Final Project Report - INFO6205_205

Event Table Organiser (Using Genetic Algorithm)

By

Tejas Prakash Shah (NUID 001449694)

Prajakta Sumbe (NUID 001353145)

GitHub Repository -

https://github.com/shah-tejas/INFO6205_205

Table of Contents

1. Introduction

Genetic Algorithms provide a method for solving constrained and unconstrained optimization problems. The algorithm is inspired by Charles Darwin’s theory of natural selection where the fittest individual are selected for reproduction in order to produce offspring of next generation.

Genetic algorithms can solve problems that are not well suited for standard optimization algorithms. Problems which are appropriate for genetic algorithms includes timetabling and scheduling problems, where there are finite number of “valid” solutions. Genetic Algorithms start with a random solution and evolve to “valid” solutions, which are acceptable based on certain criteria. Thus, we can get unique solutions for a problem solved using this methodology, and all of those could be good solutions.

2. Problem Statement

Event Tables Organiser

In any social event, the guests at the event are seated in a finite number of tables. It is essential to have guests with similar interests accomodated in the same table. There are various factors that would influence the assignment of guests to tables like personal interests, food preferences or if they are related. The seating of each guest would affect his/her experience at the event. Thus, it would be essential to have guests with similar interests seated together to make the event a success.

Using Genetic Algorithm to solve this problem would be a good strategy as we could get various optimal solutions in finite time. The solution for the algorithm would be arrangement of n number of guests at m number of tables, such that like minded or related persons are seated at each table.

For this implementation, we take 3 parameters of a person into perspective to determine if the guests seated at the tables have similar interests:

1. Views - The views a person has about various topics.
2. Relation - To determine if 2 people are related.
3. Food Preference - Food preference of a guest.

These 3 parameters form the “Genotype” of the person.

3. Implementation Details

The implementation of the algorithm to solve this problem involves different components which are explained as follows:

Population:

Population of our algorithm is different arrangements of guests into different tables. Thus, population here is an ArrayList of different Arrangements and consists of the following Entities:

A. **Person**: The Person class signifies a Guest at the party and has the attributes like views, relation and eating Preference. Each person is uniquely identified by PersonID.
B. **Seat**: The Seat class signifies each seat at the party and has a Person reference that would be assigned this seat and also Table ID for the table this seat belongs to. Each Seat is uniquely identifies by SeatID.
C. **Table**: The whole arrangement contains a finite number of Tables, which contains a List of Seats that are assigned to this table.
D. **Arrangement**:

The output (solution) of the algorithm would be an arrangement of tables, with all the guests assigned to the table. The solution would thus be a list of tables where each table has a list of Seats and each Person is assigned this seat.

We implement this structure in the Arrangement class, which contains an ArrayList of Table class. Each Table further has an ArrayList of Seat class and each Seat has a reference of Person object. The Person object has various attributes like Views, Relation and EatingPreference. Each of this attribute is an integer and is used to calculate the "fitness" of the arrangement as a whole.

Fitness Calculation: A method to calculate the overall fitness of the arrangement is defined in this class. Fitness of each table is calculated in the arrangement and we take average of all table's fitness as the fitness of the current arrangement. For each table we compare 2 person's attributes at a time and calculate the fitness on the following basis:

i. If **views** of both the persons are same, we award **+5** to fitness, else deduct the difference in views between both the persons.

ii. If the two persons are **related**, we award **+20** to fitness, else deduct fitness by **20**.

iii. If the two **persons** share the same eating preference, **+5** is awarded else **-5** is deducted from the fitness.

Based on this logic, we can document the implementation of the fitness function in pseudocode as follows:

```
totalFitness = 0
For each table in the arrangement:

     Initialise tableFitness = 0
     seatList = Seat list of current table

     For i=0 to length(seatList)-1:
          Person1 = seat(i).getPerson
          Person2 = seat(i+1).getPerson

          //Views
          If Person1.getViews == Person2.getViews
               tableFitness += 5
          Else
               tableFitness -= (Person1.getViews - Person2.getViews)

          //Relation
          If Person1.getRelation == Person2.getRelation
               tableFitness += 20
          Else
               tableFitness -= 20

          //EatingPreference
          If Person1.getEatingPreference == Person2.getEatingPreference
               tableFitness += 5
          Else
               tableFitness -= 5

     End for seatList

     totalFitness += tableFitness

End for tableList

Return totalFitness / arrangementSize
```

Arrangement Generation:

When the Arrangement is initially created, we need to initialize all the attributes of the arrangement. We do this by first generating a list of all the Seats currently available, that is the number of guests present. To feed some randomness, we **shuffle** this list of Seats and then assign each Seat to a table.

Algorithm:

The algorithm contains of different functions that initially reset our population and gradually evolve it to a feasible solution. Feasibility of the solution is judged by the fittest individual present in the population. The different functions are:

1. **Evolution:**
   During each evolution, we first find the fittest individual in the population and save it separately from other individuals of the population. For the rest of the individuals of the population we perform crossovers and mutation to improve the fitness of the population.
2. **Crossover:**
   For each crossover, 2 parents are selected using the tournament selection logic and the product of this crossover is a new individual which is added to the population.Crossover is performed by adding certain traits of the first parent to the child and then rest traits of the second parent again to the child. In the current problem context, each parent is an arrangement of seats over tables. During crossover, certain seats from first parent are added to the child arrangement and rest of the seats are added from the second parent.
3. **Parent Selection:**
   We have utilized tournament selection logic to select a parent feasible for crossover. This is done by creating a new empty Population of size decided by the parameter tournamentSize and randomly adding individuals from the original population to this population. We then select the fittest individual from this new population as the parent for crossover.
4. **Mutation:**
   We randomly mutate individuals during the evolution process. This is simply done by swapping 2 persons from 2 randomly selected tables.

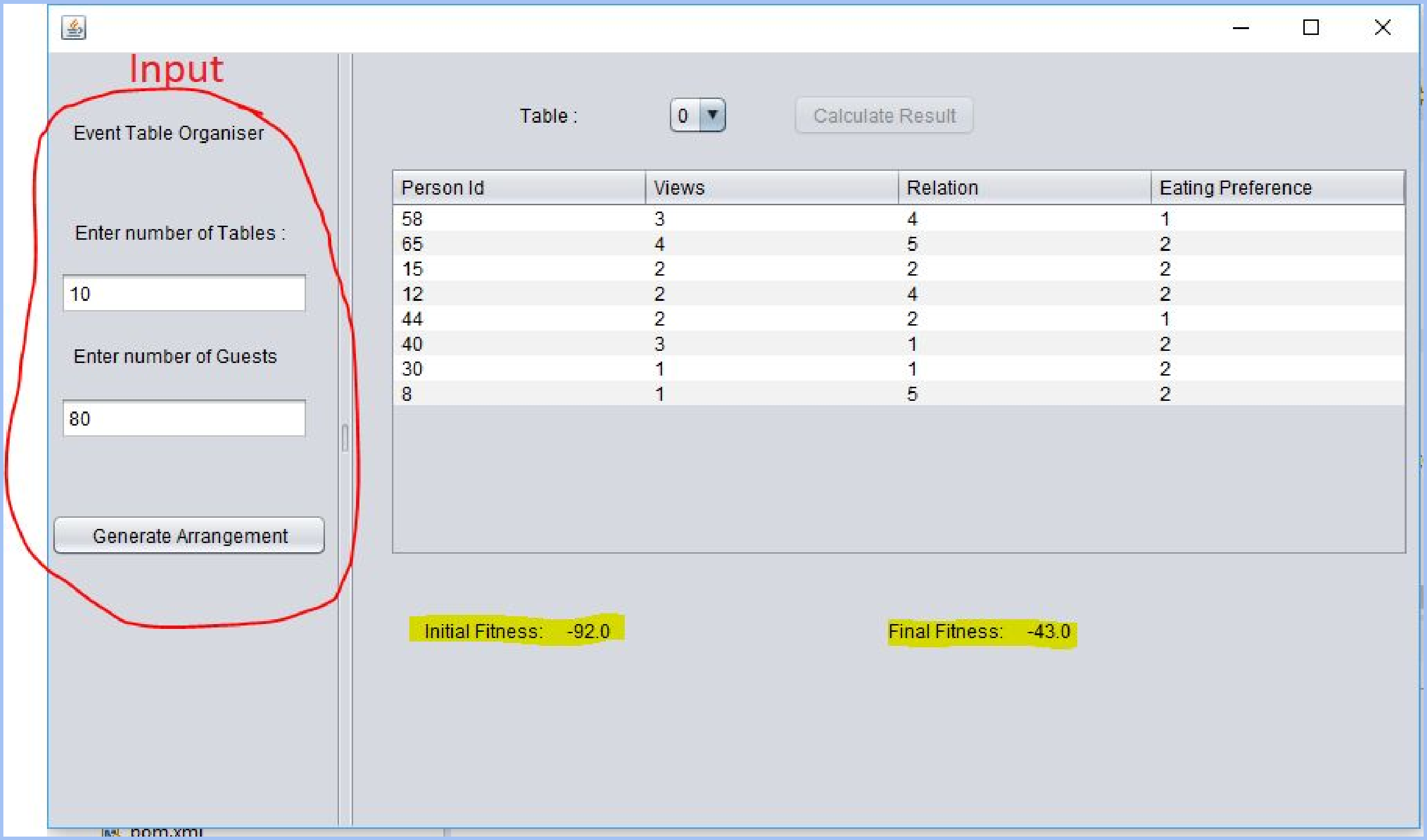
User Interface:

A simple User Interface is created for this Algorithm that takes the Number of Tables and Number of Guests as input and generates the table arrangement where each person is assigned a table. We display each table arrangement in a JTable.

Logging:

The log4j library is used for logging in the application, where we capture important information from the algorithm in the log.

4. Output & Screenshots:



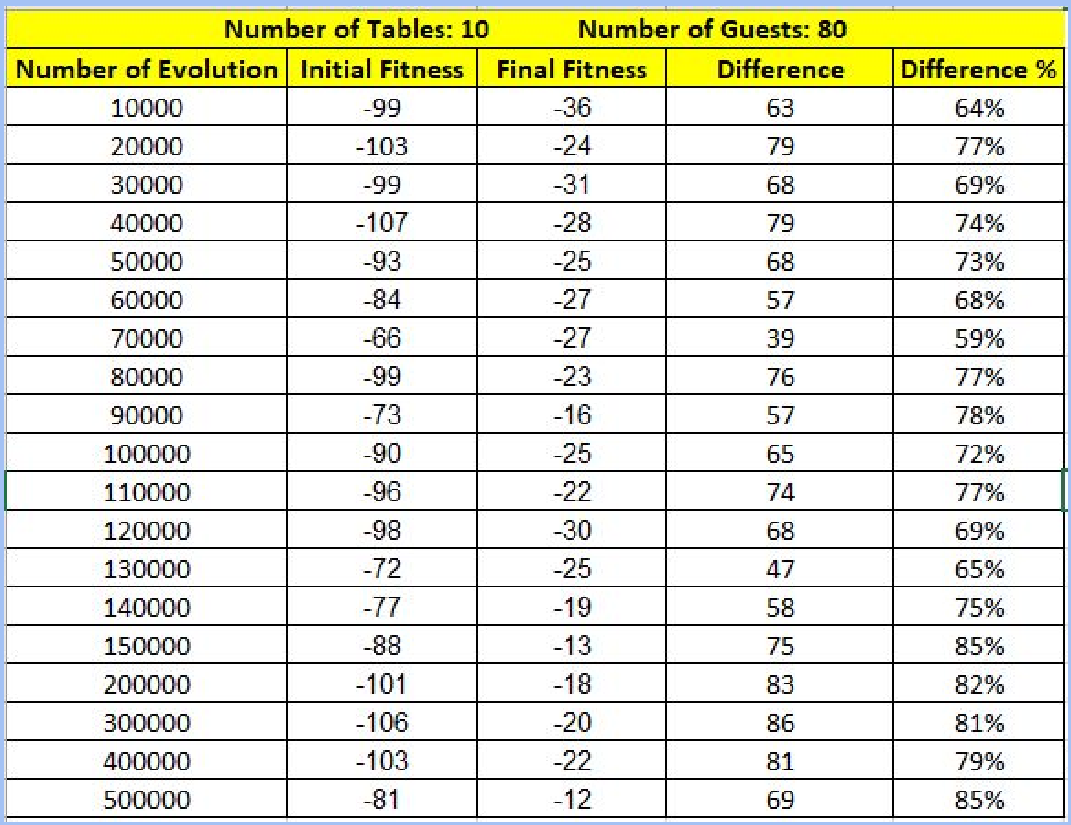
Person Id	Views	Relation	Eating Preference
58	3	4	1
65	4	5	2
15	2	2	2
12	2	4	2
44	2	2	1
40	3	1	2
30	1	1	2
8	1	5	2

Initial Fitness: -92.0

Final Fitness: -43.0

Sample Output when Number of Tables: 10 and Number of Guests: 80.

We toggle the number of evolutions performed over the population and can see the difference in the fitness:

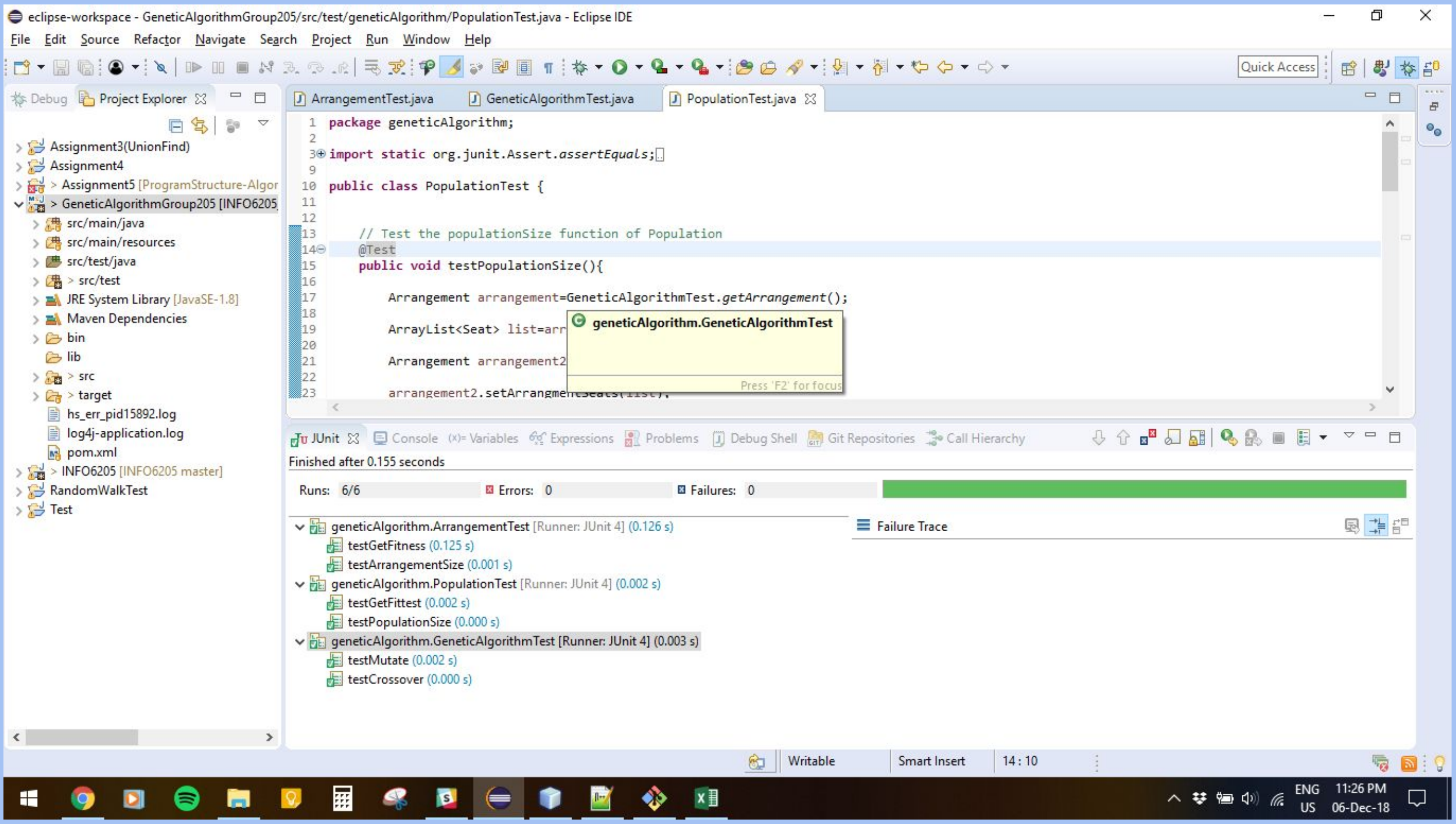


	Number of Tables: 10		Number of Guests: 80	
Number of Evolution	**Initial Fitness**	**Final Fitness**	**Difference**	**Difference %**
10000	-99	-36	63	64%
20000	-103	-24	79	77%
30000	-99	-31	68	69%
40000	-107	-28	79	74%
50000	-93	-25	68	73%
60000	-84	-27	57	68%
70000	-66	-27	39	59%
80000	-99	-23	76	77%
90000	-73	-16	57	78%
100000	-90	-25	65	72%
110000	-96	-22	74	77%
120000	-98	-30	68	69%
130000	-72	-25	47	65%
140000	-77	-19	58	75%
150000	-88	-13	75	85%
200000	-101	-18	83	82%
300000	-106	-20	86	81%
400000	-103	-22	81	79%
500000	-81	-12	69	85%

5. Test Cases:

We perform various Unit Tests to ensure our framework functions behave as intended to. Using JUnit framework, we test all the functions that are algorithm uses.

Output of all the Unit Test are passed, as follows:



eclipse-workspace - GeneticAlgorithmGroup205/src/test/geneticAlgorithm/PopulationTest.java - Eclipse IDE

```java
package geneticAlgorithm;

import static org.junit.Assert.assertEquals;

public class PopulationTest {


    // Test the populationSize function of Population
    @Test
    public void testPopulationSize(){

        Arrangement arrangement=GeneticAlgorithmTest.getArrangement();

        ArrayList<Seat> list=arr

        Arrangement arrangement2

        arrangement2.setArrangmentSeats(list);
```

geneticAlgorithm.GeneticAlgorithmTest

Press 'F2' for focus

```
Finished after 0.155 seconds

Runs: 6/6        Errors: 0        Failures: 0

geneticAlgorithm.ArrangementTest [Runner: JUnit 4] (0.126 s)
    testGetFitness (0.125 s)
    testArrangementSize (0.001 s)
geneticAlgorithm.PopulationTest [Runner: JUnit 4] (0.002 s)
    testGetFittest (0.002 s)
    testPopulationSize (0.000 s)
geneticAlgorithm.GeneticAlgorithmTest [Runner: JUnit 4] (0.003 s)
    testMutate (0.002 s)
    testCrossover (0.000 s)
```

6.Conclusion :

The solution for the above problem is not simple and cannot be found by normal optimization algorithms. On the other hand, based on our above implementation and observation, we can see that Genetic Algorithm gives a good, feasible solution in linear time.

There are various good solutions that solve the problem. We determine if the solution is good by judging the “fitness” of our solution. In our experiment, we do this by beginning with an initial population and gradually evolve it to find an optimum solution.