Faculty of Engineering and Information Technology

Computer Science Department

Artificial Intelligence

Local Search

COMP338 : Assignment #2

Prepared by:

Worood Assi – 1210412

Instructor:

Radi Jarrar

Date :

11-6-2024

Contents

Problem:

Seating Arrangement Optimization.

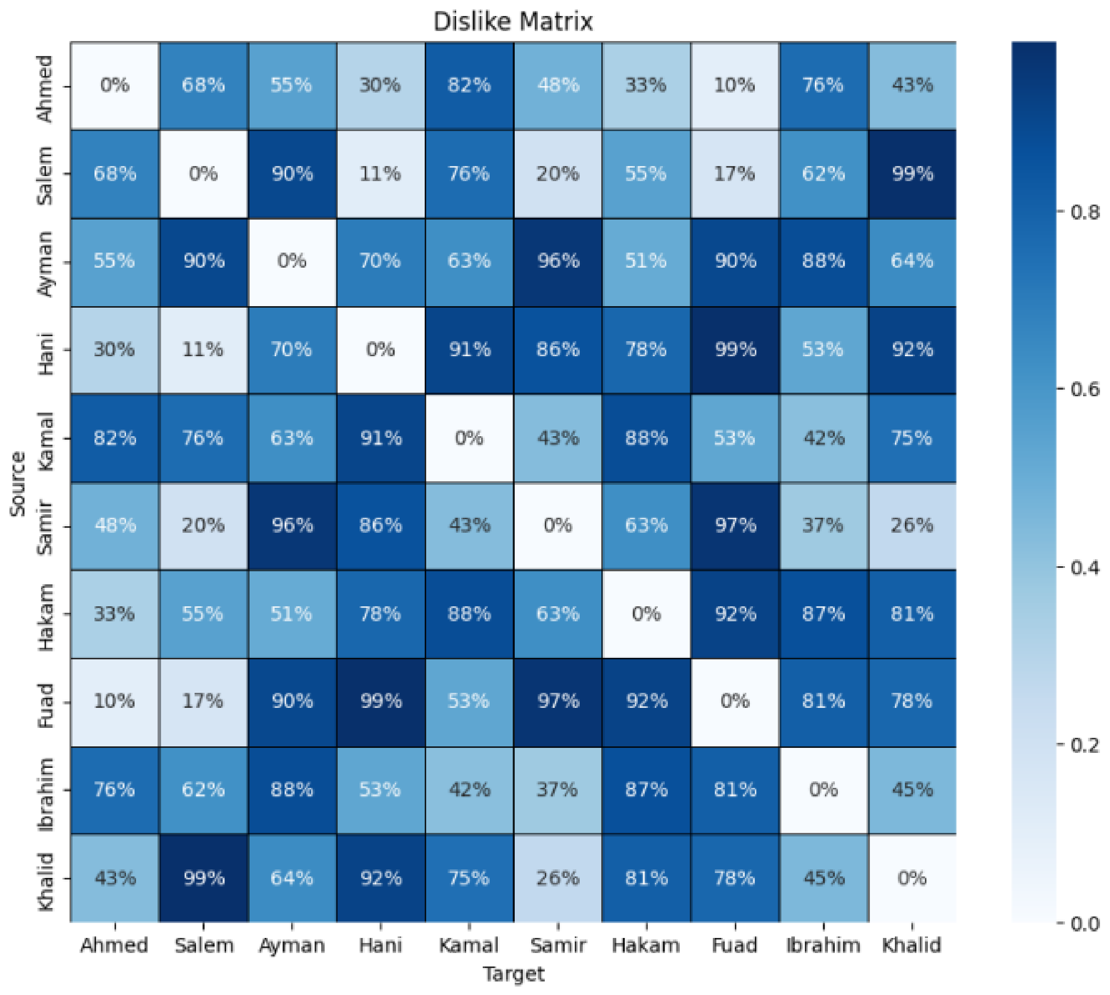
Objective:

Design and implement a program that optimizes the seating arrangement of a group of people based on their mutual dislikes. The goal is to minimize the total cost of seating people together by using three optimization techniques: **Genetic Algorithm**, **Simulated Annealing**, **and Hill Climbing.**

Problem Description:

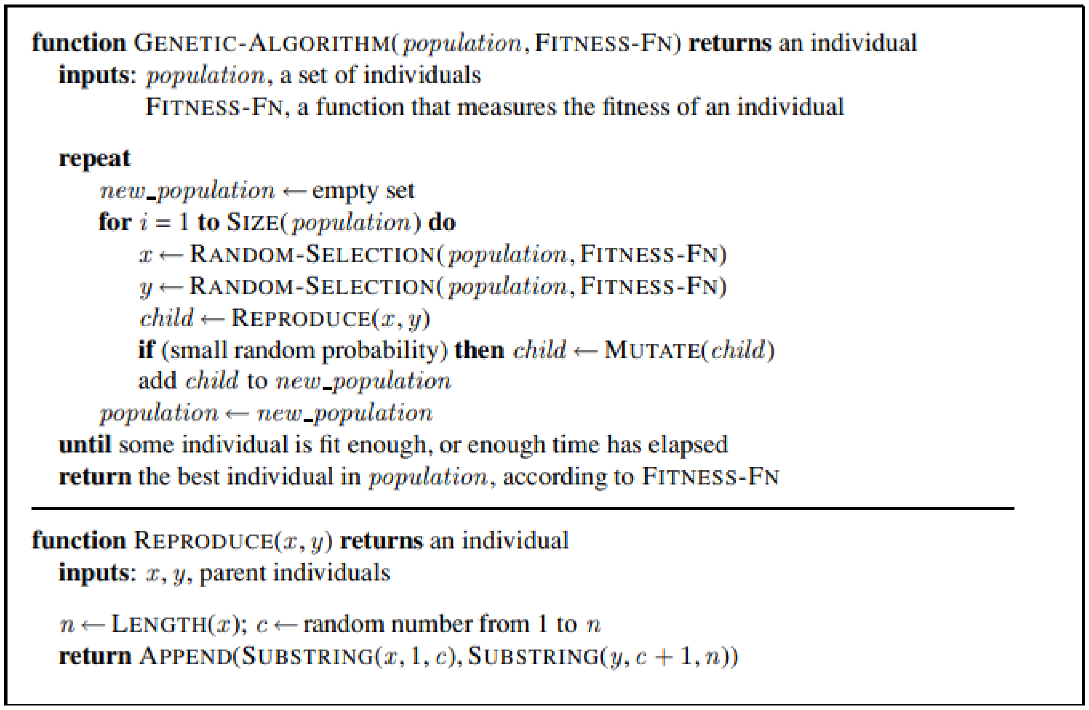
I am given a group of 10 people and a 10x10 matrix representing the dislike costs between each pair of people. The cost of seating two people together is given by the matrix, where the value at the intersection of row i and column j represents the cost of seating person i next to person j. The seating arrangement is circular, meaning the first and the last person are also considered to be seated next to each other.

Dislike Matrix:



Genetic Algorithm:

Genetic Algorithms(GAs) are adaptive heuristic search algorithms that belong to the larger part of evolutionary algorithms. Genetic algorithms are based on the ideas of natural selection and genetics. These are intelligent exploitation of random searches provided with historical data to direct the search into the region of better performance in solution space. They are commonly used to generate high-quality solutions for optimization problems and search problems.



```
function GENETIC-ALGORITHM($population$, FITNESS-FN) returns an individual
  inputs: $population$, a set of individuals
          FITNESS-FN, a function that measures the fitness of an individual

  repeat
     $new\_population \leftarrow$ empty set
     for $i$ = 1 to SIZE($population$) do
        $x \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
        $y \leftarrow$ RANDOM-SELECTION($population$, FITNESS-FN)
        $child \leftarrow$ REPRODUCE($x, y$)
        if (small random probability) then $child \leftarrow$ MUTATE($child$)
        add $child$ to $new\_population$
     $population \leftarrow new\_population$
  until some individual is fit enough, or enough time has elapsed
  return the best individual in $population$, according to FITNESS-FN

function REPRODUCE($x, y$) returns an individual
  inputs: $x, y$, parent individuals

  $n \leftarrow$ LENGTH($x$); $c \leftarrow$ random number from 1 to $n$
  return APPEND(SUBSTRING($x, 1, c$), SUBSTRING($y, c + 1, n$))
```

Operators of Genetic Algorithms:

Once the initial generation is created, the algorithm evolves the generation using following operators :

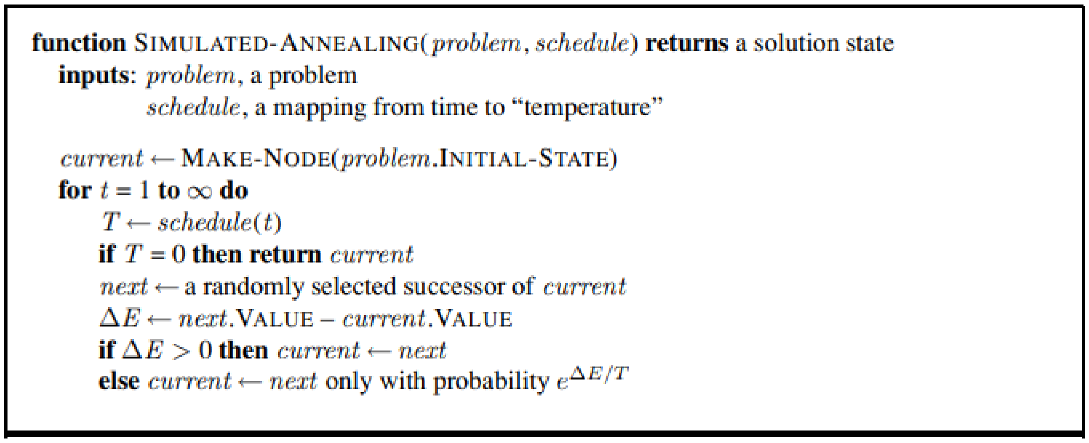
1) Selection Operator: The idea is to give preference to the individuals with good fitness scores and allow them to pass their genes to successive generations.

2) Crossover Operator: This represents mating between individuals. Two individuals are selected using selection operator and crossover sites are chosen randomly. Then the genes at these crossover sites are exchanged thus creating a completely new individual (offspring).

3) Mutation Operator: The key idea is to insert random genes in offspring to maintain the diversity in the population to avoid premature convergence.

Simulated Annealing:

Simulated Annealing is an optimization algorithm inspired by the annealing process in metallurgy. It combines elements of hill climbing and random walks to find global optima efficiently. The algorithm starts with intense shaking (high temperature) and gradually reduces it (lower temperature). In each step, it randomly explores neighboring states and accepts moves that improve the situation. The probability of accepting "bad" moves decreases as the temperature decreases. Simulated annealing has been used for solving optimization problems, including VLSI layout and factory scheduling.



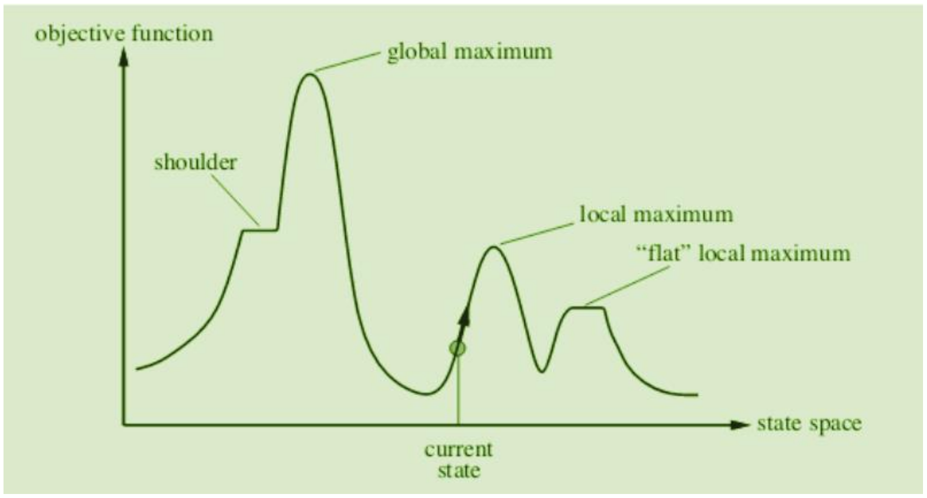
```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
  inputs: problem, a problem
          schedule, a mapping from time to "temperature"

  current ← MAKE-NODE(problem.INITIAL-STATE)
  for t = 1 to ∞ do
      T ← schedule(t)
      if T = 0 then return current
      next ← a randomly selected successor of current
      $\Delta E$ ← next.VALUE − current.VALUE
      if $\Delta E > 0$ then current ← next
      else current ← next only with probability $e^{\Delta E/T}$
```

If the move improves the solution, it is always accepted. Otherwise, it is accepted with a probability that decreases exponentially with the "badness" of the move (ΔE) and as the "temperature" (T) decreases. This means that worse moves are more likely to be accepted at the start when T is high, helping the algorithm escape local minima. As T lowers gradually, the acceptance of bad moves becomes less likely, guiding the algorithm towards an optimal solution. If T decreases slowly enough, the algorithm will find a global optimum with high probability.

Hill Climbing:

Hill climbing is a straightforward optimization algorithm used in Artificial Intelligence (AI) to find the best solution for a given problem. It belongs to the family of local search algorithms and is commonly applied to optimization problems where the goal is to identify the best solution from a set of possibilities. The algorithm starts with an initial solution and iteratively makes small changes based on a heuristic function that evaluates solution quality, continuing until it reaches a local maximum where no further improvements are possible. Variations include steepest ascent, where the best move is selected from all possible moves; first-choice, where a random move is accepted if it improves the solution. Hill climbing is useful in various optimization tasks like scheduling, route planning, and resource allocation, but it can get stuck in local maxima and lacks search space diversity. Therefore, it is often combined with other techniques, such as genetic algorithms or simulated annealing, to enhance search results.

```
function HILL-CLIMBING(problem) returns a state that is a local maximum
  current ← MAKE-NODE(problem.INITIAL-STATE)
  loop do
     neighbor ← a highest-valued successor of current
     if neighbor.VALUE ≤ current.VALUE then return current.STATE
     current ← neighbor
```



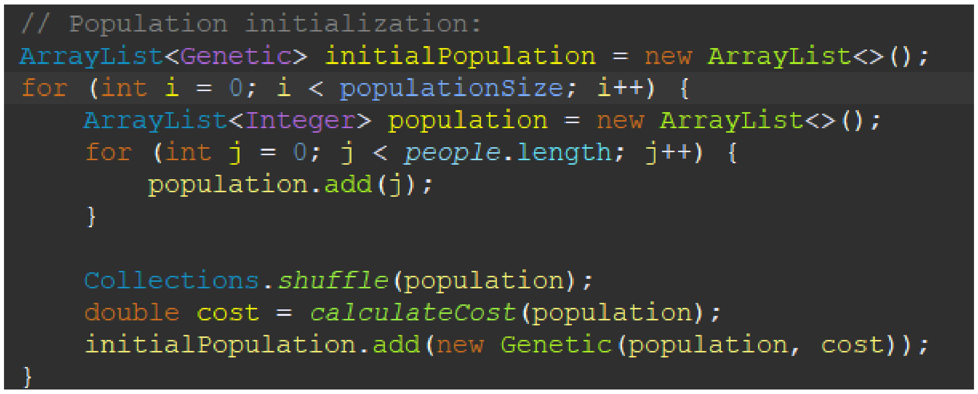
Evaluating the effectiveness of algorithms:

Genetic Algorithm:

I created a geneticAlgorithm(int populationSize, int numGenerations, double mutationRate) method that takes three parameters to find the optimal seating arrangement. The first parameter represents the number of individuals in the randomly generated population. The second parameter is the number of generations to be produced from the initial population. The third parameter is the mutation rate, which determines how frequently the positions of two people are swapped in the generated populations. The algorithm operates in four stages:

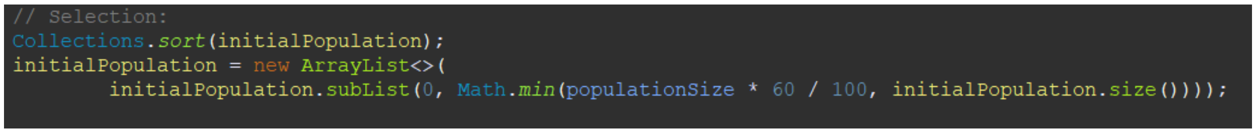
1. Population initialization: Randomly generate the initial population.
2. Selection: Use elitism to select the top 60% of the population based on their cost.
3. Crossover: Combine pairs of parents to create offspring.
4. Mutation: Randomly swap positions of two people in an arrangement with a given mutation rate.

First, I randomly generated the required number of populations (100) using the *ArrayList(Collections.shuffle)* method and calculated their cost through the calculate_cost(arrangement) method, which will be explained later.



```java
// Population initialization:
ArrayList<Genetic> initialPopulation = new ArrayList<>();
for (int i = 0; i < populationSize; i++) {
    ArrayList<Integer> population = new ArrayList<>();
    for (int j = 0; j < people.length; j++) {
        population.add(j);
    }

    Collections.shuffle(population);
    double cost = calculateCost(population);
    initialPopulation.add(new Genetic(population, cost));
}
```

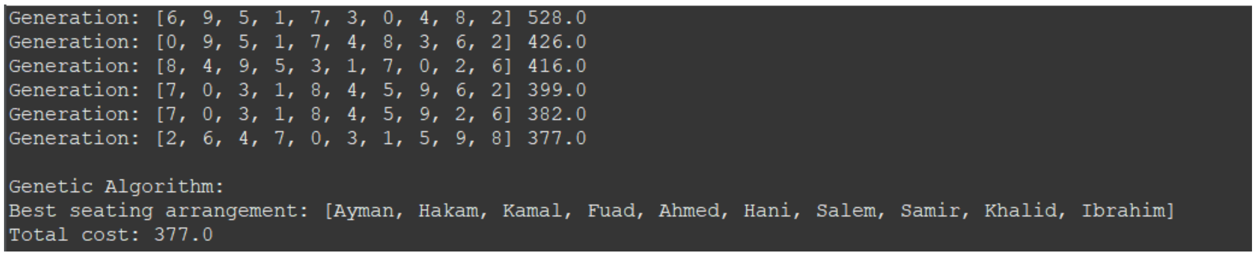
Second, I ordered these initial populations based on their cost and selected the top 60% with the lowest costs.



```java
// Selection:
Collections.sort(initialPopulation);
initialPopulation = new ArrayList<>(
        initialPopulation.subList(0, Math.min(populationSize * 60 / 100, initialPopulation.size())));
```

Then, I started the crossover process by splitting each chromosome in half and merging the first half of the first chromosome, called child1, with the second half of the second chromosome, called child2. To avoid duplication, before adding a gene from the second half to the first half, I checked if the gene already existed. If it didn't, I added it; if it did, I looked for another gene that didn't exist and added it instead. I repeated this process until I reached 1000 generations. Finally, to implement the mutation, I randomly selected two genes at a rate of 0.1 of the total number of genes and swapped them with each other.

I then calculated the cost of each new one and compared it to the previous one to determine if it was better. For example:



```
Generation: [6, 9, 5, 1, 7, 3, 0, 4, 8, 2] 528.0
Generation: [0, 9, 5, 1, 7, 4, 8, 3, 6, 2] 426.0
Generation: [8, 4, 9, 5, 3, 1, 7, 0, 2, 6] 416.0
Generation: [7, 0, 3, 1, 8, 4, 5, 9, 6, 2] 399.0
Generation: [7, 0, 3, 1, 8, 4, 5, 9, 2, 6] 382.0
Generation: [2, 6, 4, 7, 0, 3, 1, 5, 9, 8] 377.0

Genetic Algorithm:
Best seating arrangement: [Ayman, Hakam, Kamal, Fuad, Ahmed, Hani, Salem, Samir, Khalid, Ibrahim]
Total cost: 377.0
```

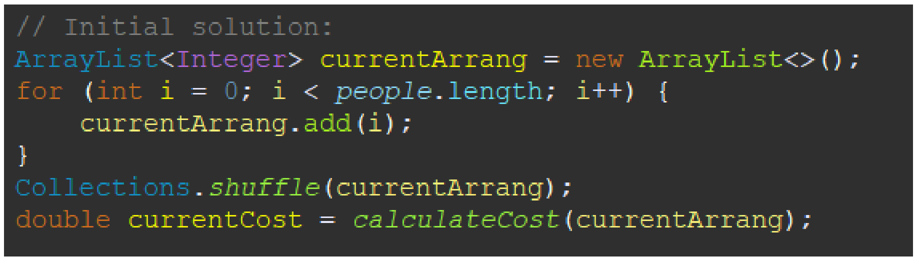
Simulated Annealing:

To solve the seating arrangement issue using annealing, I created a simulatedAnnealing(int initialTemperature, double coolingRate, int numIterations) method. the first parameter represents the initial temperature at which the algorithm will start so that it is a high temperature, which affects the algorithm's behaviour and speed, making it more exploratory and slower to reach the solution, the second is the cooling rate that will affect the temperature value and reduce it over time, and the third is the number of iterations in exploring the solution and the higher the number of iterations, the higher the probability of finding the best solution, the number is chosen according to the difficulty of the issue.

Algorithm Solution Restrictions:

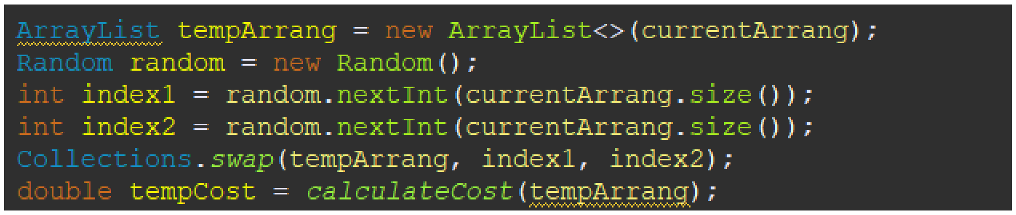
1. Initial solution: Start with a random arrangement.
2. Temperature: Decrease the temperature gradually according to the cooling rate.
3. Acceptance probability: Accept worse solutions with a probability depending on the current temperature to escape local minima.

So, first, I created a random seating arrangement and calculate its cost. Then created other arrangements,



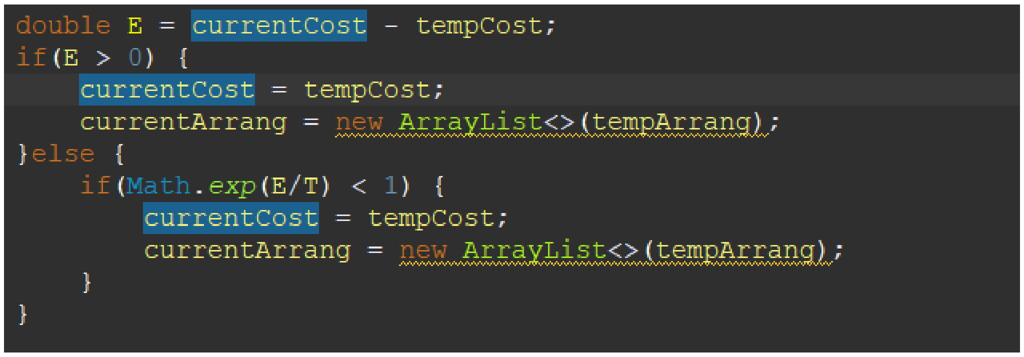
```java
// Initial solution:
ArrayList<Integer> currentArrang = new ArrayList<>();
for (int i = 0; i < people.length; i++) {
    currentArrang.add(i);
}
Collections.shuffle(currentArrang);
double currentCost = calculateCost(currentArrang);
```

Exploring further and looking for a better solution by switching the order of people.



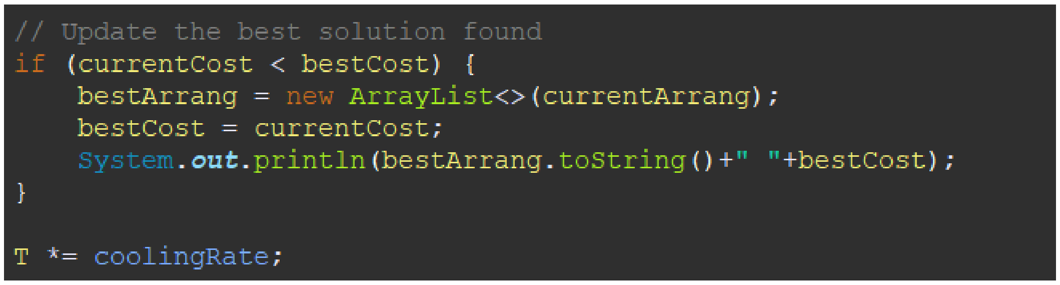
```java
ArrayList tempArrang = new ArrayList<>(currentArrang);
Random random = new Random();
int index1 = random.nextInt(currentArrang.size());
int index2 = random.nextInt(currentArrang.size());
Collections.swap(tempArrang, index1, index2);
double tempCost = calculateCost(tempArrang);
```

Here's the thing: if the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some **probability e^ΔE/T less than 1**. The probability decreases exponentially with the "badness" of the move—the amount **ΔE** by which the evaluation is worsened. The probability also decreases as the 'temperature' **T** goes down.



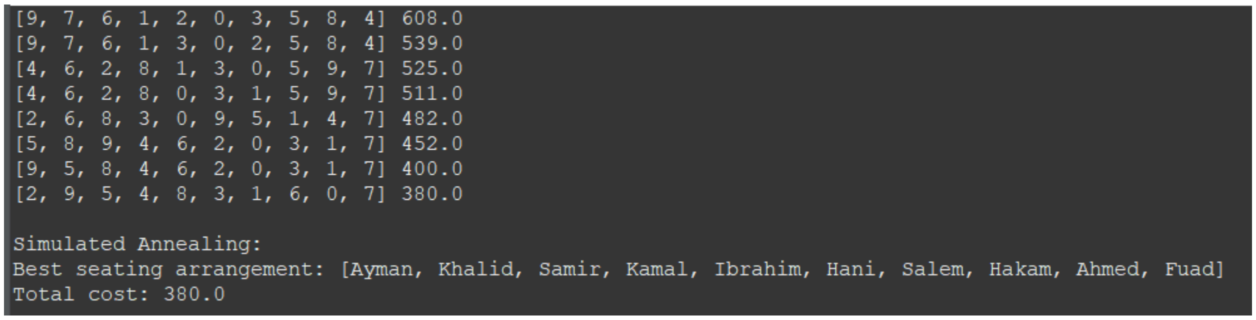
```
double E = currentCost - tempCost;
if(E > 0) {
    currentCost = tempCost;
    currentArrang = new ArrayList<>(tempArrang);
}else {
    if(Math.exp(E/T) < 1) {
        currentCost = tempCost;
        currentArrang = new ArrayList<>(tempArrang);
    }
}
```

Finally, each accepted solution is compared with the previously accepted solution to find the best solution, in each iteration, the temperature is lowered



```
// Update the best solution found
if (currentCost < bestCost) {
    bestArrang = new ArrayList<>(currentArrang);
    bestCost = currentCost;
    System.out.println(bestArrang.toString()+" "+bestCost);
}

T *= coolingRate;
```

Solution Example:



```
[9, 7, 6, 1, 2, 0, 3, 5, 8, 4] 608.0
[9, 7, 6, 1, 3, 0, 2, 5, 8, 4] 539.0
[4, 6, 2, 8, 1, 3, 0, 5, 9, 7] 525.0
[4, 6, 2, 8, 0, 3, 1, 5, 9, 7] 511.0
[2, 6, 8, 3, 0, 9, 5, 1, 4, 7] 482.0
[5, 8, 9, 4, 6, 2, 0, 3, 1, 7] 452.0
[9, 5, 8, 4, 6, 2, 0, 3, 1, 7] 400.0
[2, 9, 5, 4, 8, 3, 1, 6, 0, 7] 380.0

Simulated Annealing:
Best seating arrangement: [Ayman, Khalid, Samir, Kamal, Ibrahim, Hani, Salem, Hakam, Ahmed, Fuad]
Total cost: 380.0
```

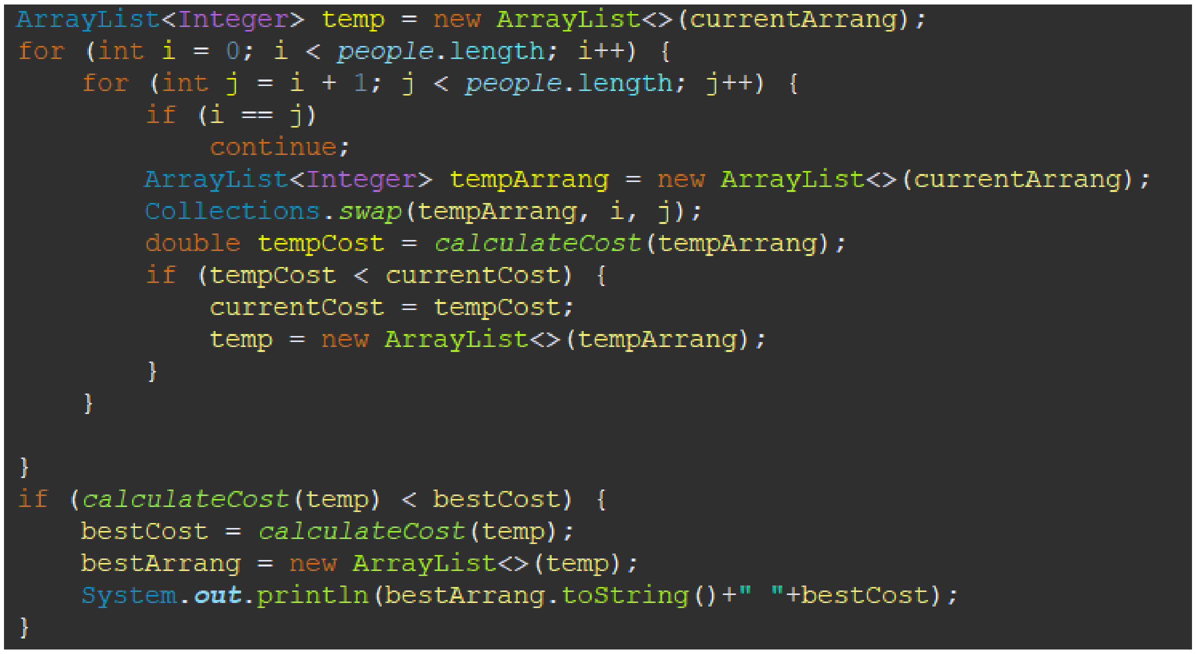
Hill Climbing:

To solve our problem using the hill climbing idea, we needed the hillClimbing(int restart) method, which selects the best seating arrangement, taking one parameter representing the number of times the algorithm is re-executed and then selecting the arrangement with the lowest cost,

Algorithm Solution Restrictions:

1. Initial solution: Start with a random arrangement.
2. Neighbors: Generate neighbors by swapping pairs of people.
3. Restarts: Perform multiple random restarts to explore different parts of the solution space.

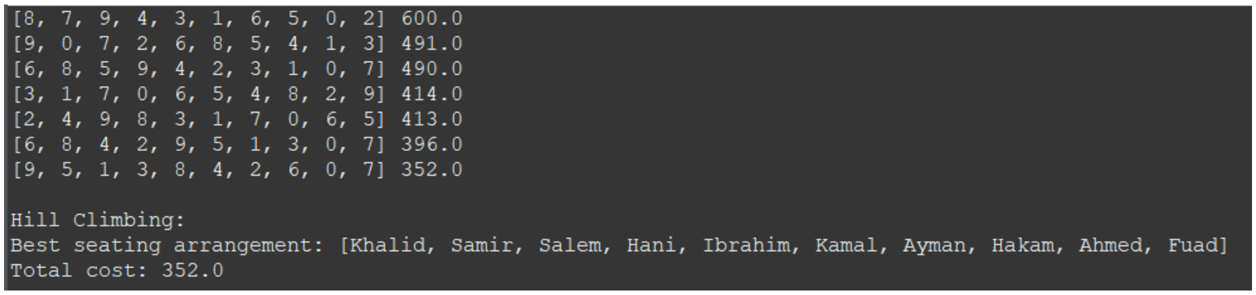
First I created a random seating arrangement and calculated its cost Then I moved to the stage of creating neighbours, where I swapped each person with other people to create different arrangements and explore the optimal solution, and when an arrangement with a lower cost is found, it is adopted as the optimal solution for this moment until a better solution is explored,



```java
ArrayList<Integer> temp = new ArrayList<>(currentArrang);
for (int i = 0; i < people.length; i++) {
    for (int j = i + 1; j < people.length; j++) {
        if (i == j)
            continue;
        ArrayList<Integer> tempArrang = new ArrayList<>(currentArrang);
        Collections.swap(tempArrang, i, j);
        double tempCost = calculateCost(tempArrang);
        if (tempCost < currentCost) {
            currentCost = tempCost;
            temp = new ArrayList<>(tempArrang);
        }
    }

}
if (calculateCost(temp) < bestCost) {
    bestCost = calculateCost(temp);
    bestArrang = new ArrayList<>(temp);
    System.out.println(bestArrang.toString()+" "+bestCost);
}
```

The execution of this algorithm is repeated as many times as the number of restart(100) to reach its goal, each time starting from a different random initial solution. By doing so, the algorithm explores different parts of the solution space. This increases the likelihood of finding a global optimum or at least a better solution than what might be found by a single run of the algorithm.

An example of exploring the optimal solution using hill climbing:

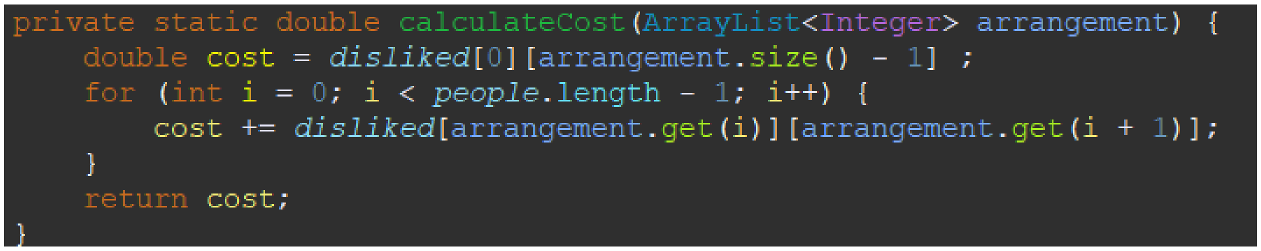


```
[8, 7, 9, 4, 3, 1, 6, 5, 0, 2] 600.0
[9, 0, 7, 2, 6, 8, 5, 4, 1, 3] 491.0
[6, 8, 5, 9, 4, 2, 3, 1, 0, 7] 490.0
[3, 1, 7, 0, 6, 5, 4, 8, 2, 9] 414.0
[2, 4, 9, 8, 3, 1, 7, 0, 6, 5] 413.0
[6, 8, 4, 2, 9, 5, 1, 3, 0, 7] 396.0
[9, 5, 1, 3, 8, 4, 2, 6, 0, 7] 352.0

Hill Climbing:
Best seating arrangement: [Khalid, Samir, Salem, Hani, Ibrahim, Kamal, Ayman, Hakam, Ahmed, Fuad]
Total cost: 352.0
```

Calculate Cost Method:

I implemented a calculateCost(arrangement) method that calculates the total cost of a given seating arrangement.

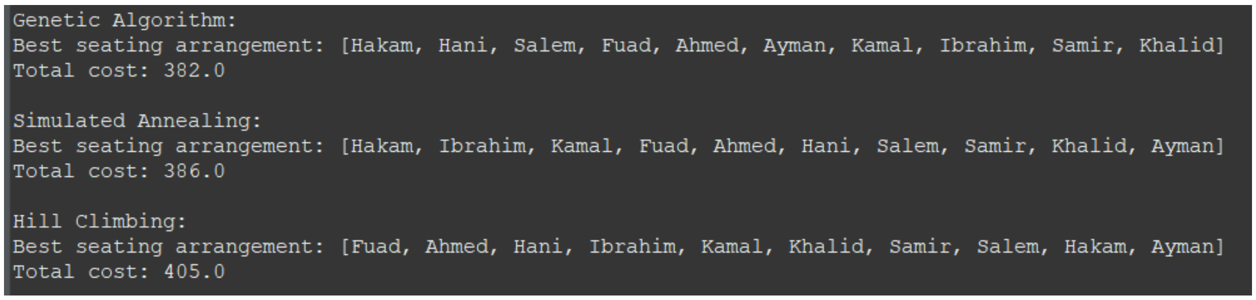
- The total cost calculation considers both directions between neighbouring and seated pairs in a circle to avoid asymmetry



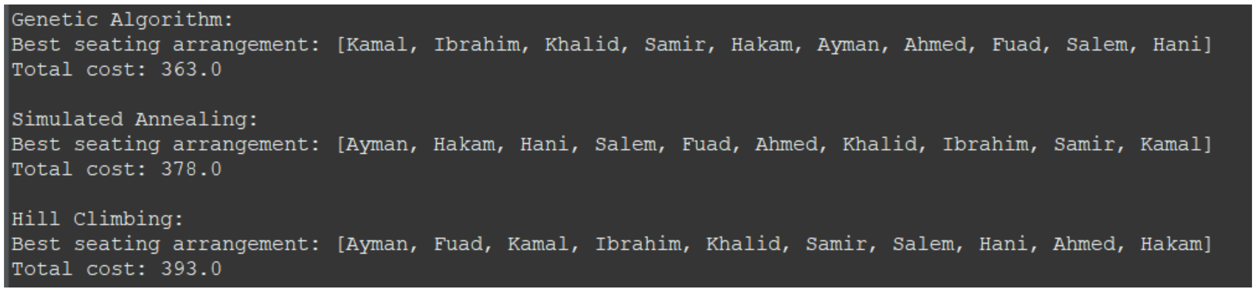
```java
private static double calculateCost(ArrayList<Integer> arrangement) {
    double cost = disliked[0][arrangement.size() - 1] ;
    for (int i = 0; i < people.length - 1; i++) {
        cost += disliked[arrangement.get(i)][arrangement.get(i + 1)];
    }
    return cost;
}
```

Summary

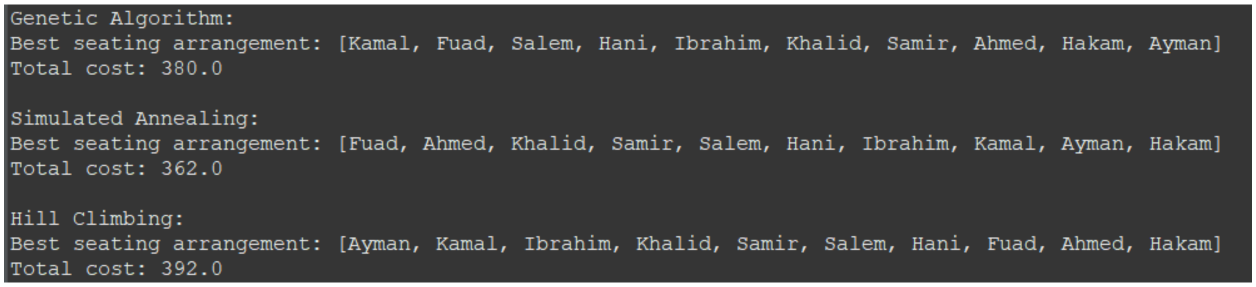
Executing the three algorithms several times shows the following results:



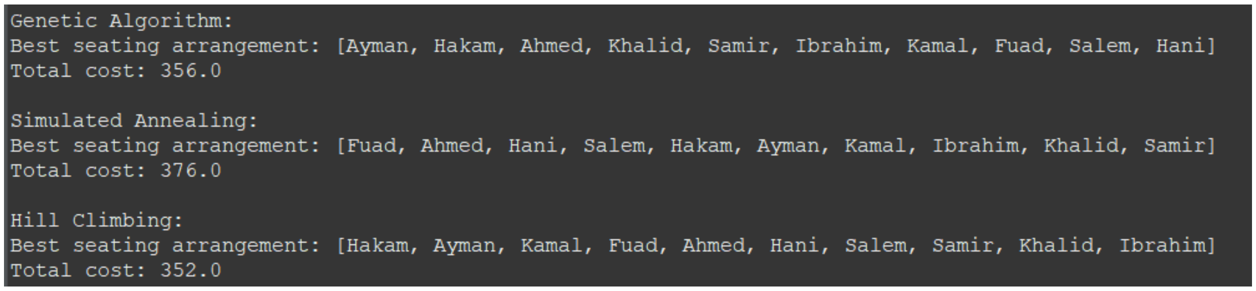
```
Genetic Algorithm:
Best seating arrangement: [Hakam, Hani, Salem, Fuad, Ahmed, Ayman, Kamal, Ibrahim, Samir, Khalid]
Total cost: 382.0

Simulated Annealing:
Best seating arrangement: [Hakam, Ibrahim, Kamal, Fuad, Ahmed, Hani, Salem, Samir, Khalid, Ayman]
Total cost: 386.0

Hill Climbing:
Best seating arrangement: [Fuad, Ahmed, Hani, Ibrahim, Kamal, Khalid, Samir, Salem, Hakam, Ayman]
Total cost: 405.0
```



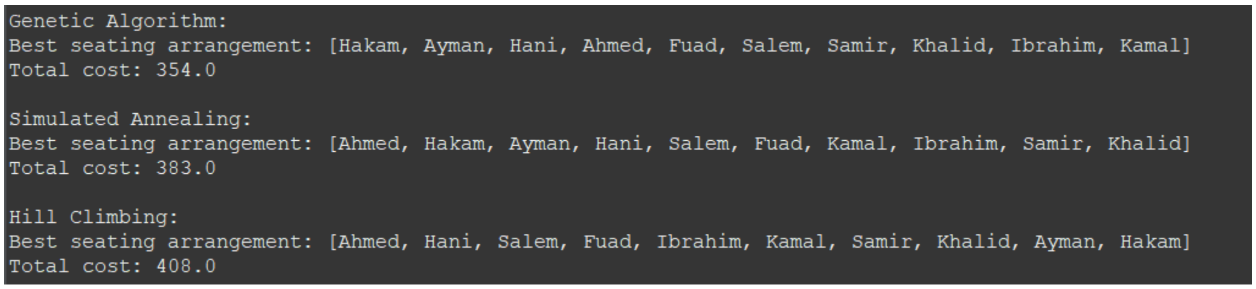
```
Genetic Algorithm:
Best seating arrangement: [Kamal, Ibrahim, Khalid, Samir, Hakam, Ayman, Ahmed, Fuad, Salem, Hani]
Total cost: 363.0

Simulated Annealing:
Best seating arrangement: [Ayman, Hakam, Hani, Salem, Fuad, Ahmed, Khalid, Ibrahim, Samir, Kamal]
Total cost: 378.0

Hill Climbing:
Best seating arrangement: [Ayman, Fuad, Kamal, Ibrahim, Khalid, Samir, Salem, Hani, Ahmed, Hakam]
Total cost: 393.0
```



```
Genetic Algorithm:
Best seating arrangement: [Kamal, Fuad, Salem, Hani, Ibrahim, Khalid, Samir, Ahmed, Hakam, Ayman]
Total cost: 380.0

Simulated Annealing:
Best seating arrangement: [Fuad, Ahmed, Khalid, Samir, Salem, Hani, Ibrahim, Kamal, Ayman, Hakam]
Total cost: 362.0

Hill Climbing:
Best seating arrangement: [Ayman, Kamal, Ibrahim, Khalid, Samir, Salem, Hani, Fuad, Ahmed, Hakam]
Total cost: 392.0
```



```
Genetic Algorithm:
Best seating arrangement: [Ayman, Hakam, Ahmed, Khalid, Samir, Ibrahim, Kamal, Fuad, Salem, Hani]
Total cost: 356.0

Simulated Annealing:
Best seating arrangement: [Fuad, Ahmed, Hani, Salem, Hakam, Ayman, Kamal, Ibrahim, Khalid, Samir]
Total cost: 376.0

Hill Climbing:
Best seating arrangement: [Hakam, Ayman, Kamal, Fuad, Ahmed, Hani, Salem, Samir, Khalid, Ibrahim]
Total cost: 352.0
```



```
Genetic Algorithm:
Best seating arrangement: [Hakam, Ayman, Hani, Ahmed, Fuad, Salem, Samir, Khalid, Ibrahim, Kamal]
Total cost: 354.0

Simulated Annealing:
Best seating arrangement: [Ahmed, Hakam, Ayman, Hani, Salem, Fuad, Kamal, Ibrahim, Samir, Khalid]
Total cost: 383.0

Hill Climbing:
Best seating arrangement: [Ahmed, Hani, Salem, Fuad, Ibrahim, Kamal, Samir, Khalid, Ayman, Hakam]
Total cost: 408.0
```

I conclude that the cost values are close but there are some differences between them due to the randomness of the algorithm,

Comparing the results of all the algorithms with each other to choose the best algorithm to find the optimal solution for the seating arrangement, I face difficulty in this, because each of them starts with a different random seating arrangement from the other, and the final result appears based on the initial solution from which it started,

Genetic algorithm begins by initializing a random arrangement and then generates different Generations randomly. It selects the best solution based on a certain fitness ratio to initiate the crossover process, swapping chromosome pieces to create new generations.

Simulated annealing also starts from a random arrangement and operates at a high temperature initially. It aims to discover the optimal solution by iteratively refining its approach. In each iteration, the temperature is gradually lowered by a fixed percentage.

Hill climbing, starting from an initial random arrangement, explores neighboring solutions in a random manner. Its goal is to find the best arrangement with the lowest cost.

<u>To choose the best algorithm:</u>

Each algorithm is better than the others at some point depending on the different results I get, because of the randomness,

But if we want to choose the best algorithm to be better than the others in most cases, it can be the genetic algorithm, because when the solution is executed more than once, it often gives the optimal solution due to its principle of operation that was explained earlier.

References:

1) Artificial Intelligence A Modern Approach Third Edition.
[https://people.engr.tamu.edu/guni/csce421/files/AI_Russell_Norvig.pdf]
2) Geeks For Geeks
    - [https://www.geeksforgeeks.org/genetic-algorithms/]
3) Hill Climbing
    - [https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/?ref=header_search]