Search Algorithms in Java Language (Timetable Generation) - Group 38

Dinis Moreira
up201503092
MIEIC
FEUP
Porto, Portugal
up201503092@fe.up.pt

Diogo Dores
up201504614
MIEIC
FEUP
Porto, Portugal
up201504614@fe.up.pt

Lus Henriques
up201604343
MIEIC
FEUP
Porto, Portugal
up201604343@fe.up.pt

***Abstract*—In this document, we describe a known problem of timetabling, formulate it as an optimisation problem, solve it with different algorithms, then we analyse the results and how we can benefit from them.**

***Index Terms*—Optimisation Algorithms, Meta-heuristics, Artificial Intelligence, Hill-Climbing Algorithm, Simulated Annealing, and Genetic Algorithms.**

I. Introduction

In this project we will implement an application capable of solving a typical university course timetabling problem, without any external interaction, using optimisation algorithms and meta-heuristics of Artificial Intelligence, namely Hill-Climbing algorithm, Simulated Annealing, and Genetic Algorithms.

II. Problem Description

The problem consists of a set of events that can be scheduled in 45 time slots distributed through 5 days of 9 hours each, a set of rooms in which the events can take place, and a set of students to attend the events. Each student attends a number of events and each room has a size and some amount of features. A feasible timetable is one in which all events have been assigned a time slot and a room. The information regarding the number of events, rooms, features and students should be read from a *.tim* file and have the following format:

First line: Number of events, number of rooms, number of features, number of students.
One line for each room: Represents the room size.
One line for each student/event: A zero or one. Zero means that the student does not attend the event, one means that he does attend the event.
One line for each room/feature: A zero if the room does not satisfy the feature, or a one if the room does satisfy the feature.
One line for each event/feature: A zero if the event does not require the feature, or a one if it does.

Additionally, the best solution found by the algorithm must be written on an output file with extension *.sln* in the following format. For each event, in the order of the problem file, one per line: the time slot number and the room number.

The time slot number is an integer between 0 and 44 representing the time slots allocated to the event.

The room is the room number assigned to the event. Rooms are numbered in the order from the problem file, starting at zero.

Finally, there is a set of hard constraints that need to be met for the solution to be valid, as well as a set of soft constraints.

Hard Constraints

- No student attends more than one event at the same time
- The room needs to be big enough for all the attending students and satisfies all the features required by the event
- Only one event is in each room at any time slot

Soft Constraints

- A student has a class in the last slot of the day
- A student has more than two classes consecutively
- A student has a single class on a day

III. Related Work

Taking into account that the suggested problem was already solved by a number of people during the International Timetabling Competition in 2003, further exploration of this competition will be valuable for us. [1]

Furthermore, this type of project already has a vast number of articles that will most likely be very useful to consult during our implementation of this problem. For instance, a thesis made by a bachelor student at the University of Amsterdam delves into the Comparison of Heuristic Approaches to Timetable Generation at Science Park [2]. We have also come across an article regarding the difference between the Hill Climbing and the Genetic algorithm in solving the timetabling problem will also prove to be useful in the future [3].

Finally, a few projects regarding the problem described above were also encountered. These, in conjunction with the concepts and knowledge obtained in class, will provide us with useful insight on how to approach the timetable generation problem [4] [5] [6].

IV. Implementation

This project was developed using the Java programming language. We chose to implement this game in this specific language, not only because all the members already felt comfortable using it, but also because it is one of the most used programming languages in the world. Java is also widely used in the Artificial Intelligence context. Therefore, we believed that utilising this language would benefit us over all other languages.

Our project's main logic is structured within five classes. The classes Event, Room and Student store the information of the given data set.

The class Event has the following constructor and attributes:

```java
private int id;
private List<Integer> requiredFeatures;

private List<Room> acceptableRooms;
private int attendeesNum;

// Solution Parameters
private int timeSlot;
private Room room;

public Event(int id) {
    this.id = id;
    this.attendeesNum = 0;
    this.requiredFeatures = new ArrayList<>();

    this.timeSlot = -1;
    this.acceptableRooms = new ArrayList<>();
}
```

As it is shown above, this class is responsible for storing the basic information of an event.

The class Room has the following constructor and attributes:

```java
private int id;
private int size;
private List<Integer> features;

public Room(int id, int size) {
    this.id = id;
    this.size = size;
    this.features = new ArrayList<>();
}
```

As it is shown above, this class is responsible for storing the basic information of a room.

The class Student has the following constructor and it's id attribute:

```java
private int id;

public Student(int id) {
    this.id = id;
}
```

We chose to store the id of each student in this class to make code reading easier.

However, the core structure of our project resides in the Solution and Problem classes.

The class Problem, serves not only as a parser for the testing files, but also as the structure responsible to analyse that information and structure it around the rest of our project, making it easily accessible at any time.

```java
final public static int numberOfDays = 5;
final public static int hoursPerDay = 9;
final public static int timeSlots = numberOfDays *
    hoursPerDay;

private List<Room> rooms;
private List<Student> students;
private List<Event> events;

private ArrayList<ArrayList<Boolean>> studentEvents;

//Constructor
public Problem() {
    this.rooms = new ArrayList<>();
    this.students = new ArrayList<>();
    this.events = new ArrayList<>();

    this.studentEvents = new ArrayList<ArrayList<
        Boolean>>();
}
```

The code excerpt above shows the constructor and attributes of this class. As it is shown, the problem's actors are all stored in this class.

The class Solution is responsible for the allocation of events to rooms and time slots, verification of the plausibility of a solution by calculating the hard and soft constraints of each problem and the overall logic of it. Furthermore, it is also responsible for outputting the solution of a problem to a file.

One last point that is important to take note as well is the way the problem files are imported and how the solution is exported to a different file.

Regarding the method how the file information is imported, below is presented the function responsible for it.

```java
public void getProblemFromFile(File file) throws
    FileNotFoundException {
    Scanner fi = new Scanner(file);
    int numEvents = fi.nextInt();
    int numRooms = fi.nextInt();
    int numFeatures = fi.nextInt();
    int numStudents = fi.nextInt();

    //Initialize students/events relation matrix
    for(int s = 0; s < numStudents; s++){
        this.studentEvents.add(new ArrayList<Boolean
            >());
        for(int e = 0; e < numEvents; e++){
            this.studentEvents.get(s).add(false);
        }
    }

    //Add students
    for(int i = 0; i < numStudents; i++){
        Student stud = new Student(i);
        this.students.add(stud);
    }

    //Add Events (without required features)
    for(int i = 0; i < numEvents; i++){
        Event e = new Event(i);
        this.events.add(e);
    }

    //Add Rooms (without features)
    for(int i = 0; i < numRooms; i++){
        Room room = new Room(i, fi.nextInt());
        this.rooms.add(room);
    }

    //Add student Event relations
    for(int s = 0; s < numStudents; s++){
        for(int e = 0; e < numEvents; e++){
            if(fi.nextInt() == 1){
                final Event event = this.events.get(
                    e);
                if(addStudentEventRelation(this.
                    students.get(s), event) == true)
                    event.setAttendeesNum(event.
                        getAttendeesNum() +1);
            }
        }
    }

    //Add features to rooms
    for(int r = 0; r < numRooms; r++){
        for(int f = 0; f < numFeatures; f++){
            if(fi.nextInt() == 1){
                this.rooms.get(r).addFeature(f);
            }
        }
    }

    //Add required features to events
    for(int e = 0; e < numEvents; e++){
        for(int f = 0; f < numFeatures; f++){
            if(fi.nextInt() == 1){
                this.events.get(e).addFeature(f);
            }
        }
    }

    // Add to Event all acceptable Rooms
    for(Event event : this.events) {
        for(Room room : this.rooms) {
            final Boolean attendeeCheck = room.
                getSize() >= event.getAttendeesNum()
                ;
            final Boolean featuresCheck =
                roomHasRequiredFeatures(room, event)
                ;
            if(attendeeCheck && featuresCheck) {
                event.addAcceptableRoom(room);
            }
        }
    }

    //Check if there are more values
    if(!fi.hasNext()){
        System.out.println("Successfully red file");
        return;
    }
    System.out.println("WARNING: Bad file values");

}
```

As demonstrated by the comments in the excerpt of code above, the process is fairly simple. Using the data given in the first line we can easily isolate the number of rooms, events and students. With this information, we are able to gather all the data in the file, knowing where each part of information about the problem begins and ends.

In order to complete every task mentioned in the project's description, when a valid solution is reached, we write the solution of the selected problem into its own file. The function responsible for this task is presented below.

```
public void outputSolutionToFile(String fileName) {
    File file = new File(fileName);
    PrintWriter printWriter;
    try {
        printWriter = new PrintWriter(file);
    } catch (FileNotFoundException e) {
        return;
    }
    for (Event e : this.eventList) {
        // Pad number with max of 4 spaces
        printWriter.printf("%4d %4d\n", e.
            getTimeSlot(), ((e.getRoom() == null) ?
            -1 : e.getRoom().getID()));
    }
    printWriter.close();
}
```

This function is simply in charge of filling a file where each line represents each of the events declared in the problem, and the information within each line represents, respectively, the time slot and the identifier of the room the event was assigned.

V. Search Algorithms

In this project, the implemented algorithms were the ones described in the project's description: Hill-Climbing, Simulated Annealing and Genetic algorithm.

The Hill-Climbing algorithm (or, in our case, Hill-Descent) is implemented in the class *HillClimbing.java* and needs to be initialized with the following parameter: problem (a Problem's class object). The code below shows our general implementation of this algorithm.

```
public Solution getSolution(){
    Solution sol = new Solution(prob);
    Solution betterSol;

    System.out.println();
    sol.generateRandomSolution();

    while(sol.getScore() > 0){
        betterSol = getBetterSolution(sol);

        if(betterSol == null){
            System.out.println("Finished with score
                = " + sol.getScore());
            if(sol.getNumberOfHardInfractions() ==
                0){
                System.out.println("Valid Solution")
                    ;
            }
            else{
                System.out.println("Invalid Solution
                    ");
            }
            break;
        }
        else{
            sol = betterSol;
            System.out.println("Best neighbour score
                = " + sol.getScore());

        }
    }

    return sol;
}
```

First of all, we needed to generate a random solution (also described as a node) in order to have a starting point for our algorithm, this starting point has a great influence in the final result of the algorithm. We do this by generating a random time slot and an index for a room. We assign these two values to an event and check if the random allocation is valid. If it is not, a new random time slot and index will be generated, and the process is repeated until a valid time slot and room index is obtained. The random solution is finally generated when every event has a random room and a time slot.

Then, we analyse the generated solution's score. If this score is equal to 0, that means that the randomly generated solution was ideal. But in most cases it is not, and then the algorithm searches for a neighbour with a lesser score, between a certain number of neighbours, a neighbour being a similar solution with only one event with a different time slot or room. If the problem has less then 100 events, all possible neighbours are explored, if not, then a number of neighbours equal to the number of events in the problem are generated randomly, and in either case, the best neighbour is selected.

This process is repeated until the node's score is 0 or the algorithm cannot find a neighbour with a lesser score. In that case, the solution is validated by checking if none of the hard constraints was violated.

The Simulated Annealing algorithm is implemented in the class *SimulatedAnnealing.java* and needs to be initialized with the following parameter: problem (a Problem's class object). The code below shows our general implementation of this algorithm.

```
public Solution getSolution() {
    Solution sol = new Solution(prob);
    Solution randomSol;

    System.out.println();
    sol.generateRandomSolution();

    double temperature = Double.MAX_VALUE;
    while (sol.getScore() > 0) {
        randomSol = getRandomSolution(sol);

        if (randomSol == null) {
            System.out.println("Finished␣with␣score␣
                =␣" + sol.getScore());
            if (sol.getNumberOfHardInfractions() ==
                0) {
                System.out.println("Valid␣Solution")
                    ;
            } else {
                System.out.println("Invalid␣Solution
                    ");
            }
            break;
        } else {
            final double deltaE = randomSol.getScore
                () - sol.getScore();
            if(deltaE > 0)
                sol = randomSol;
            else {
                final double probability = Math.exp(
                    deltaE/temperature);
                final double randomDouble = this.
                    random.nextDouble();
                if(randomDouble < probability)
                    sol = randomSol;
            }

            System.out.println("Best␣neighbour␣score
                ␣=␣" + sol.getScore());

        }
        temperature /= 2;
    }

    return sol;
}
```

This algorithm's logic is very similar to the one presented in the Hill Climbing. The first step is to generate a random solution. The way this procedure is done is described above in the Hill Climbing algorithm.

Then, we analyse the generated solution's score. If this score is equal to 0, that means that the randomly generated solution was ideal. If not, the algorithm calculates the energy difference between the new, randomly generated, node and the previous node. If this difference is greater than 0, that means the new node is a better solution than the previous. In case it isn't, there is a probability that the algorithm will explore the new node. This probability is calculated using the temperature variable, which is decreased by half at the end of every loop.

This process is repeated until the node's score is 0 or the algorithm cannot find a neighbour with a lesser score. In that case, the solution is validated by checking if none of the hard constraints was violated.

Finally, the Genetic algorithm is implemented in the class *Genetic.java* and needs to be initialized with the following parameters: problem (a Problem's class object), populationSize (the wanted size of the population) and mutationProbability (the chance of a child being mutated). The code below shows our general implementation of this algorithm.

```
public Solution getSolution(){
    Random rand = new Random();
    Solution bestSol = null;
    TreeSet<Solution> population =
        generateInitialPopulation();

    Solution p1, p2, child;

    bestSol = population.first();

    while(bestSol.getScore() > 0){
        TreeSet<Solution> newPopulation = new
            TreeSet<Solution>();

        bestSol = population.first();

        System.out.println(bestSol.getScore());

        for(int i = 0; i < populationSize; i++){
            p1 = getRandomPopulationElement(
                population);
            p2 = getRandomPopulationElement(
                population);

            child = reproduce(p1, p2);

            int isMutated = rand.nextInt(101);

            if(mutationProbability > isMutated){
                mutateSolutionOneEvent(child);
            }

            child.calculateSolutionEval();

            newPopulation.add(child);
        }
```

```
            population = newPopulation;
        }
        return bestSol;
}
```

In order to start the algorithm, an initial population has to be generated. That process is done by populating a *TreeSet* with N random solutions, where N is the specified population size.

The following steps are repeated until the score of the first element (the best element) of the *TreeSet* is 0 or the algorithm iterates through 100 generations without getting a better solution then the best one yet.

The algorithm iterates over the population, getting two random elements from the first half of it (the best parents). With these two elements, a child is generated. This child starts with every event the first parent has. However, for every event contained in the child there is 1/2 of a chance that the time slot and room are inherited from the second parent.

After the generation of this child, there is a probability (related to the mutationProbability given as an argument, or calculated dynamically in some of the available options) that this child will be mutated. In that case, one of the events of the child gets assigned to a random time slot and a supported room.

This new child, mutated or not, will be part of a new population, which will then substitute the original population over the next iteration.

During the development of this project, we observed that, with the progression of the generations, each one of them is increasingly less diverse, making it more difficult at more advanced generations to produce better children, as all the available parents were, each time, more similar to each other.

To increase the diversity of later generations we developed a variant of the genetic algorithm with a dynamic chance of mutation, witch starts at 0% for the initial random population and increases by 1% for each consecutive generation that does not produce a better child than the best one yet (with a maximum of 50%), and goes back to 0% each time the generation does produce the best solution yet.

VI. Experiences and Results

In order to easily visualize the results obtained, each one of the tables below will illustrate the outcome of the different algorithms for five experiments of different complexity degrees. In our repository these are represented by the names ourFileX, where X is the number of the experiment. A description regarding what information each test file has can be found below.

ourFile1 - This test file is composed of 7 events, 4 rooms, 3 features and 9 students.
ourFile2 - This test file is composed of 10 events, 5 rooms, 4 features and 5 students.
ourFile3 - This test file is composed of 10 events, 5 rooms, 4 features and 5 students.
ourFile4 - This test file is composed of 40 events, 20 rooms, 20 features and 50 students.
ourFile5 - This test file is composed of 40 events, 20 rooms, 20 features and 50 students.

With the current goal of standardizing the results obtained, each algorithm, in the first three experiments, was ran five times. In the fourth and fifth ones, each algorithm was executed three times, because of complexity of this experiment and, consequently, the large amount of time it took to run each test. Each run time was calculated by averaging the results from these executions.

In each experiment, we have decided to benchmark the run time of each execution and its distance to the perfect solution, the value of this distance is equal to the number of violations to the soft constraints plus the number of hard constraints violated multiplied by 1000, so a hard constraint can have a much higher importance than a soft one. We believe that, by using these two components, we could get a good idea of how each algorithm performed overall.

Regarding the Genetic algorithm, we needed to have different experiments where we changed the population number and the probability of a mutation, as well as our version with the dynamic chance of mutation. Therefore, we created the experiments Genetic 1 - 6, whose meaning we describe below.
Genetic 1 - Genetic algorithm with a population of 10 and a mutation probability of 3%.
Genetic 2 - Genetic algorithm with a population of 30 and a mutation probability of 3%.
Genetic 3 - Genetic algorithm with a population of 10 and a mutation probability of 12%.
Genetic 4 - Genetic algorithm with a population of 30 and a mutation probability of 12%.
Genetic 5 - Genetic algorithm with a population of 10 and a dynamic mutation probability.
Genetic 6 - Genetic algorithm with a population of 30 and a dynamic mutation probability.

TABLE I
EXPERIMENT FOR FILE OURFILE1.TIM

Algorithm	Execution (s)	Distance to perfect solution
Hill-Climbing	0.081	1.6
Simulated Annealing	0.108	2.0
Genetic 1	0.712	2.0
Genetic 2	3.587	1.2
Genetic 3	1.124	1.6
Genetic 4	0.828	0.8
Genetic 5	0.181	2.0
Genetic 6	0.861	1.2

TABLE II
EXPERIMENT FOR FILE OURFILE2.TIM

Algorithm	Execution (s)	Distance to perfect solution
Hill-Climbing	0.033	1.8
Simulated Annealing	0.054	0.6
Genetic 1	1.258	1.0
Genetic 2	0.353	0.2
Genetic 3	0.496	0.0
Genetic 4	1.387	0.0
Genetic 5	0.218	0.2
Genetic 6	0.619	0.0

TABLE III
EXPERIMENT FOR FILE OURFILE3.TIM

Algorithm	Execution (s)	Distance to perfect solution
Hill-Climbing	0.037	0.8
Simulated Annealing	0.072	1.2
Genetic 1	4.399	0.2
Genetic 2	5.780	0.6
Genetic 3	2.148	0.8
Genetic 4	1.455	0.0
Genetic 5	0.586	0.2
Genetic 6	1.142	0.2

TABLE IV
EXPERIMENT FOR FILE OURFILE4.TIM

Algorithm	Execution (s)	Distance to perfect solution
Hill-Climbing	90.557	11.0
Simulated Annealing	190.080	17.7
Genetic 1	45.609	16.0
Genetic 2	121.640	16.0
Genetic 3	11.661	17.3
Genetic 4	40.798	14.3
Genetic 5	26.460	17.3
Genetic 6	56.541	9.6

TABLE V
EXPERIMENT FOR FILE OURFILE5.TIM

Algorithm	Execution (s)	Distance to perfect solution
Hill-Climbing	216.847	17.7
Simulated Annealing	353.239	28.7
Genetic 1	97.294	19.7
Genetic 2	287.538	15.3
Genetic 3	27.881	26.0
Genetic 4	44.030	18.7
Genetic 5	24.167	25.7
Genetic 6	78.471	18.0

Since the first three tests are examples with a fairly low complexity, the results of these problems are not too surprising. The Hill-Climbing algorithm performs in the same way throughout the experiments. The Simulated Annealing algorithm, since its logic and search method are similar to the Hill Climbing one, it would only be expected that its performance would be about the same.

It is in the Genetic algorithm that we notice a bigger discrepancy between the results. First of all, we can notice that Genetic 3 and 4 are the tests which exhibit a lower execution time. This is most probably caused by the higher probability of mutation of a node. However, since the results of this algorithm depend a lot on randomly generated values, there is a small chance that the better results could just be a coincidence in this small problem instances. It is also because of this mutation chance, that the results in the Genetic 2 test are so different between each of these three problems.

But if we look at the results obtained in the fourth and fifth files, we can notice a big difference between the execution times of the Genetic algorithms and the other two. Since these last files are much more complex than the other three, this leads us to conclude that the Genetic algorithm, although it struggles to compete in terms of run time with the Hill Climbing and Simulated Annealing algorithms when tested with small experiments, is much more efficient when applied to larger experiments, such as the ones present in ourFile4 and ourFile5. As for the differences between the different variants of the genetic algorithm, we can conclude that the variants with a larger population size were able to achieve better solutions than the variants with a smaller population, although at the expense of taking more time to reach these results. This happens because a bigger population allows for greater diversity between solutions in each generation, and consequently a better mix of characteristics between them. So, if our goal is to get the best possible solution, we should try running the genetic algorithm with a larger population size, but if we want a quick solution, a smaller population is our best bet. As for the different mutation probabilities, we can see that the version with a higher mutation probability (12%) was able to achieve similar results in a shorter amount of time compared to the version with a lower mutation probability (12%), probably due to the greater diversity achieved in later generations allowing the solutions to evolve faster. Our experiments with the genetic version with a dynamic mutation degree weren't able to achieve significantly better results than the version with a mutation probability of 12%. Obtaining slightly better solutions in a slightly longer time.

VII. CONCLUSION

To summarise, we believe that the concepts and knowledge acquired in the classes, allied to the aid of the documentation and logic used behind the related projects that were found, immensely helped us in the development of the project. We also conclude that our understanding of the logic behind these three algorithm has exponentially increased.

Furthermore, we believe that the algorithms were implemented successfully, since their performance is similar to the performance of other algorithms developed in the articles below.

Lastly, in our opinion, we have achieved all the goals proposed at the beginning of this project and have successfully utilized them in it.

REFERENCES

[1] International Timetabling Competition, 2013, [online], available at http://sferics.idsia.ch/Files/ttcomp2002/oldindex.html, consulted May 2019.

[2] S. Swatman, “A Comparison of Heuristic Approaches to Timetable Generation at Science Park”, June 2016, consulted May 2019.

[3] S. Rahim, ”Hill Climbing versus genetic algorithm optimization in solving the examination timetabling problem”, A. Bargiela, R. Qu, January 2013, consulted May 2019.

[4] Pranav Khurana, Time Table generation using Genetic Algorithms (Java-Struts2)”, 2017, [online], available at https://github.com/pranavkhurana/Time-table-scheduler, consulted in May 2019.

[5] Armin Kazemi, OptimizationAlgorithms”, 2017, [online], available at https://github.com/arminkz/OptimizationAlgorithms, consulted in May 2019.

[6] Yurii Lahodiuk, Generic implementation of genetic algorithm in Java.”, 2014, [online], available at https://github.com/lagodiuk/genetic-algorithm, consulted in May 2019.