Christian Honicker

Dr. Armstrong

CSC 462: Artificial Intelligence

11 December 2023

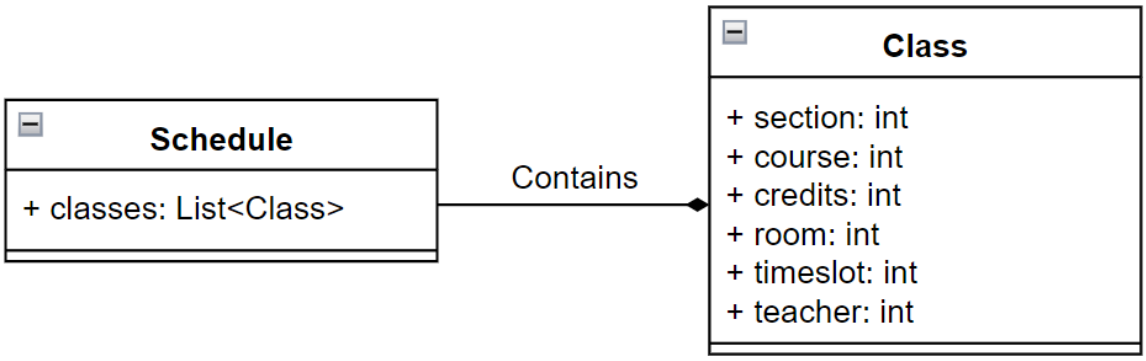
Genetic Algorithm for Developing a College Course Curriculum

1 Introduction

In his paper "The University Class Scheduling Problem", Calvin Hoang Thach proposed a number of different models for mathematically developing a class curriculum for a college course curriculum. While he has decided to use linear programming as the means for creating his schedules, I have adapted some of his models into a genetic algorithm that uses evolution as a means of developing these curriculums. In particular, this paper will demonstrate how I've adapted his Teacher Tricriteria Model (Thach 24-25) into a genetic algorithm.

2 Algorithm Description

2.1 Hypothesis Representation



I chose to represent each hypothesis as a Java class called Schedule. Each Schedule contains a List of another class called Class. Class contains six relevant fields:

- section: Unique course ID
- course: Course number
- credits: number of credits this class is worth (either 3 or 4)
- room: room ID that this class takes place in
- timeslot: timeslot ID that this class takes place during
- teacher: ID of the teacher who teaches this class

Every Class in a Schedule has a unique section value, and a Class's course & credits values are consistent based on the section value. Only room, timeslot, and teacher can mutate and vary between Classes of the same section value.

2.2 Fitness Function

Every Schedule object also contains a FitnessCalculator object, which it uses to calculate its fitness value (lower is better). Exactly what type of FitnessCalculator is used determines what model the algorithm ends up using. Fitness calculation is further split up into two processes: calculating constraints and calculating objective. My reasoning for splitting the fitness function up like this is severalfold, but ultimately boils down to ease of development & allowing for flexibility in weighing different factors.

2.2.1 Calculating Constraints

Because I developed several models iteratively, the constraint function is split up between multiple classes. For the sake of this algorithm description, I will condense all of these classes

into a single algorithm. The following is a pseudocode description of the constraint calculation algorithm, marked with the expressions each part corresponds to in Thach's paper:

```
calculateConstraints(Schedule s):
    Let f = 0
    For every class in the same room, if their timeslots overlap, f++          [2.1]
    For every 3 credit class in a 4 credit timeslot (or vice versa), f++       [2.3, 2.4]
    For every class in the same room, if their timeslots share a category (morning,
    afternoon, evening), f++                                                   [2.5]
    For every class with the same teacher, if their timeslots overlap, f++     [2.18]
    For every class with a teacher that likes whiteboards, if the room has a chalkboard
    (or vice versa), f++                                                       [2.22, 2.23]
    For every class with a teacher that likes a specific timeslot category, if the class's
    timeslot is not in that category, f++                                      [2.24, 2.25, 2.26]
    For every class with a teacher that likes MWF timeslots, if the class is during a
    TR timeslot (or vice versa), f++                                           [2.27, 2.28]
    Return f
```

To put the function more simply, for every constraint violation found in the schedule, f is incremented by 1. This value of f is then returned.

2.2.2 Calculating Objective

Again, since the objective function is split up between multiple classes, this algorithm description will be combining those classes into one algorithm. The following is a pseudocode representation of the objective function, as per Section 2.2.5 of Thach's paper:

```
calculateObjective(Schedule s):
    // MWF objective
    Let w = 0
    For each Class c in s: If c.timeslot is a MWF timeslot: w++; Else w–
    w = |w|
    // Difference objective
    Let q = 0
    Let coursesPerTeacher = new int[numTeachers]
    For each Class c in s: coursesPerTeacher[c.teacher]++
    Let target = numCourses / numTeachers
```

```
For each i in coursesPerTeacher: q += |i - target|
// Satisfaction objective
Let t = 0
For each Class c in s: t += satisfactionMatrix[c.teacher][c.section]
Return weight[0] * w + weight[1] * q + weight[2] * t
```

Again, to put this function simply, it calculates three separate objective functions: w relates to balancing MWF vs TR classes, q relates to equalizing the number of courses each teacher is assigned to, and t is calculated according to the Teacher Satisfaction Matrix. These values are then multiplied by their respective weights, and then summed.

2.2.3 Combining Constraints and Objective

The constraint & objective values are combines as such, using an additional value CONSTRAINT_MULTIPLIER determined by the user, as such:

```
calculateFitness(Schedule s):
    Return calculateObjective(s) + CONSTRAINT_MULTIPLIER *
    calculateConstraints(s)
```

The fitness from constraints is multiplied by CONSTRAINT_MULTIPLIER, then the fitness from objective is added to it.

2.3 Creating the First Generation

In order to create the first generation, a number of Schedule objects equal to a value POPULATION_SIZE determined by the user are randomly generated. When a Schedule is randomly generated, it creates a number of Class objects equal to the length of the course list table passed into it, randomly generating the room, timeslot, and teacher. This random generation is done inside Schedule's constructor and a static method in Class:

```
public Schedule(courselist, rooms, timeslots, teachers):
    Let classes = new ArrayList<Class>()
    for each int[] c in courselist:
```

```
                classes.add(randomClass(c[0], c[1], c[3], rooms, timeslots, teachers)

    static Class randomClass(s, co, cr, r, ti, te):
        Return new Class(s, co, cr,
            rand.nextInt(1, r), rand.nextInt(1, ti), rand.nextInt(1, te)
```

2.4 Killing Method

Once the first generation is created, the evolution loop begins. The first step of this loop is to sort the hypotheses by fitness, then to kill off members of the generation until it's half its original size. The following method is used to kill half a generation:

```
killHalf(List<Schedule> gen):
    Let target = gen.size / 2
    While gen.size > target:
        Let num = rand.nextInt(0, gen.size() ^ 2)
        Let index = sqrt(num)
        If index <= 0: continue
        gen.remove(index)
```

The goal of this method is to kill off hypotheses based on a gradient: the worse a hypothesis is, the more likely it is to be killed, but a given hypothesis is never guaranteed to be killed. Instead, a hypothesis's odds of getting killed are roughly proportional to the square of its placement in the generation. Furthermore, this method guarantees that the best hypothesis survives.

2.5 Reproduction Method

Reproduction is the second step of the evolution loop. In this step, crossover and mutation are both utilized in order to bring the generation back to its original size. This step also makes use of a number of user-defined values: CROSSOVER_POINTS, MUTATION_CHANCE, and MAJOR_MUTATION_CHANCE. The reproduction methods are as such:

```
reproduce(List<Schedule> gen):
    Let foo = a clone of gen
    While foo is not empty:
        Let s1, s2 = members taken from foo
        Let s3 = new Schedule(s1, s2)
        Let s4 = new Schedule(s2, s1)
        Let r1, r2 = rand.nextInt(0, 99)
        // Mutation
        If r1 < MAJOR_MUTATE_CHANCE:
            Let bar = rand.nextInt(2, 10)
            s3.mutateMultiple(bar)
        Else If r1 < MUTATE_CHANCE:
            s3.mutate()
        (Repeat the above mutation steps for s4/r2)
        gen.add(s3, s4)

Schedule(s1, s2):
    Let c1 = s1.classes, c2 = s2.classes
    Let classes = new ArrayList<Class>()
    Let crossPoints = a list of CROSSOVER_POINTS unique integers, bound by
    s1.size
    For i from 1 to c1.size:
        If crossPoints contains i: classes.add(c2.get(i))
        Else: classes.add(c1.get(i))

Schedule.mutate():
    Let index = rand.nextInt(0, classes.size)
    Let c = classes.get(index).clone()
    Let property = rand.nextInt(0, 2)
    switch(property):
        Case 0 (Room): c.room = rand.nextInt(1, numRooms)
        Case 1 (Timeslot): c.timeslot = rand.nextInt(1, numTimeSlots)
        Case 2 (Teacher): c.teacher = rand.nextInt(1, numTeachers)
    classes.remove(index) ; classes.add(c, at index)

Schedule.mutateMultiple(num):
    For i from 1 to num: mutate()
```

What this part of the algorithm is doing is taking two hypotheses, performing crossover to produce children, then randomly deciding whether to mutate them or not. These children are then added to the generation. This is done until there are no more hypotheses left to crossover.

From here, the generation is sorted by fitness, now with the new reproduced hypotheses, and the evolution loop starts anew. This evolution loop continues until either some fitness threshold is reached by the generation's best hypothesis, or until some generation count threshold is reached.

2.6 Random Number Generator

I wasn't sure where else to put this, but I wanted to talk about how this algorithm uses the Random Number Generator, or RNG. An RNG shared singleton, referred to as rand throughoutn this paper, is used by every single class & method that requires random number generation. This singleton can be seeded, allowing for the user to produce deterministic evolution results—the same random seed, along with the same preset parameters, will produce the same results every time.

3 Results

The following is the best hypothesis I was able to produce from the evolutionary algorithm. First, the settings used to produce this hypothesis, in case you want to reproduce my results:

```
{
    "DATA_FOLDER": "simulated-data",
    "POPULATION_SIZE": 10000,
    "MAX_GENS": 200,
    "FITNESS_THRESHOLD": 0.0,
    "CROSSOVER_POINTS": 3,
    "MUTATION_CHANCE": 10,
    "MAJOR_MUTATION_CHANCE": 2,
```

```
    "CONSTRAINT_WEIGHT": 100,
    "WEIGHT_W": 0.33, "WEIGHT_Q": 0.33, "WEIGHT_T": 0.33,
    "SEED": 0
}
```

My reasoning for these settings are as follows:

- POPULATION_SIZE: I found that a population size of 10000 provided a nice balance between variety and efficiency, as each generation takes, at most, just over a second to compute.
- MAX_GENS: While the number of generations it took for a simulation to plateau varied from model to model, I found that the Tricriteria Model frequently plateau'd before Generation 150, and almost always before Generation 200.
- FITNESS_THRESHOLD: As I wasn't sure what the theoretical minimum was for this model, I set it to 0, something unlikely to be achieved, to see how low the simulation can go.
- CROSSOVER_POINTS: I felt like if I had too many crossover points at the reproduction step, the resulting children would almost certainly be worse than their parents. As such, I kept this value low.
- MUTATION_CHANCE: Kept this at a reasonable value so that most children would be stable with no mutations, but there's still enough mutated children to introduce variety to each generation.
- MAJOR_MUTATION_CHANCE: Same reasoning as MUTATION_CHANCE
- WEIGHT_W/Q/T: The first idea I tried in regards to weighing these objective functions against one another was to just weight each one separately, and that ended up working out.

- SEED: Thankfully, I did not need to search long for a good seed.

And now for the schedule itself:

```
Best speciman:
Course 1 (1-1)   | Room 10 | Teacher 10 | MWF 2 - 2:50pm
Course 2 (1-2)   | Room 8  | Teacher 9  | MWF 4 - 4:50
Course 3 (1-3)   | Room 11 | Teacher 3  | MWF 5 - 5:50pm
Course 4 (2-1)   | Room 4  | Teacher 5  | TR 6 - 7:50pm
Course 5 (2-2)   | Room 6  | Teacher 4  | TR 1 - 2:50pm
Course 6 (2-3)   | Room 4  | Teacher 5  | TR 8 - 9:50am
Course 7 (3-1)   | Room 1  | Teacher 6  | TR 6 - 7:50pm
Course 8 (3-2)   | Room 9  | Teacher 7  | WF 7 - 8:50pm
Course 9 (3-3)   | Room 1  | Teacher 2  | TR 1 - 2:50pm
Course 10 (4-1)  | Room 2  | Teacher 2  | MW 3 - 4:50pm
Course 11 (4-2)  | Room 9  | Teacher 9  | MWF 7am - 8:05am
Course 12 (4-3)  | Room 7  | Teacher 9  | MWF 9:30am - 10:35am
Course 13 (5-1)  | Room 7  | Teacher 3  | MW 6 - 7:50pm
Course 14 (5-2)  | Room 3  | Teacher 5  | TR 10 - 11:50am
Course 15 (5-3)  | Room 6  | Teacher 1  | TR 10 - 11:50am
Course 16 (6-1)  | Room 5  | Teacher 6  | TR 8 - 9:50pm
Course 17 (6-2)  | Room 3  | Teacher 8  | TR 3 - 4:50pm
Course 18 (6-3)  | Room 9  | Teacher 2  | TR 3 - 4:50pm
Course 19 (7-1)  | Room 4  | Teacher 10 | MWF 3:45pm - 4:50pm
Course 20 (7-2)  | Room 6  | Teacher 9  | WF 8 - 9:50pm
Course 21 (7-3)  | Room 8  | Teacher 3  | TR 8 - 9:50am
Course 22 (8-1)  | Room 11 | Teacher 1  | TR 8 - 9:50am
Course 23 (8-2)  | Room 8  | Teacher 7  | MW 5 - 6:50pm
Course 24 (8-3)  | Room 10 | Teacher 7  | MWF 10:45am - 11:50am
Course 25 (9-1)  | Room 3  | Teacher 6  | MF 8 - 9:50pm
Course 26 (9-2)  | Room 10 | Teacher 3  | TR 5 - 6:50pm
Course 27 (9-3)  | Room 7  | Teacher 2  | MF 1 - 2:50pm
Course 28 (10-1) | Room 5  | Teacher 4  | TR 3 - 4:50pm
Course 29 (10-2) | Room 11 | Teacher 10 | MWF 12pm - 1:05pm
Fitness: 41.10206896551725
```

The full output log is included as part of the source code.

Since the fitness is below the CONSTRAINT_MULTIPLIER value of 100, we know at least the calculateConstraints portion of the fitness function is returning 0, thus no constraints are being violated. And as a reminder, lower fitness is better.

For context, let's compare this with a random specimen, generated via the same method that generates specimens for the first generation of evolution:

```
Random speciman:
Course 1 (1-1)   | Room 1  | Teacher 10 | MW 8:30 - 9:45pm
Course 2 (1-2)   | Room 6  | Teacher 4  | MW 7 - 8:15pm
Course 3 (1-3)   | Room 3  | Teacher 10 | MWF 9 - 9:50pm
Course 4 (2-1)   | Room 1  | Teacher 8  | MF 4 - 5:15pm
Course 5 (2-2)   | Room 3  | Teacher 6  | MW 1 - 2:50pm
Course 6 (2-3)   | Room 2  | Teacher 6  | MF 2:30 - 3:45pm
Course 7 (3-1)   | Room 10 | Teacher 4  | MF 11:30am - 12:45pm
Course 8 (3-2)   | Room 8  | Teacher 8  | MWF 8 - 8:50am
Course 9 (3-3)   | Room 11 | Teacher 4  | MW 7 - 8:50pm
Course 10 (4-1)  | Room 7  | Teacher 4  | TR 7 - 8:15am
Course 11 (4-2)  | Room 2  | Teacher 8  | MWF 11 - 11:50am
Course 12 (4-3)  | Room 11 | Teacher 3  | TR 2:30 - 3:45pm
Course 13 (5-1)  | Room 5  | Teacher 9  | WF 8:30 - 9:45pm
Course 14 (5-2)  | Room 3  | Teacher 6  | TR 1 - 2:50pm
Course 15 (5-3)  | Room 2  | Teacher 9  | MWF 8 - 8:50am
Course 16 (6-1)  | Room 2  | Teacher 7  | MF 1 - 2:50pm
Course 17 (6-2)  | Room 9  | Teacher 10 | TR 7 - 8:15am
Course 18 (6-3)  | Room 4  | Teacher 7  | WF 11am - 12:50pm
Course 19 (7-1)  | Room 6  | Teacher 9  | MW 5:30 - 6:45pm
Course 20 (7-2)  | Room 11 | Teacher 8  | MW 6 - 7:50pm
Course 21 (7-3)  | Room 4  | Teacher 6  | MWF 5pm - 6:05pm
Course 22 (8-1)  | Room 2  | Teacher 8  | MW 7 - 8:50pm
Course 23 (8-2)  | Room 9  | Teacher 9  | WF 3 - 4:50pm
Course 24 (8-3)  | Room 8  | Teacher 3  | MW 7 - 8:50pm
Course 25 (9-1)  | Room 4  | Teacher 7  | MW 1 - 2:15pm
Course 26 (9-2)  | Room 9  | Teacher 3  | MW 4 - 5:15pm
Course 27 (9-3)  | Room 9  | Teacher 8  | MF 11:30am - 12:45pm
Course 28 (10-1) | Room 9  | Teacher 10 | MWF 1 - 1:50pm
Course 29 (10-2) | Room 9  | Teacher 10 | TR 2:30 - 3:45pm
Fitness: 8555.314827586208
```

This randomly generated schedule has 8500 of its fitness points coming from constraints, and 55.3 points coming from the objective. Our best specimen solidly beats out the random speciman in both of these aspects, with 0 points from constraints and 41.1 points from objective.

4 Comparison with Thach's Results

Here's the solution Thach derived via linear programming methods for the Teacher Tricriteria Model (Thach 49, 61), alongside the schedule's calculated fitness:

```
Thach's solution:
Course 1 (1-1)   | Room 10 | Teacher 2  | TR 4 - 5:15pm
Course 2 (1-2)   | Room 9  | Teacher 4  | TR 4 - 5:15pm
Course 3 (1-3)   | Room 11 | Teacher 8  | TR 4 - 5:15pm
Course 4 (2-1)   | Room 11 | Teacher 7  | MWF 8:45pm - 9:50pm
Course 5 (2-2)   | Room 8  | Teacher 2  | MWF 3:45pm - 4:50pm
Course 6 (2-3)   | Room 11 | Teacher 10 | MF 11am - 12:50pm
Course 7 (3-1)   | Room 11 | Teacher 7  | MWF 9:30am - 10:35am
Course 8 (3-2)   | Room 5  | Teacher 5  | TR 5 - 6:50pm
Course 9 (3-3)   | Room 9  | Teacher 3  | MWF 5pm - 6:05pm
Course 10 (4-1)  | Room 1  | Teacher 6  | TR 5 - 6:50pm
Course 11 (4-2)  | Room 10 | Teacher 9  | MWF 10:45am - 11:50am
Course 12 (4-3)  | Room 11 | Teacher 9  | MWF 5pm - 6:05pm
Course 13 (5-1)  | Room 5  | Teacher 6  | TR 8 - 9:50pm
Course 14 (5-2)  | Room 4  | Teacher 5  | TR 8 - 9:50pm
Course 15 (5-3)  | Room 11 | Teacher 4  | TR 1 - 2:50pm
Course 16 (6-1)  | Room 10 | Teacher 8  | TR 1 - 2:50pm
Course 17 (6-2)  | Room 11 | Teacher 3  | TR 8 - 9:50pm
Course 18 (6-3)  | Room 11 | Teacher 1  | TR 10 - 11:50am
Course 19 (7-1)  | Room 10 | Teacher 10 | MWF 3:45pm - 4:50pm
Course 20 (7-2)  | Room 6  | Teacher 3  | TR 10 - 11:50am
Course 21 (7-3)  | Room 9  | Teacher 3  | MWF 2:30pm - 3:35pm
Course 22 (8-1)  | Room 10 | Teacher 9  | MWF 8:45pm - 9:50pm
Course 23 (8-2)  | Room 6  | Teacher 7  | MW 5 - 6:50pm
Course 24 (8-3)  | Room 11 | Teacher 1  | TR 8 - 9:50am
Course 25 (9-1)  | Room 5  | Teacher 6  | MWF 8:45pm - 9:50pm
Course 26 (9-2)  | Room 5  | Teacher 5  | TR 10 - 11:50am
Course 27 (9-3)  | Room 9  | Teacher 2  | TR 1 - 2:50pm
Course 28 (10-1) | Room 10 | Teacher 10 | MWF 1:15pm - 2:20pm
Course 29 (10-2) | Room 11 | Teacher 9  | MWF 2:30pm - 3:35pm
Fitness: 3336.482068965517
```

The fitness values might seem to imply at first that the evolution-derived solution is better, but further analysis from calculating these with different CONSTRAINT_WEIGHT values gives different results: namely, if CONSTRAINT_WEIGHT is set to 0 (that is, constraints aren't even

considered), Thach's schedule's fitness goes all the way down to 36.5, a whopping 3300 point drop. Meanwhile the evolution schedule's fitness does not change from 41.1, meaning Thach's schedule is better under these conditions. Clearly, our constraint function is disagreeing with Thach's schedule in some way. After poking around the constraints functions (and fixing some bugs and copy errors along the way), I found the following contributors to Thach's schedule's fitness:

- A room cannot be used more than once per timeslot category [2.5]: maybe this constraint got removed from Thach's models at a later point and I didn't notice, or maybe I took this constraint too literally, but Thach's models seem to be a lot more loose with this constraint than I was, seeing how this constraint accounted for 3200 fitness points.
- Two classes in the same room cannot overlap in timeslots [2.1]: apparently one pair of classes is causing this constraint to trip, adding 100 fitness points. However, I can't seem to find what pair is causing this though, or if it's due to a copy error on my part. In any case this is likely due to some sort of error.

Adding these up gives the 3300 fitness points that Thach's schedule was getting due to constraints. Accounting for these constraints & subtracting the fitness points accounted by them can indicate that Thach's schedule is, in fact, better than one that the genetic algorithm has produced. But the different still isn't that large; 41.1 vs 36.6—a 4.6 point gap—is indicative of just a few spots of unoptimality, and is less of a gap than that between the evolution schedule and a random schedule—a 14.2 point difference if you don't account for constraints.

5 Conclusion

We have successfully demonstrated the usage of a genetic algorithm to produce a course curriculum for a university. Although the resultswe achieved are still worse in some aspects as compared to the schedules Calvin Thach produced via linear programming, they're not that far off, and they're still measurably better than generating schedules randomly. And while the algorithm might have its faults, and there may be a better seed out there that I haven't found yet, I'm still satisfied with what this algorithm was able to do.

Works Cited

Thach, Calvin. "The University Class Scheduling Problem." (2020).