**Introduction**

Making a class schedule is one of those NP hard problems. The problem can be solved using a heuristic search algorithm to find the optimal solution, but it only works for simple cases. For more complex inputs and requirements, finding a considerably good solution can take a while, or it may be impossible. This is where genetic algorithms come in to the game. In this article, I assume that you are familiar with the basic concepts of genetic algorithms, and I won't describe them in detail because it has been done so many times before.

**Background**

When you make a class schedule, you must take into consideration many requirements (number of professors, students, classes and classrooms, size of classroom, laboratory equipment in classroom, and many others). These requirements can be divided into several groups by their importance. Hard requirements (if you break one of these, then the schedule is infeasible):

* A class can be placed only in a spare classroom.
* No professor or student group can have more then one class at a time.
* A classroom must have enough seats to accommodate all students.
* To place a class in a classroom, the classroom must have laboratory equipment (computers, in our case) if the class requires it.

Some soft requirements (can be broken, but the schedule is still feasible):

* Preferred time of class by professors.
* Preferred classroom by professors.
* Distribution (in time or space) of classes for student groups or professors.

Hard and soft requirements, of course, depend on the situation. In this example, only hard requirements are implemented. Let's start by explaining the objects which makes a class schedule.

**Objects of Class Schedule**

**Professor**

The Professor class has an ID and the name of the professor. It also contains a list of classes that a professor teaches.

**Students Group**

The StudentsGroup class has an ID and the name of the student group, as well as the number of students (size of group). It also contains a list of classes that the group attends.

**Classroom**

The Room class has an ID and the name of the classroom, as well as the number of seats and information about equipment (computers). If the classroom has computers, it is expected that there is a computer for each seat. IDs are generated internally and automatically.

**Course**

The Course class has an ID and the name of the course.

**Class**

CourseClass holds a reference to the course to which the class belongs, a reference to the professor who teaches, and a list of student groups that attend the class. It also stores how many seats (sum of student groups' sizes) are needed in the classroom, if the class requires computers in the classroom, and the duration of the class (in hours).

**Chromosome**

The first thing we should consider when we deal with a genetic algorithm is how to represent our solution in such a way that it is feasible for genetic operations such as crossover and mutation. Also, we should know how to specify how good our solution is. In other words, we should be able to calculate the fitness value of our solution.

**Representation**

How can we represent the chromosome for a class schedule? Well, we need a slot (time-space slot) for each hour (we assume that time is in one hour granules), for every room, every day. Also, we assume that classes cannot begin before 9am, and should finish before or at 9pm (12 hours total), and working days are from Monday to Friday (5 days total). We can use an std::vector with a size 12\*5\*number\_of\_rooms. The slot should be an std::list because during the execution of our algorithm, we allow multiple classes during the same time-space slot. There is an additional hash map which is used to obtain the first time-space slot at which a class begins (its position in vector) from the address of the class' object. Each hour of a class has a separate entry in the vector, but there is only one entry per class in the hash map. For instance, if a class starts at 1pm and lasts for three hours, it has entries in the 1pm, 2pm, and 3pm slots.

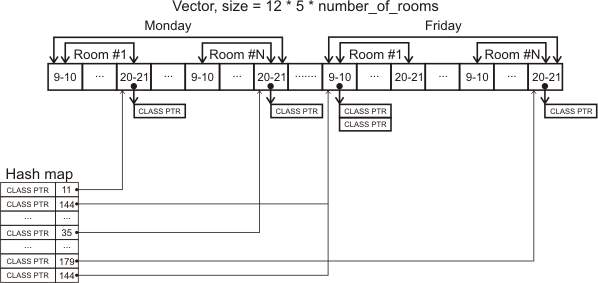


Figure 1 - Chromosome Representation

Chromosomes are represented by the Schedule class, and it stores the representation of a class schedule in these two attributes:

http://www.codeproject.com/images/minus.gifCollapse |

// Time-space slots, one entry represent one hour in one classroom

vector<list<CourseClass\*>> \_slots;

// Class table for chromosome

// Used to determine first time-space slot used by class

hash\_map<CourseClass\*, int> \_classes;

Additionally, the chromosome should store its fitness value and the parameters which are used by genetic operations.

The fitness value is stored here:

Collapse



// Fitness value of chromosome

float \_fitness;

// Flags of class requiroments satisfaction

vector<bool> \_criteria;

Chromosome parameters:

http://www.codeproject.com/images/minus.gifCollapse

// Number of crossover points of parent's class tables

int \_numberOfCrossoverPoints;

// Number of classes that is moved randomly by single mutation operation

int \_mutationSize;

// Probability that crossover will occure

int \_crossoverProbability;

// Probability that mutation will occure

int \_mutationProbability;

**Fitness**

Now we need to assign a fitness value to the chromosome. As I previously said, only hard requirements are used to calculate the fitness of a class schedule. This is how we do it:

* Each class can have 0 to 5 points.
* If a class uses a spare classroom, we increment its score.
* If a class requires computers and it is located in the classroom with them, or it doesn't require them, we increment the score of the class.
* If a class is located in a classroom with enough available seats, guess what, we increment its score.
* If a professor has no other classes at the time, we increment the class's score once again.
* The last thing that we check is if any of the student groups that attend the class has any other class at the same time, and if they don't, we increment the score of the class.
* If a class breaks a rule at any time-space slot that it occupies, its score is not incremented for that rule.
* The total score of a class schedule is the sum of points of all classes.
* The fitness value is calculated as schedule\_score/maximum\_score, and maximum\_score is number\_of\_classes\*5.

The fitness values are represented by single-precision floating point numbers (float) in the range 0 to 1.

**Crossover**

A crossover operation combines data in the hash maps of two parents, and then it creates a vector of slots according to the content of the new hash map. A crossover 'splits' hash maps of both parents in parts of random size. The number of parts is defined by the number of crossover points (plus one) in the chromosome's parameters. Then, it alternately copies parts form parents to the new chromosome, and forms a new vector of slots.

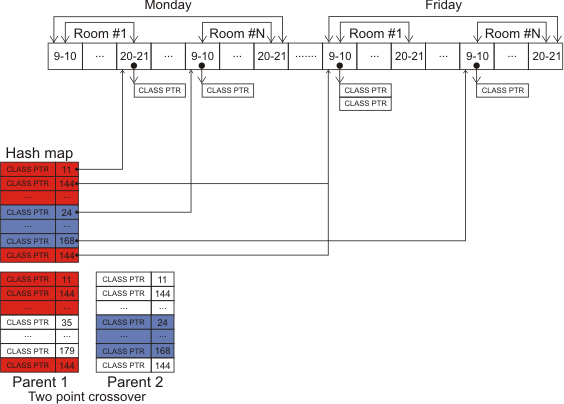


Figure 2 - Crossover operation

http://www.codeproject.com/images/minus.gifCollapse

// Performes crossover operation using to chromosomes

// and returns pointer to offspring

Schedule\* Crossover(const Schedule& parent2) const;

**Mutation**

A mutation operation is very simple. It just takes a class randomly and moves it to another randomly chosen slot. The nmber of classes which are going to be moved in a single operation is defined by the mutation size in the chromosome's parameters.

http://www.codeproject.com/images/minus.gifCollapse

// Performs mutation on chromosome

void Mutation();

**Algorithm**

The genetic algorithm is fairly simple. For each generation, it performs two basic operations:

1. Randomly selects N pairs of parents from the current population and produces N new chromosomes by performing a crossover operation on the pair of parents.
2. Randomly selects N chromosomes from the current population and replaces them with new ones. The algorithm doesn't select chromosomes for replacement if it is among the best chromosomes in the population.

And, these two operations are repeated until the best chromosome reaches a fitness value equal to 1 (meaning that all classes in the schedule meet the requirement). As mentioned before, the genetic algorithm keeps track of the M best chromosomes in the population, and guarantees that they are not going to be replaced while they are among the best chromosomes.

http://www.codeproject.com/images/minus.gifCollapse

// Genetic algorithm

class Algorithm

{

private:

// Population of chromosomes

vector<Schedule\*> \_chromosomes;

// Inidicates wheahter chromosome belongs to

// best chromosome group

vector<bool> \_bestFlags;

// Indices of best chromosomes

vector<int> \_bestChromosomes;

// Number of best chromosomes currently saved in

// best chromosome group

int \_currentBestSize;

// Number of chromosomes which are replaced in

// each generation by offspring

int \_replaceByGeneration;

// Pointer to algorithm observer

ScheduleObserver\* \_observer;

// Prototype of chromosomes in population

Schedule\* \_prototype;

// Current generation

int \_currentGeneration;

// State of execution of algorithm

AlgorithmState \_state;

// Synchronization of algorithm's state

CCriticalSection \_stateSect;

// Pointer to global instance of algorithm

static Algorithm\* \_instance;

// Synchronization of creation and destruction

// of global instance

static CCriticalSection \_instanceSect;

public:

// Returns reference to global instance of algorithm

static Algorithm& GetInstance();

// Frees memory used by gloval instance

static void FreeInstance();

// Initializes genetic algorithm

Algorithm(int numberOfChromosomes,

int replaceByGeneration,

int trackBest,

Schedule\* prototype,

ScheduleObserver\* observer);

// Frees used resources

~Algorithm();

// Starts and executes algorithm

void Start();

// Stops execution of algoruthm

void Stop();

// Returns pointer to best chromosomes in population

Schedule\* GetBestChromosome() const;

// Returns current generation

inline int GetCurrentGeneration() const { return \_currentGeneration; }

// Returns pointe to algorithm's observer

inline ScheduleObserver\* GetObserver() const { return \_observer; }

private:

// Tries to add chromosomes in best chromosome group

void AddToBest(int chromosomeIndex);

// Returns TRUE if chromosome belongs to best chromosome group

bool IsInBest(int chromosomeIndex);

// Clears best chromosome group

void ClearBest();

};

**Observing**

The ScheduleObserver class handles the events that are triggered by the genetic algorithm. This class sends messages to the view window of the application. Also, you can block the caller's thread until the execution of the algorithm is not finished or stopped, by calling the WaitEvent() method.

http://www.codeproject.com/images/minus.gifCollapse

// Handles event that is raised

// when algorithm finds new best chromosome

void NewBestChromosome(const Schedule& newChromosome);

// Handles event that is raised when state

// of execution of algorithm is changed

void EvolutionStateChanged(AlgorithmState newState);

// Block caller's thread until algorithm finishes execution

inline void WaitEvent() //...

If you plan to change the NewBestChromosome method, keep in mind that if you want to keep the best chromosome to display it, you must make a hard copy (by using the MakeCopy method of the Schedule class), because the algorithm can delete that chromosome in the next generation.

**Configuration**

**Configuration File**

Types of objects:

1. professor (#prof tag) - describes a professor.
2. course (#course tag) - describes a course.
3. room (#room tag) - describes a room.
4. group (#group tag) - describes a students group.
5. course class (#class tag) - describes a class, and binds the professor, course, and students group.

Each object begins with its tag and finishes with the #end tag, all tags must be in separate lines. In the body of an object, each line contains only one key and value pair (attribute) separated by an = character. Each attribute should be specified just one time, except for the group attribute in the #group object which can have multiple group attributes. Tag and key names are case sensitive. Here is a list of the objects' attributes:

1. #prof
   * id (number, required) - ID of the professor.
   * name (string, required) - name of the professor.
2. #course
   * id (number, required) - ID of the course.
   * name (string, required) - name of the course.
3. #room
   * name (string, required) - name of the room.
   * size (number, required) - number of seats in the room.
   * lab (boolean, optional) - indicates if the room is a lab (has computers); if not specified, the default value is false.
4. #group
   * id (number, required) - ID of the students group.
   * name (string, required) - name of the students group.
   * size (number, required) - number of students in the group.
5. #class
   * professor (number, required) - ID of a professor; binds a professor to a class.
   * course (number, required) - ID of a course; binds a course to a class.
   * group (number, required) - ID of a students group; binds the students group to a class; each class can be bound to multiple students groups.
   * duration (number, optional) - duration of class (in hours); if not specified, the default value is 1.
   * lab (boolean, optional) - if the class requires computers in a room; if not specified, the default value is false.

Note that the professor, students group, and course objects must be defined before they are bound to a course class object.

**Example of a Configuration File**

http://www.codeproject.com/images/minus.gifCollapse

#prof

id = 1

name = John Smith

#end

#course

id = 1

name = Introduction to Programming

#end

#room

name = R1

lab = true

size = 24

#end

#group

id = 1

name = 1O1

size = 19

#end

#class

professor = 1

course = 1

duration = 2

group = 1

group = 2

#end

#class

professor = 1

course = 1

duration = 3

group = 1

lab = true

#end

#class

professor = 1

course = 1

duration = 3

group = 2

lab = true

#end

**Parsing a Configuration**

Parsing of a configuration file is done by the Configuration class. The ParseFile method opens and parses a configuration file. It searches for object tags and calls the appropriate method for a parsing object. The ParseFile method also clears a previously parsed object.

http://www.codeproject.com/images/minus.gifCollapse

public:

void ParseFile(char\* fileName);

private:

Professor\* ParseProfessor(ifstream& file);

StudentsGroup\* ParseStudentsGroup(ifstream& file);

Course\* ParseCourse(ifstream& file);

Room\* ParseRoom(ifstream& file);

CourseClass\* ParseCourseClass(ifstream& file);

To parse a file:

http://www.codeproject.com/images/minus.gifCollapse

Configuration::GetInstance().ParseFile( "GaSchedule.cfg" );

Parsed objects are kept in a hash map except for course classes, so they can be accessed easily and fast.

http://www.codeproject.com/images/minus.gifCollapse

private:

hash\_map<int, Professor\*> \_professors;

hash\_map<int, StudentsGroup\*> \_studentGroups;

hash\_map<int, Course\*> \_courses;

hash\_map<int, Room\*> \_rooms;

list<CourseClass\*> \_courseClasses;

The Configuration class also contains the methods for retrieving the parsed information and objects.

http://www.codeproject.com/images/minus.gifCollapse

public:

inline Professor\* GetProfessorById(int id) //...

inline int GetNumberOfProfessors() const //...

inline StudentsGroup\* GetStudentsGroupById(int id) //...

inline int GetNumberOfStudentGroups() const //...

inline Course\* GetCourseById(int id) //...

inline int GetNumberOfCourses() const //...

inline Room\* GetRoomById(int id) //...

inline int GetNumberOfRooms() const //...

inline const list<CourseClass\*>& GetCourseClasses() const //...

inline int GetNumberOfCourseClasses() const //...