 kg.

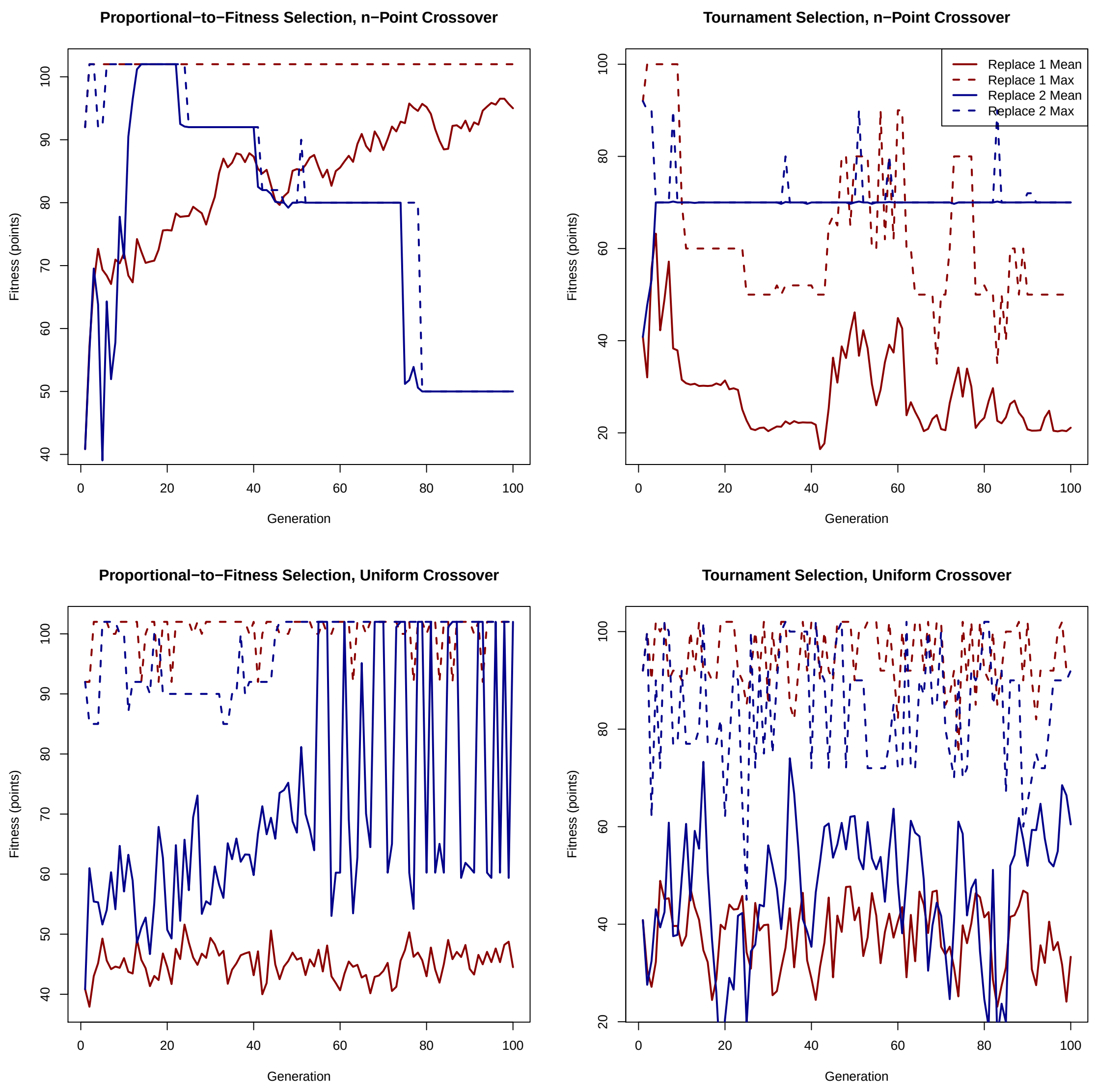

Figure 6: Knapsack problem with varying selection, crossover, and replacement operators

3 Appendix

3.1 Sudoku Solver R Code

```
rm(list = ls(all=T))
# Solving Sudoku
# create a Sudoku puzzle
s <- matrix(0,ncol=9,nrow=9)
s[1,c(6,8)] <- c(6,4)
s[2,c(1:3,8)] <- c(2,7,9,5)
s[3,c(2,4,9)] <- c(5,8,2)
s[4,3:4] <- c(2,6)
s[6,c(3,5,7:9)] <- c(1,9,6,7,3)
s[7,c(1,3:4,7)] <- c(8,5,2,4)
s[8,c(1,8:9)] <- c(3,8,5)
s[9,c(1,7,9)] <- c(6,9,1)
s

# free spaces
free_spaces <- (s == 0)
free_spaces

# get init sol
get_initial_x <- function(s){
  # identify free spaces
  free_spaces <- (s == 0)
  # fixed non-free spaces
  cur_x <- s
  # randomly choose a number between 1 and 9 for free spaces
  cur_x[free_spaces] <- sample(1:9, sum(free_spaces), replace=T)
  return(cur_x)
}

# init sol
init_x <- get_initial_x(s)
init_x

# function to count duplicates
count_duplicates <- function(x) {
  n_dup <- length(x) - length(unique(x))
  return(n_dup)
}

# eval solution
evaluate_x <- function(x){ # within-row duplications
  row_dups <- sum(apply(x,1,count_duplicates)) # within-col duplications
  col_dups <- sum(apply(x,2,count_duplicates))
  # within-block duplications
  block_dups <- 0
  for (i in 1:3){
    for(j in 1:3){
      small_x <- x[(3*(i-1) + 1):(3*i), (3*(j-1) + 1):(3*j)]
      thisblock_dups <- count_duplicates(as.vector(small_x))
      block_dups <- block_dups + thisblock_dups
    }
  }
  total_dups <- row_dups + col_dups + block_dups
  return(total_dups)
}
# eval init sol
evaluate_x(init_x)

# perturb function
perturb_x <- function(x, free_spaces) {
  # select a free site
  site_to_change <- sample(1:81,1,prob=free_spaces)
  # change that site at random
  x[site_to_change] <- sample(1:9,1,replace=T)
  return(x)
}

# simulatiion
set.seed(100) # for repeatability
start_temp <- 1e3
temp_factor <- 0.995

cur_x <- get_initial_x(s)
cur_fx <- evaluate_x(cur_x)
cur_x
cur_fx
```

```
##############################
## Base Case
# initial results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:3000){
# generate a candidate solution
prop_x <- perturb_x(cur_x, free_spaces = free_spaces)
# 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 <- rbind(all_x,as.vector(cur_x))
}

plot(all_fx,type="l")

plot(1:3000,start_temp * temp_factor ^ (1:3000))
tail(all_fx) # 23 errors
############################################################
### CHANGES
##############################################
## Case 2.1
## logarithmic cooling sched
# initial results data frames
cur_x <- get_initial_x(s)
all_fx <- c()
```

```
all_x <- data.frame()
# for a fixed number of iterations
set.seed(100)
for(i in 1:3000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x, free_spaces = free_spaces) # evaluate the candidate so
  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 <- rbind(all_x,as.vector(cur_x))
}

plot(all_fx,type="l")

plot(1:3000,start_temp * temp_factor ^ (1:3000))
tail(all_fx) # 80 + errors

############################################
## Case 2.2
## Quadratic cooling sched
# initial results data frames
cur_x <- get_initial_x(s)
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
set.seed(100)
for(i in 1:3000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x, free_spaces = free_spaces) # evaluate the candidate so
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp/(1+temp_factor*(i^2))
  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 <- rbind(all_x,as.vector(cur_x))
}

plot(all_fx,type="l")

plot(1:3000,start_temp * temp_factor ^ (1:3000))
tail(all_x)
cur_x
tail(all_fx) # 14 errors

#########################################
## Case 3.1
# improve initial solution by row
get_initial_x_rows <- function(s){
  # identify free spaces
  free_spaces <- (s == 0)
  # initialize mnatrix to return
  cur_x <- matrix(NA, nrow = 9, ncol = 9)

  # for each row
  for (i in 1:nrow(s)){
    # the row
    row <- s[i,]
    # which spaces in row are free?
    free <- free_spaces[i,]
    # which numbers are not used in row?
    # if all zeroes , can use 1,..,9
    if(sum(free)==length(row)){not_taken <- c(1:9)}else{
      # if not all zeroes, can use those not in row
      not_taken <- c(1:9)[-row[!free]]
    }
    # replace open spaces with available numbers
    row[free] <- sample(x = not_taken, size = sum(free), replace = F)
    # replace in matrix to return
    cur_x[i,] <- row
  }
  return(cur_x)
}

# rerun with this adjustment
# simulatiion
set.seed(100) # for repeatability
start_temp <- 1e3
temp_factor <- 0.995

cur_x <- get_initial_x_rows(s)
cur_fx <- evaluate_x(cur_x)

# initial results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:6000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x, free_spaces = free_spaces)
  # 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 <- rbind(all_x,as.vector(cur_x))
}

plot(all_fx,type="l")

plot(1:6000,start_temp * temp_factor ^ (1:6000))
tail(all_fx) # 12 errors - improvement

########################################
## Case 3.2
# improve initial solution by block
get_initial_x_blocks <- function(s){
  # initialize mnatrix to return
  cur_x <- matrix(NA, nrow = 9, ncol = 9)

  for (b in 1:3){
    for (a in 1:3){
      # for each block
      block <- s[(3*b-2):(3*b),(3*a-2):(3*a)]
      # identify free spaces
      free <- (block == 0)
      # which numbers are not taken
      # if all zeroes , can use 1,..,9
      if(sum(free)==9){not_taken <- c(1:9)}else{
        # if not all zeroes, can use those not in row
        not_taken <- c(1:9)[-block[!free]]
      }
      # replace open spaces with available numbers
      block[free] <- sample(x = not_taken, size = sum(free), replace = F)
      # replace in matrix to return
      cur_x[(3*b-2):(3*b),(3*a-2):(3*a)] <- block
    }
  }
  return(cur_x)
}

# rerun with this adjustment
# simulatiion
set.seed(100) # for repeatability
start_temp <- 1e3
temp_factor <- 0.995

cur_x <- get_initial_x_blocks(s)
cur_fx <- evaluate_x(cur_x)

# initial results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:6000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x, free_spaces = free_spaces)
  # 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 <- rbind(all_x,as.vector(cur_x))
}

plot(all_fx,type="l")
```

```r
plot(1:6000,start_temp * temp_factor ^ (1:6000))
tail(all_fx) # 10 errors

########################################
## Case 4.1
# perturb function - second version
perturb_x2 <- function(x, free_spaces) {
  # select a free site
  # choose a row
  i <- sample(1:9,1)
  # choose a col (where there is a free site ij)
  j <- sample(1:9,1,prob=free_spaces[i,])
  # row of site
  row <- s[i,]
  # column of site
  col <- s[,j]
  # which numbers can we choose from?
  taken <- c(row, col)[-which(c(row,col)==0)]

  # change that site to a legal value given fixed sites
  x[i,j] <- sample(c(1:9)[-taken],1)

  return(x)
}

# simulatiion
set.seed(100) # for repeatability
start_temp <- 1e3
temp_factor <- 0.995

cur_x <- get_initial_x_blocks(s)
cur_fx <- evaluate_x(cur_x)

# initial results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:3000){
  # generate a candidate solution
  prop_x <- perturb_x2(cur_x, free_spaces = free_spaces)
  # 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 <- rbind(all_x,as.vector(cur_x))
}

plot(all_fx,type="l")

plot(1:3000,start_temp * temp_factor ^ (1:3000))
tail(all_fx) # 10 errors
################################
## Case 4.2
# as 4.1 with quadratic cooling sched
# simulatiion
set.seed(100) # for repeatability
start_temp <- 1e3
temp_factor <- 0.995

cur_x <- get_initial_x_blocks(s)
cur_fx <- evaluate_x(cur_x)

# initial results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:6000){
  # generate a candidate solution
  prop_x <- perturb_x2(cur_x, free_spaces = free_spaces)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp/(1+temp_factor*(i^2))
  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 <- rbind(all_x,as.vector(cur_x))
}

plot(all_fx,type="l")

plot(1:6000,start_temp * temp_factor ^ (1:6000))
tail(all_fx) # 8 errors
##########################################
## Case 4.3
# as 4.1 with quadratic cooling sched
# try varying alphas
alphas <- seq(0.8, 0.999, length.out = 5)
# storage of results
res <- list()
for (k in 1:length(alphas)){
  # simulatiion
  set.seed(100) # for repeatability
  start_temp <- 5e3
  temp_factor <- alphas[k]

  cur_x <- get_initial_x_blocks(s)
  cur_fx <- evaluate_x(cur_x)
  # initial results data frames
  all_fx <- c()
  all_x <- data.frame()
  # for a fixed number of iterations
  for(i in 1:8000){
    # generate a candidate solution
    prop_x <- perturb_x2(cur_x, free_spaces = free_spaces)
    # evaluate the candidate solution
    prop_fx <- evaluate_x(prop_x)
    # calculate the probability of accepting the candidate
    anneal_temp <- start_temp/(1+temp_factor*(i^2))
    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 <- rbind(all_x,as.vector(cur_x))
  }
  # add to results
  res[[k]] <- all_fx
}

cols <- c("darkred", "red", "orange", "yellow", "green ",
          "blue", "darkblue", "purple", "violet", "pink")
pdf(file = "Case43.pdf", compress = F, width = 7, height=7)
plot(x = c(1:8000), y = res[[1]],type="l", col=cols[1], ylim = c(0,60), lwd=2,
     xlab="Iteration", ylab="Errors")
for (i in 2:length(res)){
  lines(res[[i]], col=cols[i], lwd=2)
}
legend("topright",legend = paste(alphas), col = cols, lty = 1, lwd = 2)
dev.off()
```

```
# plot temps
X <- c(1:5000)
plot(x = X, y = start_temp/(1+alphas[1]*(X^2)), type="l", col=cols[1], lwd=2,
     xlab="Run", ylab="Temperature", ylim=c(0,500))
for (i in 2:length(alphas)){
  lines(x=X, y = start_temp/(1+alphas[i]*(X^2)), lwd=2, col=cols[i])
}
##########################################
## Case 4.4
# choose best alpha = 0.94925

  # simulatiion
  set.seed(100) # for repeatability
  start_temp <- 5e3
  temp_factor <- alphas[4]

  cur_x <- get_initial_x_blocks(s)
  cur_fx <- evaluate_x(cur_x)

  # initial results data frames
  all_fx <- c()
  all_x <- data.frame()
  # for a fixed number of iterations
  for(i in 1:15000){
    # generate a candidate solution
    prop_x <- perturb_x2(cur_x, free_spaces = free_spaces)
    # evaluate the candidate solution
    prop_fx <- evaluate_x(prop_x)
    # calculate the probability of accepting the candidate
    anneal_temp <- start_temp/(1+temp_factor*(i^2))
    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 <- rbind(all_x,as.vector(cur_x))
  }

pdf(file = "Case44.pdf", compress = F, width = 7, height=7)
plot(x = c(1:15000), y = all_fx,type="l", col=cols[1], ylim = c(0,60), lwd=2,
     xlab="Iteration", ylab="Errors")
dev.off()
all_fx[which.min(all_fx)]

####################################
## Case 5
# brute force
# simulatiion
set.seed(100) # for repeatability
start_temp <- 1e3
temp_factor <- 0.995

cur_x <- get_initial_x(s)
cur_fx <- evaluate_x(cur_x)

# initial results data frames
all_fx <- c()
all_x <- data.frame()
# for a fixed number of iterations
for(i in 1:20000){
  # generate a candidate solution
  prop_x <- perturb_x2(cur_x, free_spaces = free_spaces)
  # 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 <- rbind(all_x,as.vector(cur_x))
}

pdf(file = "Case5.pdf", width = 7, height = 7, compress = F)
plot(all_fx,type="l", xlab="Iteration", ylab = "Errors", lwd=2, col = cols[1])
dev.off()

plot(1:20000,start_temp * temp_factor ^ (1:20000))

all_fx[which.min(all_fx)]
# 4 errors with 20000 runs
#save.image(file = "sudoku_pert2.RData")

#########################
## Case 6 - not included
## brute force with adjustments
# simulatiion
set.seed(100) # for repeatability
start_temp <- 5e3
temp_factor <- alphas[4]

cur_x <- get_initial_x_blocks(s)
cur_fx <- evaluate_x(cur_x)

# initial results data frames
all_fx2 <- c()
all_x2 <- data.frame()
# for a fixed number of iterations
for(i in 1:20000){
  # generate a candidate solution
  prop_x <- perturb_x2(cur_x, free_spaces = free_spaces)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp/(1+temp_factor*(i^2))
  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_fx2 <- c(all_fx2, cur_fx)
  all_x2 <- rbind(all_x2, as.vector(cur_x))
}

pdf(file = "Case6.pdf", compress = F, width = 7, height=7)
plot(x = c(1:20000), y = all_fx2,type="l", col=cols[1], ylim = c(0,60), lwd=2,
     xlab="Iteration", ylab="Errors")
dev.off()
all_fx2[which.min(all_fx2)]
##############END
```

3.2 Travelling Salesman Problem R Code

```
# TSP
rm(list = ls(all=T))
# load data
y <- as.matrix(eurodist)
# defining functions
# get intital solution
get_initial_x <- function(ncity){
  tour <- sample(1:ncity)
  # ensure tour returns to first city
  tour <- c(tour, tour[1])
  return(tour)
}
cur_tour <- get_initial_x(nrow(y))
cur_tour

# evaluation function
```

```
evaluate_x <- function(dists, tour){
  sum(dists[cbind(tour[-length(tour)], tour[-1])])
}
evaluate_x(dists = y, tour = cur_tour)

# perturbation functions
# reverse operator
perturb_x <- function(tour){
  # select two cities at random
  ch <- sort(sample(2:(length(tour)-1),2,replace=FALSE))
  # reconnect other way
  tour[ch[1]:ch[2]] <- tour[ch[2]:ch[1]]
  return(tour)
}

#############################
### Simulation
# Case 1.1 (base case)
# set start temperature and geometric cooling factor
set.seed(100) # for repeatability
start_temp <- 50000
temp_factor <- 0.98
# get an initial solution
cur_x <- get_initial_x(ncol(y))
# evaluate the solution
cur_fx <- evaluate_x(dists = y, tour = cur_x)
# initialize results data frames
all_fx <- c()
all_x <- data.frame()
all_temp <- c()

# for a fixed number of iterations
set.seed(2020)
for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(dists = y, tour = 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 <- rbind(all_x,cur_x)
  all_temp <- c(all_temp, anneal_temp)
}

# plot
pdf(file = "TSPCase11.pdf", width = 7, height = 7, compress = F)
plot(all_fx, type="l", lwd=2, col="darkred",
     xlab="Iteration", ylab="Distance") # converges quickly
dev.off()
plot(all_temp, type="l")
# results
best_solution <- which.min(all_fx)
best_tour <- all_x[best_solution,]
optimal_solution <- all_fx[best_solution]
optimal_solution #12 919
best_tour
colnames(y)[as.numeric(best_tour)]
###############################
## Case 1.2
# explore feature space more
# increase cooling factor
# set start temperature and geometric cooling factor
set.seed(100) # for repeatability
start_temp <- 50000
temp_factor <- 0.999
# get an initial solution
cur_x <- get_initial_x(ncol(y))
```

```
# evaluate the solution
cur_fx <- evaluate_x(dists = y, tour = cur_x)
# initialize results data frames
all_fx <- c()
all_x <- data.frame()
all_temp <- c()

# for a fixed number of iterations
set.seed(2020)
for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(dists = y, tour = 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 <- rbind(all_x,cur_x)
  all_temp <- c(all_temp, anneal_temp)
}

# plot
pdf(file = "TSPCase12.pdf", width = 7, height = 7, compress = F)
plot(all_fx,type="l", xlab="Iterations", ylab="Distance", col = "darkred")
dev.off()
# plot temp
plot(all_temp, type="l") # slower decrease
# results
```

```
best_solution <- which.min(all_fx)
best_tour <- all_x[best_solution,]
optimal_solution <- all_fx[best_solution]
optimal_solution #12 842
best_tour
colnames(y)[as.numeric(best_tour)]

################################
## Case 2.1
# use swap perturbation function
# swap operator
perturb_x2 <- function(tour){
  # select two cities at random
  ch <- sort(sample(2:(length(tour)-1),2,replace=FALSE))
  # swap them in the chain
  p1 <- tour[ch[1]]
  tour[ch[1]] <- tour[ch[2]]
  tour[ch[2]] <- p1
  return(tour)
}
# set start temperature and geometric cooling factor
set.seed(100) # for repeatability
start_temp <- 50000
temp_factor <- 0.999
# get an initial solution
cur_x <- get_initial_x(ncol(y))
# evaluate the solution
cur_fx <- evaluate_x(dists = y, tour = cur_x)
# initialize results data frames
all_fx <- c()
all_x <- data.frame()
all_temp <- c()

# for a fixed number of iterations
set.seed(2020)
for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x2(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(dists = y, tour = 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 <- rbind(all_x,cur_x)
  all_temp <- c(all_temp, anneal_temp)
}

# plot
plot(all_fx,type="l") # convergence is quick
plot(all_temp,type="l")
# results
best_solution <- which.min(all_fx)
best_tour <- all_x[best_solution,]
optimal_solution <- all_fx[best_solution]
optimal_solution # 13 363
best_tour
colnames(y)[as.numeric(best_tour)]
############################
## Case 3.1
# same settings as case 1.2 but log cooling schedule
set.seed(100) # for repeatability
start_temp <- 50000
temp_factor <- 0.999
# get an initial solution
cur_x <- get_initial_x(ncol(y))
# evaluate the solution
cur_fx <- evaluate_x(dists = y, tour = cur_x)
# initialize results data frames
```

```
all_fx <- c()
all_x <- data.frame()
all_temp <- c()

# for a fixed number of iterations
set.seed(2020)
for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(dists = y, tour = 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 <- rbind(all_x,cur_x)
  all_temp <- c(all_temp, anneal_temp)
}

# plot
plot(all_fx,type="l") # convergence is quick
plot(all_temp, type="l")
# results
best_solution <- which.min(all_fx)
best_tour <- all_x[best_solution,]
optimal_solution <- all_fx[best_solution]
optimal_solution # 20 955 - worse
best_tour
colnames(y)[as.numeric(best_tour)]
```

```
############################
## Case 3.2
# same settings as case 1.2 but log cooling schedule
set.seed(100) # for repeatability
start_temp <- 50000
temp_factor <- 0.999
# get an initial solution
cur_x <- get_initial_x(ncol(y))
# evaluate the solution
cur_fx <- evaluate_x(dists = y, tour = cur_x)
# initialize results data frames
all_fx <- c()
all_x <- data.frame()
all_temp <- c()

# for a fixed number of iterations
set.seed(2020)
for(i in 1:10000){
  # generate a candidate solution
  prop_x <- perturb_x(cur_x)
  # evaluate the candidate solution
  prop_fx <- evaluate_x(dists = y, tour = prop_x)
  # calculate the probability of accepting the candidate
  anneal_temp <- start_temp/(1+temp_factor*(i^2))
  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 <- rbind(all_x,cur_x)
  all_temp <- c(all_temp, anneal_temp)
}
```

```
# plot
plot(all_fx,type="l") # convergence is quick
plot(all_temp, type="l")
# results
best_solution <- which.min(all_fx)
best_tour <- all_x[best_solution,]
optimal_solution <- all_fx[best_solution]
optimal_solution # 12 984
best_tour
colnames(y)[as.numeric(best_tour)]
########## END
```

3.3 Knapsack Problem R Code

```
# function to generate initial population
initial_pop <- function(varsize, popsize){
  # a random binary matrix with varsize columns, popsize rows
  matrix(data = sample(c(0,1),varsize*popsize,replace = T),
         nrow = popsize, ncol = varsize)
}

# fitness function - calculate number of points
evaluate_pop <- function(pop, item_points, item_weight, weightlimit=20){
  # find fitness of each member of population
  fit <- pop%*%item_points
  # find weight of each member
  weight <- pop%*%item_weight
  # set fitness=0 when weight exceeds weight limit
  fit[weight > weightlimit] <- 0

  return(fit)
}

# selection functions
# proportional-to-fitness selection
select1 <- function(pop,fitness){
  # total fitness
  totfit <- sum(fitness)
  # probabilities
  probi <- fitness/totfit
  # expected count
  exp <- probi*length(fitness)
  # actual count
  actual <- round(exp,digits = 0)

  # parents index (incl selection frequency)
  parents_ind <- sample(1:nrow(pop), size = nrow(pop),
                        prob = actual, replace = T)

  # return parents
  pop[parents_ind,]
}

# tournament selection (ordinal) - incomplete
select2 <- function(pop,fitness=evals,p=0.3,k=10){
  # initialise individuals to return
  selected_inds <- matrix(NA, nrow = nrow(pop), ncol = ncol(pop))

  for (i in 1:nrow(selected_inds)){
    # select k members of population for tournament
    ind <- sample(1:nrow(pop), size = k)
    tournament <- pop[ind,]
    # get their fitness
    fit <- sort(fitness[ind], decreasing=T)
    # rank them
    tourn_ordered <- tournament[order(fit,decreasing = T),]
    # select best with prob p
    # 2nd best with prob p*(1-p)
    # 3rd best with prob p*(1-p)^2 etc.
    # probabilities
    probs <- p*(1-p)^(c(0:(k-1)))
    # choose individual to return
    new_ind <- pop[sample(1:k,size =1, prob = probs),]
    # add to selected inds
    selected_inds[i,] <- new_ind
  }

  return(selected_inds)
}
```

```
# crossover functions
# n point crossover
crossover1 <- function(parents, n=1){
  # crossover points
  cross_points <- sample(1:(ncol(parents)-1), size = nrow(parents)/2, replace = T)

  # matrix of offspring generated
  offspring <- matrix(NA,nrow = nrow(parents), ncol = ncol(parents))

  # for each pair
  for (i in 1:50){
    # parents
    p1 <- parents[(2*i-1),]
    p2 <- parents[(2*i),]

    # offspring
    new1 <- p1
    new1[(cross_points[i]+1):length(p1)] <- p2[(cross_points[i]+1):length(p1)]

    new2 <- p2
    new2[(cross_points[i]+1):length(p2)] <- p1[(cross_points[i]+1):length(p2)]

    offspring[(2*i-1),] <- new1
    offspring[(2*i),] <- new2
  }

  return(offspring)
}

# uniform crossover
crossover2 <- function(parents){
  # storage of offspring
  offspring <- matrix(NA, nrow = nrow(parents), ncol = ncol(parents))
  # ind of parent pairs
  ind <- sample(1:100, size = 100, replace = F)
  for (i in 1:50){
    # first pair of parents
    p1 <- parents[2*i-1,]
    p2 <- parents[2*i,]
    # flip a coin for each gene
    coin <- sample(c(0,1),ncol(parents),replace = T)
    # new offspring
    # 1=p1
    # 0=p2
    new1 <- p1
    new1[!as.logical(coin)] <- p2[!as.logical(coin)]
    # other offspring is inverse of new1
    new2 <- as.numeric(!as.logical(new1))

    # add new1, new2 to offspring matrix
    offspring[2*i-1,] <- new1
    offspring[2*i,] <- new2
  }
  return(offspring)
}

# mutation functions
# local modification
mutation <- function(pop, mutation_rate){
  # number to mutate
  n <- nrow(pop)*mutation_rate
  # get positions of mutations
  positions <- matrix(c(sample(1:nrow(pop),n), sample(1:ncol(pop),n)),
                      ncol=2,nrow=n)
  colnames(positions) <- c("i", "j")

  # generate mutated offspring
  for (p in 1:n){
    # number to mutate
    site <- pop[positions[p,1], positions[p,2]]

    # replace in population with opposite
    if (site==1){pop[positions[p,1], positions[p,2]] <-0}else{
      pop[positions[p,1], positions[p,2]] <- 1
    }

  }

  return(pop)
```

```r
}

# replacement functions
# generational model - GGA
# each member survives one generation
replace1 <- function(parents, offspring){
  return(offspring)
}

# SSGA
# replace some of the parent population
replace2 <- function(parents, offspring, fitness=evals){
  # get the best 100 out of parents and offspring
  all <- rbind(parents, offspring)
  # get fitness of offspring
  fit <- evaluate_pop(pop = offspring, item_points = pts, item_weight = wts)
  # all fitness
  fitness_all <- c(fitness, fit)
  # rank fitness values
  # rank gives 1 to min
  ranks <- rank(fitness_all)
  # ordered chromosomes
  # smallest fitness at top
  all_ord <- all[ranks,]
  # select best 100
  # last 100 in all_ord matrix
  return(all_ord[c((nrow(parents)+1):nrow(all)),])
}

# setup
pts = c(10, 20, 15, 2, 30,10, 30)
wts = c(1, 5, 10, 1, 7, 5, 1)
n_vars = length(wts)
n_pop = 100
# starting pop
set.seed(2020)
start <- initial_pop(varsize=n_vars,popsize=n_pop)

##############################
# storage for results of all runs
```

```r
res <- list()

## RUN 1
# PFS slection (1), n-point crossover
this_pop = start
weightlimit = 20
max_evals = c()
mean_evals = c()
max_evals2 = c()
mean_evals2 = c()
n_gen = 100
individ = matrix(NA, ncol=n_vars, nrow=n_gen)


# first replacement function
set.seed(2020)
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select1(pop=this_pop,fitness=evals)
  offspring_crossover = crossover1(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  #store best individ
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace1(parents=next_parents, offspring=offspring_mutation)
  max_evals = c(max_evals,max(evals))
  mean_evals = c(mean_evals,mean(evals))
}
# 2nd replacement function
this_pop = start
individ = matrix(NA, ncol=n_vars, nrow=n_gen)
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select1(pop=this_pop,fitness=evals)
  offspring_crossover = crossover1(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  #store best individ
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace2(parents=next_parents, offspring=offspring_mutation)
  max_evals2 = c(max_evals2,max(evals))
  mean_evals2 = c(mean_evals2,mean(evals))
```

```
}

# store results
res[[1]] <- cbind(mean_evals,max_evals, individ)
res[[2]] <- cbind(mean_evals2,max_evals2, individ)
names(res) <- paste("graph 1, replacement", c(1,2))
cols <- c("darkred", "darkblue")

## plot
pdf(file = "allGA.pdf", width = 12, height = 12, compress = F)
par(mfrow=c(2,2))
#pdf(file = "GAgraph1.pdf",width = 7, height = 7, compress = F)
# first replacement
plot(1:n_gen,mean_evals, type='l', xlab="Generation", ylab="Fitness (points)" ,
     ylim=c(min(c(max_evals,mean_evals)),max(c(max_evals,mean_evals))),
     col=cols[1], lwd=2,
     main = "Proportional-to-Fitness Selection, n-Point Crossover")
lines(1:n_gen,max_evals,lty=2, col=cols[1], lwd=2)
# second replacement
lines(1:n_gen, mean_evals2, type='l', xlab="Generation", ylab="Fitness (points)" ,
      ylim=c(min(c(max_evals2,mean_evals2)),max(c(max_evals2,mean_evals2))),
      col=cols[2], lwd=2)
lines(1:n_gen,max_evals2,lty=2, col=cols[2], lwd=2)
#legend("bottom", lwd=2, legend = paste("Replace", c(1,1,2,2), rep(c("Mean", "Max
#dev.off()

#####################################
## RUN 2
# tournament selection (2), n-point crossover(1)
this_pop = start
max_evals = c()
mean_evals = c()
max_evals2 = c()
mean_evals2 = c()
individ = matrix(NA, ncol=n_vars, nrow=n_gen)

# first replacement function
set.seed(2020)
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select2(pop=this_pop,fitness=evals)
  offspring_crossover = crossover1(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace1(parents=next_parents, offspring=offspring_mutation)
  max_evals = c(max_evals,max(evals))
  mean_evals = c(mean_evals,mean(evals))
}
# 2nd replacement function
this_pop = start
individ = matrix(NA, ncol=n_vars, nrow=n_gen)
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select2(pop=this_pop,fitness=evals)
  offspring_crossover = crossover1(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace2(parents=next_parents, offspring=offspring_mutation)
  max_evals2 = c(max_evals2,max(evals))
  mean_evals2 = c(mean_evals2,mean(evals))
}

# store results
res[[3]] <- cbind(mean_evals,max_evals,individ)
res[[4]] <- cbind(mean_evals2,max_evals2,individ)
names(res)[c(3,4)] <- paste("graph 2, replacement", c(1,2))

## plot
#pdf(file = "GAgraph2.pdf",width = 7, height = 7, compress = F)
# first replacement
plot(1:n_gen,mean_evals, type='l', xlab="Generation", ylab="Fitness (points)" ,
     ylim=c(min(c(max_evals,mean_evals)),max(c(max_evals,mean_evals))),
     col=cols[1], lwd=2,
     main = "Tournament Selection, n-Point Crossover")
lines(1:n_gen,max_evals,lty=2, col=cols[1], lwd=2)
# second replacement
lines(1:n_gen, mean_evals2, type='l', xlab="Generation", ylab="Fitness (points)" ,
      ylim=c(min(c(max_evals2,mean_evals2)),max(c(max_evals2,mean_evals2))),
      col=cols[2], lwd=2)
lines(1:n_gen,max_evals2,lty=2, col=cols[2], lwd=2)
```

```r
legend("topright", lwd=2,
       legend = paste("Replace", c(1,1,2,2), rep(c("Mean", "Max"),2)),
       lty = rep(c(1,2),2), col = c(cols[1],cols[1],cols[2],cols[2]))
#dev.off()
#####################################
## RUN 3
# PFS selection (1), n-point crossover(2)
# more runs were required
this_pop = start
max_evals = c()
mean_evals = c()
max_evals2 = c()
mean_evals2 = c()
individ = matrix(NA, ncol=n_vars, nrow=n_gen)

# first replacement function
set.seed(2020)
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select1(pop=this_pop,fitness=evals)
  offspring_crossover = crossover2(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  # store best individ
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace1(parents=next_parents, offspring=offspring_mutation)
  max_evals = c(max_evals,max(evals))
  mean_evals = c(mean_evals,mean(evals))
}
# 2nd replacement function
this_pop = start
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select1(pop=this_pop,fitness=evals)
  offspring_crossover = crossover2(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  # store best individ
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace2(parents=next_parents, offspring=offspring_mutation)
  max_evals2 = c(max_evals2,max(evals))
  mean_evals2 = c(mean_evals2,mean(evals))
```

```
}

# store results
res[[5]] <- cbind(mean_evals,max_evals, individ)
res[[6]] <- cbind(mean_evals2,max_evals2, individ)
names(res)[c(5,6)] <- paste("graph 3, replacement", c(1,2))

## plot
#pdf(file = "GAgraph3.pdf",width = 7, height = 7, compress = F)
# first replacement
plot(1:n_gen, mean_evals, type='l', xlab="Generation", ylab="Fitness (points)" ,
     ylim=c(min(c(max_evals,mean_evals)),max(c(max_evals,mean_evals))),
     col=cols[1], lwd=2,
     main = "Proportional-to-Fitness Selection, Uniform Crossover")
lines(1:n_gen,max_evals,lty=2, col=cols[1], lwd=2)
# second replacement
lines(1:n_gen, mean_evals2, type='l', xlab="Generation", ylab="Fitness (points)" ,
      ylim=c(min(c(max_evals2,mean_evals2)),max(c(max_evals2,mean_evals2))),
      col=cols[2], lwd=2)
lines(1:n_gen,max_evals2,lty=2, col=cols[2], lwd=2)
#legend("topright", lwd=2,legend = paste("Replace", c(1,1,2,2), rep(c("Mean", "Ma
#dev.off()

#####################################
## RUN 4
# torunament selection (2), uniform crossover(2)
# more runs were required
this_pop = start
max_evals = c()
mean_evals = c()
max_evals2 = c()
mean_evals2 = c()
individ = matrix(NA, ncol=n_vars, nrow=n_gen)

# first replacement function
set.seed(2020)
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select2(pop=this_pop,fitness=evals)
  offspring_crossover = crossover2(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  # store best individ
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace1(parents=next_parents, offspring=offspring_mutation)
  max_evals = c(max_evals,max(evals))
  mean_evals = c(mean_evals,mean(evals))
}
# 2nd replacement function
this_pop = start
for(gen in 1:n_gen){
  evals = evaluate_pop(pop=this_pop, item_points=pts, item_weight=wts)
  next_parents = select2(pop=this_pop,fitness=evals)
  offspring_crossover = crossover2(parents=next_parents)
  offspring_mutation = mutation(pop=offspring_crossover,mutation_rate=0.05)
  # store best individ
  individ[gen,] =  this_pop[which.max(evals),]
  this_pop = replace2(parents=next_parents, offspring=offspring_mutation)
  max_evals2 = c(max_evals2,max(evals))
  mean_evals2 = c(mean_evals2,mean(evals))
}

# store results
res[[7]] <- cbind(mean_evals,max_evals,individ)
res[[8]] <- cbind(mean_evals2,max_evals2, individ)
names(res)[c(7,8)] <- paste("graph 3, replacement", c(1,2))

## plot
#pdf(file = "GAgraph4.pdf",width = 7, height = 7, compress = F)
# first replacement
plot(1:n_gen, mean_evals, type='l', xlab="Generation", ylab="Fitness (points)" ,
     ylim=c(min(c(max_evals,mean_evals)),max(c(max_evals,mean_evals))),
     col=cols[1], lwd=2,
     main = "Tournament Selection, Uniform Crossover")
lines(1:n_gen,max_evals,lty=2, col=cols[1], lwd=2)
# second replacement
lines(1:n_gen, mean_evals2, type='l', xlab="Generation", ylab="Fitness (points)" ,
      ylim=c(min(c(max_evals2,mean_evals2)),max(c(max_evals2,mean_evals2))),
      col=cols[2], lwd=2)
lines(1:n_gen,max_evals2,lty=2, col=cols[2], lwd=2)
#legend("topright", lwd=2,legend = paste("Replace", c(1,1,2,2), rep(c("Mean", "Ma
dev.off()

##################################
### which produced best solution?
# storage for best fitnesses
best_fit <- c()
best_inds <- matrix(NA, ncol = n_vars, nrow = length(res))
for (i in 1:length(res)){
  best_fit <- c(best_fit, max(res[[i]][,2]))
  best_inds[i,] <- res[[i]][which.max(res[[i]][,2]),-c(1,2)]
}
names(best_fit) <- names(res)
best_fit # 102
# best solution
best_inds[which.max(best_fit),]
# weight
sum(best_inds[which.max(best_fit),]*wts)
############## END
```