A Genetic Algorithm Approach for Finding a Good Course Schedule

Knut-Edvart Ellingsen[1] and Manuel Penaloza
Department of Mathematics and Computer Science
South Dakota School of Mines and Technology
501 East St. Joseph Street
Rapid City, SD 57701-3995
[1]ellingsen@knut-edvart.com **or** ke_ellingsen@hotmail.com

Abstract

Scheduling classes is a time consuming job for administrators. Many constraints are defined for classrooms, faculty members, and courses. For instance, a course may require a classroom with some minimum number of seats and with some audiovisual equipment. A faculty member may not prefer teaching two or more courses in a row, or he or she may prefer teaching before 3 PM.

The goal of this study was the use of genetic algorithm (GA) principles for finding a "good" schedule that results in an efficient use of each classroom, in relation to time, space, and constraints. The GA program initializes a single schedule randomly representing a single individual. This individual is altered by mutations, and its fitness is evaluated using a fitness function. The modified individual is either kept or discarded. The final schedule for the courses is displayed along with its fitness value.

Introduction

Some optimization problems require heuristic methods for finding near optimal solutions within some reasonable amount of time. Several search algorithms have been used elsewhere for solving scheduling problems. For instance, a very large-scale neighborhood search algorithm is described in [1] for airline fleet scheduling. A genetic algorithm (GA) is another search method based on the mechanics of biological systems. Genetic algorithms provide robust search in complex spaces [2]. GA starts by defining a random population of points (individuals) of the search space and generate a new better fitted population by applying three basic operations: reproduction or selection, crossover and mutations.

GA was used previously in a scheduling problem that consists in finding a near optimal schedule for K classes, taught by M instructors in a single classroom [3]. Each class consists of several sections. The class sections were taught from Monday through Friday, of six hours each day. The "best" schedule minimizes the penalty for not satisfying several soft and hard constraints.

In [3], each individual of the population represents a particular class schedule. The initial population consisted of a small set of schedules. The selection of individuals for mating was done in two phases: pre-selection by tournament, and by random selection. The tournament selected few schedules with the highest fitness function. The fitness function evaluated the different hard and soft constraints resulting in a fitness value. To complete the number of mates, other schedules were selected randomly. Crossover took two schedules and swapped the times of two random classes, one from each schedule. Mutation switched times of two sections for a given class.

In this work, we used a population with a single individual (schedule) and applied only mutations to this single schedule. This approach was used by Levi [4] for the implementation of a fast algorithm named HereBoy for searching large search spaces. Hereboy required up to100 times fewer iterations than other solutions that defined a population with more than one individual. The following sections describe the implementation of our work, followed by results and conclusion.

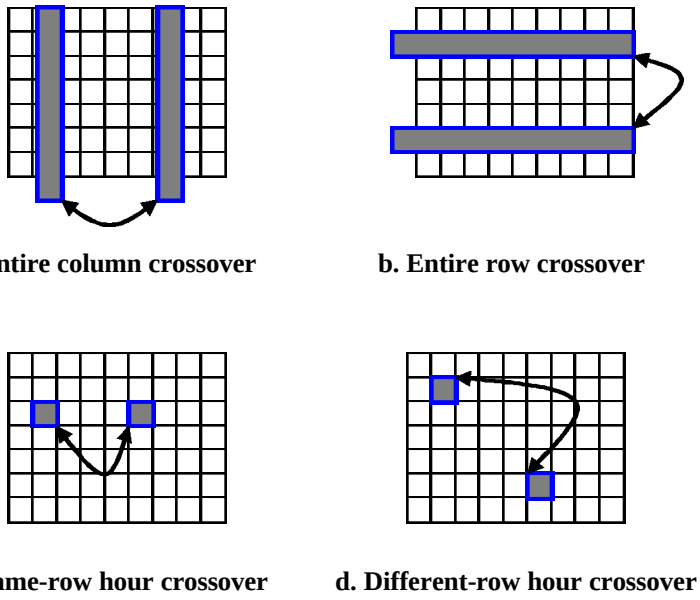
Implementation

The population consisted of a single individual (schedule), and the GA algorithm, implemented in a C++ program, only performed the selection and mutation GA operations. Having a single schedule avoided problems such as checking that class sections were not repeated or got lost on the schedules, when performing the mutation operations. The single schedule used here was randomly generated using information about classes, class sections, instructors, and classrooms stored in separate binary files. Data was used from the course listing at the South Dakota School of Mines (SDSM&T) for the current academic semester. The attributes for each of these records are described later in the Data Structures subsection.

A course at the SDSM&T, are commonly taught either on Monday, Wednesday, and Friday, or on Tuesday, and Thursday. To make our program work with this model, we subdivided the schedule into two sub-schedules called MWF-schedule and TTH-schedule. Description of these two schedules is given in the Operation subsection.

Selection was performed after the schedule was altered by mutations. The fitness function for the mutated schedule was evaluated. Selection consisted in selecting either the mutated schedule or the schedule before the mutations. If the mutated schedule had a higher fitness value that the unmutated schedule, this new schedule was selected. Otherwise, the unmutated schedule was kept. The program repeated this process as many times as indicated by a user.

Several mutation operations were defined. Some of these mutations were actually crossover operations when the population consists of more than one schedule. For instance, a regular crossover takes two candidate schedules and divides them, swapping components to produce two new candidates. Figure 1 shows several crossover variants. Our solution is based only on changes in a single schedule. Therefore, these crossover operations are simply mutations.



a. Entire column crossover **b. Entire row crossover**

c. Same-row hour crossover **d. Different-row hour crossover**

Figure 1: Several variants of the crossover operation.

There are four different kinds of mutations used in the program. Three of them are regular crossover used in GA programs having multiple individual populations. *Row_reselect*, *Column_reselect*, *Classroom_reselect* are the three kinds of crossovers used in our GA program. One of them will occur in 70% of the reproduction cases. *Row_reselect* (Figure 1b) is a function that randomly picks out one classroom, and changes the schedule for that classroom with a different classroom, which is also randomly picked. *Column_reselect* (Figure 1a) is a function that randomly selects two

scheduled hours of teaching and swaps all the classes assigned to these two scheduled hours. *Classroom_reselect* (Figure 1c) is a function that randomly picks out one classroom and swaps two classes within that classroom. The fourth mutation function (Figure 1d) simply called mutation randomly takes one class from one classroom and swaps it with a class of a different classroom.

Eleven different constraints were identified and each one was implemented by a separate C++ function. The constraints defined for this program were classified as fixed (4), hard (4), or soft (3). Fixed-, hard-, and soft constraints had high, medium, and low priorities, respectively. Every course/instructor pair of a schedule that satisfies the fixed-, hard-, or soft constraints was assigned a fitness value of 100, 70, or 30, respectively. That is, the maximum fitness value is 200 for an optimal course/instructor pair, and for an optimal course schedule. For example, if a course/instructor pair satisfies three fixed constraints, 2 hard constraints, and all three soft constraints, it would have a fitness value of (3/4)100+(2/4)70+30, for a total of 140. The schedule's fitness value is the average computed for all the course/instructor pairs in the schedule. The next subsections describe each constraint and the functions that evaluate each of them. Each function returns a value that depends on how many times the constraint is violated, it ranges from 0 (completely violated) to 100 (no violation). From this point, the word *course* refers to a *class section.*

Fixed Constraints

There are four different fixed-constraints, and they can have a combined maximum fitness value of 100 all together. The fixed constraints are constraints with the highest priorities. The GA program checked for these fixed constraints with the assistance of the functions *classesNeedSpecialRoom(), classroomSize(), twoHoursInRowClasses(),* and *useClassroomAvailableTime().* Every function starts with an accumulative value of 0.

classesNeedSpecialRoom() counts the occurrences of lab courses, and then checks that each lab course is taught in a lab room. For every lab course, if it is taught in a lab room, 100 points are added to this function's accumulative value, otherwise no points are added to the value. After the entire lab courses are examined, the accumulative function's value is divided by the number of lab courses.

classroomSize() checks that the classroom capacity is large enough for the courses that are assigned to that classroom. When a classroom can hold the students enrolled in a course, then 100 points are added to the function's accumulative value. Otherwise, no points are added. Finally, the total value is calculated, and divided by the number of courses assigned to this classroom.

twoHoursInRowClasses() checks that courses that are supposed to be taught two hours in a row, are really being taught that way, and that both hours take place in the same classroom. For every course taught in two hours in a row, if the condition is satisfied by the schedule, then the function adds 100 points to its accumulative value. Otherwise, no

points are added to this value. Finally, the total value is calculated, and divided by the number of courses taught two hours in a row.

useClassroomAvailableTime() checks that a course is assigned to an available classroom. If the course is assigned to an available room, then 100 points are added to the value. Otherwise, no points are added to the value. Finally, the total value is calculated and divided by the number of courses.

Hard constraints

There are four different hard constraints, which can have a combined maximum fitness value of 70 points. These constraints have lower priorities than the fixed constraints. To check for hard constraints the GA program used the functions *facultymemberOneClas(), teachingParticularTimes(), sameTimeBothSchedules(), and facultymeeting().*

facultymemberOneClas() checks that a faculty member teaches only one course at a time. First, it updates the schedule with faculty ids instead of courses. The function counts the number off faculty ids and for each of them, it adds 100 points to the function's accumulative value. Then it checks whether a faculty member is assigned to more than one course at a time. The function goes through all the faculty members, for each faculty member, it checks that no more than one course is assigned to him or her at one time period. If a faculty member is scheduled for more than one course at one time then 100 points are subtracted for each conflicted course. The function's total accumulative value is calculated and divided by the number of courses.

teachingParticularTimes() checks to see if a course is assigned to a faculty member at an unwanted time period. Some faculty members do not like to teach early morning or after 3PM. The schedule is first updated, and the course id is replaced with a faculty member id. Then, it checks whether a course is assigned to a faculty member before or after he or she wants to teach any course. If the faculty member is assigned a course when he or she does not want to teach it, then 0 points are added to the accumulated value. Otherwise, 100 points are added to the value. Finally, the total accumulative value is calculated and divided by the number of courses.

s*ameTimeBothSchedules()* checks that courses taught in more than three days per week, are assigned the same time period and classroom on the mwf and tth schedules. If a course is assigned to the same classroom and at the same time on both sub-schedules, then 100 points are added to thee function's accumulative value. If a course is taught at the same time or at the same classroom then 50 points are added to the accumulative value. If the courses are assigned to different classrooms at different times, then 0 points are added to the value. Finally, the total accumulative value is calculated by dividing the accumulative value by the number of courses that require more than three days.

facultymeeting() checks whether a course is scheduled during faculty meetings. The faculty of the Mathematics and Computer Science at SDSM&T meets every Tuesday

between 10 AM and 11 AM. This constraint only affects the tth_schedule. The function checks all classrooms and makes sure no courses are scheduled during department meeting. If a course violates this constraint, then no points are added to the function's accumulative value. Otherwise, 100 points will be added to the value. The final value for this function is computed after the total value is divided on the number of classrooms.

Soft constraints

Soft constraints are constraints with the lowest priorities. There are three different soft constraints, which can have a combined maximum fitness value of 30 points all together. To check for soft constraints the program used the functions *noFollowingClasses(), againstSpecificClassroom(), and needSpecialEquipment().*

noFollowingClasses() checks to see if two courses in a row are assigned to the same faculty member. Some professors prefer to teach them one after another, and others do not. First the schedule is updated with faculty ids instead of course numbers. The program then finds the professors who do wish to teach consecutive classes and puts them in a new table. The function does the same for those professors who do not choose to teach one class after the other. If the schedule determines two classes to be taught in a row by a professor that does not wish this to happen, 10 points are added to the function's accumulative value. These conditions will be counted, and if there are more than 10 conditions that occur the value becomes equal to 100. Subsequently, this total value is then subtracted from 100 and the remaining value becomes this function's fitness value.

againstSpecificClassroom() checks for faculty members that do not want to teach courses in specific classrooms. If the faculty member is assigned to a classroom he/she does not like then no points are added to the function's accumulative value, while in any other circumstance, 100 points are added to the function's value. The total value is divided by the number of courses, and becomes this function's value.

needSpecialEquipment() checks that the classroom assigned to a course contains the equipment that is required for that course. If a room does not have the required equipment then no points are added to the function's accumulative value, and if the room is properly equipped, then 100 points are added to the value. Finally, the total value is divided by the number of courses.

Evaluation

This is done by a function that takes the values returned from each of the function's constraints and calculates a number, which is the fitness value for the population. The maximum fitness value for a population after being evaluated is 200. This function used three variables, *constraintFixed, constraintHard*, and *constraintSoft*. First the function adds all the values returned from the constraint's functions. These values are then divided by the number of each type of constraint. Then, the total value for the hard

constraints is multiplied by 0.7, and the total value for the soft constraints is multiplied by 0.3. When this is done, the total values for each type of constraint are added together, resulting in the fitness value for the population.

Data Structures

To keep track of the data used in this program, the data are divided on three C++ classes and stored on three separate files. These three files stored data about the classrooms, courses, and faculty. The private members of each C++ class consist of variables used for storing data. The public members are two constructors (one empty and one with parameters), destructor, functions for get information, functions for set information, and a function to display the information.

Course C++ class contains information about course id, name, credits, whether each should be taught two hours in a row or not, maximum number of students, whose faculty member will teach the course, and the equipment that is required. Figure 2, shows the information stored about the courses. The equipments required for a course are indicated with a capital letter (B = blackboard, C = computer, P = projector, and W = whiteboard).

```
CourseID: 4324,   Name: CSC-762 Neural Networks
FacultyMemberID: 114
Credits       : 3,   Size: 40
Equipment: CBP,   Hours in a row: 0

CourseID: 5259,   Name: CSC-150L Computer Science I lab
FacultyMemberID: 120
Credits       : 0,   Size: 20
Equipment: CB,   Hours in a row: 1
```

Figure 2: Information stored about courses.

The faculty C++ class contains information about faculty id, name, which room the instructor prefers, whether he or she allows courses being taught consecutively, and time range preferred by each professor to teach the courses. If a professor allows classes taught consecutively this is indicated with the integer 1, and 0 if not. Figure 3, shows the output when viewing the faculty info.

Classroom C++ class contains information about classroom name, capacity, equipment, room type, and when the room is available. Room type could either be normal or a lab. If the room is available all the time a 0 indicates this. Figure 4 shows a subset of the output when viewing the classroom info.

```
FacultyID: 109,   Name: Jeff S. McGough
Prefer not this room                : M205
Prefer no classes before            : 11
Prefer no classes after             : 15
Allow classes taught consecutively: 1

FacultyID: 110,   Name: Manuel Penaloza
Prefer not this room                : M310
Prefer no classes before            : 10
Prefer no classes after             : 17
Allow classes taught consecutively: 1
```

Figure 3: Information stored about the faculty members.

```
Name                : M205
Roomtype: normal,   Capacity: 40,   Equipment: BW
Not available after : 0
Not available before: 0

Name                : M310
Roomtype: normal,   Capacity: 40,   Equipment: CBP
Not available after : 15
Not available before: 9
```

Figure 4: Information stored about the classrooms.

Operation

As mentioned in the implementation part, courses are most often taught on two different day sequences, Mondays, Wednesdays, and Fridays, or on Tuesdays and Thursdays. With these characteristics, we decided to make use of two arrays, one for each of these day sequences. For simplicity they are called as mwf_schedule and tth_schedule. Each field in these arrays contains either a course number or 0 if no course is assigned to that field. Each column represents a time slice of the day, and each row represent a different classroom. There are nine time slices since teaching is done between 8am and 5 pm. Figure 5 shows the data dependency between the C++ data classes and the schedules.

The mwf_schedule and tth_schedule are treated as two different populations. They have both their own fitness values. Separate genetic operations will be performed on each of them. The schedules will be evaluated several times, and there will be performed genetic operations on them. After the GA operations are completed, they are combined to set up a global schedule for each one of the classrooms.

Results

The system was tested several times with different number and types of courses, and different mutation rates. The system was usually tested with using 1000 rounds, which

after several tests seemed to be a good number. By adding more rounds, it seemed that the results didn't improve that much.

The number of courses seems not to be that important for the fitness value. The final test data consist of 35 courses, the system has also been tested with 45 courses, and there were just slightly different results. When using 45 courses, the fitness value after 1000 rounds where about 5 to 10 points lower than when was using 35 courses.

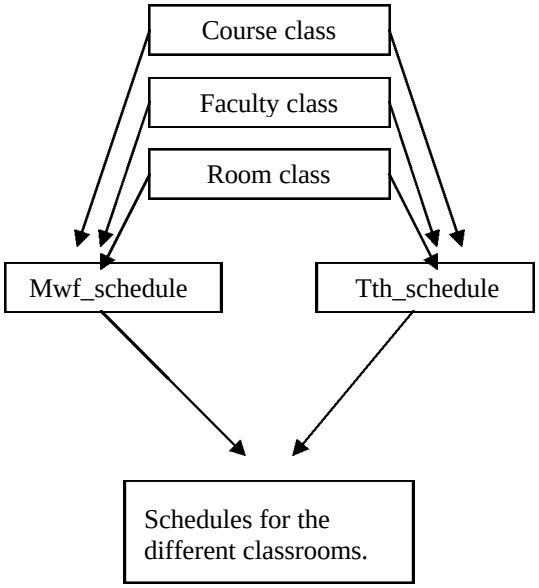


Figure 5: Data dependency between C++ data classes and schedules.

Our initial schedule was initialized by given random values to both our sub-schedules. The initial fitness value for the mwf_schedule is most common around 160, and for the tth_schedule around 135. Figure 6 shows the values for an initial tth_schedule. In this Figure, Each row is a different classroom, and the columns indicate the time of the day the course is taught. Each course is referred to with a number. Here the user had typed 1000 as the number of rounds to run the system. This value and the initial fitness values are displayed.

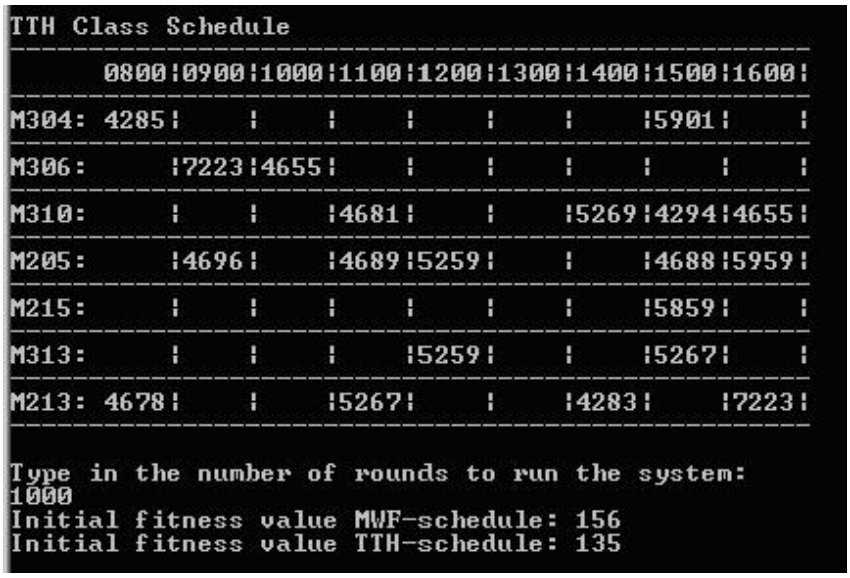


Figure 6: The initial tth_schedule.

Then, after 1000 runs, the fitness value for the mwf_schedule lies usually between 185 and 190, and for the tth_schedule, it is most common between 175 and 185. Figure 7, shows the results for the mwf_schedule after running the system 1000 rounds. It displays the schedule, its fitness value, and the number of times each of the different kinds of mutations has occurred. We got slightly better results on the mwf_schedule, because a higher number of lab courses are taught on Tuesdays and Thursdays, and these have to be assigned to special lab rooms. In addition, the faculty meeting each Tuesday influences the fitness value for the tth_schedule.

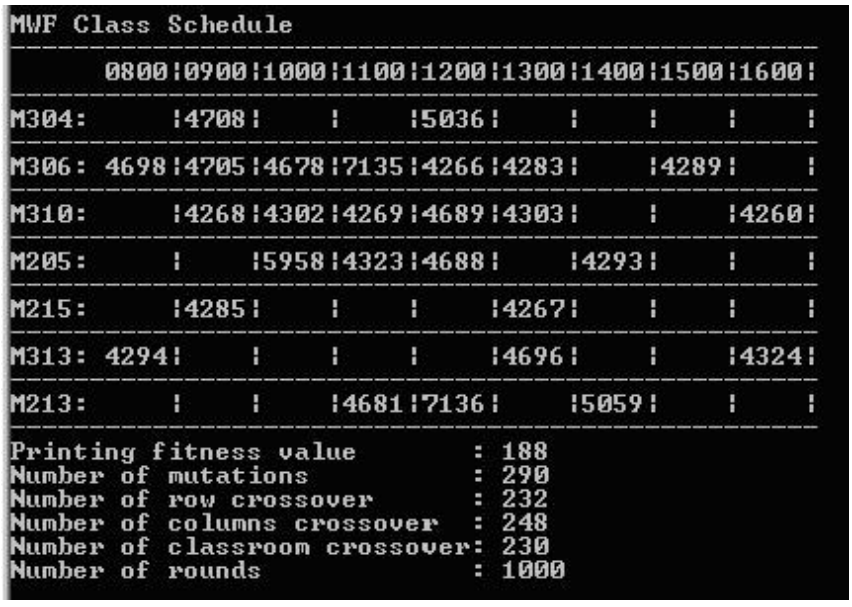
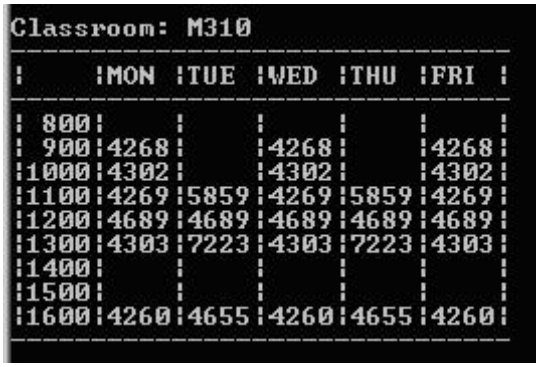


Figure 7: Results for the mwf_schedule after the system run for 1000 rounds.

After the number of runs has completed, the system displays the schedules for each classroom. Figure 8 shows the final schedule for one classroom. Then, the user will be asked the question whether he/she wants to add more runs for the system, to achieve a better result.



Classroom: M310

	MON	TUE	WED	THU	FRI
800					
900	4268		4268		4268
1000	4302		4302		4302
1100	4269	5859	4269	5859	4269
1200	4689	4689	4689	4689	4689
1300	4303	7223	4303	7223	4303
1400					
1500					
1600	4260	4655	4260	4655	4260

Figure 8: The schedule for a single classroom is displayed after the program is run.

The system's performance is low because of the number of testing performed for the constraints. On an 800 MHz AMD Duron Processor with 256 MHz RAM, it takes about 2.5 minutes to run the program with 1000 rounds.

A system like this would probably works better if the courses do not have to be taught at the same time in the same classroom several times a week. By removing the constraint *sameTimeBothSchedules(),* both schedules will get a higher initial value, approximately 175 for mwf_schedule, and 150 for tth_schedule. Then after running the system 1000 runs the value will be close to 200 hundred for both schedules. Applying another 1000 runs, we actually got the maximum fitness value of 200 for the mwf_schedule.

Conclusion

By using genetic algorithms we got pretty good results for our test data. It seems like the system is able to find a good schedule for each classroom. Different test data will affect the results. Our tests of the system showed that with more courses and higher requirements for these courses produced fair results. By increasing the number of classrooms, decreasing the number of courses, and relaxing the constraints we were able to produce good results. For instance, by removing the constraint *sameTimeBothSchedules()* the results improved and the fitness value got very close to its maximum fitness value with our test data.

The frequency of mutations is also important. By increasing the rate of mutations (where we change one randomly selected course in the array with another randomly selected course) from 3% to 30%, we got approximately 10% percent better fitness value after 1000 rounds.

References

1. Ahuja, Rabindra, and Orlin, James (2002). Very Large-Scale Neighborhood Search in Airline Fleet Scheduling. SIAM News, November 2002.
2. Goldberg, David (1989). *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley Longman, Inc.
3. Arjan Tijms, drs. And Johan van der Meulen (2002). Genetic Algorithm – Assignment 4b, University Leiden.
4. Levi, Delon (2000). HereBoy: A Fast Evolutionary Algorithm. The second NASA/DoD Workshop on Evolvable Hardware (EH'00). July 13-15, 2000. Palo Alto California.