Optimizing Round Table Seating Arrangement

A Study in Artificial Intelligence

Prepared by: Hanadi Asfour **Date:** 6/6/2024

Problem:

The objective is to try to find the optimal seating arrangement for a group of people around a circular table by minimizing the conflict between the pairs. Three different algorithms were used to do so: Genetic Algorithm, Simulated Annealing, and Hill Climbing.

Theory of Search Algorithms:

1. Genetic Algorithm (GA): Uses techniques like selection, crossover, and mutation to evolve solutions towards the optimal arrangement over multiple generations.

2. Simulated Annealing (SA): Probabilistic technique that aims to avoid local minima by allowing uphill moves depending on a probability controlled by the temperature parameter given that gradually decreases.

3. Hill Climbing (HC): A local search algorithm that iteratively improves the solution by exploring neighboring states. It seeks to find the local optimum by always moving towards a better neighbor (less cost).

Solution Approach:

An array of double numbers was populated by the dislike values from the first assignment matrix. The three searches were implemented as the following:

- **<u>Genetic Algorithm</u>:**

A population of random seating arrangements was generated. In each generation, the fittest 60% of the individuals (with the least conflict) are selected to form the next generation. New arrangements (children) are made by combining parts of two parent arrangements at a randomly chosen crossover point. The children go through a mutation depending on mutation rate, which swaps two random positions in the parent arrangement. This process continues for 1000 generations. The final population is then evaluated, and the arrangement with the highest fitness (least conflict) is chosen as the best solution.

- Population size: 100
- Number of generations: 1000
- Mutation rate: 0.1
- Crossover point: randomly generated every pairing.

CODE:

```java
public class GeneticAlgorithm {
	private Random rand = new Random();// for selecting random indexes
	private DislikeTable matrix = Main.matrix;// holds utility functions and data

	public int[] geneticAlgorithm(int populationSize, int numGenerations, double mutationRate) {

		// initializing the population with random arrangements
		int[][] population = initializePopulation(populationSize);

		// keep looping until the specifies number of generation were generated
		for (int generation = 0; generation < numGenerations; generation++) {

			// select the top 60% elite arrangements with the least conflicts
			int[][] selectedElite = selection(population);

			// container for upcoming generations of arrangements
			int[][] newGen = new int[populationSize][10];

			// loop the population to generate the net generation
			for (int i = 0; i < populationSize; i += 2) {

				// selecting a random two parents from the top 60% of the population
				int[] parent1 = selectedElite[rand.nextInt(selectedElite.length)];
				int[] parent2 = selectedElite[rand.nextInt(selectedElite.length)];

				// index point for crossover [1,2,3...(size-1)]
				int point = rand.nextInt(parent1.length - 2) + 1;

				// generating the children resulted from the crossover of the parents
				int[] child1 = crossover(parent1, parent2, point);
				int[] child2 = crossover(parent2, parent1, point);

				// applying mutation to children
				mutate(child1, mutationRate);
				mutate(child2, mutationRate);

				// adding children to the new generation container
				newGen[i] = child1;
				newGen[i + 1] = child2;
			}
```

```java
			population = newGen;// setting the current population to the new generation
		}

		// to hold the resulted best arrangement (least conflict)
		int[] bestArrangement = null;
		double bestCost = -1;// least cost (impossible)

		// loop the end population to find the best discovered arrangement (best cost)
		for (int[] arrangement : population) {
			double fitness = matrix.calculateFitness(arrangement);

			if (fitness > bestCost) {// found a better cost
				bestArrangement = arrangement;
				bestCost = fitness;
			}
		}

		return bestArrangement;// return best found arrangement
	}


	// this method fills the population with random seating arrangements
	public int[][] initializePopulation(int size) {
		int[][] population = new int[size][10];
		for (int i = 0; i < size; i++)
			population[i] = matrix.generateRandomArrangement();

		return population;
	}

	// selects the top 60% elite of the population given, which are the arrangements
	// with the best cost (least conflict)
	public int[][] selection(int[][] population) {

		// finding the cost of each arrangement in the given population
		double[] fitness = new double[population.length];
		for (int i = 0; i < population.length; i++)
			fitness[i] = matrix.calculateFitness(population[i]);

		int size = (int) (population.length * 0.6);// the number of selected arrangements
		int[][] elites = new int[size][];// to hold the best of the best

		// selecting the top 60% of the population
		for (int i = 0; i < size; i++) {// just the selected size
			int max = 0;// hold index

			for (int j = 1; j < population.length; j++) // comparing with the whole pop
				if (fitness[j] > fitness[max])
					max = j;


			// saving the individual with the highest fitness function cost
			elites[i] = population[max];

			// preventing this arrangement to be selected again next round
			fitness[max] = -1;// setting fitness to a very low number
		}

		return elites; // Return the selected population
	}
```

```java
    // crossover the two parent arrangements to produce the new one
    public int[] crossover(int[] parent1, int[] parent2, int point) {
        int size = parent1.length;// size of arrangement
        int[] child = new int[size];// produced child
        boolean[] seated = new boolean[size];// keep track of seen people in arrangement

        // copy first part from parent1 to child
        for (int i = 0; i <= point; i++) {
            child[i] = parent1[i];
            seated[child[i]] = true;// mark as taken
        }

        // fill remaining part in the same order it appeared in the parent
        int p2Index = 0;
        for (int i = point + 1; i < size; i++) {
            // until an unassigned person appears
            while (seated[parent2[p2Index]])
                p2Index++;

            child[i] = parent2[p2Index];// add to child arrangement
            seated[child[i]] = true;// set as seated
        }

        return child;// return resulted arrangement
    }



    // applies a mutation by switching two individuals from the arrangement given
    public void mutate(int[] arrangement, double rate) {

        if (rand.nextDouble() <= rate) {// only mutate within the rate
            int a, b;// indexes to mutate(switch)

            do {
                a = rand.nextInt(arrangement.length);
                b = rand.nextInt(arrangement.length);
            } while (a == b);// making sure indexex never equal

            int temp = arrangement[a];
            arrangement[a] = arrangement[b];
            arrangement[b] = temp;
        }
    }
}
```

- **<u>Simulated Annealing</u>:**

The algorithm starts with an initial random arrangement, assuming it as the best arrangement, and calculates its cost. The temperature is gradually decreased using a cooling rate, it affects the acceptance of worse arrangements. Every iteration, a neighboring arrangement is created by randomly swapping two people in the current arrangement. The algorithm moves to this new state if it is better (lower cost) or depending on the temperature, it accepts it as a worse state. This helps in avoiding local minima. This process continues for many iterations. The best-found state is updated and considered the solution.

- Initial temperature: 1000
- Cooling rate: 0.99
- Number of iterations: 10000

CODE:

```java
public class SimulatedAnnealing {

	private Random rand = new Random();// for selecting random indexes
	private DislikeTable matrix = Main.matrix;// holds utility functions and data

	public int[] simulatedAnnealing(double initialTemperature, double coolingRate, int numIteration) {
		int[] current = matrix.generateRandomArrangement();// initial state
		double currentCost = matrix.calculateCost(current);// cost of the initial arrangement

		int[] bestArr = current.clone();// assuming the best is the initial state
		double bestCost = currentCost;

		double temp = initialTemperature;// will start to decrease by the cooling rate

		for (int i = 0; i < numIterations; i++) {// repeat by number of iterations
			int[] neighbor = getNeighbor(current);// state to move to next
			double neighborCost = matrix.calculateCost(neighbor);

			// accept the neighboring state only if the next state is :
			// 1) better than the current
			// 2) worse but the random number is less than the equation (delta E / temp)
			if (neighborCost < currentCost ||
				 rand.nextDouble() < Math.exp((currentCost - neighborCost) / temp)) {

				// assigning the current as the neighboring state
				current = neighbor;
				currentCost = neighborCost;


				// check if it is the ultimate best arrangement found
				if (currentCost < bestCost){
					bestArr = current;
					bestCost = currentCost;
				}
			}

			temp *= coolingRate;// decrease worse case acceptance rate
		}

		return bestArr;// return best found arrangement
	}
```

```
	// returns a new state by switching two individuals in the given arrangement
	public int[] getNeighbor(int[] arrangement) {
		int[] neighbor = arrangement.clone();
		int a, b;// indexes to switch

		do {
			a = rand.nextInt(arrangement.length);
			b = rand.nextInt(arrangement.length);
		} while (a == b);// making sure indexes never equal

		// swap individuals
		int temp = neighbor[a];
		neighbor[a] = neighbor[b];
		neighbor[b] = temp;
		return neighbor;
	}

}
```

- **<u>Hill Climbing</u>:**

The algorithm also starts with a random arrangement and keeps generating neighboring solutions by swapping pairs of people just like the Simulated annealing. It keeps track of the arrangement with the lowest conflict found. If a neighboring arrangement has a lower cost than the current one, it becomes the new current arrangement. This process continues until no neighboring arrangement has a lower cost, which means we have got to a local minimum. To avoid getting stuck in local minima, we perform a number of random restarts, exploring different parts of the solution space each time. The best arrangement found across all restarts is returned as the solution.

- Number of random restarts: 100

CODE:

```
public class HillClimbing {

	private DislikeTable matrix = Main.matrix;// holds utility functions and data

	public int[] hillClimbing(int numRestarts) {
		int[] bestArr = null;// to hold the best encountered arrangement
		double bestCost = Double.MAX_VALUE;

		// loop until all of the restarts were completed
		for (int restart = 0; restart < numRestarts; restart++) {
			int[] currentArr = matrix.generateRandomArrangement();// initial state
			double currentCost = matrix.calculateCost(currentArr);// initial cost

			// loop until lest costly neighbor is not less than current
			while (true) {
				// generate all neighboring arrangements to the current
				int[][] neighbors = getNeighbors(currentArr);
				int[] next = currentArr;// potential next arrangement state
				double nextCost = currentCost;

				for (int[] neighbor : neighbors) {// finding neighbor with the least cost
					double neighborCost = matrix.calculateCost(neighbor);
					if (neighborCost < nextCost) {
						next = neighbor;
						nextCost = neighborCost;
					}
				}
				// if the least cost found neighbor is better than the currently held
				// arrangement then assign it as the current
				if (nextCost < currentCost) {
					currentArr = next;
					currentCost = nextCost;
				} else // cost not decreased from the current state
					break;
			}

			// tracking the best arrangement detected in every loop restart
			if (currentCost < bestCost) {
				bestArr = currentArr;
				bestCost = currentCost;
			}
		}
```

```java
        return bestArr;// returning the best arrangement with the least costs
}

// generate all possible neighboring arrangements(swapping 2 people)from the given current state
public int[][] getNeighbors(int[] arrangement) {

        // The number of combinations of n elements taken 2 at a time
        int combinationsSize = arrangement.length * (arrangement.length - 1) / 2;
        int[][] neighbors = new int[combinationsSize][];// initializing to store neighbors
        int index = 0;// track neighbor index

        // generating all of combinations when swapping 2 individuals
        for (int i = 0; i < arrangement.length; i++) {
                for (int j = i + 1; j < arrangement.length; j++) {
                        int[] neighbor = arrangement.clone();// copy the current arrangement

                        // swap individuals at i and j
                        int temp = neighbor[i];
                        neighbor[i] = neighbor[j];
                        neighbor[j] = temp;
                        neighbors[index++] = neighbor; // save this neighbor
                }}
        return neighbors;}
```

- **Other Relevant Functions:**

1. Calculate cost function, which calculates the cost of the seating arrangement

CODE:

```
//calculates the cost of the seating arrangement given
//the higher the cost, the higher the conflict, and the worse the arrangement is
//used for hill climbing and simulated annealing
public double calculateCost(int[] arrangement) {
        double totalCost = 0;
        int numPeople = arrangement.length;
        for (int i = 0; i < numPeople; i++) {
                int personA = arrangement[i];
                int personB = arrangement[(i + 1) % numPeople];
                totalCost += dislikeMatrix[personA][personB] + dislikeMatrix[personB][personA];
        }
        return totalCost;
}
```

2. Calculate fitness, which is the inverse of the cost function. Calculates the amount of "likeness" between pairs.

CODE:

```
//method that calculated the fitness of the arrangement instead of the cost
//the higher the fitness, the smaller the conflict, the better the arrangement
//used for the genetic algorithm
public double calculateFitness(int[] arrangement) {
        double sum = 0;
        int numPeople = getNumPeople();
        for (int i = 0; i < numPeople; i++) {
                int personA = arrangement[i];
                int personB = arrangement[(i + 1) % numPeople];
                sum += (1- dislikeMatrix[personA][personB]) + (1-
                dislikeMatrix[personB][personA]);
        }
        return sum;
}
```

3. Generate random seating arrangement, which shuffles a matrix of indexes representing the people on a table.

CODE:

```
//generates a random seating arrangement for initial states
public int[] generateRandomArrangement() {
        int[] arr = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };//contains all indexes
        // loop the sequence arr to shuffle it
        for (int i = arr.length - 1; i > 0; i--) {
                int index = rand.nextInt(i + 1);//random index ahead
                //swap current with the random index
                int temp = arr[index];
                arr[index] = arr[i];
                arr[i] = temp;
        }
        return arr;//return shuffled arrangement
    }
```

- **Main Part:**

```
public static void main(String[] args) {
        // TODO Auto-generated method stub

        // initializing optimizing algorithms classes
        GeneticAlgorithm genetic = new GeneticAlgorithm();
        SimulatedAnnealing simulated = new SimulatedAnnealing();
        HillClimbing hill = new HillClimbing();


        //////////////////
        // * <Genetic> *//
        //////////////////
        int[] bestSeatingG = genetic.geneticAlgorithm(100, 1000, 0.1);
        double bestCostG = matrix.calculateCost(bestSeatingG);

        //print results
        System.out.println("-----------------------------------------------------");

        System.out.println("Genetic Algorithm Best Seating: \n" +
        Arrays.toString(matrix.getArrangmentOfNames(bestSeatingG)));
        System.out.println("Total cost: " + bestCostG + "\n\n");
        System.out.println("-----------------------------------------------------");



        //////////////////////////////
        // * <Simulated Annealing> *//
        //////////////////////////////
        int[] bestSeatingS = simulated.simulatedAnnealing(1000, 0.99, 10000);
        double bestCostS = matrix.calculateCost(bestSeatingS);

        //print results
        System.out.println("Simulated Annealing Best Seating: \n" +
        Arrays.toString(matrix.getArrangmentOfNames(bestSeatingS)));
        System.out.println("Total cost: " + bestCostS + "\n\n");
        System.out.println("-----------------------------------------------------");


        ////////////////////////
        // * <Hill Climbing> *//
        ////////////////////////
        int[] bestSeatingH = hill.hillClimbing(100);
        double bestCostH = matrix.calculateCost(bestSeatingH);

        //print results
        System.out.println("Hill Climbing Best Seating: \n" +
        Arrays.toString(matrix.getArrangmentOfNames(bestSeatingH)));
        System.out.println("Total cost: " + bestCostH + "\n\n");
        System.out.println("-----------------------------------------------------");



        }
```

Console Results:

RUN #1 :

```
-----------------------------------------------------
Genetic Algorithm Best Seating:
[Ayman, Hakam, Samir, Salem, Hani, Ahmad, Fuad, Kamal, Ibrahim, Khalid]
Total cost: 7.780000000000001


-----------------------------------------------------
Simulated Annealing Best Seating:
[Salem, Fuad, Ahmad, Hakam, Ayman, Khalid, Samir, Kamal, Ibrahim, Hani]
Total cost: 6.999999999999999


-----------------------------------------------------
Hill Climbing Best Seating:
[Salem, Fuad, Ahmad, Hakam, Ayman, Khalid, Samir, Kamal, Ibrahim, Hani]
Total cost: 6.999999999999999
```

RUN #2 :

```
-----------------------------------------------------
Genetic Algorithm Best Seating:
[Khalid, Ayman, Hakam, Ahmad, Fuad, Salem, Hani, Ibrahim, Kamal, Samir]
Total cost: 7.0


-----------------------------------------------------
Simulated Annealing Best Seating:
[Hani, Salem, Samir, Ibrahim, Khalid, Ahmad, Fuad, Kamal, Ayman, Hakam]
Total cost: 8.22


-----------------------------------------------------
Hill Climbing Best Seating:
[Fuad, Ahmad, Hakam, Ayman, Khalid, Samir, Kamal, Ibrahim, Hani, Salem]
Total cost: 6.999999999999999
```

RUN #3 :

```
-----------------------------------------------------
Genetic Algorithm Best Seating:
[Hani, Ahmad, Fuad, Kamal, Ibrahim, Samir, Khalid, Ayman, Hakam, Salem]
Total cost: 7.579999999999999


-----------------------------------------------------
Simulated Annealing Best Seating:
[Hakam, Ayman, Khalid, Ibrahim, Kamal, Fuad, Ahmad, Hani, Salem, Samir]
Total cost: 7.779999999999999


-----------------------------------------------------
Hill Climbing Best Seating:
[Fuad, Ahmad, Hakam, Ayman, Khalid, Samir, Kamal, Ibrahim, Hani, Salem]
Total cost: 6.999999999999999
```

Optimal Solution:

[Salem, Fuad, Ahmad, Hakam, Ayman, Khalid, Samir, Kamal, Ibrahim, Hani]

Conflict Cost:
6.999999999999999 ≅ 7.0

Algorithm:
Hill climbing in run #1 (genetic algorithm in run #2 and simulated annealing in run #1 too)

Analysis:

The overall average results were as the following:

- Genetic Algorithm: Total cost average across all runs = 7.453
- Simulated Annealing: Total cost average across all runs = 7.666
- Hill Climbing: Total cost average across all runs = 6.999999999999999 ≅ 7.0

The Hill Climbing algorithm consistently gave the optimal solution with the lowest cost of 7.0 in every run. This is because hill climbing goes through the solution space and finds the local optima, then with the 100 random restarts, it can explore different parts of the solution space and finds the ultimate minimum between the arrangements.

The genetic algorithm and simulated annealing solutions seem to alternate between different arrangements in each run. This is due to the stochastic nature of their algorithms, where randomness plays a role in the exploration of the solution space. genetic algorithm's crossover and mutation operations, as well as simulated annealing's acceptance of worse solutions with certain probabilities.

I don't think any of the algorithms can be considered the "worst" as they all converge to solutions with relatively low total costs and can produce the optimal solution in some of their runs. However, it was notable that the genetic and simulated annealing algorithms tend to produce non-optimal or higher costs than hill climbing algorithms in most of their runs.

The reason for this in the genetic algorithm is that some produced generations may have lost the "good" features from their parents when performing the crossings and mutations in a probabilistic manner. Which produced less optimal outcomes. Similarly, simulated annealing can struggle to escape local optima sometimes due to its probabilistic acceptance of worse solutions, leading to not so optimal results in some runs.

Simulated annealing on average produced higher results than the genetic algorithm. It accepts worse solutions with a certain probability which leads to exploring a large range of states that includes higher-cost arrangements. Especially at the beginning of the process when the temperature is high, suboptimal solutions are accepted a lot.

Problems Faced:

In the genetic algorithm, it was challenging to decide on the number and location of the crossover points between the parent arrangements. I decided to choose a single random crossover point because it introduced stochasticity into the algorithm even more (but not too much like it would be if multiple random points were selected). This could help in generating diverse children and exploring different solutions, but it also increased the chance of producing suboptimal offspring. The half point crossover was considered, but then it was noted that on average, the solution cost was higher than the one with a random cross over. That’s why the single random point technique was used.