Assignment 4

Pranav Kasela 846965

Contents


```
require(GA)
require(ggplot2)
require(reshape2)
require(dplyr)
require(Rcpp)
```

Function definition

Initially the functions that were being using to calculate the makespan and fitness were the written in R, but one of the colleagues: *Federico Moiraghi*, was kind enough to provide his code compiled in C++ for the time function to speed up the process. Some parameters are modified in the function before reusing his code.

Since the time is > 0, the inverse of the makespan can be used as the fitness function for **GA** algorithm, since maximizing the fitness is equivalent to minimizing the makespan.

```
#-- Function before C++
makespan <- function(perm,distMatrix){
  n_jobs     <- ncol(distMatrix)
  n_machines <- nrow(distMatrix)
  dist       <- matrix(NA, nrow=n_machines, ncol=n_jobs)
  dist[1,]   <- cumsum(distMatrix[1,perm])
  dist[,1]   <- cumsum(distMatrix[,perm[1]])
  for (i in 2:n_machines){
    for (j in 2:n_jobs){
      dist[i,j] <- distMatrix[i,perm[j]] +
        max(dist[i,j-1],dist[i-1,j])
    }
  }
  makespan   <- dist[n_machines,n_jobs]
  return(makespan)
}

fitness <- function(perm,distMatrix){
  return(1/makespan(perm,distMatrix))
}
```

From hereon the time function used will be makespanCpp, while the fitness code will be fitnessCpp.

```
#-- function in C++
cppFunction('double fitnessCpp(NumericVector perm,
                           NumericMatrix distMatrix)
    {
        int nrow = distMatrix.nrow();
        int ncol = distMatrix.ncol();
        int norder = perm.size();
        NumericVector order(norder);
        for (int i = 0; i < norder; i++) order[i] = perm[i]-1;
        NumericMatrix time_matrix(nrow, norder);
        time_matrix[0] = distMatrix[nrow * order[0]];
        for (int r = 1; r < nrow; r++)
            time_matrix[r] = time_matrix[r - 1] +
            distMatrix[nrow * order[0] + r];
        for (int c = 1; c < norder; c++)
            time_matrix[nrow * c] = time_matrix[nrow * (c - 1)] +
            distMatrix[nrow * order[c]];
        for (int r = 1; r < nrow; r++)
            for (int c = 1; c < norder; c++)
                if (time_matrix[nrow * c + (r - 1)] > time_matrix[nrow * (c - 1) + r])
                    time_matrix[nrow * c + r] = time_matrix[nrow * c + (r - 1)] +
            distMatrix[nrow * order[c] + r];
        else
            time_matrix[nrow * c + r] = time_matrix[nrow * (c - 1) + r] +
            distMatrix[nrow * order[c] + r];
        return 1/time_matrix[nrow * norder - 1];
    }')

cppFunction('double makespanCpp(NumericVector perm,
                           NumericMatrix distMatrix)
    {
        int nrow = distMatrix.nrow();
        int ncol = distMatrix.ncol();
        int norder = perm.size();
        NumericVector order(norder);
        for (int i = 0; i < norder; i++) order[i] = perm[i]-1;
        NumericMatrix time_matrix(nrow, norder);
        time_matrix[0] = distMatrix[nrow * order[0]];
        for (int r = 1; r < nrow; r++)
            time_matrix[r] = time_matrix[r - 1] +
            distMatrix[nrow * order[0] + r];
        for (int c = 1; c < norder; c++)
            time_matrix[nrow * c] = time_matrix[nrow * (c - 1)] +
            distMatrix[nrow * order[c]];
        for (int r = 1; r < nrow; r++)
            for (int c = 1; c < norder; c++)
                if (time_matrix[nrow * c + (r - 1)] > time_matrix[nrow * (c - 1) + r])
                    time_matrix[nrow * c + r] = time_matrix[nrow * c + (r - 1)] +
            distMatrix[nrow * order[c] + r];
        else
            time_matrix[nrow * c + r] = time_matrix[nrow * (c - 1) + r] +
            distMatrix[nrow * order[c] + r];
        return time_matrix[nrow * norder - 1];
```

```
    }')
```

Table with 20 jobs for 5 machines for the performance checking

The testing of the functions are done with the smallest table available on the website. The algorithm used are the Genetic Algorithm from the R package and the Simulated Annealing that will be implemented manually to check their performance.

GA algorithm (from the GA package)

```
#This is the pre-test given in the assignment
time_matrix <- matrix(c(29,30,27,2,37,62,21,6,95,59,70,82,
                 85,11,62,80,65,55,67,57),nrow = 4) #used once to test and never again

time_matrix <- as.matrix(read.csv("j20-m5",sep=" ",header = FALSE))
n_jobs <- ncol(time_matrix)


time_taken_GA_20_5 <- microbenchmark::microbenchmark(
  GA.fit <- ga(type = "permutation",
    fitness = fitnessCpp,
    distMatrix = time_matrix,
    lower = 1,
    upper = n_jobs,
    popSize = 600,
    maxiter = 10000,
    run = 300,
    pmutation = 0.2,
    keepBest = TRUE,
    monitor = NULL,
    seed = 1234),
times = 1
)

summary(GA.fit)
```

```
## -- Genetic Algorithm -------------------
##
## GA settings:
## Type                  =  permutation
## Population size       =  600
## Number of generations =  10000
## Elitism               =  30
## Crossover probability =  0.8
## Mutation probability  =  0.2
##
## GA results:
## Iterations             = 359
## Fitness function value = 0.00077101
## Solutions =
##       x1 x2 x3 x4 x5 x6 x7 x8 x9 x10  ...  x19 x20
```

```
## [1,]  15 11  9 14  5  7  1  3 19  4        10  20
## [2,]  15  6 11 14  7 17  9  5  8  1        10  20
## [3,]   1 17 15 14 19  9 16  6 11 13        10  20
## [4,]  15 11  9 14  5  8  7 17  1  3        10  20
## [5,]  15 14 19  1  3  9  4 17  5 16        10  20
## [6,]  15  6 11 14  5  8  7 17  1 19        10  20
## [7,]  15 11  9 14  5  7  1  8  6 19        10  20
## [8,]  15  9 14  5  1  6 19  4  7 16        10  20
## [9,]   1  9  4 19 15 17 14 16  5  3        10  20
## [10,] 15 17 14  4  9  3  1 19  5 16        10  20
##  ...
## [29,] 15  6 11  9  7 17  1 19 14  5        10  20
## [30,] 15  6 11 17 14  9  1  7  8  3        10  20
```

```
ris_GA_20_5 <- makespanCpp(GA.fit@solution[1,],time_matrix) #best time
ris_GA_20_5
```

```
## [1] 1297
```

```
out <- plot(GA.fit, main = "GA progression")
```

```
melt(out[,c(1:3,5)],id.var="iter") %>%
  mutate(inv.value=1/value) -> df1

ggplot(df1, aes(x = iter, y = inv.value,
                group = variable, colour = variable)) +
  xlab("Generation") + ylab("Makespan") +
  geom_line(aes(lty = variable)) +
  scale_colour_brewer(palette = "Set1") +
  labs(title = "GA Progression with 20 jobs in 5 machines")
```

The **GA** algoritm finds the best solution as 1297, with an initial population of 600, a mutation probability of 20% and a crossover probability of 80%. An initial trend of improvement can be seen in the figure above, bit after 50 iterations, this improvement stops.

SA algorithm (manually implemented)

The change that has been made in the SA algorithm is the swapJobs function, in this case since the permutation will be done on a lot of elements (= number of jobs):

- If the number of the jobs is greater than 10, the swap will be done on a random number of elements between 2 and 5;
- If the number of jobs is smaller or equal to 10, the swap will be done on 2 elements.

The algorithm, since it's computationally easy, will be done 10 times, to avoid heavy dependencies from it's probabistic nature.

```
swapJobs <- function(perm){
    perm <- as.numeric(perm)
    n <- length(perm)
    if(n>10)
      n_change <- sample(2:min(ceiling(n/5),5),1) #no more than 5 changes at a time
    else
      n_change <- 2
    change <- sort(sample.int(n,n_change))
    newperm <- replace(perm,change,perm[sort(change,decreasing = TRUE)])
    return(as.numeric(newperm))
}
```

```r
SA <- function(tour, distMatrix, maxIterNoChange = 2000, T_ini = 50, T_min = 1){
  path <- tour
  n <- length(path)
  tmin <- T_min    # minimum temperature
  alpha <- 0.999  # update factor
  T <- T_ini
  tini <- T_ini   # starting temperature
  dist <- makespanCpp(path, distMatrix)
  bestLength <- dist
  traceBest <- c(dist)
  traceCurrentLength <- c(dist)
  iterNoChange = 0
  while(T >= tmin){        # if the temperature is not at its minimum
    iterNoChange = iterNoChange+1
    newpath <- swapJobs(path) #swap
    dist_new <- makespanCpp(newpath, distMatrix)
    if(dist_new <= bestLength){
      path <- newpath
      dist <- dist_new
      bestLength <- dist
      iterNoChange <- 0
    }
    else {

      if (exp((dist-dist_new)/T)>runif(1, 0, 1)){
        dist <- dist_new
        path <- newpath
        iterNoChange <- 0
      }

    }
    traceBest <- append(traceBest, bestLength)
    traceCurrentLength <- append(traceCurrentLength, dist)
    T <- T*alpha # the temperature is updated
    if(iterNoChange >= maxIterNoChange){ break}
  }
  res = list(route=path, traceBest = traceBest, trace = traceCurrentLength)
  return(res)
}


start <- as.numeric(sample(1:n_jobs,n_jobs)) #start randomly
best_res   <- SA(start, time_matrix, maxIterNoChange = 10000)

for (i in 1:9){
  start <- as.numeric(sample(1:n_jobs,n_jobs)) #start randomly
  res   <- SA(start, time_matrix, maxIterNoChange = 10000)
  if (tail(res$traceBest,1) < tail(best_res$traceBest,1))
    best_res <- res
}
ris_SA_20_5 <- tail(best_res$traceBest,1) #best value found
ris_SA_20_5
```

```
## [1] 1297
```

When trying with the 100x20 table, the **GA** algorithm finished in approximately 5 minutes, while **SA** algorithm finished in a few seconds giving more or less the same result (GA best makespan was 6557 while SA best makespan was 6594). The result for the 100x20 table is not provided here for the simplicity of the report.

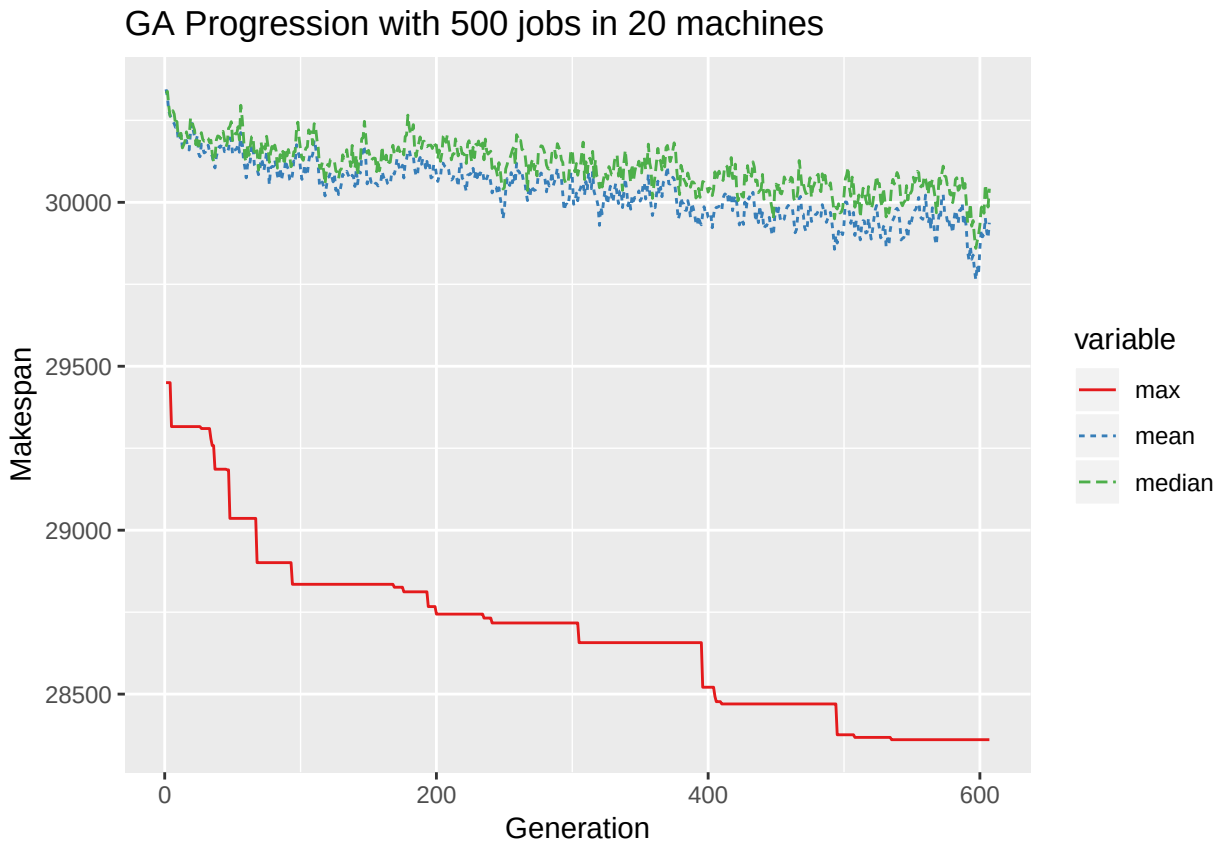
Final Results using 500 jobs with 20 machines.

```
time_matrix <- as.matrix(read.csv("j500-m20",sep=" ",header = FALSE))
n_jobs <- ncol(time_matrix)

time_taken_GA_500_20_1 <- microbenchmark::microbenchmark(
  GA.fit <- ga(type = "permutation",
    fitness = fitnessCpp,
    distMatrix = time_matrix,
    lower = 1,
    upper = n_jobs,
    popSize = 300,
    maxiter = 10000,
    run = 100,
    pmutation = 0.2,
    keepBest = TRUE,
    monitor = NULL,
    seed = 1234),
times = 1
)

ris_GA_500_20 <- makespanCpp(GA.fit@solution[1,],time_matrix)
ris_GA_500_20
```

```
## [1] 28361
```

```
out <- plot(GA.fit, main = "GA progression")
```

```
melt(out[,c(1:3,5)],id.var="iter") %>%
  mutate(inv.value=1/value) -> df1

ggplot(df1, aes(x = iter, y = inv.value,
                group = variable, colour = variable)) +
  xlab("Generation") + ylab("Makespan") +
  geom_line(aes(lty = variable)) +
  scale_colour_brewer(palette = "Set1") +
  labs(title = "GA Progression with 500 jobs in 20 machines")
```



In this case a improving trend can be seen throughout the generation and the median is always higher than the mean, which indicates that a fraction of the population has low values of makespan, which in this case is a good thing.

Since the imporving trend didn't seem to stablize even in the end, it might indicate that we need more iteration before stopping the algorithm, also the initial population is incresed to 500 to have more variability for the algorithm.

```
time_matrix <- as.matrix(read.csv("j500-m20",sep=" ",header = FALSE))
n_jobs <- ncol(time_matrix)

time_taken_GA_500_20 <- microbenchmark::microbenchmark(
  GA.fit <- ga(type = "permutation",
    fitness = fitnessCpp,
    distMatrix = time_matrix,
    lower = 1,
    upper = n_jobs,
    popSize = 500,
    maxiter = 10000,
    run = 200,
    pmutation = 0.2,
    keepBest = TRUE,
    monitor = NULL,
    seed = 1234),
times = 1
)

ris_GA_500_20_2 <- makespanCpp(GA.fit@solution[1,],time_matrix)
```

```
ris_GA_500_20_2
```

```
## [1] 28304
```

```
out <- plot(GA.fit, main = "GA progression")
```

```
melt(out[,c(1:3,5)],id.var="iter") %>%
  mutate(inv.value=1/value) -> df1

ggplot(df1, aes(x = iter, y = inv.value,
                group = variable, colour = variable)) +
  xlab("Generation") + ylab("Makespan") +
  geom_line(aes(lty = variable)) +
  scale_colour_brewer(palette = "Set1") +
  labs(title = "GA Progression with 500 jobs in 20 machines")
```

Using more iteration, more generations are created and the optimal solution is improved, and the is a more stable trend at the ending, increasing even more the iterations, the soution might improve but more time is required for the completion of the algorithm, instead of increasing the number of iteration or the initial population the SA algorithm is tested, which is faster than the GA.

Using the same idea as before, the SA algorithm is executed 10 times and only the best one is kept.

```
best_res = list(traceBest = Inf)

time_taken_SA_500_20 <- microbenchmark::microbenchmark(
  for (i in 1:10){
    start <- as.numeric(sample(1:n_jobs,n_jobs)) #start randomly
    res   <- SA(start, time_matrix, maxIterNoChange = 30000,
```

```
                T_ini = 10000,T_min = 1)
    if (tail(res$traceBest,1) < tail(best_res$traceBest,1))
      best_res <- res
  },times=1)
ris_SA_500_20 <- tail(best_res$traceBest,1) #best value found
ris_SA_500_20
```

```
## [1] 27417
```

```
ggplot(data = data.frame(iteration=1:length(best_res$traceBest),
                         optimal=best_res$traceBest),
       aes(iteration,optimal)) +
  geom_line() +
  xlab("Iteration") + ylab("Best Makespan") +
  scale_colour_brewer(palette = "Set1") +
  labs(title = "SA Progression with 500 jobs in 20 machines")
```

The **SA** algorithm solution is better than the one given by the **GA** algorithm, and timewise **SA** is very fast when compared to the **GA** algorithm.

For the Extra point, NEH algorithm for the suboptimal solution

The NEH algorithm is a constructive algoritm, due to Nawaz, Ensore and Ham, basically in this algorithm consists in: 1. Ordering the n jobs by decreasing sums of processing time on the machines. 2. Take the first job and schedule it initially as the first job. 3. For all the ordered jobs from 2 to n where n = total number of jobs, do the following: for k=2:n insert the k-th job at the position, which minimizes the partial makespan.

The algorithm time complexity, say m=number of machines and n=number of jobs: The first step is done in summing all the jobs for m machines each $O(nm)$, the second step is O(1), during the third step, for the k-th job the time required is $O(km)$, which is the time required by the makespan function, plus the time to find the minimum of the times is $O(k) \subset O(km)$, since it is for all k, the total time is $O(nm) + \sum_{k=2}^{n} O(km) = O(nm) + O(m \sum_k k) = O(n^2 m)$.

In this implementation the makespan function is not comletely optimized so the time order might be bigger, but since it was implemented in C++, it still executes really fast. There are methods to compute the makespan in a time order of $O(km)$ for possible position of the k-th job.

```
insert_at <- function(x,pos,val){
  "inserts at a given position (pos) the value (val) in the array x"
  if (pos==1)
    return(c(val,x))
  len <- length(x)
  if (pos==(len+1))
    return(c(x,val))
  return(c(x[1:pos-1],val,x[pos:len]))
}

NEH <- function(distMatrix,fun.obj,REPORT=0){
  #Part 1
  sum <- rbind(colSums(distMatrix),1:ncol(distMatrix))
  sum_order <- as.numeric(sum[2,order(sum[1,],
                                      decreasing = TRUE)])
  #Part 2
  job <- sum_order[1]

  #Part 3
  for (i in 2:ncol(distMatrix)){
    temp_ris <- lapply(1:(length(job)+1), function(x) {
                           temp_job = insert_at(job,x,sum_order[i])
                           t <- fun.obj(1:(length(job)+1),
                                        distMatrix[,temp_job])
                           return(list(temp_job=temp_job, t=t))
                       })
    t <- unlist(lapply(temp_ris, '[[','t'))
    job <- lapply(temp_ris, '[[','temp_job')[[which.min(t)]]
    if(REPORT!=0 && i%%REPORT==0)
      print(paste0("Done ",as.character(i)," jobs"))
  }
  return(list(sol   = job,
              value = fun.obj(job,distMatrix) ))
}
```

```
time_taken_NEH_500_20 <- microbenchmark::microbenchmark(
  ris <- NEH(distMatrix = time_matrix, fun.obj = makespanCpp),
times = 1
)

ris_NEH_500_20 <- ris$value
ris_NEH_500_20
```

```
## [1] 26670
```

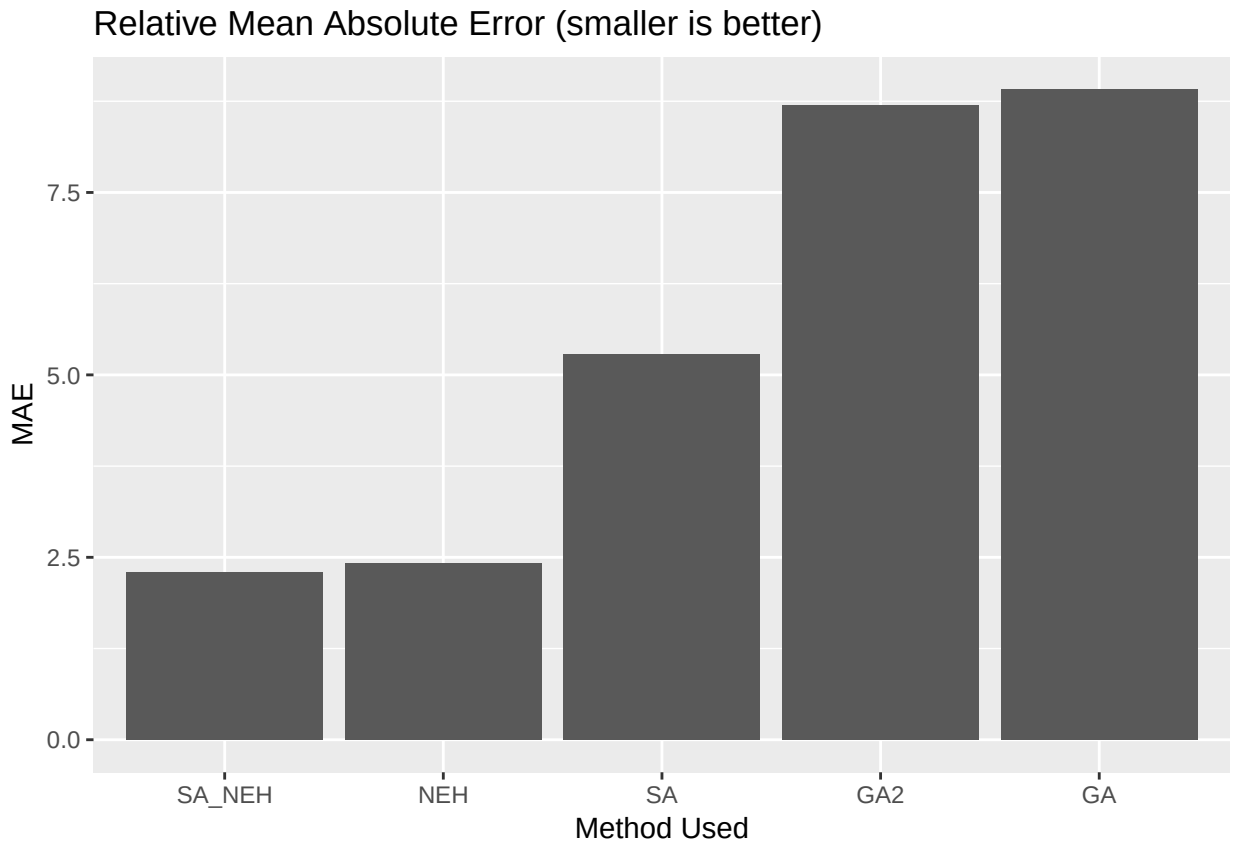
The temperature of NEH can be setted as a low value, so it will search for a lower solution near the initial

solution which is the one returned by the NEH method, improving possibly further the optimal solution, the lowest value this additional method on NEH has obtained has been 26652 during the testing phases, but the result is pretty random so it may not reach that value during the new compilation of the report, or it may even find a lower value.

```
best_SA_after_NEH <- Inf
time_taken_SA_NEH_500_20 <- microbenchmark::microbenchmark(
  for (i in 1:10){
    sa.sol <- SA(ris$sol,time_matrix,
                 maxIterNoChange = 10000,T_ini = 10,T_min = 1)
    if (tail(sa.sol$traceBest,1) < best_SA_after_NEH)
      best_SA_after_NEH <- tail(sa.sol$traceBest,1)
  },times=1)
best_SA_after_NEH
```

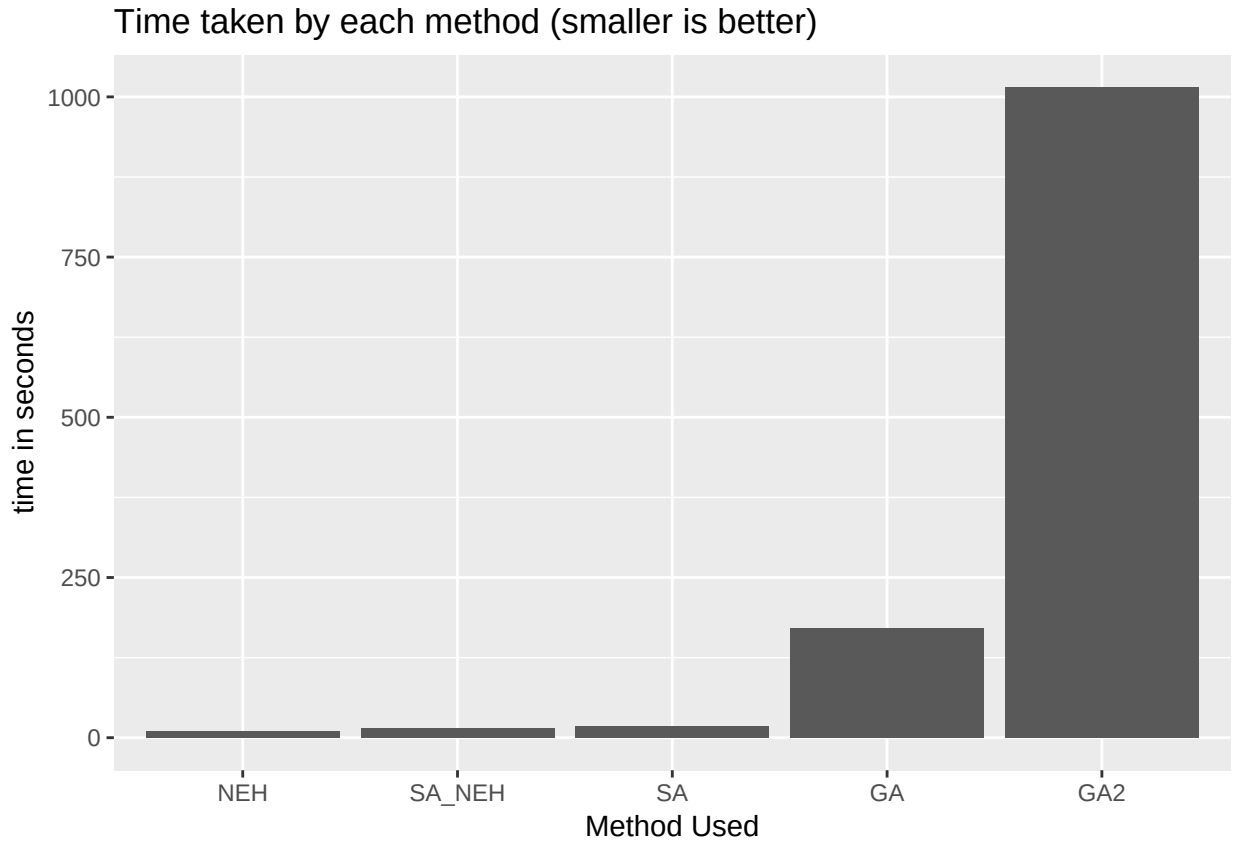
```
## [1] 26639
```

```
best <- 26040 #-- Value from the website
residue_GA     <- (abs(ris_GA_500_20 - best)/best)*100
residue_GA2    <- (abs(ris_GA_500_20_2 - best)/best)*100
residue_SA     <- (abs(ris_SA_500_20 - best)/best)*100
residue_NEH    <- (abs(ris_NEH_500_20 - best)/best)*100
residue_NEH_SA <- (abs(best_SA_after_NEH - best)/best)*100

df_bar <- data.frame(method = c("GA","GA2","SA","NEH","SA_NEH"),
                     val = c(residue_GA,residue_GA2,
                       residue_SA,residue_NEH,
                       residue_NEH_SA))
ggplot(data=df_bar, aes(x=reorder(method,val),y=val)) +
  geom_bar(stat = "identity") +
  labs(title = "Relative Mean Absolute Error (smaller is better)")+
       xlab("Method Used")+ ylab("MAE")
```



The MAE is clearly lower for NEH (and after executing SA after the solution found by NEH), while the GA has an error bigger than 8.5%, the SA after NEH usually improves the solution by a little bit (sometimes it doesn't help it, it's pretty random as expected).

```
df_time <- data.frame(method = c("GA","GA2","SA","NEH","SA_NEH"),
                      time = c(time_taken_GA_500_20_1$time/10^9,
                               time_taken_GA_500_20$time/10^9,
                               time_taken_SA_500_20$time/10^9,
                               time_taken_NEH_500_20$time/10^9,
                               time_taken_SA_NEH_500_20$time/10^9+
                                 time_taken_NEH_500_20$time/10^9))
ggplot(data=df_time, aes(x=reorder(method,time),y=time)) +
  geom_bar(stat = "identity") +
  labs(title = "Time taken by each method (smaller is better)")+
   xlab("Method Used")+ ylab("time in seconds")
```



The time is lower for the NEH method which also gives the best solution, while the GA 2nd version with more population and runs takes more than 1000 seconds to execute not giving a satisfying solution.