Genetic Algorithm: The Event Scheduling Problem

Team Number: 313

Team Members: Raghavi Kirouchenadradjou - 001826638, Neha Pednekar – 001883033

**PROBLEM STATEMENT**

Generation of Event Schedules can be a time consuming and a challenging task. We have used Genetic Algorithm to find a solution to the Event Scheduling problem. Considering a set of data, where we have a list of different organizations, events taking place which are conducted by these organizations, different event themes or titles under every event(For example, dancing and singing under Arts and Culture Event, Scientific Seminars under Scientific Events, etc.), organizers conducting these events, number of attendees showing presence for a specific event and available rooms within a venue for these events, an event schedule can be produced with minimal or no conflicts. In Event Scheduling problem, we begin with an initial population and select parents for mating. We will then be applying mutation and crossover operations on the parents to generate new off-springs which in turn will replace the existing individuals. This is a repetitive process until we find an optimal set of solutions within a search space. In simple words, this process will help us find an optimal event schedule.

**GENETIC ALGORITHM DESIGN IMPLEMENTATION FOR EVENT SCHEDULING**

**Genetic code**

In Genetic Algorithm, we have a population which is a subset of all possible encoded solutions to the Event Scheduling problem. Chromosomes represent one such solution to the problem. In the Event Scheduling problem, a chromosome is an event which comprises of various event titles, organizations, organizers, event venues and so on. To begin with the GA, we first need to initialize the population. We are initializing the population with selection of randomized values from the sample data declared in the Data class. A fitness function or evaluation function will then determine how close a given solution is to the optimal solution of the desired problem.

**Gene Expression**

Depending upon the tournament size declared, we will be choosing the most fit chromosome for the process of further mutating and crossing over. In cross over, we simply select the fittest chromosomes and pass them on to the further generations. We select the elite event schedules in the crossover population. Then, we select one fittest event schedule among the schedules of a given tournament size and another event schedule in a similar manner. If the crossover rate is greater than the random value and if the probability is greater than half, we select chromosomes from the first event schedule or else we will select chromosomes from the second event schedule. Also, if the crossover rate is less than the random value, we will select the event schedules from the existing population.

In Mutation, one or more genes are changed within the chromosomes. We initialize the mutating population and set the elite individuals in the mutate population just like we did in the crossover population. We set the other remaining individuals randomly from the initial population of there is a possibility of the mutation rate.

**Fitness Function**

Fitness function is the measure of how good or optimal is a solution. It depends upon the conflicts that arise while generating an event schedule like when two organizers organize events in the same room or the number of attendees is greater than the venue capacity. More the number of conflicts, less fit is the event schedule and vice versa.

We have calculated the fitness function in our code like as follows

public double getFitness() {

if (isFitnessChanged == true) {

fitness = calculateFitness();

isFitnessChanged = false;

}

return fitness;

}

If fitness is changed, we again calculate the fitness using the below given function:

private double calculateFitness() {

numOfConflicts = 0;

events.forEach(x -> {

if (x.getRoom().getSeatCapacity() < x.getTitle().getMaxNumberOfAttendees()) {

numOfConflicts++;

}

events.stream().filter(y -> events.indexOf(y) >= events.indexOf(x)).forEach(y -> {

if (x.getEventTime() == y.getEventTime() && x.getId() != y.getId()) {

if (x.getRoom() == y.getRoom()) {

numOfConflicts++;

}

if (x.getOrganizer() == y.getOrganizer()) {

numOfConflicts++;

}

}

});

});

return 1 / (double) (numOfConflicts + 1);

}

If there are no conflicts between the generated list of event schedules, the fitness will be 1 and the process will be stopped. At this point, we have successfully generated an event schedule in which no event clashes with another event.

**ANALYSIS, OBSERVATIONS AND GRAPHS**

Below given are the readings for five different trials, the generations taken to generate the fittest event schedule, the total time take and the corresponding fitness and conflicts which will be 1 and 0 anyways at the end.

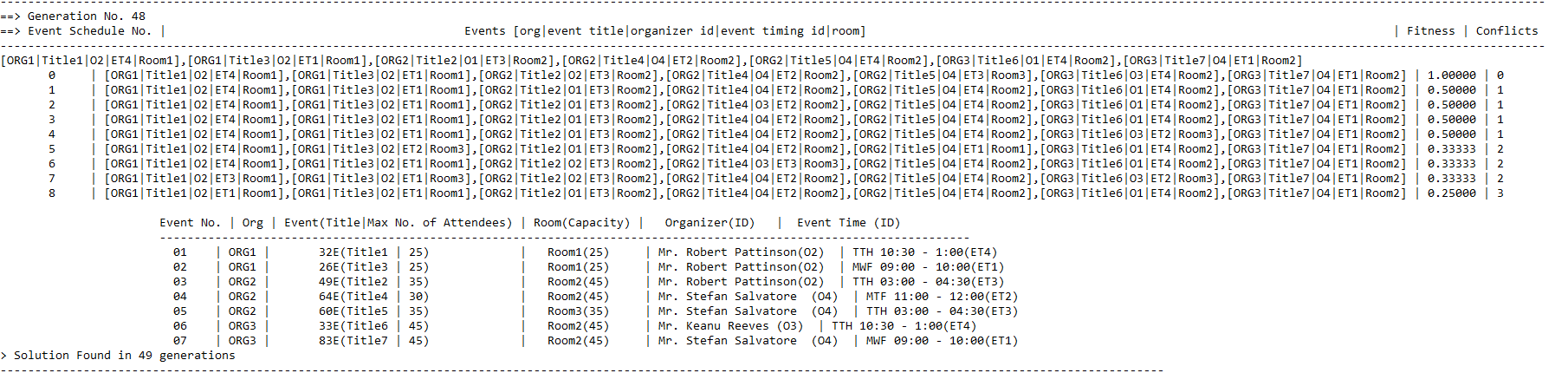
The fitness is 1 for all the trials at the end as there are 0 conflicts between all the event schedules. The time taken for the event scheduling and to get a perfect fitness for a particular number of generations is directly proportional to the number of generations. The Trail with number of generations with 3 is an ideal case. The best case would be to get 0 conflicts in 1 generation.

**Time Complexity**

The sortByFitness() function which uses a system sort takes O(n) time complexity where n is the size of the population.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Trial #** | **Generations Taken** | **Total Time Taken(ms)** | **Fitness** | **Conflicts** |
|  |  |  |  |  |
| 1 | 3 | 0.104 | 1 | 0 |
|  |  |  |  |  |
| 2 | 12 | 0.175 | 1 | 0 |
|  |  |  |  |  |
| 3 | 20 | 0.203 | 1 | 0 |
|  |  |  |  |  |
| 4 | 37 | 0.209 | 1 | 0 |
|  |  |  |  |  |
| 5 | 49 | 0.277 | 1 | 0 |

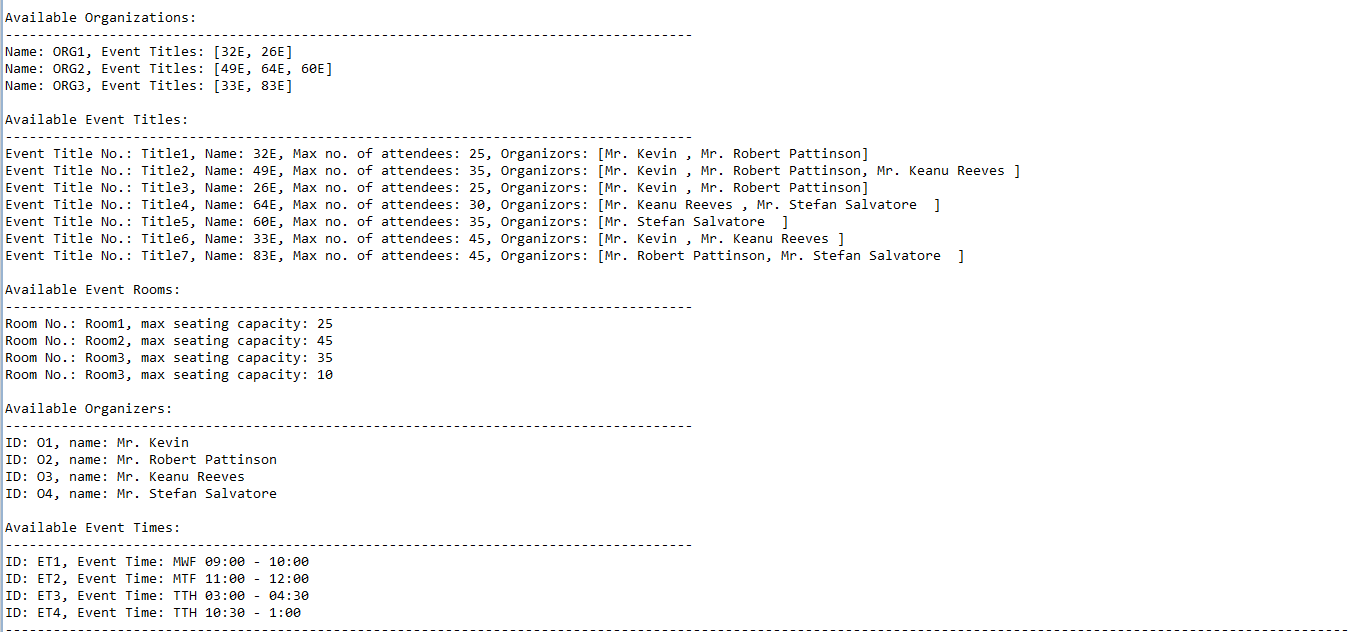
**Trial 1 Snapshot**

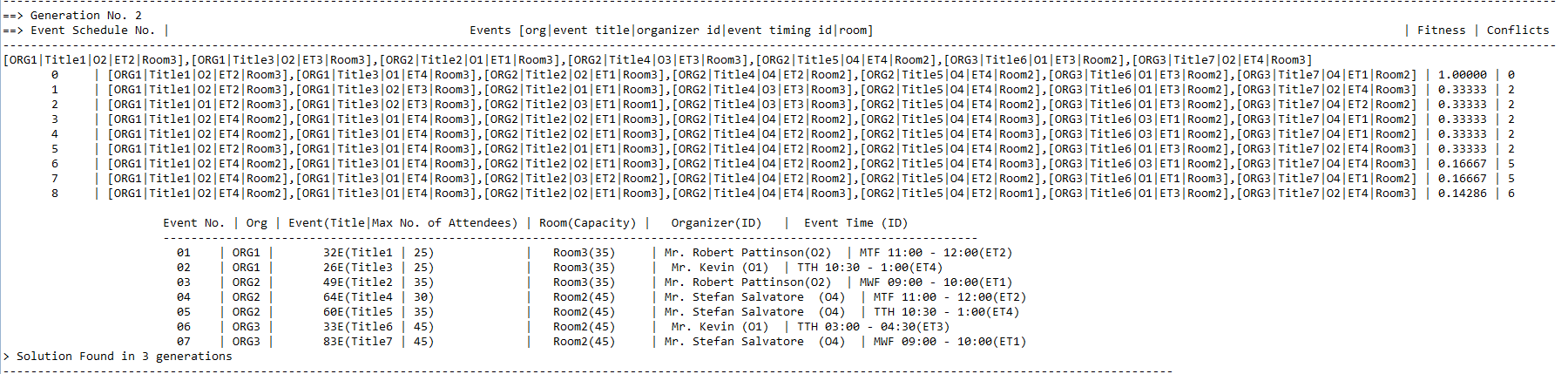


**Trial 2 Snapshot**

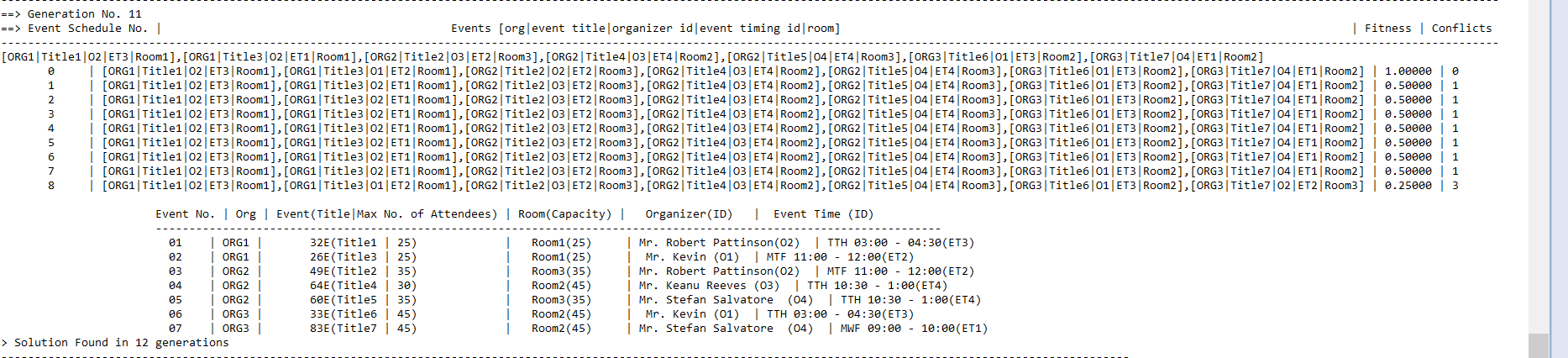
Trial 2 was one of the best cases that we encountered with fitness 1 achieved in only 3 generations.

We have also attached the data that was sent as an input to get this corresponding output.

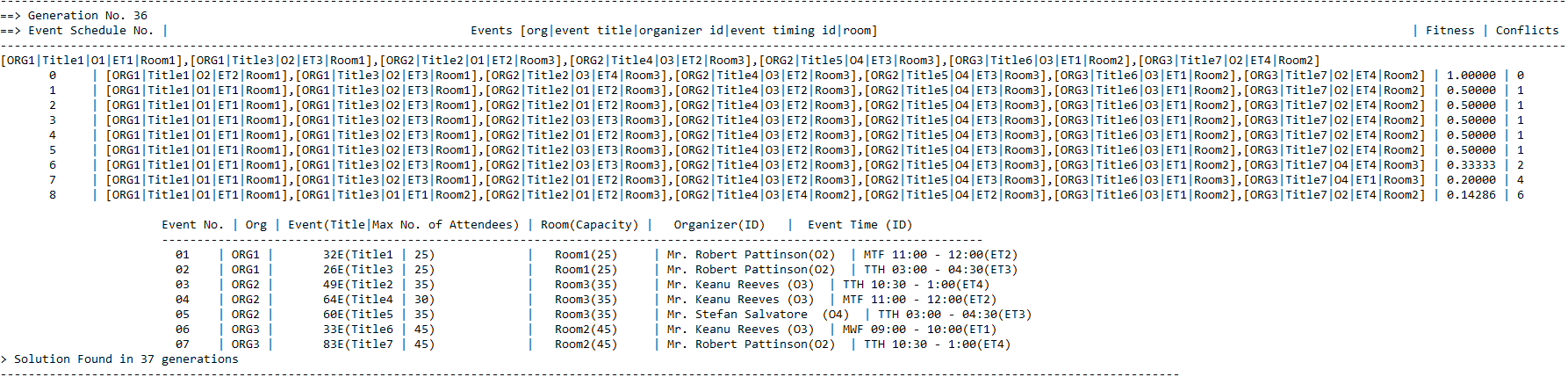




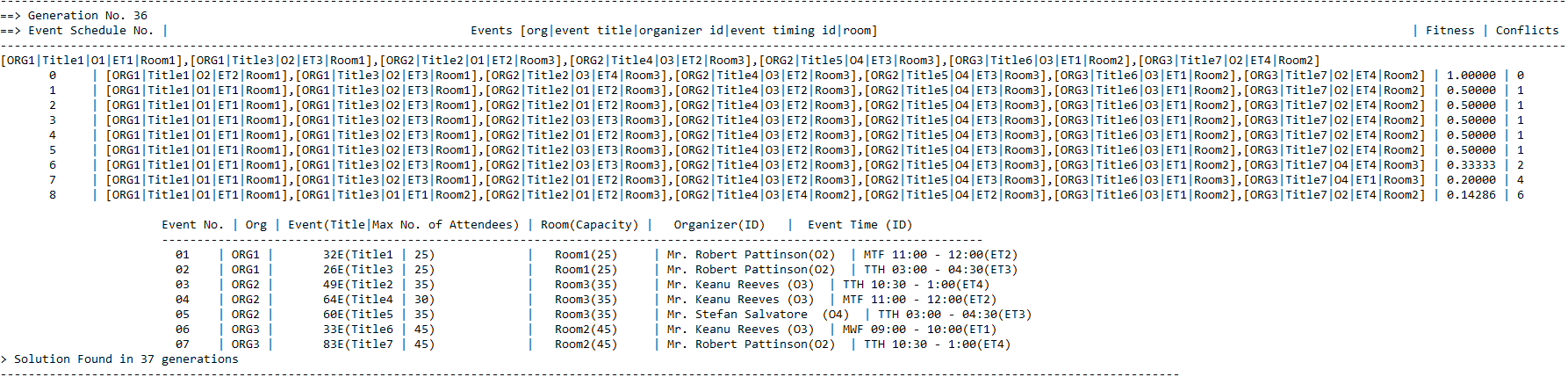
**Trial 3 Snapshot**



**Trial 4 Snapshot**



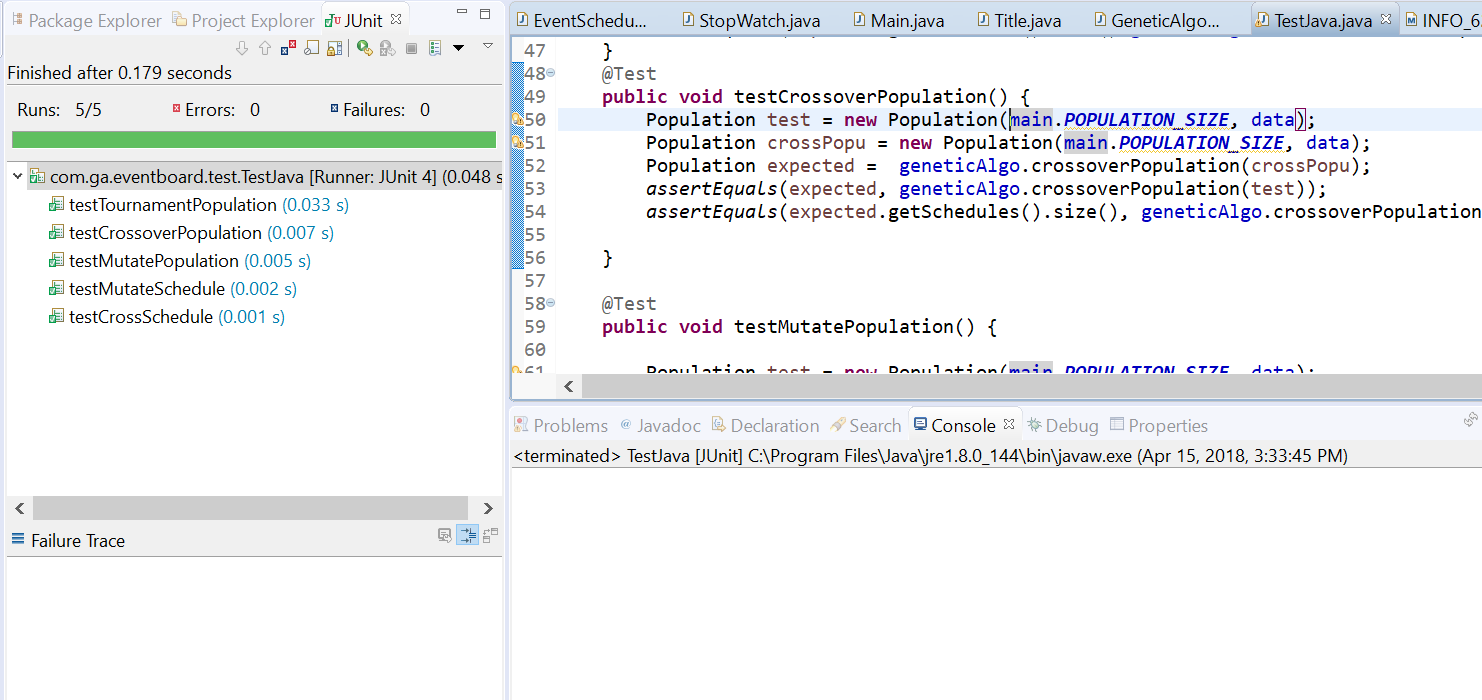
**Trial 5 Snapshot**



**Graphs**

**UNIT TEST CASES**

All unit test cases to test the Genetic algorithm generic methods were passed.



**REFERENCES**

* <https://en.wikipedia.org/wiki/Genetic_algorithm>
* <https://www.tutorialspoint.com/genetic_algorithms/index.htm>
* <https://www.mathworks.com/help/gads/what-is-the-genetic-algorithm.html>
* <https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol1/hmw/article1.html>
* <https://en.wikipedia.org/wiki/Genetic_algorithm_scheduling>