Post Block Assessment 1

Francois van Zyl : 18620426

27/07/2020

Question 1a)

Local search was implemented with binary candidate solutions. The move-operator was defined as a single bit operator, and the method of best-improvement was used to select the neighbours. For ease of explaining this code, my program code contains text in between the code chunks that will explain the relative parts. This was done to mitigate the effect of having comments that cluttered the chunks too much. The code is as follows, and I started by intializing two global variables, namely the length of the vectors that will be used and the capacity constraint limit that is specified in the question.

```
library(keras) # Gain access to piping operator
```

```
## Warning: package 'keras' was built under R version 3.6.3
```

```
# Question 1 ------------------------------------------------------------
rm(list = ls())
setwd("/Users/jd-vz/Desktop/MEng/Optimisation/PBA1/")
length_bits <- 10 # Length of bits used e.g. 0101010101
constraint_limit <- 12 # The capacity constraint
```

After initializing these variables, I proceeded to define the function that will be used to find the neighbours of a candidate solution. This process is performed by a single bit move operator, that works by inverting a single bit from the initial candidate solution. The function is displayed below, and it accepts variables called **flip_me** and **index**. The variable flip_me contains the intial candidate solution, and index specifies which bit will be inverted to obtain its' neighbour. **If** this bit at the index is a 0, it is inverted to a 1; **else** if the bit at the index is a 1, it is inverted to a 0. The function then returns flipped candidate solution as a single neighbour. It is important to note that this is just one neighbour and will have to be called multiple times with a varying index.

```
# Perform single bit complement ------------------------------------------
single_bit_complement <- function(flip_me, index){
  ifelse(test = flip_me[[index]] == 1, # if the value of the vector is 1
         yes = flip_me[[index]] <- 0, # invert it to a 0
         no = flip_me[[index]] <- 1) # else, if the value is zero, flip it to a 1
  return(flip_me)
}
```

The function below was defined to calculate the objective value of a set of bits. It takes two inputs, namely **all_solns** and an **option** variables. All_solns refers to the candidate solutions, which in my implementation may exist as either a matrix with each row containing a neighbouring solution or as a vector with a single row. These so-called options exist to prevent RStudio of throwing a non-conformable array error, and the

options perform exactly the same operation except the vector implementation does not loop through all the rows since it only has one row. Therefore, the flow of this function is as follows for the Matrix option, the specified objective function values are sorted into a matrix and multiplied with the matrix of neighbouring solutions, and summed to find the respective objective function values for each neighbouring solution. This flow is the same for the vector option, except it only acts on one solution. The function then returns the respective objective function values.

```
# Get the objective function for a solution -------------------------------
get_objective <- function(all_solns, option = "Matrix"){
  if(option == "Matrix") { # Matrix Objective Function
    const <- c(8,12,9,14,16,10,6,7,11,13) %>% as.matrix(nrow = 1) # Shape constants
    obj_value <- matrix(nrow = nrow(all_solns), ncol = 1) # Store objective value
    for(i in 1:nrow(all_solns))
    {
      obj_value[i] <- sum(const*all_solns[i,]) # Calculate for all entries
    }
    return(obj_value) # Return this vector
  }
  if(option == "Vector") { # Vector objective function
    const <- c(8,12,9,14,16,10,6,7,11,13) %>% as.matrix(nrow = 1) # Shape constants
    all_solns <- all_solns %>% as.matrix(ncol = 1)
    return(sum(const*all_solns)) # Calculate and return scalar
  }
}
```

Since there is a capacity constraint, I implemented the following function to inspect which solutions are feasible. The function takes an input variable **all_solns**, which specifies a matrix of bit vectors representing the current and neighbouring solutions. Thereafter the constraint constants are piped into a matrix, and an initial feasibility counter to true for all solutions received. The function then loops **for** each row of the matrix of candidate solutions, it checks whether the constraint is violated and **if** the constraint is being violated it assigns the feasibility counter as false. The function then returns this feasibility vector.

```
# Check whether the constraints hold ----------------------------------------
check_constraint <- function(all_solns){
  const <- c(3,2,1,4,3,3,1,2,2,5) %>% as.matrix(nrow = 1) # Shape capacity constants
  feasible <-  matrix(data = TRUE, nrow = nrow(all_solns), ncol = 1) # Initialize all counters as True
  for(i in 1:nrow(all_solns))
  {
    if(sum(const * all_solns[i,]) > constraint_limit)
      feasible[i] <- FALSE
  }
  return(feasible)
}
```

The function that was used to generate the neighbouring solutions of a candidate solution is displayed below. The function takes as input a bit vector **s**, which is initially replicated into one matrix for the same amount of rows and columns that the bit vector contains. This matrix is therefore essentially 10 replicas of our solution s that contains one solution per row. The function then loops for each row within the initial neighbouring matrix, and calls the single_bit_complement function that was discussed earlier on each row at a different position. By applying the move operator like this, all the neighbouring solutions can be discovered for an input bit vector and are returned from the function.

```
# Generate a neighbourhood for a solution ---------------------------------
generate_neighbourhood <- function(s)
{
  # Initialize all neighbours as being the same as current solution
  neighbours <- matrix(rep(s, length_bits), nrow = length_bits, ncol = length_bits, byrow = T)
  for(i in 1:length_bits) # For each bit
  {
    # For each row i, swap the bit at position i within neighbours
    neighbours[i,] = single_bit_complement(neighbours[i,], i)
  }
  # Return the neighbours
  return(neighbours)
}
```

The local search function is displayed below. Recalling from class, the algorithmic template for the Local Search Algorithm is as follows.

1) Initialize counter to zero
2) Generate Neighbourhood from candidate solution
3) If no better solutions within neighbourhood, exit algorithm.
4) If a better solution found, update best solution.
5) Increment counter
6) Repeat from Step 2.

The code is displayed on the next page. I included references to these steps from the template so that the examiner can see where the template logic is being applied. The function accepts the starting solution **s_initial** as an input variable. At the start of the function's execution, I initialized the counter to 0, accepted the initial starting solution as the solution, and initialized the best and current solutions to arbitrary values so that the function could enter the while loop. The while loop was specified to continue until the current best solution is equal to the best found solution, implying no more improving neighbours exist. If the current solution is not better than the best found solution, the while loop condition is met, and the function enters the iterative process. At the start of the while loop, I calculated the objective function value at the current solution, or bit vector *s*, where the program currently is by using the predefined function get_objective. Thereafter, I generated the neighbouring solutions by using the predefined function generate_neighbourhood. Recall that this function calls the single_bit_complement for me, to generate all possible neighbouring solutions. Thereafter the current solution and neighbouring solutions are merged into all_solns. The objective value for every solution within all_solns is calculated by using get_objective, including the current solution [note we've already calculated this and saved it into current_soln_value]. Similarly, the predefined check_constraint is used to check which of these solutions is valid. These solutions, objective values and feasibility checks are then merged into a dataframe, after which all infeasible options are excluded. The best possible solution is chosen as being the solution with the highest objective function value [i.e. best-improving neighbour selection]. The best found solution objective function value is saved into best_soln_value, and best_soln is reformatted such that the loop can continue. The candidate solution is updated as s, and if the while loop is entered the function will use it to calculate the next current_soln_value. Finally the counter is incremented. At this point the while loop will check whether the

current_soln_value found in the previous iteration is equal to the new best_soln_value that was found, and if they are equal there are no more improving neighbours and the algorithm exists. The algorithm then outputs a list containing the best found solution, the best found solution value, and the amount of iterations required.

```
# Local Search Function ----------------------------------------------------
local_search <- function(s_initial)
{
  best_soln_value <- 0
  current_soln_value <- Inf
  s <- s_initial
  counter <- 0
  # While current solution is not the best solution ------------------------
  while(!(current_soln_value == best_soln_value))
  { # Calculate current solution's objective value ----------------------------
    current_soln_value = get_objective(s, option = "Vector")
    neighbours <- generate_neighbourhood(s) # Generate neighbourhood
    all_solns <- rbind(s, neighbours) # Merge all possible solutions
    # Get objective value and feasibility ------------------------------------
    obj_value <- get_objective(all_solns, option = "Matrix") # Objective values for solutions
    feasible_check <- check_constraint(all_solns) # Check which of these solutions are feasible
    data <- data.frame(all_solns, obj_value, feasible_check) # Merge
    data <- data[feasible_check,] # Only feasible options remain
    best_soln <- data[which.max(data$obj_value),] #Choose best soln as soln with highest objective valu
    best_soln_value <- best_soln$obj_value # Save best found objective value
    # Clear formatting for loop to continue ----------------------------------
    best_soln$obj_value <- best_soln$feasible_check <- NULL # Formatting
    s <- best_soln %>% as.numeric() # Update the bit vector
    counter <- counter + 1 # Counter
  }
  # Returns the final solution as bits, the objective function value,
  # as well as the amount of iterations required
  return(list(Bit_ = s, Value = best_soln_value, Iter = counter))
}
```

Question 1b)

The algorithm is defined above, and the results can be investigated as follows for the three initial starting solutions that were provided.

```
# Investigate ------------------------------------------------------------
s_1 <- c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0) # Solution 1
s_2 <- c(0, 0, 0, 0, 0, 0, 0, 0, 1, 0) # Solution 2
s_3 <- c(0, 1, 1, 1, 0, 1, 0, 1, 0, 0) # Solution 3
```

```
display_data <- data.frame(S_1 = unlist(s_1 %>% local_search()),
                           S_2 = unlist(s_2 %>% local_search()),
                           S_3 = unlist(s_3 %>% local_search())) # Calling local search for each
knitr::kable(t(display_data)) # Neat display of results
```

	Bit_1	Bit_2	Bit_3	Bit_4	Bit_5	Bit_6	Bit_7	Bit_8	Bit_9	Bit_10	Value	Iter
S_1	0	0	0	1	1	0	0	0	0	1	43	4
S_2	0	1	1	1	1	0	0	0	1	0	62	5
S_3	0	1	1	1	0	1	0	1	0	0	52	1

The output of local search for the three initial starting solutions are displayed above. The initial starting position s_1 can be seen to have had the lowest objective function value, and required 4 iterations. The second starting position s_2 achieved the highest objective function value and also required the most iterations. The final starting position only required one iteration, and achieved an objective function value better than the first solution, but worse than the second solution.

Question 2a)

I started this problem by clearing my workspace, and loading the defined variables. These declarations are displayed below.

```
# Question 2 ------------------------------------------------------------
rm(list = ls())
library(keras)
# Define Variables ------------------------------------------------------
population_size <- 8 # Population size of candidate solutions
candidate_size <- 6 # Candidate solution size
tour_select_size <- 3 # Tournament selection size
num_parents_crossover <- 6 # The number of parents that will be used for crossover
TSP <- matrix(c(0, 41, 26, 31, 27, 35,
                41, 0, 29, 32, 40, 33,
                26, 29, 0, 25, 34, 42,
                31, 32, 25, 0, 28, 34,
                27, 40, 34, 28, 0, 36,
                35, 33, 42, 34, 36, 0), nrow = 6) # The distance matrix a
```

After defining these variables, I proceeded to construct some functions to aid in keeping the program as modular as possible. I started by defining the create_initial_population function, which takes no input and returns a 8 by 6 matrix that returns a randomly generated matrix that contains possible permutations of the sequence of 1 to 6. This returned matrix will act as the initial population that will need to be generated upon the first execution of the genetic algorithm.

```
# Create population -----------------------------------------------------
create_initial_population <- function()
{
  population <- matrix(nrow = population_size, ncol = candidate_size, byrow = T) # Empty 8 by 6 matrix
  for(i in 1:population_size) population[i,] <- sample.int(candidate_size, candidate_size)
  # Fill rows in the matrix with permutation vector of length 6, thus each row is one solution.
  return(population) # Matrix containing 8 random permutation vectors of length 6
}
```

Please do note that each row within the population matrix corresponds to a solution.

The following function will be used to evaluate the fitness of a set of candidate solutions. The function takes as input a population matrix, and sets up a counter called *fitness* to keep track of each row within the population matrix. The function then enters a nested for loop, where it iterates for every solution (row) and every column (value within solution) within the population matrix. The fitness can then be calculated as the sum of the distance that can be found from the TSP distance matrix. The TSP is indexed at the values contained within i-th row of the population matrix (solution) at the j-th column or element, which allows for the calculation of the distances between the row and column of the TSP matrix. Within the TSP the row corresponds to where the candidate is, and the column corresponds where the candidate is trying to go. If the last column is reached, the TSP returns to its starting position. After the nested for loop finishes, this function returns a vector called fitness containing the total distance travelled corresponding to a solution within the population matrix. At this point I confess, I defined a higher fitness as a lower distance that is returned and therefore when selecting the highest fitness, the minimum fitness should be chosen. This was negligent on my behalf, but please note for the rest of the report that the **maximum fitness corresponds to taking the minimum of the vector called fitness.**

```
# Evaluate Fitness ----------------------------------------------------------
evaluate_fitness <- function(population)
{
  fitness <- vector(mode = "numeric", length = nrow(population))
  for(i in 1:nrow(population)) # Row corresponds to where candidate is
  { # Column corresponds to where candidate is going
    for(j in 1:(ncol(population)-1))
    {
      fitness[i] <- fitness[i] + TSP[population[i,j],
                                     population[i,j + 1]] # Add distance traversed
      # Need to return to start
      if(j == (ncol(population)-1)) fitness[i] <- fitness[i] +  TSP[population[i,j+1],
                                                                    population[i,1]]
    }
  }
  return(fitness) # Vector containing fitness values
}
```

I then defined the function that would be used to perform the tournament selection. This function accepts as a similar matrix to the matrix returned from create_initial_population, with dimensions 8 by 6. In the start, it initializes a variable best_candidates to keep track of who won the tournament selection, with dimensions 6 by 6 (since three pairs of parents are required for crossover). The function enters a loop that draws 3 random indices, without replacement, from the total rows of the population. The probability of being drawn is drawn from a uniform random distribution. However, the function then subsets the candidates for the tournament from the population matrix, and evaluates their fitness using the predefined function evaluate_fitness. The fittest individuals are then chosen **according to the minimum of the vector called fitness**, and removed from the pool. Therefore, the probability of winning the tournament and being chosen as a parent is proportional to the fitness value. This loop continues until 6 parents are obtained and the function returns the winners of the tournament.

```r
# Tournament Selection ----------------------------------------------------
tournament_selection <- function(population) # 8 by 6 matrix
{
  best_candidates <- matrix(nrow = num_parents_crossover, ncol = candidate_size) # Initialize matrix
  for(i in 1:num_parents_crossover) # 6 tournaments
  {
    idx <- sample.int(nrow(population), tour_select_size) # Get 3 random indices
    parent_candidates <- population[idx,] # Subset contenders
    candidates_fitness <- evaluate_fitness(parent_candidates) #E Evaluate their fitness
    best_candidates[i,] <- parent_candidates[which.min(candidates_fitness),] # Choose fittest
    remove_idx <- idx[which.min(candidates_fitness)]
    population <- population[-remove_idx, ] # Remove the winners from pool to prevent cycling
  }
  return(best_candidates) # Return winners
}
```

The following function will be used to perform two-point crossover. It accepts as input a variable parents, which will be the 6 parents, or 3 parent-pairs, that won the tournament selection process. The input will therefore be a 6 by 6 matrix, with the number of columns corresponding to the candidate size and the number of rows corresponding to the number of solutions. The function starts by defining the offspring variable. A for loop is set up to ensure that the loop executes upon the correct pairs within the parents matrix, namely {1, 2}, {3,4}, and {5,6}. These respective pairs are used per loop iteration, and two random cutting indices are generated. The first cutting index can range from 1 to 4, and the second cutting index can range from (first cutting index + 1) to 5. By using these indices, the lower and higher part of the first parent can be extracted and the middle can be filled in **with unordered values obtained from the second parent** that have not occurred within the first parent yet. This was achieved by finding the intersection (*nip_1*) of values that exist within parent 2 that do not exist within the lower or higher sections of parent 1. Each loop, the function appends the offspring generated and after completion returns them as a 3 by 6 matrix.

```r
# Crossover function ------------------------------------------------------
two_point_crossover <- function(parents)
{
  offspring <- c() # Placeholder initialization
  for(i in c(1,3,5)) # Starting places in the loop
  {
    # Get parent pair -------------------------------------------------------
    parent_1 <- parents[i , ] # Access variables {1, 3, 5}
    parent_2 <- parents[i + 1 , ] # Access variables {2, 4, 6}
    # Generate random points to perform crossover ---------------------------
    cut_idx_1 <-  round(runif(1, min = 1, max = (candidate_size - 2))) # Sample both cutting points
    cut_idx_2 <- round(runif(1, min = cut_idx_1 + 1, max = (candidate_size - 1)))
    # Find unique elements to merge -----------------------------------------
    low_1 <- parent_1[1:cut_idx_1] # Lower section of parent 1
    high_1 <- parent_1[cut_idx_2:candidate_size] # Higher section of parent 1
    nip_1 <- intersect(parent_2[!(parent_2 %in% low_1)],
                       parent_2[!(parent_2 %in% high_1)]) # Only require values that exist jointly
    # Create offspring ------------------------------------------------------
    child_1 <- c(low_1, nip_1, high_1) # Use unique elements to create child
    offspring <- rbind(offspring, child_1) # Append to offspring
  }
  rownames(offspring) <- NULL # To ensure report stays neat
  return(as.matrix(offspring)) # Return the 3 offspring that were found
}
```

I then proceeded to define the function that will be used for random mutation of one offspring. The function is displayed below, and takes as input the 3 by 6 offspring matrix that was previously defined. The function selects a random offspring (referred to as mutated_offspring in the code), and generates two swapping indices from a random uniform distribution without replacement. The value of the offspring is then swapped through the usage of a temporary object. After swapping the values at the indices, the mutated offspring is returned and overwrites the initial offspring that was mutated. The function then returns the altered 3 by 6 matrix as an output.

```
# Mutation Function ------------------------------------------------------
mutation <- function(offspring) # 3 by 6 matrix accepted
{
  idx <- sample.int(n = nrow(offspring), size = 1) # Sample 1 random integer
  mutated_offspring <- offspring[idx,] # Random offspring selected
  idx_swap <- sample.int(ncol(offspring), 2) # 2 random swapping indices
  temp <- mutated_offspring[idx_swap[1]]  # Store first random value
  mutated_offspring[idx_swap[1]] <- mutated_offspring[idx_swap[2]]
  mutated_offspring[idx_swap[2]] <- temp # Replace second value by first value
  offspring[idx,] <- mutated_offspring # Overwrite old offspring
  return(offspring)  # 3 by 6 matrix returned
}
```

The following function defines the elitist replacement strategy that was implemented. It accepts the population and generated/mutated offspring matrices from previously. First it pools all the possible population members into one by merging the population and generated/mutated offspring, and then it evaluates the fitness of the total pooled population. This population is then sorted according to their respective fitness values with the lowest distance travelled (or the highest fitness) at the top rows. The top 8 values are then chosen from this population and a 8 by 6 matrix containing the survivors is returned.

```
# Elitism replacement strategy -------------------------------------------
elitism_replacement <- function(population, offspring_mutated)
{
  population<- rbind(population, offspring_mutated) # Pool population
  elitist_candidates <- cbind(population, evaluate_fitness(population)) # Evaluate fitness
  elitist_candidates <- elitist_candidates[order(elitist_candidates[,ncol(elitist_candidates)],
                                                 decreasing = F),] # Lowest distance at top
  elitists <- elitist_candidates[1:population_size,1:candidate_size] # Choose top 8
  return(elitists)
}
```

Now that the functions are defined, I referred to the GA template provided to us in class to determine the algorithm flow.

1) Initialize counters and find best initial solution
2) Evaluate the fitness of all 8 solutions
3) Find 6 parents through tournament selection
4) Find offspring through crossover with 3 parent pairs
5) Mutate one offspring
6) Evaluate the fitness of the population candidates
7) Select the 8 best members with an elitism replacement strategy
8) Update incumbant if necessary
9) Increment counter
10) Repeat until the stopping criterion is met

The GA I implemented is displayed below. The function does not take an input, and initializes counters for the number of iterations and the number of iterations experienced without observing a change in the incumbant. The function then enters a while loop, which acts as the stopping critrion and checks for how long there has been no change in the incumbant solution value. At and only at the very first time step, the population is generated through create_initial_population. Thereafter the current incumbant and incumbant solution are found at the minimum of the fitness vector returned from evaluate_fitness, which in my implementation represents the maximum of the actual fitness. Thereafter, the function enters the main body of the code, which is the selection, crossover, mutation and replacement functions that are called. Note that I do not have an evaluate_fitness function as sometimes defined, but to keep the code organized, I instead call the evaluate_fitness function from within functions that require it. After these evolutionary operations have been performed, I calculate the new incumbant value from the population. If this incumbant value is the same as it was before the four evolutionary operations, I increment the counter that represents the generations that have passed without changed. However, if it does change, the counter is reset to 0. Finally the generation counter is incremented. The last part of code is simply a way of sorting the output to get the output requested in the assignment. The algorithm and its output are displayed on the following pages. The algorithm output is quite cluttered, and to ease the marking process I will briefly explain the representation.

1) Population – the upper left chunk which has row labels **Soln__1** to **Soln__8** and column labels **X1** to **X6**
2) Fit – The fitness value of individual solutions
3) AvgFit – The average fitness value of the population
4) Best – The incumbant solution *value*
5) Gen – The generation under consideration
6) Parents – The upper right chunk which has column labels **P.1** to **P.6**
7) Offspring – The bottom left chunk which has column labels **O.1** to **O.6**
8) Mutated Offspring – The bottom right chunk which has column labels **OM.1** to **OM.6**

```r
genetic_algorithm <- function()
{ #
  t <- no_change <- 0 # Initialize counters
  Output <- list() # Initialize output list
  while(!(no_change == 10)) # Stopping criterion
  {
    if(t == 0)   population <- create_initial_population() # Generate population
    incumbant_soln <- population[which.min(evaluate_fitness(population)),] # Incumbant
    incumbant_soln_value <- min(evaluate_fitness(population)) # Incumbant associated value

    # Selection ------------------------------------------------------------
    parents <- tournament_selection(population) # Touranment Selection of parents
    # Crossover ------------------------------------------------------------
    offspring <- two_point_crossover(parents) # 2-point Crossover of parents
    # Mutation -------------------------------------------------------------
    offspring_mutated <- mutation(offspring) # Mutation with 1 random swap
    # Replacement ----------------------------------------------------------
    population <- elitism_replacement(population, offspring_mutated) # Elitism

    # Check stopping criteria ----------------------------------------------
    new_incumbant_val <-  min(evaluate_fitness(population))
    ifelse(incumbant_soln_value == new_incumbant_val,
           no_change <- no_change + 1,
           no_change <- 0)
    t <- t + 1 # Increment counter

    # Prepare history ------------------------------------------------------
    Parents = rbind(parents, matrix(NA, nrow = 2, ncol = candidate_size))
    offspring <- rbind(offspring, matrix(NA, nrow = 5, ncol = candidate_size))
    offspring_mutated <- rbind(offspring_mutated, matrix(NA, nrow = 5, ncol = candidate_size))
    Output[[t]] <- data.frame(population,
                              Fit = evaluate_fitness(population),
                              AvgFit = mean(evaluate_fitness(population)),
                              Best = incumbant_soln_value,
                              Gen = t,
                              fill = rep("###", population_size),
                              P = Parents,
                              fill = rep("###", population_size),
                              O = offspring,
                              fill = rep("###", population_size),
                              OM = offspring_mutated,
                              row.names = c("Soln_1", "Soln_2", "Soln_3", "Soln_4",
                                            "Soln_5", "Soln_6", "Soln_7", "Soln_8"))
  }
  return(Output) # Return history of outputs
}




genetic_algorithm() # Execute the GA function
```

```
## [[1]]
##        X1 X2 X3 X4 X5 X6 Fit  AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177 191.375  177   1  ###   6   3   4   2   5   1
## Soln_2  2  6  5  4  1  3 183 191.375  177   1  ###   6   1   5   4   3   2
## Soln_3  4  5  6  1  3  2 186 191.375  177   1  ###   2   6   5   4   1   3
## Soln_4  4  2  3  5  1  6 191 191.375  177   1  ###   4   5   6   1   3   2
## Soln_5  4  2  3  5  1  6 191 191.375  177   1  ###   4   2   3   5   1   6
## Soln_6  6  3  4  2  5  1 201 191.375  177   1  ###   6   4   5   3   2   1
## Soln_7  6  4  5  3  2  1 201 191.375  177   1  ###  NA  NA  NA  NA  NA  NA
## Soln_8  6  3  4  2  5  1 201 191.375  177   1  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   6   3   4   2   5   1    ###    6    3    4    2    5    1
## Soln_2    ###   2   6   5   4   1   3    ###    4    6    5    2    1    3
## Soln_3    ###   4   2   3   5   1   6    ###    4    2    3    5    1    6
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[2]]
##        X1 X2 X3 X4 X5 X6 Fit  AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177 189.125  177   2  ###   2   6   5   4   1   3
## Soln_2  2  6  5  4  1  3 183 189.125  177   2  ###   4   5   6   1   3   2
## Soln_3  2  6  5  4  1  3 183 189.125  177   2  ###   4   2   3   5   1   6
## Soln_4  4  5  6  1  3  2 186 189.125  177   2  ###   4   2   3   5   1   6
## Soln_5  4  2  3  5  1  6 191 189.125  177   2  ###   6   3   4   2   5   1
## Soln_6  4  2  3  5  1  6 191 189.125  177   2  ###   6   1   5   4   3   2
## Soln_7  6  3  4  2  5  1 201 189.125  177   2  ###  NA  NA  NA  NA  NA  NA
## Soln_8  6  4  5  3  2  1 201 189.125  177   2  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   2   6   5   4   1   3    ###    2    6    5    4    1    3
## Soln_2    ###   4   2   3   5   1   6    ###    4    2    1    5    3    6
## Soln_3    ###   6   3   4   2   5   1    ###    6    3    4    2    5    1
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[3]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177    185  177   3  ###   2   6   5   4   1   3
## Soln_2  2  6  5  4  1  3 183    185  177   3  ###   6   1   5   4   3   2
## Soln_3  2  6  5  4  1  3 183    185  177   3  ###   2   6   5   4   1   3
## Soln_4  2  6  5  4  1  3 183    185  177   3  ###   4   2   3   5   1   6
## Soln_5  4  5  6  1  3  2 186    185  177   3  ###   4   5   6   1   3   2
## Soln_6  4  5  6  1  3  2 186    185  177   3  ###   4   2   3   5   1   6
## Soln_7  4  2  3  5  1  6 191    185  177   3  ###  NA  NA  NA  NA  NA  NA
## Soln_8  4  2  3  5  1  6 191    185  177   3  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   2   6   5   4   1   3    ###    2    6    5    4    1    3
## Soln_2    ###   2   6   5   4   1   3    ###    3    6    5    4    1    2
## Soln_3    ###   4   5   6   1   3   2    ###    4    5    6    1    3    2
```

```
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[4]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177    183  177   4  ###   2   6   5   4   1   3
## Soln_2  2  6  5  4  1  3 183    183  177   4  ###   6   1   5   4   3   2
## Soln_3  2  6  5  4  1  3 183    183  177   4  ###   2   6   5   4   1   3
## Soln_4  2  6  5  4  1  3 183    183  177   4  ###   2   6   5   4   1   3
## Soln_5  2  6  5  4  1  3 183    183  177   4  ###   4   5   6   1   3   2
## Soln_6  2  6  5  4  1  3 183    183  177   4  ###   4   5   6   1   3   2
## Soln_7  4  5  6  1  3  2 186    183  177   4  ###  NA  NA  NA  NA  NA  NA
## Soln_8  4  5  6  1  3  2 186    183  177   4  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   2   6   5   4   1   3    ###    2    6    5    4    1    3
## Soln_2    ###   2   6   5   4   1   3    ###    2    6    5    4    1    3
## Soln_3    ###   4   5   6   1   3   2    ###    4    5    1    6    3    2
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[5]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177    182  177   5  ###   6   1   5   4   3   2
## Soln_2  6  5  1  4  3  2 181    182  177   5  ###   2   6   5   4   1   3
## Soln_3  2  6  5  4  1  3 183    182  177   5  ###   2   6   5   4   1   3
## Soln_4  2  6  5  4  1  3 183    182  177   5  ###   2   6   5   4   1   3
## Soln_5  2  6  5  4  1  3 183    182  177   5  ###   2   6   5   4   1   3
## Soln_6  2  6  5  4  1  3 183    182  177   5  ###   2   6   5   4   1   3
## Soln_7  2  6  5  4  1  3 183    182  177   5  ###  NA  NA  NA  NA  NA  NA
## Soln_8  2  6  5  4  1  3 183    182  177   5  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   6   5   1   4   3   2    ###    6    5    1    4    3    2
## Soln_2    ###   2   6   5   4   1   3    ###    2    6    5    4    1    3
## Soln_3    ###   2   6   5   4   1   3    ###    2    6    1    4    5    3
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[6]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177 181.25  177   6  ###   6   1   5   4   3   2
## Soln_2  6  1  5  4  3  2 177 181.25  177   6  ###   2   6   5   4   1   3
## Soln_3  6  5  1  4  3  2 181 181.25  177   6  ###   2   6   5   4   1   3
## Soln_4  2  6  5  4  1  3 183 181.25  177   6  ###   2   6   5   4   1   3
## Soln_5  2  6  5  4  1  3 183 181.25  177   6  ###   2   6   5   4   1   3
## Soln_6  2  6  5  4  1  3 183 181.25  177   6  ###   6   5   1   4   3   2
```

```
## Soln_7  2  6  5  4  1  3 183 181.25  177   6  ###  NA  NA  NA  NA  NA  NA
## Soln_8  2  6  5  4  1  3 183 181.25  177   6  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   6   1   5   4   3   2    ###    6    1    5    4    3    2
## Soln_2    ###   2   6   5   4   1   3    ###    2    6    5    4    1    3
## Soln_3    ###   2   6   5   4   1   3    ###    1    6    5    4    2    3
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[7]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177 180.25  177   7  ###   6   1   5   4   3   2
## Soln_2  6  1  5  4  3  2 177 180.25  177   7  ###   6   1   5   4   3   2
## Soln_3  6  1  5  4  3  2 177 180.25  177   7  ###   6   5   1   4   3   2
## Soln_4  6  5  1  4  3  2 181 180.25  177   7  ###   2   6   5   4   1   3
## Soln_5  6  5  1  4  3  2 181 180.25  177   7  ###   2   6   5   4   1   3
## Soln_6  2  6  5  4  1  3 183 180.25  177   7  ###   2   6   5   4   1   3
## Soln_7  2  6  5  4  1  3 183 180.25  177   7  ###  NA  NA  NA  NA  NA  NA
## Soln_8  2  6  5  4  1  3 183 180.25  177   7  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   6   1   5   4   3   2    ###    6    1    5    4    3    2
## Soln_2    ###   6   5   1   4   3   2    ###    6    5    1    4    3    2
## Soln_3    ###   2   6   5   4   1   3    ###    2    6    3    4    1    5
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[8]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177 179.25  177   8  ###   6   1   5   4   3   2
## Soln_2  6  1  5  4  3  2 177 179.25  177   8  ###   6   5   1   4   3   2
## Soln_3  6  1  5  4  3  2 177 179.25  177   8  ###   6   5   1   4   3   2
## Soln_4  6  1  5  4  3  2 177 179.25  177   8  ###   6   1   5   4   3   2
## Soln_5  6  5  1  4  3  2 181 179.25  177   8  ###   6   1   5   4   3   2
## Soln_6  6  5  1  4  3  2 181 179.25  177   8  ###   2   6   5   4   1   3
## Soln_7  6  5  1  4  3  2 181 179.25  177   8  ###  NA  NA  NA  NA  NA  NA
## Soln_8  2  6  5  4  1  3 183 179.25  177   8  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   6   1   5   4   3   2    ###    6    1    5    4    3    2
## Soln_2    ###   6   5   1   4   3   2    ###    6    5    1    4    3    2
## Soln_3    ###   6   1   5   4   3   2    ###    6    1    4    5    3    2
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[9]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
```

```
## Soln_1  6  1  5  4  3  2 177    178  177   9  ###   6   1   5   4   3   2
## Soln_2  6  1  5  4  3  2 177    178  177   9  ###   6   1   5   4   3   2
## Soln_3  6  1  5  4  3  2 177    178  177   9  ###   6   1   5   4   3   2
## Soln_4  6  1  5  4  3  2 177    178  177   9  ###   6   1   5   4   3   2
## Soln_5  6  1  5  4  3  2 177    178  177   9  ###   6   5   1   4   3   2
## Soln_6  6  1  5  4  3  2 177    178  177   9  ###   6   5   1   4   3   2
## Soln_7  6  5  1  4  3  2 181    178  177   9  ###  NA  NA  NA  NA  NA  NA
## Soln_8  6  5  1  4  3  2 181    178  177   9  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   6   1   5   4   3   2    ###    6    1    5    4    3    2
## Soln_2    ###   6   1   5   4   3   2    ###    6    1    5    4    3    2
## Soln_3    ###   6   5   1   4   3   2    ###    6    5    1    2    3    4
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
##
## [[10]]
##        X1 X2 X3 X4 X5 X6 Fit AvgFit Best Gen fill P.1 P.2 P.3 P.4 P.5 P.6
## Soln_1  6  1  5  4  3  2 177    177  177  10  ###   6   1   5   4   3   2
## Soln_2  6  1  5  4  3  2 177    177  177  10  ###   6   1   5   4   3   2
## Soln_3  6  1  5  4  3  2 177    177  177  10  ###   6   1   5   4   3   2
## Soln_4  6  1  5  4  3  2 177    177  177  10  ###   6   1   5   4   3   2
## Soln_5  6  1  5  4  3  2 177    177  177  10  ###   6   1   5   4   3   2
## Soln_6  6  1  5  4  3  2 177    177  177  10  ###   6   1   5   4   3   2
## Soln_7  6  1  5  4  3  2 177    177  177  10  ###  NA  NA  NA  NA  NA  NA
## Soln_8  6  1  5  4  3  2 177    177  177  10  ###  NA  NA  NA  NA  NA  NA
##        fill.1 O.1 O.2 O.3 O.4 O.5 O.6 fill.2 OM.1 OM.2 OM.3 OM.4 OM.5 OM.6
## Soln_1    ###   6   1   5   4   3   2    ###    6    1    5    4    3    2
## Soln_2    ###   6   1   5   4   3   2    ###    6    1    5    4    3    2
## Soln_3    ###   6   1   5   4   3   2    ###    6    1    4    5    3    2
## Soln_4    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_5    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_6    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_7    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
## Soln_8    ###  NA  NA  NA  NA  NA  NA    ###   NA   NA   NA   NA   NA   NA
```