Name : Anshuman Kumar Sharma
Roll No. : 21419CMP007
Subject : Cloud Computing
Semester : MSc. Semester - III (2021 - 2023)
Under the Guidance of : *Dr. Gaurav Baranwal*

Report 1: Scheduling

Why Scheduling is NP Hard ?

VM (Virtual Machine) scheduling is considered an NP-hard problem because it involves finding an optimal solution among a large number of possibilities, and it's difficult to determine the best solution in a reasonable amount of time. The problem is complex because it involves multiple constraints such as resource availability, resource allocation, and workload demands, which must be taken into consideration when scheduling VMs. The problem becomes even more challenging as the number of VMs and resources increases. To find an optimal solution, exhaustive search and optimization algorithms may take an exponential amount of time, making it infeasible to find a solution in a reasonable time frame, making it an NP-hard problem.

There are several algorithms for VM scheduling, we have implemented
Optimal Algorithm (OA): This algorithm uses brute force to find the optimal result.

Genetic Algorithm (GA) : This algorithm is an optimization method inspired by the concept of natural selection and genetics.

Greedy Algorithm (GrA) : This algorithm uses weight / value heuristic to find best solution.

Pseudo Code:

DriverMain Algorithm

```
Input: SIZE, LOOP

GENETIC <- new instance of GeneticAlgorithm
OPTIMAL <- new instance of Optimal
GREEDY <- new instance of Greedy
WTCSV <- new instance of WriteToCsv
PD <- new instance of PoissionDistribution

RANDOM_BURST_TIME_MAX <- 200
```

```
RANDOM_BURST_TIME_MIN <- 100

RANDOM_PRICE_MAX <- 200
RANDOM_PRICE_MIN <- 100

for i <-0 to LOOP do,
     temp <- PD.getPoissionRandom(SIZE)
     executionTimeArr <- Array of size temp
     priceArr <- Array of size temp
     capacity <- randomly generate a capacity
     for j <- 0 to temp do,
          executionTime j <- randomly generate burst time
          priceArr j <- randomly generate price
     end
     A <- OPTIAML.optimal(executionTimeArr, priceArr, capacity)
     B <- GREEDY.greedy(executionTimeArr, priceArr, capacity)
     C <- GENETIC.genetic(executionTimeArr, priceArr, capacity)
     Display the data
     WRCSV save the data to a file
end
```

SuperScheduling Algorithm

```
MAIN(ARGS)

DM <- new instance of DRIVERMAIN
SIZE <- 30
for i <- 5 to SIZE do,
     DM.driver(i, 10000)
```

Greedy Algorithm

```
Input: EXECUTION_TIME_ARR, PRICE_ARR, CAPACITY
Output: GREEDY_VALUE

start <- start time
itemValue <- [EXECUTION_TIME_ARR 0 to n, PRICE_ARR 0 to n]
itemvalue <- Sort item value wrt Ratio
maxVal <- 0
currentCapacity <- 0
for i <- 0 to len(itemValue) do
     if CAPACITY - current item >= 0 then
```

```
          currentCapacity = currentCapacity + current item

end <- end time
greedyExecutionTime <- end - start
return [maxValue, greedyExecutionTime]
```

Optimal Algorithm

```
Input: EXECUTION_TIME_ARR, PRICE_ARR, CAPACITY
Output: OPTIMAL_VALUE

start <- start time
maxProfit <- 0
maxProfitCombination <- []
noOfTasks <- no of tasks
totalNoOfCombinations = 2^noOfTasks
for i <- 0 to totalNoOfCombinations do
     temp <- Array of Combinations
     tempProfit <- 0
     tempCapacity <- 0
     for j <- 0 len(temp) do
          if temp j == 1 then
               Add profit
          else
               Skip
     end
if tempCapacity <= CAPACITY then
     if maxProfit < tempProfit then
          maxProfit <- tempProfit
          maxProfitCombination <- max profit combination
end <- end time
optimalExecutionTime <- end - start
return [maxValue, optimalExecutionTime]
```

Genetic Algorithm

```
Input: EXECUTION_TIME_ARR, PRICE_ARR, CAPACITY
Output: GENETIC_VALUE

start <- start time
t <- create Tasks
p.createPopulation <- Create Population from t
p.calculateFitness <- Calculate Fitness Value
```

```
p.sort <- Sort Genes wrt to there Fitness Value
while i <- 0 do
     if previous 5 generations are same, then
          stop
     else
          p.generation <- generation + 1
          p.selection <- Initialise Selection
          p.mutation <- Do Mutation
          p.calculateFitness <- Calculate Fitness Value
          p.sort <- Sort Genes wrt to there Fitness Value
          i <- i + 1
end
end <- end time
geneticExecutionTime <- end - start
return [p.population[0], ogeneticExecutionTime]
```

calculateFitness Algorithm

```
Input: TASKS

for i <- 0 to len(population) do
     temp <- 0
     tempPrice <- 0
     for j <- 0 len(chormo) do
          Calculate the Fitness value based on Execution Time
     end
end
```

selection Algorithm

```
for i <- 0 len(population) do
     if probability < 0.1 then
          Do CrossOver
     i <- i + 1
end
```

crossOver Algorithm

```
Input: A, B
Output: NewChromo
crossOverPoint <- generate a random point between the length of
chromo
NewChromo <- Repalce A from the crossOverPoint with B
return NewChromo
```

mutation Algorithm

```
for i <- 0 to len(population) do
     if probability < 0.05 then
          Do Single Bit Crossover
end
```

Experiment Analysis

System Configuration:

```
description: Desktop Computer
     product: Virtual Machine (None)
     vendor: Microsoft Corporation
     version: Hyper-V UEFI Release v4.1
     serial: 0000-0014-3196-3636-3917-2163-45
     width: 64 bits

Distributor ID:    Ubuntu
Description:    Ubuntu 22.04.1 LTS
Release:    22.04

cpu
          description: CPU
          product: Intel(R) Xeon(R) Platinum 8272CL CPU @2.60GHz
          vendor: Intel Corp.
          physical id: 4
          bus info: cpu@0
          version: 6.85.7
          serial: None
          slot: None
          size: 2600MHz
          capacity: 3700MHz
          width: 64 bits
          clock: 100MHz

memory
          description: System Memory
          physical id: 6
          slot: System board or motherboard
          size: 8GiB
```

```
disk:0
          description: SCSI Disk
          version: 1.0
          size: 30GiB (32GB)
```

Programming Language: Java

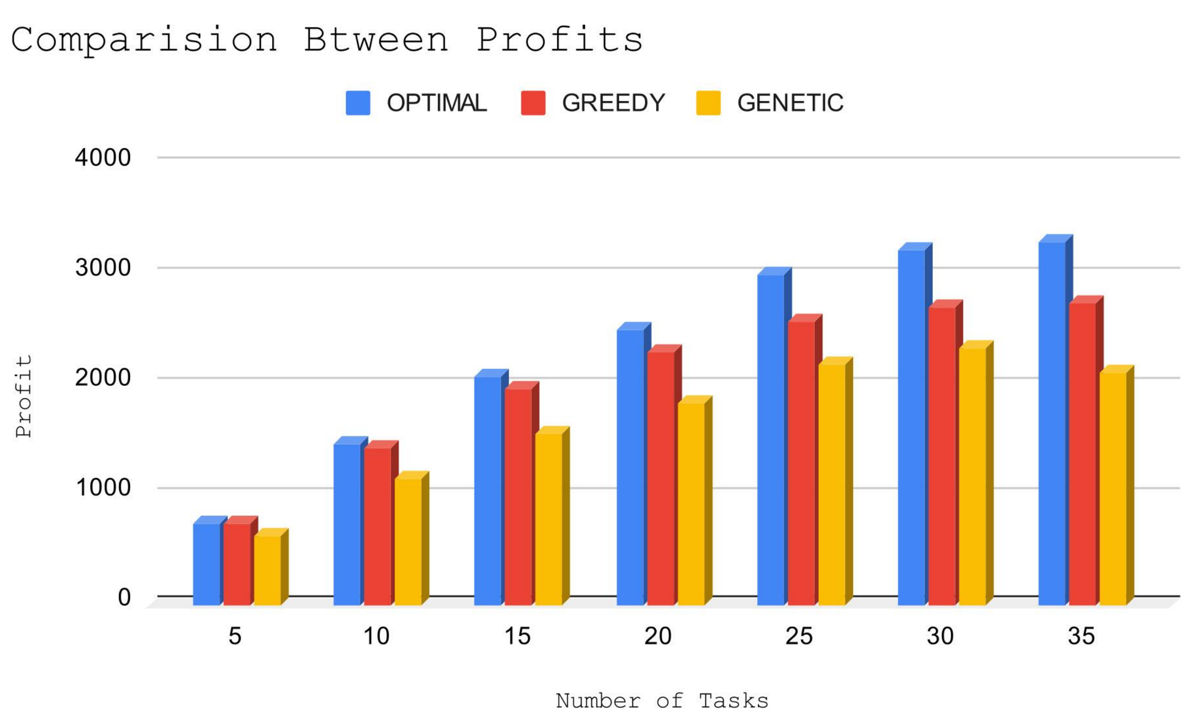
Task Length and Why ?

I have taken these lengths because if we take more tasks length then it will be not possible to visualize readings obtained by greedy and genetic algorithms in comparison with Brute force method on Execution Time.

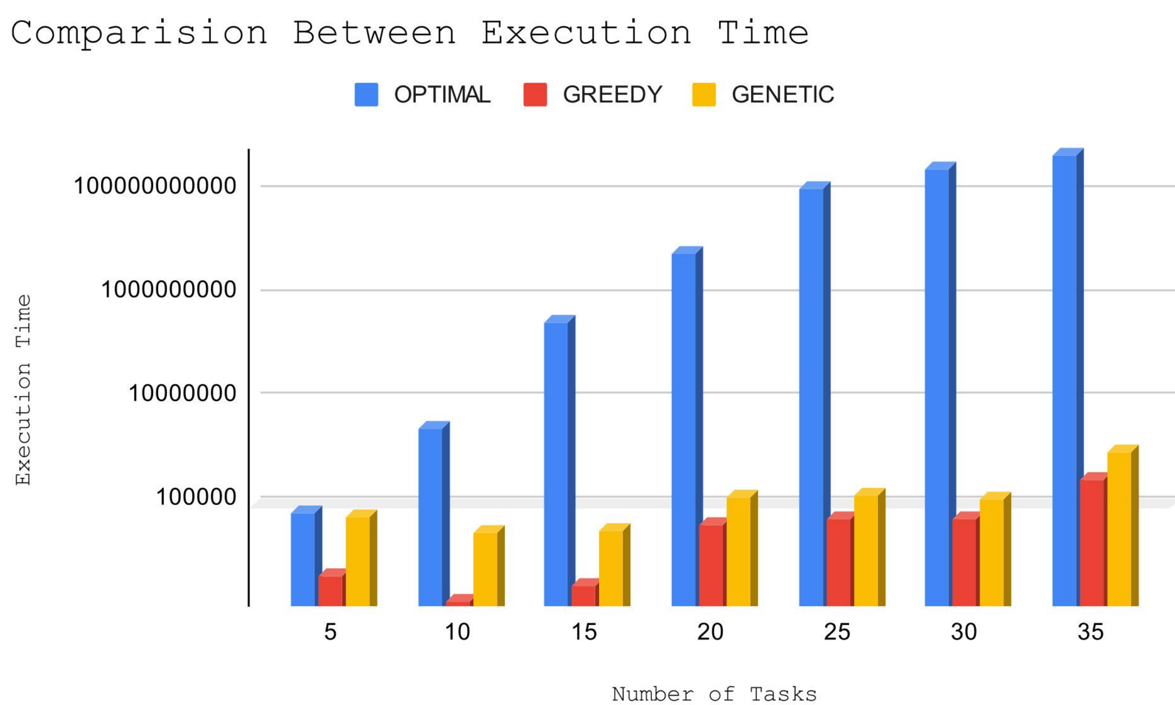
User Price and Why?

I have taken Random values for every users and in range 100 to 200, for simplicity.

Graphs:



Here we can see that Optimal is giving the best solution compared to Greedy and Genetic Algorithms.



As we can see that Optimal Algorithm was giving the best solution, but in this graph we can see that it's execution time is increasing exponentially wrt number of users. The time complexity of the Genetic algorithm is very similar in every case, because it stops when the population is repeating.

After comparing Both the Graphs Greedy Algorithm is giving the best solution in a short period of time.

Report 2: Rank Reversal in TOPSIS

MCDM stands for Multi-Criteria Decision Making. It is a systematic approach to decision making that involves evaluating multiple criteria or factors to determine the best course of action. In MCDM, the decision maker considers multiple criteria that are relevant to the decision at hand and uses mathematical models and algorithms to determine the best outcome based on those criteria. The criteria can include both qualitative and quantitative factors, such as cost, time, risk, and feasibility. MCDM is widely used in fields such as engineering, management, operations research, and environmental science, where decisions are often complex and involve multiple interrelated factors. The goal of MCDM is to provide a transparent and consistent approach to decision making that can be easily understood and justified, thereby improving the quality of decision making and increasing the chances of success.

Rank reversal is a phenomenon that occurs when a decision maker, in the process of selecting an alternative from a set of choices, is confronted with new alternatives that were not thought about when the selection process was initiated. It depends on the relationship between this new alternative and the old ones under each criterion.

Pseudo code:

Topsis
dataNormilisation Algorithm
Input: MATRIX
Output: MATRIX (Normalised)

```
temp <- MATRIX (make a copy of MATRIX array)
tempRootSum <- [] len = MATRIX[0]
for i <- 0 to len(temp) do
     for j <- 0 to len(temp[0]) do
          temp ij = temp ij^2
     end
end
for i <-0 to len(temp[0]) do
```

```
     tempSum <- 0
     for j <- 0 to len(temp) do
          tempSum <- tempSum + tempSum_ij
     end
     tempSum <- tempSum^0.5
     tempRootSum <- tempSum
end
for i <- 0 to len(MATRIX) do
     for j <- 0 to len(MATRIX[0]) do
          Matrix_ij = matrix_ij / tempRootSum_j
     end
end
return MATRIX
```

weightedMatrix Algorithm

```
Input: MATRIX, WEIGHTS
Output MATRIX (weighted Matrix)

for i <- 0 to len(MATRIX) do
     for j <- 0 to len(MATRIX[0]) do
          MATRIX_ij = MATRIX_ij * WEIGHTS
     end
end
return MATRIX
```

calPositiveSolution Algorithm

```
Input: MATRIX, BENEFICIAL (Boolean Array)
Output: POSITIVESOLUTION (Array)
POSITIVESOLUTION <- []
for i <- 0 to len(MATRIX[0]) do
     tempMaxMin <- MATRIX_0i
     for j <- 0 to len(MATRIX) do
          if BENEFICIAL == true then
               if tempMaxMin < MATRIX_ji then
                    tempMaxMin <- MATRIX_ji
          else
               if tempMaxMin > MATRIX_ji then
                    tempMaxMin <- MATRIX_ji
     end
     POSITIVESOLUTION_i <- tempMaxMin
```

```
end
return POSITIVESOLUTION
```

calNegativeSolution Algorithm

```
Input: MATRIX, BENEFICIAL (Boolean Array)
Output: NEGATIVESOLUTION (Array)
NEGATIVESOLUTION <- []
for i <- 0 to len(MATRIX[0]) do
     tempMaxMin <- MATRIX 0i
     for j <- 0 to len(MATRIX) do
          if BENEFICIAL != true then
               if tempMaxMin < MATRIX ji then
                    tempMaxMin <- MATRIX ji
          else
               if tempMaxMin > MATRIX ji then
                    tempMaxMin <- MATRIX ji
     end
     NEGATIVESOLUTION i <- tempMaxMin
end
return NEGATIVESOLUTION
```

calSeprationMatrix ALgorithm

```
Input: MATRIX, SEPRATIONMATRIX, SOLUTION
Output: SEPERATIONMATRIX
for i <- 0 to len(MATRIX) do
     tempSum <- 0
     for j <- 0 to  len(MATRIX[0]) do
          tempCal <- matrix ij - SOLUTION j
          tempSum <- tempSum + tempCal^2
     end
     tempSum = tempSum^0.5
     SEPERATIONMATRIX i = tempSum
end
return SEPERATIONMATRIX
```

calRelativeCofficientValue Algorithm

```
Input: RELATIVE_COFFICIENT_VALUES, POSITIVE_SEPERATION_MATRIX,
NEGATIVE_SEPERATION_MATRIX
Output: RANK
for i <- 0 to RELATIVE_COFFICIENT_VALUES do
```

```
     RELATIVE_COFFICIENT_VALUES i = NEGATIVE_SEPERATION_MATRIX i
/ (NEGATIVE_SEPERATION_MATRIX i + POSITIVE_SEPERATION_MATRIX i)
end
return RELATIVE_COFFICIENT_VALUES
```

calRank Algorithm

```
Input: RANK, RELATIVE_COFFICIENT_VALUES
OUTPUT RANK
temp <- RELATIVE_COFFICIENT_VALUES (copy)
for i <- 0 to len(RELATIVE_COFFICIENT_VALUES) do
     maxVal <- MAX(temp)
     for j <- 0 to len(RELATIVE_COFFICIENT_VALUES) do
          if maxVal == temp j then
               RANK j <- tempCount
               tempCOunt <- tempCount + 1
               temp j <- 0
     end
end
Return RANK
```

Rank Reversal in Topsis:

```
Matrix 1:              Ranking (Calculated by the Algorithm)
    [[ 1, 5 ],            3
     [ 4, 2 ],            1
     [ 3, 3 ]]            2

Matrix 2:              Ranking (Calculated by the Algorithm)
    [[ 1, 5 ],            1
     [ 4, 2 ],            3
     [ 3, 3 ],            2
     [ 5, 1 ]]            4
```

First we calculated the Rank using the Algorithm
As we can observe that the first alternative, which was previously the worst, has now become the best. This is the Rank Reversal problem in TOPSIS Algorithm. It is due to the Normalize Function used in Topsis.