

## M450 EMA

### Project Title

Multi-year Module Scheduler for OU students

### Problem Description

#### *Nature and context of the problem*

##### **What is the problem?**

Many OU students know which modules they wish to study, but don't necessarily know the best time to take each one in order to complete their studies efficiently. OU modules are not all interchangeable. They do not all start at the same time of year, and some last longer than others and/or require more hours of study per month. Additionally, some modules require other modules to be taken first. Making a long-term plan for when to take each module can therefore be quite complicated and arduous.

##### **Why is it considered a problem?**

Without a long-term schedule, a student may take longer than necessary to reach their final goal, or end up taking on periods of heavier study than they wished in order to avoid this delay. For instance, a module that runs from February to October can be followed directly by another module that runs from October to June, taking 17 months to complete both, with just a week or two of overlap. The same modules taken in the reverse order, however, would take 2 years to complete, because of the gap between them – and whether that gap can be productively filled will depend on which other modules the student intends to study. Clearly, for any set of modules to be taken, some orders for taking them are better than others, making this an optimisation problem.

##### **What will be the benefits of solving it?**

Students will be able to plan their future studies efficiently, and achieve their qualification as early as possible (or alternatively, at their preferred pace).

##### **What might the solution look like?**

A schedule should be a simple calendar-like diagram, with an x-axis showing time several years into the future, and modules plotted as rectangles against the months during which they are recommended to be studied. A solution should include several schedules for a user to choose from, reflecting different “modes” of adherence to the limit on study hours.

##### **What, specifically, was it the aim to deliver?**

A software application that allows a user to choose modules they wish to study from a (limited) list, the maximum number of hours per month they would generally like to spend

studying, and the extent to which they are prepared to occasionally exceed this limit in order to complete their studies sooner. A handful of easy-to-read schedules are delivered in both graphic and written form, each showing a suggested date to start each module.

### *Proposed Solution*

The issue of helping OU students schedule their studies is a broad one, and my initial efforts at scoping included aspects such as importing module details from the OU website, mapping modules against qualifications, and a full GUI. Over time, all the above were removed from scope, as the risks they posed to the project became apparent. Along with aspects that I had ruled out of scope from the beginning, I therefore ultimately arrived at the following scope decisions:

#### **Out of scope: Timetabling study at the micro level**

This system only provides start dates for each module, ie it suggests which presentation of a module to take. It does not help users to plan how they should distribute their available hours of study time during a module.

#### **Out of scope: Dealing with OU qualifications**

This system does not help users determine which modules should be studied in order to gain a certain qualification. Users are responsible for specifying the modules they wish to study. The system does not verify whether this combination is valid for a qualification. Given these decisions, it also follows that the system does not concern itself with whether a user has already passed some modules required for a qualification; the user should restrict their list of modules to those still to be taken.

#### **Out of scope: Future changes in modules being offered**

The system assumes that all modules and qualifications being offered in 2011 will continue to be offered indefinitely, at the same times each year. Although this is a serious limitation to the system's usefulness, the decision was made on the basis that attempting to account for future changes – which are not well known even by the OU – would pose an unacceptable risk to the project's success, forcing a reliance on information that would be difficult or impossible to obtain.

#### **Out of scope: Module costs**

Module costs are an important factor in many students' studies, and minimizing cost is an important objective in many optimisation problems. However, the possible change in module cost over the duration of a multi-year schedule cannot be predicted with any accuracy, especially as policies of government subsidy change. In any case, the order in which modules are taken is unlikely to impact total cost greatly. In addition, some students are eligible for financial assistance with their studies. For all these reasons, cost is out of scope for the system.

#### **Out of scope: GUI**

Although I had originally wanted to provide a full GUI, the new skills I would have needed to acquire and the time it would have taken to research and test a proper interface would have

damaged my ability to focus on the core project. I therefore decided to make do with a rudimentary Swing based interface (without any kind of input validation) to collect the student preferences.

### **Out of scope: Importing live online module data**

A fully functioning version of this system would ideally hold data for each module offered by the OU. Data for each module is freely available on [data.open.ac.uk](http://data.open.ac.uk) under a Creative Commons Attribution 3.0 Unported License, and I was originally hoping to use this data to import a complete module database into my system. However, I later removed this from scope for two reasons. Firstly, I had no knowledge of the SPARQL protocol through which this data was available, and not enough time to learn it without jeopardizing other parts of my project. Secondly, although information was available through SPARQL for the module's code and level, it was not available in any easily importable form for the module's start month(s) and duration. These pieces of information are crucial for the system to function, and since they can easily be found by eye on each module's webpage, I chose instead to limit my system to a subset of all the available modules, input by hand. I used 25 modules from the Mathematics, Computing and Psychology departments of the OU, of differing lengths, levels, and start and end dates. Not relying on online data removes the risk of future system failure if the online data should cease to be available or accurate.

### **Out of scope: OU pre-requisites and within-level module order recommendations**

While the system does take the level of each module (1, 2 or 3) into account when choosing a suitable order (see "In scope" below), it does not accommodate any OU rules or advice about module order beyond this. For instance, although both MU123 (Discovering Mathematics) and MST121 (Using Mathematics) are Level 1 courses, the OU recommends studying MU123 before MST121. Some modules have pre-requisites, for instance you are advised not to attempt M363 (Software Engineering with Objects) without a pass in M256 (Software Development with Java). Given that these rules are not well-defined or centrally available, but require close reading of every module webpage, I decided it would be too difficult to source all these factors and too complex to incorporate them into the system.

### **In scope:**

Having ruled the above aspects out of scope, I was able to focus in on the core task for my system, defined as follows:

*Given*

- a set of modules
- a desired monthly workload limit
- a tolerance for exceeding that workload limit

*the system should recommend several schedules of study optimised against different weightings of*

- shortness
- adherence to workload limit
- level order (ie taking lower level modules before higher ones).

### *Analysis of likely impact*

While the system will benefit students by allowing them to more easily plan their entire course of study, it would have to be carefully managed and maintained to ensure the recommendations it provides are accurate and appropriate, and remain so over time. The OU is constantly evolving: requirements for qualifications change, modules are retired or moved to a different time of year, and new modules are introduced. This creates an ethical concern, in that users might potentially base their academic plans on a schedule produced by the system, which may not in fact be the best study plan for them, or may not lead to the qualification they believe it will, or may not even be possible to complete. This is an inherent limitation of any long-term planning tool in a fast-changing environment, and therefore it will be important to ensure that people are aware of this limitation, and understand that the system's output is only a tool to help them make their study decisions. This would perhaps require a disclaimer if the system were to be made publicly available.

As a standalone program, this system is not likely to have any impact on how broader OU systems operate, but if the tool were to be adopted or endorsed by the OU, then the need to ensure on-going accuracy would become even more important. It would probably be necessary for users to register their details (which creates data protection issues to deal with) so that alerts could be sent to them if changes were being introduced that could affect their plans. A system for sending out such alerts would have to be put into place. For maximum benefits, the OU would need to integrate the system with a student's record, so that as modules are successfully completed they are removed from the schedule, and if they are failed, an updated schedule can be calculated. Also, the OU would ideally add new fields to its open source data platform for start dates, duration and pre-requisites, and populate these fields, so that data for all modules can be pulled into the system on an on-going basis. In the case of pre-requisites, this might involve quite significant work, as it would require centralizing and more clearly defining information which currently appears to be spread across departments and not well defined.

### *Account of Related Literature*

My study of the literature was focused around two tasks:

1. Comparing the merits of different approaches to solving this kind of problem (exhaustive and heuristic search, constraint satisfaction techniques, rule-based expert system, genetic algorithm)
2. Exploring specific methods used by others for solving and optimizing similar problems with a Genetic Algorithm (my chosen approach)

### *Comparing the merits of different approaches to solving this kind of problem*

*My key sources for this task was Fang (1994), a 1994 PhD thesis from a student at the AIAI at Edinburgh University, which I discovered through a reference in Nammuni (2002). It makes good learning material because, unlike many journal papers, it is comprehensive,*

*explanatory and includes data and detailed results which help me understand how the work was actually implemented. Its age means it provides a solid overview of the fundamentals of the field rather than the more complex techniques that most newer sources focus on.*

Most scheduling problems are NP-complete, meaning that they cannot be solved by any known polynomial time algorithms (Carey and Johnson, 1979). It is therefore necessary to turn to other techniques to solve them.

### **Exhaustive and Heuristic Search**

As many authors have noted (eg Fang, 1994 and Cartwright, 1994), brute force or undirected search methods are not feasible for any but the smallest scheduling problems, because the solution space of NP-complete problems is far too large to examine candidates exhaustively. Cartwright (1994) gives the Travelling Salesman Problem as an example of how even the most conceptually simple scheduling problem quickly becomes difficult to solve, because the time taken to exhaustively examine all the candidate solutions rises exponentially as the number of cities increases. He notes that a problem with 4 cities has just 24 solutions, while a problem with 10 cities has  $3 \times 10^6$  solutions, and a problem with 20 cities has  $2 \times 10^{18}$  solutions. It is easy to see how this issue makes exhaustive search infeasible for my project at any useful scale. Since each module has a minimum of 8 possible starting dates, a problem with just 4 modules has only  $8^4 = 4096$  solutions. But the problem of scheduling ten modules has  $8^{10} =$  more than a billion solutions. While heuristic search can reduce the scale of the problem by directing us to promising parts of the search space, scheduling problems are often complicated by the details of the specific scheduling task. Finding good heuristics is hard as they vary for each problem (Fang, 1994).

### **Constraint Satisfaction Techniques**

A CSP (Constraint Satisfaction Problem) is a set of variables each of which has a discrete and finite set of possible values, and a set of constraints between these variables. A solution is a set of variable-value assignments that satisfy all the constraints (Fang 1994). Constraint Satisfaction approaches have to be tailor-made for each problem and fundamentally altered for any change in constraints, so are brittle. Constraint Satisfaction approaches also cannot deal well with soft constraints. They find any solution that does not violate the set of constraints, without caring / optimizing for how well it meets soft constraints, ie. preferences (Fang 1994). In terms of my problem, this makes CSP an unsatisfactory choice, as my problem is focused almost entirely on soft constraints. Most scheduling problems have hard constraints that must not be violated, such as that two jobs cannot be scheduled at the same time, or that certain individual jobs must be completed by a certain date. In my problem, though, a student may take several modules simultaneously, and there is no date by which any module must be finished. The only hard constraint is that a module can only be scheduled for a start date on which it is actually presented, and this can be dealt with in the representation itself, so that the search process need not concern itself with this issue. All other constraints in my problem are soft: we would *prefer* to stick to the workload limit, but we may be prepared to exceed this limit by a small (or not so small) amount in return for an earlier finish date. We would *prefer* to study modules in level order, but we will consider otherwise good solutions that deviate from this somewhat. A CSP approach cannot accommodate these trade-offs.

### **Rule-Based Expert Systems**

Rule-Based Expert systems can represent both hard and soft constraints. However, they require known heuristics for the problem in order for the rules to be developed. They are rarely used for optimisation problems, being better suited to “what’s the answer” type problems (Fang 1994).

### **Genetic Algorithms**

Genetic Algorithms are a powerful and much-used tool for tackling all kinds of scheduling problems (see Gröbner and Wilke 2001, Nammuni et al 2002, Murakami et al 2010). GAs have several properties which make them suitable for scheduling problems in general, and my problem in particular. Firstly, by searching in parallel from a population of points, rather than a single point, they are able to cover much more of the search space than other search methods (Fang 1994), which makes them effective for problems with large spaces such as mine. Secondly, GAs are excellent at dealing with soft constraints, because they treat *all* constraints as soft constraints. (GAs handle hard constraints simply by penalizing them extremely heavily so that non-viable solutions will not survive.) For my problem, GAs should therefore be able to find good schedules that come as close as possible to meeting student’s preferences. Thirdly, GAs are more flexible / less brittle than most other approaches, in that only the fitness function, rather than the whole representation, needs to be altered to implement a change in constraints (Fang 1994). In my case, this allows a student to ask for a number of schedules optimised against different preferences, and with a simple adjustment of fitness function weightings the algorithm will be able to treat them as identical problems.

### *Exploring specific methods used for solving and optimizing similar problems with a Genetic Algorithm (my chosen approach)*

*My key sources for this task were Mitchell (1994), Gröbner and Wilke (2001), and Nammuni, Levine and Kingston (2002). The first is an undergraduate text book which covers details of implementing a GA, and the second and third are somewhat more recent papers which describe advances in “hybrid genetic algorithms”: those that incorporate local / problem-specific knowledge to enhance a GA’s performance.*

### **Encodings, genetic operators and parameters**

Mitchell (1996) discusses in detail how a Genetic Algorithm can be implemented, including aspects such as encodings, different kinds of selection (roulette, tournament, rank, Boltzmann), different genetic operators and their appropriate parameters (one versus two point crossover, mutation, swap) and other techniques such as elitism. Details of how I have used this text appear throughout the *Account of Outcome* section.

### **Repair Operators**

Gröbner and Wilke (2001) describe a hybrid genetic algorithm which incorporates some local knowledge in the form of “repair operators” which are applied after selection, crossover and mutation, but before fitness calculation. Gröbner and Wilke’s main reason for using such

operators is to make non-viable solutions into viable ones, for instance, they use a ‘cancel’ operator which, if an employee has been assigned to multiple shifts on one day (which is illegal for their problem) cancels all but one of them. However they do also use repair operators for optimisation, such as their ‘interchange’ operator, which swaps the assignments of two employees to achieve “a more homogenous sequence of shifts for the employees”. Importantly, improvements made by the repair operators are coded into the genome so that they will persist in future generations, establishing what they call “a kind of Lamarckian evolution.”

Gröbner and Wilke find that repair operators can increase the quality and speed of the algorithm, and I used their ideas to implement a “Gap Eliminator” repair operator to try to achieve the same for my problem. The authors point out that repair operators render the search local, and therefore need to be introduced “gradually” to avoid restricting the search space too early (although I did not find this to be a problem in my case). One downside, though, is that repair operators are unique to each problem and would have to be adapted for changes in criteria. This detracts from one of the key advantages of GAs which is their flexibility. The authors state that whether the trade-off is worth it depends upon the extent of the improvements they bring.

### **Combining a GA with an ontology**

Nammuni, Levine and Kingston (2002) uses an ontology to deal with partial matching – in this case, finding tutors whose skills, while perhaps not optimal for the class to be taught, are close enough to make them acceptable matches. In the paper, two skills (such as “C++” and “Java”) which come from the same parent node (“General Programming”) are assigned a weight (0.25) that is used in the fitness function to determine penalty points for matchings. So, if a Java class is assigned to a tutor who does not have Java skills, the normal penalty of, say, 20 points, is reduced to  $0.25 \times 20 = 5$  points. More distantly related skills (such as “C++” and “Lisp”, which share only a grandparent node of “Programming”) get higher weights (in this case 0.5), so the assignment of a Java class to a tutor skilled only in Lisp would get  $0.5 \times 20 = 10$  penalty points: more than the tutor with java skills, but still only half the penalty points of a tutor without any programming skills.

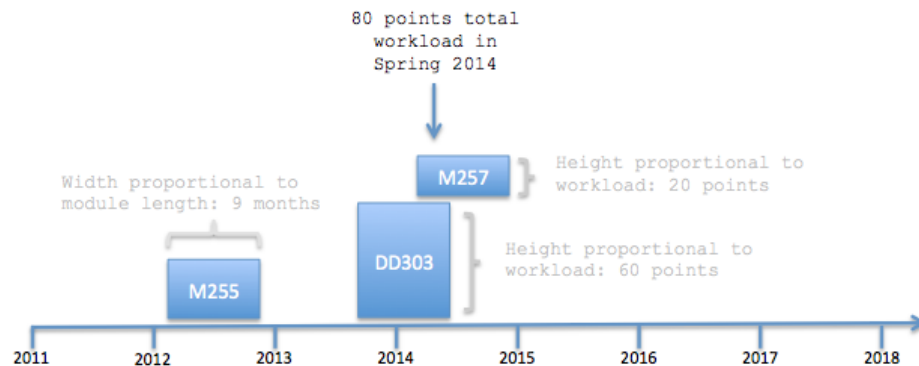
I used this paper to explore the idea of using an ontology combined with a GA as a way to handle module order / prerequisites in my system. It struck me that Nammuni’s tutors (who are physically different people with different skillsets) are equivalent to my single student at different stages of his or her learning (who are conceptually “different” people with different skillsets). I thought it might be possible to arrive at good solutions using a fitness function that judges a schedule chronologically starting at time  $t=0$ , assesses the appropriateness of the next scheduled module against the student’s “current” skillset, updates the “current” skillset after each module on the assumption of a pass, and then assesses the appropriateness of the next module against this new updated skillset. I even envisaged a system that uses its knowledge of the student’s skillset at any one time, combined with past performance, to predict whether modules will or will not be passed if taken at a certain time, and to schedule accordingly. In the end, I did not pursue the ontology idea, as the necessary data to formulate an ontology was not available, and it was clear that attempting something this complex would overstretch my resources.

## Account of Outcome: Analysis

I began by asking two questions. Firstly, what are the characteristics of a good solution, and secondly, how should a solution be encoded for use with a GA?

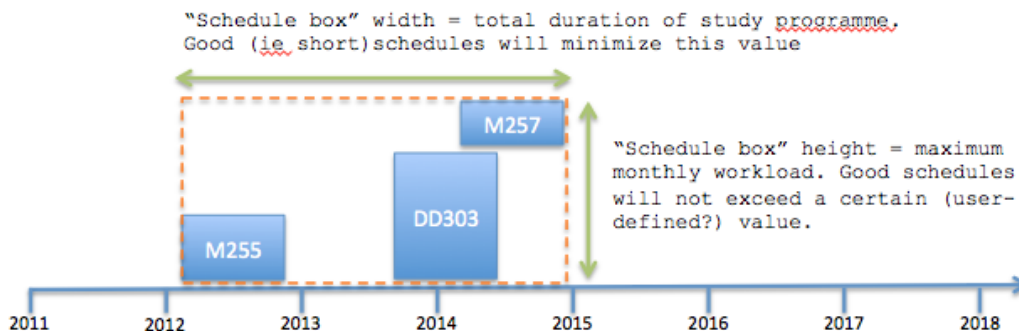
### *What are the characteristics of a good solution?*

In figures 1 and 2, I sketched out how a solution might be visualised, and what a good solution would look like. Rectangles representing modules are plotted on an x-axis representing time. The rectangles have a width proportional to the module's duration (eg 9 months) and a height proportional to the module's workload (eg 20 points). If the schedule has any modules which are being studied simultaneously for one or more months, the corresponding rectangles are stacked on top of each other.



**Figure 1: Schematic for a candidate solution / schedule**

Clearly, an ideal schedule would be close to rectangular shaped, in that modules would be fairly evenly distributed within it. For most users, the “best” rectangle would probably be one that achieves minimum width (total duration of study) without exceeding a certain limit on height (ie on total workload at any one time). By definition, this rectangle would also be the most “tightly packed” (see figure 2, below).



**Figure 2: Schematic demonstrating that a good solution will take the form of a tightly packed rectangle**



### *How should a solution be encoded for use with a genetic algorithm?*

Mitchell (1996) states that there are no clear guidelines for which kind of encoding works best in which situation, and that performance seems to vary depending on the problem being modelled and the details of the GA being used. The first GAs used bit strings to encode possible solutions, and according to Schema Theory, binary encoding was an important part of how GAs worked (M366, Block 5). However, subsequent studies using non-binary encodings seem to perform equally well (Mitchell 1996) and are simpler to implement. As my project progressed, I realised the longer strings of the binary coding were also working against my attempts to reduce the algorithm's completion time. If I were starting this project again with the knowledge I have now, I would use integer coding as discussed in Cartwright (1994). However, at the beginning of this project, and with experience from M366 of working with GAs that use bit strings, I saw no reason to question the use of binary encoding. I focused therefore simply on the best way to use a bit string to encode the problem for a use with a GA.

After initial consideration of a system that had each position represent a starting date and each value representing a module (see Appendix 2) I decided on the opposite approach. In my chosen encoding, the position of each gene represents a Module to be taken, and the value of each gene represent a possible starting date. The advantage of this is that modules will never be duplicated as each is only represented in one position on the chromosome. The disadvantage is that illegal starting dates may be represented. My initial fix for this was simply to treat an illegal starting date as if it represented the first subsequent legal starting date. (Later, when it became clear this was computationally costly, I introduced a repair operator to re-encode the first subsequent legal starting date into the bit string itself – see *Account of Outcome: Evaluation* section).

The next question was how long the bit string should be, which depends on both how many modules are to be included in the solution, and how many bits are needed to represent the starting date (month and year) for each module.

#### **Representing the month**

The OU's website indicates that the vast majority of modules begin in February or October, with a few offered in May and November. As I could not find any modules with starting dates outside of these four months, I decided I could afford to use just 2 bits to encode a module's starting month (00 for February, 01 for May, 10 for October, 11 for November). My reasoning was twofold: firstly, keeping the total length of the bit string down reduces computation time (Mitchell 1996), and secondly, allowing only four months to be represented minimizes the chances that mutation and crossover will introduce non-legal starting months (although given that most modules have only one starting month, non-legal months will still occur 3 times more often than legal ones, hence my eventual decision to add a repair operator).

#### **Representing the year**

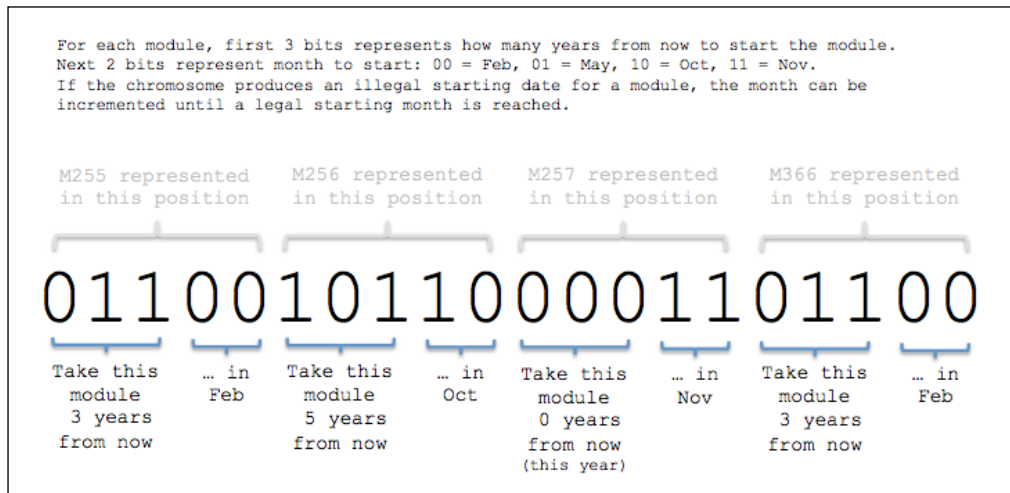
In an attempt both to keep bit strings short and to keep schedules manageable, I decided to use 3 bits, this providing a time horizon of 8 years into the future. Most OU students

complete their studies in 6 years (part-time study) or 3–4 years (full-time study) so 8 years seemed sufficient to accommodate most users (Open University, 2011). The system must be given a “Year Zero” when studies can begin (which I have hardcoded as 2012, but which could be set dynamically) and it interprets the bits as representing the number of years after this Year Zero (0-8 inclusive) the module should be started.

### Number of modules to be represented

I wanted the user to be able to choose the number of modules to include in the schedule. This means that the total length of the bit string varies from run to run. For instance, a schedule of 10 modules is represented by a bit string of length 50 (three bits for each module’s starting year plus two bits for its starting month).

Figure 3 shows an example of how four modules are represented in a bit string under my chosen encoding. Note that modules are represented in alphabetical order.



*Figure 3: Chosen representation of a solution*

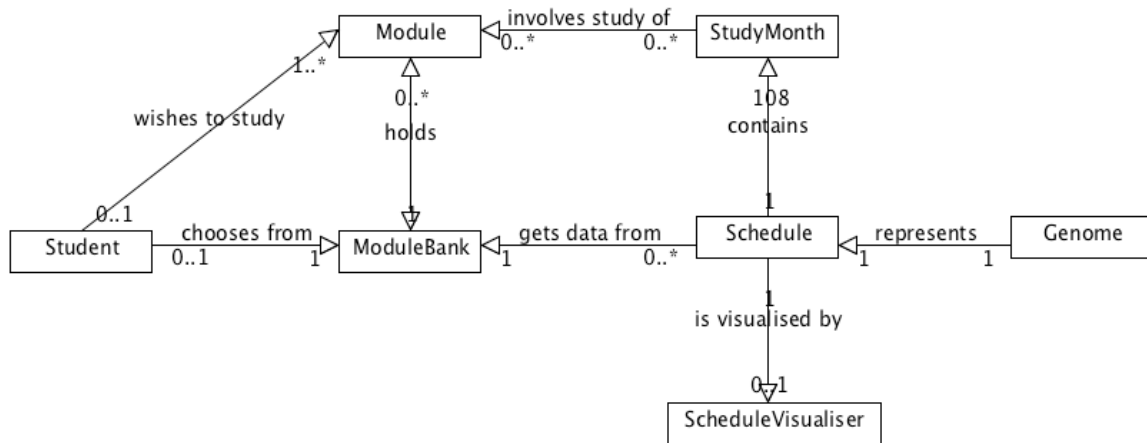
## Account of Outcome: Synthesis

Development involved three increments:

1. Creating the basic system, capable of representing a solution as a genome and visualizing it as a schedule
2. Adding algorithm functionality for processing multiple genomes over multiple generations, and developing an initial fitness function
3. System enhancements to make it easier to set / adjust parameters and evaluate results

### *Creating the basic system, capable of representing a solution as a genome and visualizing it as a schedule*

Figure 4 below shows the interaction between classes in this part of my system.



**Fig 4: Partial class diagram for the system (see Appendix 1 for full class summary)**

The first step was to create a ModuleBank class which could read and hold the relevant information about a module: the module's code, level, number of points, duration (in months), and the month or months of the year when it is possible to start the module. Using the OU website as my source, I picked a variety of modules of differing lengths, levels and start times (initially 10 modules, later expanded to 25) and I simply entered this information into a text file by hand (see figure 5 below). The ModuleBank class takes the filename of the text file in its constructor, reads in the data, and creates one Module object for each line of the text file.

[Code,	level,	points,	duration,	startable months(s) ]
M255	2	30	9	1, 9
DD303	3	60	9	1
M253	2	10	6	4, 10
MST121	1	30	9	1, 9

**Figure 5: Example lines of the text file from which the ModuleBank reads.**

To represent a solution, I created a Genome class to hold and manipulate the binary string. I wanted the Genome class to be general and potentially reusable, and not concern itself with what it was representing, so I linked each Genome object to an instance of a separate Schedule class. A Schedule object collaborates with ModuleBank to get Module data that allows it to translate the bit string it receives from the Genome object into a timetable. While the Genome only encodes a start date, the Schedule class has to take into account the duration of the module, and also consider the real starting dates when a module is offered. For instance, a Genome can encode a starting month for M255 of May, but M255 is in practice only offered with a start date of Feb or Oct. When the Schedule object consults the ModuleBank object, if it is discovered that the date encoded for that module is not legal, the

Schedule object uses the next feasible start date instead. As mentioned earlier, I later added repair operators to re-encode the legal starting date into the Genome's bit string, so this adjustment didn't have to be re-made in every generation.

Each Schedule object holds 108 StudyMonth objects, representing an 8-year period of time. The StudyMonth class extends *java.util.GregorianCalendar*, providing an easy way to manipulate dates. Each StudyMonth holds links to the Module objects that are scheduled to be in progress at that time, and keeps track of the total workload at that time. An example of the collaboration between Schedule, Module and StudyMonth objects is shown in Code Extract 1 below.

### Code Extract 1: The RecordModule method in the Schedule class

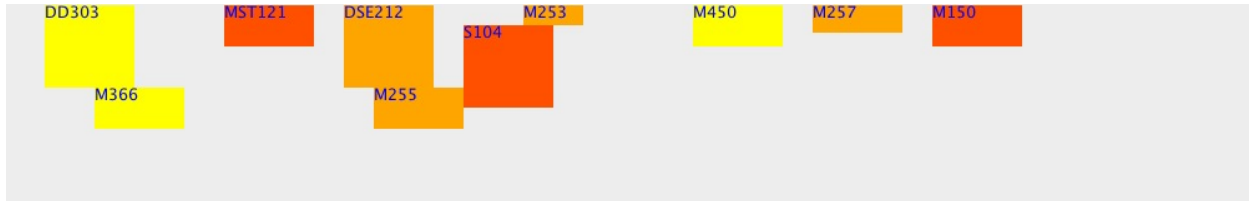
```
/**
 * Takes a Module and the Calendar object representing the month
 * in which the module should be started, and, for every appropriate
 * StudyMonth object held in the instance variable myMonths, records
 * the fact that this module is in progress, and increments the workload
 * for each of those StudyMonth objects accordingly.
 */
private void recordModule (Module mod, Calendar startMonth)
{
    StudyMonth aMonth;
    Iterator<StudyMonth> it = myMonths.iterator();
    boolean finished = false;

    while ( it.hasNext() && finished == false )
    {
        aMonth = it.next();

        if (aMonth.equals( startMonth ) )
        {
            // we've reached the month when this module should be started
            // so add the Module details
            aMonth.modulesInProgress.add(mod);
            aMonth.incrementTotalMonthlyWorkload( mod.getMonthlyWorkload() );

            // ... and for all the subsequent months til the module ends
            for (int i = 0; i < mod.getDuration()-1; i++)
            {
                aMonth = it.next();
                aMonth.modulesInProgress.add(mod);
                aMonth.incrementTotalMonthlyWorkload( mod.getMonthlyWorkload() );
            }
            finished = true; // so we don't go on pointlessly iterating
            // after the one and only occurrence of the module has been handled
        }
    }
}
```

To visualise schedules on the screen, both for my own diagnostic purposes and ultimately for the user, I created a ScheduleVisualiser class that extends JFrame (see Figure 6 below).



**Figure 6: A ScheduleVisualiser for the Schedule object belonging to a randomly generated Genome object**

Modules are represented as rectangles which are plotted along an x-axis of time, which begins in Jan 2012 and ends in Dec 2020 (8 years being the maximum length for a Schedule object, due to the decision to represent the number of years from now as a 3-bit binary number). The position on the y-axis depends on other modules being studied at the time; the module will be drawn at the first position on the y-axis at which it does not overlap any others (see Code Extract 2 below for how I achieved this, which involved teaching myself how to implement recursion, with help from an online tutorial by aathishankaran, 2008). The height of the diagram gives an approximate indication of the workload at that point in the schedule. Modules are colour coded by level: red for level 1, orange for level 2, and yellow for level 3. This means that a good schedule should be approximately rainbow-coloured: red then orange then yellow from left to right.

I decided to omit details such as axes and labels so that later classes could use ScheduleVisualiser by composition, and “wrap” around it whatever details are relevant. For instance, the StatVisualiser class uses an instance of ScheduleVisualiser to display the best solution found in a series of runs, and adds details about the solution’s fitness and the generation in which it was found, whereas the Output class displays recommended schedules for the user along with a written list of start dates for each module.

#### Code Extract 2: The addRectangle method and its helper method getOutOfTheWay in the ScheduleVisualiser class

*Note that the instance variable scheduleToDraw is the calling Schedule object, and the instance variable myData is a SortedMap<Module, Calendar> extracted from the calling Schedule object. ModRect is an inner class that extends java.awt.Rectangle.*

```
/*
 * Takes the code for a module, gets the modules details,
 * and calculates the arguments needed to create a rectangle
 * to represent this module. Adjusts the rectangle so that it
 * does not overlap with others in the collection,
 * and then returns the rectangle.
 */
private ModRect addRectangle(Module modToDraw)
{
    // create a rectangle to represent the module, with the following arguments:

    // x: shift right by the number of months since 'year zero'
    Calendar startDate = myData.get(modToDraw);
    int fromYears = (startDate.get(Calendar.YEAR) - scheduleToDraw.STARTING_YEAR) * 12;
    int fromMonths = startDate.get(Calendar.MONTH);
    int x = (fromYears + fromMonths) * SCALE_FACTOR_HOR;

    // y: Initially 0, but we'll increase it later if necessary
    int y = 0;
```

```

// width: corresponds to module's duration
int width = modToDraw.getDuration() * SCALE_FACTOR_HOR;

// height: corresponds to module's workload
int height = (int) (modToDraw.getMonthlyWorkload() * SCALE_FACTOR_VERT);

// set the color according to the level of the module
Color color = getColorForLevel(modToDraw);

// Create the rectangle
ModRect modRect = new ModRect(x, y, width, height, color, modToDraw.getCode());

// Check if it overlaps with any other rectangles,
// and if so, increase y until it does not

// First, copy the instance's set of ModRect's to a new variable
Set<ModRect> rectanglesToCheck = new HashSet<ModRect>(rectanglesToDraw);

// Now send that set of ModRect objects as an argument to checking method
getOutOfTheWay( modRect, rectanglesToCheck );

return modRect;
}

private void getOutOfTheWay(ModRect modRect, Set<ModRect> rectanglesToCheck)
{
    Iterator<ModRect> it = rectanglesToCheck.iterator();
    ModRect existingRect;
    while ( it.hasNext() )

    {
        existingRect = it.next();

        if (modRect.intersects(existingRect) )
        {
            // move it out the way
            while (modRect.intersects(existingRect) )
            {
                modRect.translate(0,1);
            }

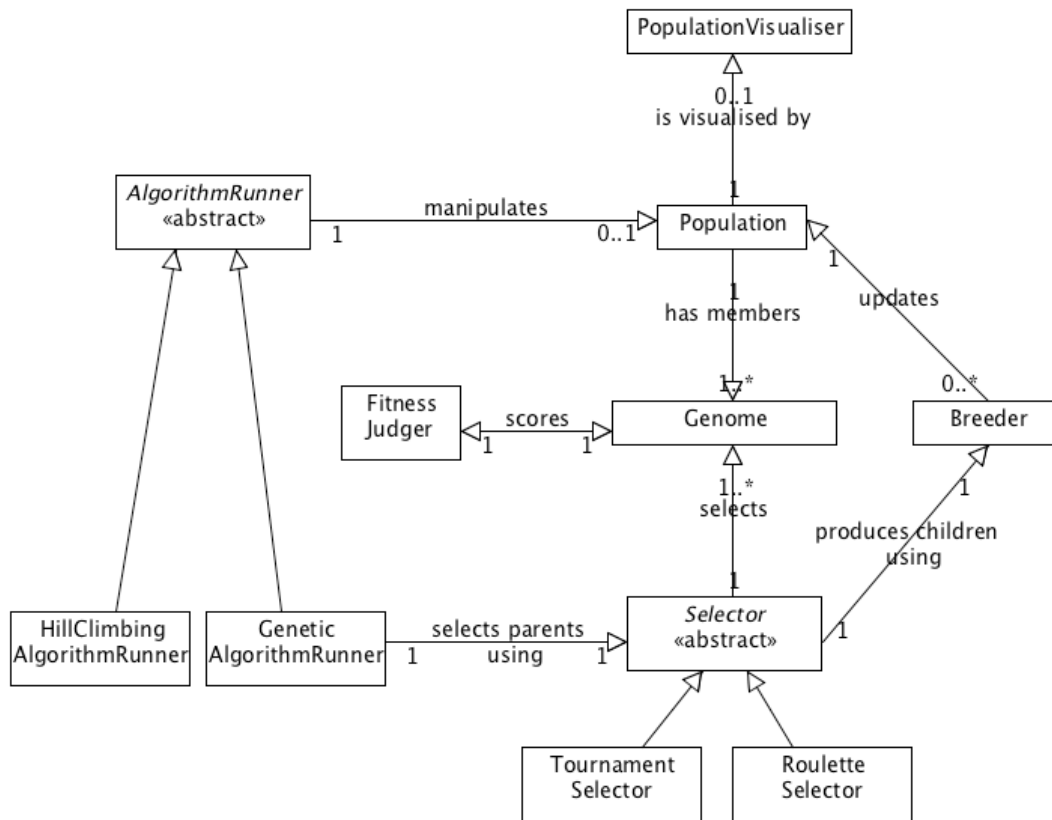
            // remove the ModRect we just checked from the set
            // and check we didn't move ourselves into any of the others
            it.remove();
            Set<ModRect> smallerSet = new HashSet<ModRect>(rectanglesToCheck);

            getOutOfTheWay( modRect, smallerSet); // my first recursion!
        }
    }
}

```

*Adding algorithm functionality for processing multiple genomes over multiple generations, and developing an initial fitness function*

Figure 7 below shows the interaction between classes in this part of my system.



**Fig 7: Partial class diagram for the system (see Appendix 1 for full class summary)**

I created a **Population** class to hold a Collection of **Genome** objects and generate an initial population, and a **PopulationVisualiser** class (utilising **ScheduleVisualiser**) to display the fittest genomes in the population.

One of the harder parts of the project was the **FitnessJudger** class, which uses links to **Genome** and **Student** (not shown) to assign a fitness score to each **Genome**, based partly on the student's preferences. Although the class is not complicated, my fitness calculations were a source of many bugs (for example, see Appendix 3: Fixing my "reverse evolution" bug), and the parameters that control the weighting of different fitness factors required much tweaking to obtain good performance. The three factors that are combined to give the **Genome**'s fitness are shortness, adherence to student's workload limit, and appropriate level order (see Code Extract 3 below for how I decided to implement these). Note the use of "perfect score" constants from which the penalty amount is subtracted, each of which is set to the arbitrary value of 1000. This was my improvised technique for generating positive integer values. If I were starting again, I would avoid the need for these constants either by using the inverse of the penalty as the fitness, possibly along with a scaling factor (as discussed by Cartwright, 1994) or by taking the cumulative penalty as the fitness value and simply treating lower scores as fitter than higher, instead of vice versa (as used in many studies, eg Gröbner and Wilke 2001, and Nammuni et al 2002).

Code Extract 3: The three methods of the FitnessJugder class used to score the three different fitness factors: shortness, adherence to workload limit, appropriate level order.

```

/**
 * Returns a score which reflects a penalty for
 * every month in the Schedule up to the last active month
 */
private int judgeLength()
{
    int score = PERFECT_SHORTNESS_SCORE -
        (sched.totalDuration * LENGTH_PENALTY_MULTIPLIER);
    return score;
}

/**
 * Returns a score which reflects a penalty for each month
 * which exeeeds student's max hours per month
 */
private int judgeMonthlyHoursAgainstStudentMax()
{
    int runningBandwidthPenalty = 0;
    int BandwidthPenaltyThisMonth = 0;

    for (StudyMonth aMonth: sched.myMonths)
    {
        // see how many hours we are over the Student's limit
        int excessiveHours = aMonth.getTotalMonthlyWorkload()
            - student.maxMonthlyHours;

        // penalise the schedule for exceeding the Student's limit this month
        if (excessiveHours > 0)
        {
            BandwidthPenaltyThisMonth = excessiveHours
                * BANDWIDTH_PENALTY_MULTIPLIER;

            runningBandwidthPenalty = runningBandwidthPenalty+BandwidthPenaltyThisMonth;
        }
    }
    return PERFECT_BANDWIDTH_SCORE -
        (runningBandwidthPenalty * BANDWIDTH_PENALTY_MULTIPLIER);
}

/**
 * Returns a score which reflects a penalty for each module
 * which is scheduled AFTER a module of higher level
 */
private int judgeLevel()
{
    Map<Module, Calendar> theMap = genome.getModulesWithDates();

    int runningLevelOrderPenalty = 0;

    // get a key value pair from the map and assign to local variables
    for (Map.Entry<Module,Calendar> e : theMap.entrySet())
    {
        Module mod = e.getKey();
        Calendar startDate = e.getValue();

        for (Map.Entry<Module,Calendar> otherEntry : theMap.entrySet())
        {
            Module otherMod = otherEntry.getKey();
            Calendar otherStartDate = otherEntry.getValue();

```



```

        if (
            otherMod.compareTo(mod) < 0 // if otherMod is easier
            && otherStartDate.getTimeInMillis()
                > startDate.getTimeInMillis() // but starts later
        )
        {
            //penalise!
            runningLevelOrderPenalty = runningLevelOrderPenalty
                + PENALTY_PER_MODULE_OUT_OF_LEVEL_ORDER;
        }
    }
    return PERFECT_LEVEL_SCORE - runningLevelOrderPenalty;
}

```

I designed a Selector class to be responsible, in collaboration with the Breeder class, for creating the next generation of Genome objects. It is widely agreed that elitism leads to significant performance improvements (eg Mitchell 1996 and Gen and Cheng 2000), so I initially had the Selector object preserve the fittest 5% genomes in the population, and this turned out to be a good level, which I returned to after experimentation with other values. The rest of the new population is generated by selecting parent Genome objects which are passed to a Breeder object. Initially, my Selector class achieved this using roulette selection, but later, given the ubiquity of tournament selection in the literature and my hope it would improve efficiency, I created specialised RouletteSelector and TournamentSelector classes and abstracted their common functionality to the Selector class from which they inherit.

The Breeder object performs single point crossover and mutation at given probabilities. Based on common values mentioned in Mitchell, I initially set the chance of crossover to 0.6. (I eventually altered this slightly – see *Account of Outcome: Evaluation* section.) The chance of mutation I set to 0.2 for a whole genome, ie a genome is selected for mutation with probability 0.2, and mutation is then performed at one random position. On reflection, I probably should have used a bitwise mutation rate, where each bit is mutated with some (lower) probability. (Many exercises in Mitchell 1996 use a bitwise mutation rate of 0.001). This would have maintained a consistent mutation rate across runs with different numbers of modules and hence different length bit strings. However, for simplicity I followed the technique taught in M366 Block 5. I chose the higher mutation probability of 0.2 on the basis that many mutations in the genome (the majority of those in positions representing months) would not alter the schedule, due to only one month being a legal start date for most modules.

For the purposes of performance comparison with the GA, I implemented a Hill Climbing algorithm, using the steps for a “random mutation hill-climbing” method given in Mitchell (1996), which are:

1. Start with a single randomly generated string. Calculate its fitness.
2. Randomly mutate one locus on the current string.
3. If the fitness of the mutated string is equal to or higher than the fitness of the original string, keep the mutated string. Otherwise, keep the original string.
4. Go to step 2.

As specified in Mitchell (1996), to make the algorithms comparable, each “generation” of the Hill Climbing Algorithm performs step 2 above  $n$  times, where  $n$  is the number of individuals in the GA population.

### *System enhancements to make it easier to set / adjust parameters and evaluate results*

At this point I decided to think of my system as having two different modes: “Scientist mode”, which was for me or other computer scientists to use, and which would allow parameters to be chosen (eg GA versus hill climbing, GA with roulette selection versus GA with tournament selection, number of generations and individuals), multiple runs to be performed, and their results saved. In this mode, a fixed set of modules is used and the adherence to bandwidth mode is set permanently to Moderate, so that the differences between algorithms can be studied.

The other mode is “Student mode”, in which my system operates like the actual user-focused system that this project was scoped to deliver. Here, algorithm types and selection methods and technical parameters are hidden and fixed (a 100 generation GA with roulette selection and repair operators is used; see *Evaluation* section for discussion of why), but the user can choose which modules to include in the schedule, how many hours a month they wish to study, and which adherence to workload mode(s) they would like to see schedule for (Conservative, Moderate, Aggressive).

The code for Scientist mode creates a StatCollector object, which creates an instance of AlgorithmChooser to collect desired parameters from the scientist (see Fig 8 below). The chosen parameters are passed to a ParamSetter object, which is used for the entire set of runs. The StatCollector object then uses the ParamSetter object to create an AlgorithmRunner object for each run. Each AlgorithmRunner object records data for each of its generations in a PopRecord object, and at the end of each run the StatCollector object writes data from the PopRecord objects to a file.

Choose parameters for the Algorithm

☒ Genetic Algorithm      ☐ Hill Climbing Algorithm

☒ Roulette Selection      ☐ Tournament Selection

How many generations?      50

How many individuals / steps?      50

How many runs?      8

Choose filename to save data     

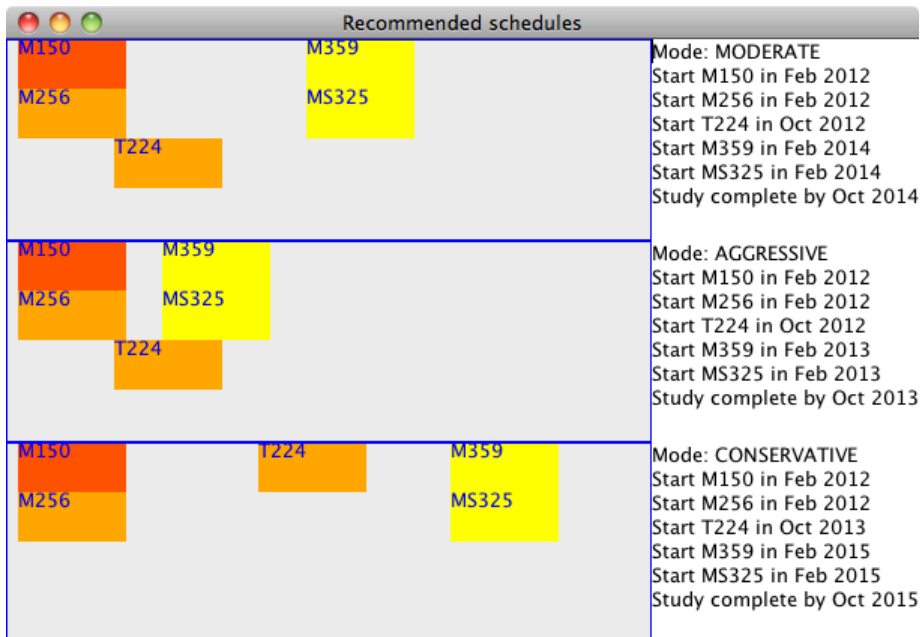
GO

**Fig 8: An instance of *AlgorithmChooser*, created by the *StatCollector* object when running the system in Scientist mode**

The code for Student mode creates an *Initializer* object, which sets up the system with default technical parameters, and then creates a *StudentPreferenceChooser* object to collect input from the student user (see Fig 9 below).

**Fig 9: An instance of *StudentPreferenceChooser*, when running the system in Student mode**

After running the algorithm, the *Initializer* class creates an *Output* object that displays the schedule(s) for the student (see fig 10 below).



**Fig 10: An instance of *Output*, showing an example of the final output delivered by my system to the user**

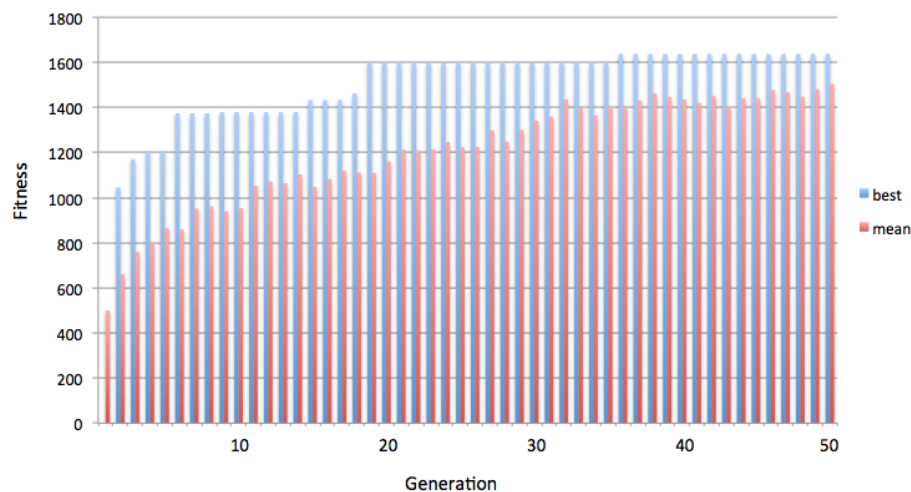
## Account of Outcome: Evaluation

My evaluation was focused on answering the following questions:

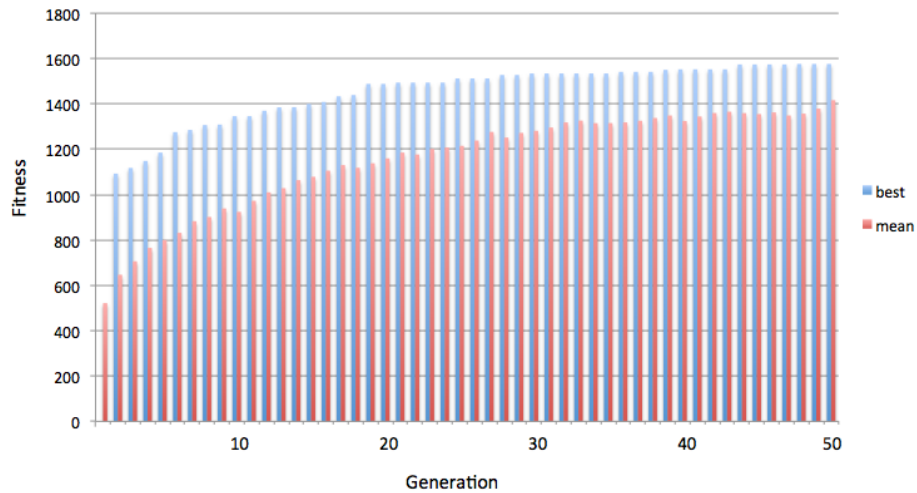
1. Does the GA find good solutions?
2. Does the GA find better solutions than a hill-climbing algorithm?
3. Do repair operators improve the GA's performance?

### *Does the GA find good solutions?*

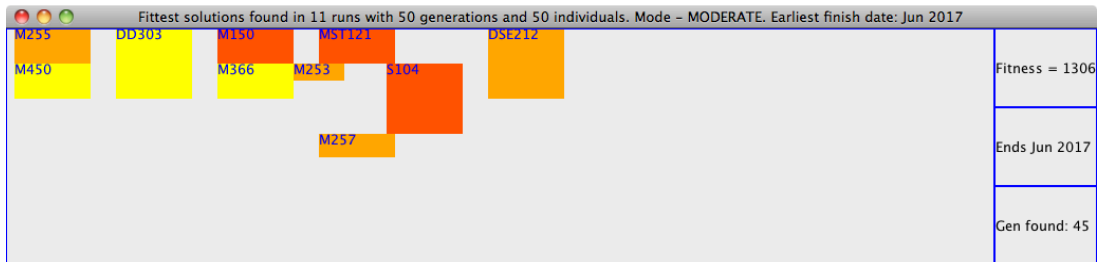
The file produced by the StatCollector class allows graphs to be plotted showing the convergence towards a solution, both for a single run (see figure 11) or averaged across a number of runs (see figure 12). These show quantitatively that fitness improves over time, and therefore that the GA is successful at finding fit solutions. Looking at the actual schedules delivered, we can also say qualitatively that the GA does indeed manage to find considerably better solutions (see figure 13) than a typical randomly generated one.



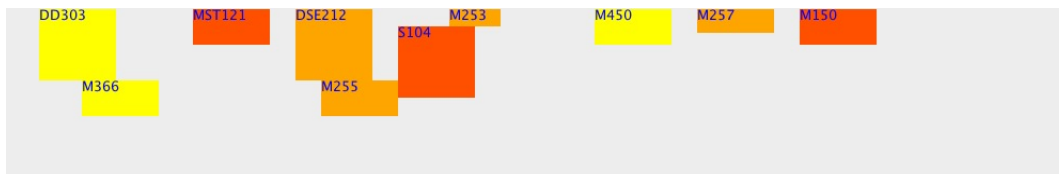
**Figure 11: Graph showing best and mean fitness at each generation of a typical run of a Genetic Algorithm with 50 generations and 50 individuals. Note that the best fitness increases in spurts such as in generations 6, 19 and 36 as good new solutions emerge suddenly from the selection process, while the mean fitness of the population increases smoothly as members converge on a solution.**



**Figure 12: Graph showing best and mean fitness at each generation of a Genetic Algorithm, averaged across 5 runs, each with 50 generations and 50 individuals. Note that the spikiness seen in figure 11 is smoothed out by the averaging.**



**Figure 13: Example of a solution found by the GA (WITHOUT level order as a factor in the fitness function). To the human eye, it appears to succeed at minimizing completion time and only briefly violating the student's workload limit. Note that it is clearly superior to the randomly generated solution shown in Fig 6 (reprinted below)**

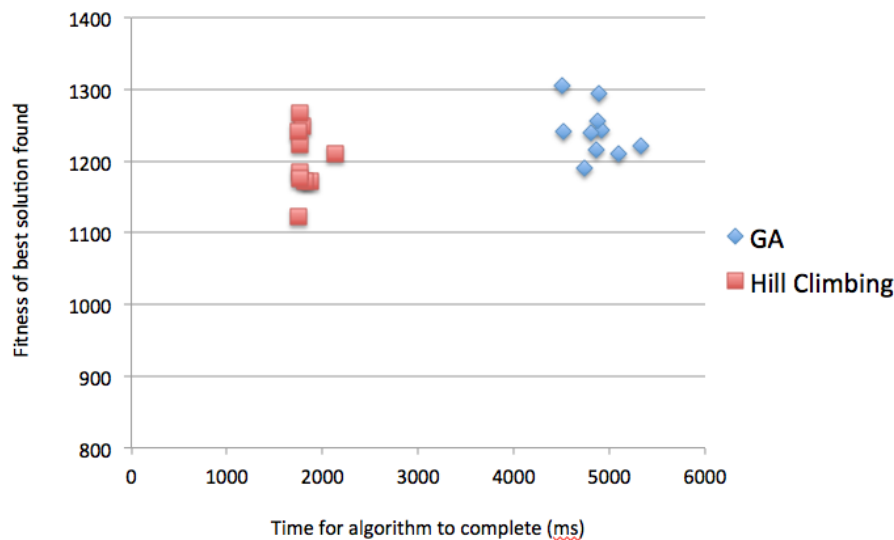


**Figure 6 (reprinted): A ScheduleVisualiser for the Schedule object belonging to a randomly generated Genome object**

### *Does the GA find better solutions than a hill-climbing algorithm?*

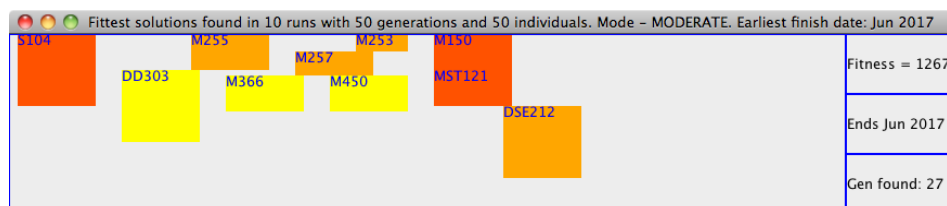
I first compared the performance of the Genetic Algorithm with the Hill Climbing algorithm *before* I had implemented module order into the fitness factor, ie solutions were only

optimised for shortness and adherence to workload limits. At that point, the Genetic Algorithm did indeed seem capable of finding fitter results than the Hill Climbing Algorithm, with the GA finding solutions in each run an average of 40 points fitter, and a best solution across 10 runs that is 39 points fitter (see Figure 14 below, and Tables 1 and 2 in Appendix 4). This difference is statistically significant at the 95% confidence level (Student T-test,  $t = 2.23$ ,  $sd = 40.6$ ,  $df = 18$ ,  $p < 0.05$ , calculated using the online calculator at [http://www.physics.csbsju.edu/stats/t-test\\_bulk\\_form.html](http://www.physics.csbsju.edu/stats/t-test_bulk_form.html)).



**Figure 14: A scatterplot of results from 10 runs of the Genetic Algorithm and 10 runs of the Hill Climbing Algorithm, with  $n=50$  and 50 generations (WITHOUT level order as a factor in the fitness function)**

There were some important caveats to this finding. Firstly, statistical significance is of limited relevance given that the quantification of fitness is somewhat arbitrary. Indeed, looking at the best schedules produced (compare figure 13 above with figure 15 below), which both finish in June 2017, they both appear to human eyes to be fairly equally attractive.

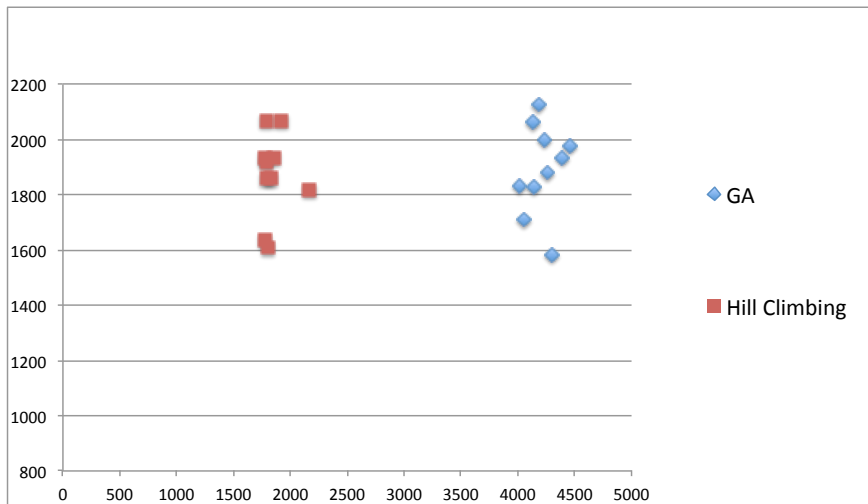


**Figure 15: Best solution found by Hill Climbing Algorithm (WITHOUT level order as a factor in the fitness function)**

Secondly, the Genetic Algorithm was taking more than twice as long to execute as the Hill Climbing Algorithm (see Appendix 4). Although I was aware that some of this extra time was due to imperfect programming choices on my part, which could perhaps be rectified, it is to be expected that a GA would be more computationally costly, and therefore it needs to produce clearly superior results to justify its use over a Hill Climbing algorithm.

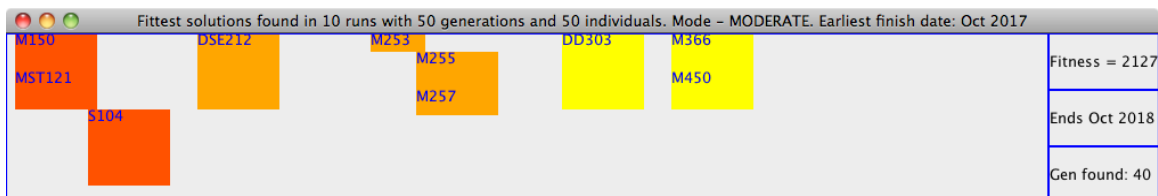
My next step, therefore, was to attempt to improve the GA's performance further. I tried tweaking crossover (ending up with 0.55), mutation (ending up back at 0.2), and elitism (ending up back at 5%) rates, as well as implementing tournament selection, but I found no significant improvement in fitness. Neither did tournament selection increase the algorithm's speed as I had hoped it might.

More disappointingly, once I had factored level order into the fitness function, I found that the performance advantage of the GA over the Hill Climbing algorithm disappeared, with no significant difference between the fitness of the two data sets in a Student T-test ( $t = 0.323$ ,  $sd = 159$ ,  $df = 18$ ,  $p > 0.05$ ). See figure 16 below.

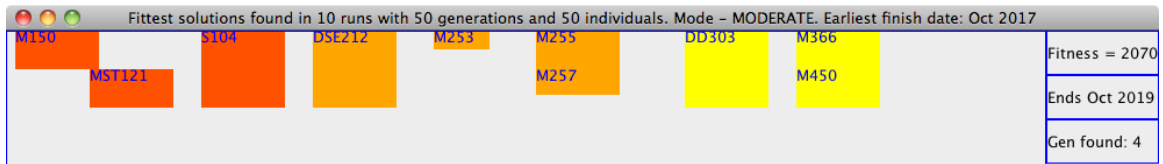


**Figure 16:** A scatterplot of results from 10 runs of the Genetic Algorithm and 10 runs of the Hill Climbing Algorithm, with  $n=50$  and 50 generations (WITH level order as a factor in the fitness function)

However, the best solution found by the GA was still better than the best solution found by the Hill Climbing algorithm, finishing a year earlier (see figures 17 and 18). This is consistent with the observation of Fang (1994) that while a GA is often capable of finding better results in the long term, a local hill climber can sometimes produce *acceptable* performance much more quickly. This becomes even more relevant when we note that my Hill Climbing algorithm usually found its best solution within about 7 “generations”, ie 350 steps. I had not programmed my system to halt the algorithm after a certain number of generations/steps with no improvement; had I done so, it would have reduced the completion time for the Hill Climbing algorithm by more than half, making it over four times faster than the GA.

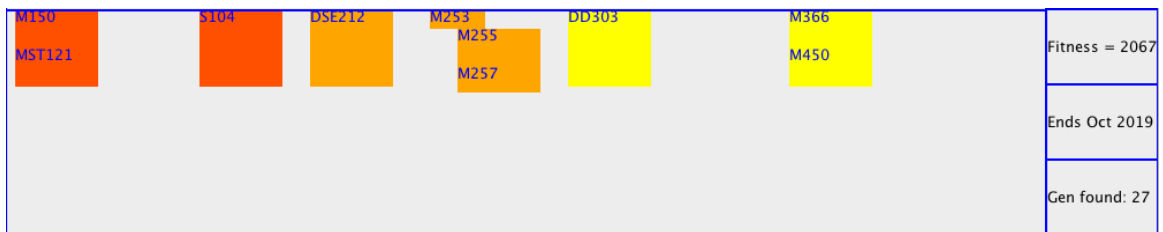


**Figure 17:** Best solution found by Genetic Algorithm



**Figure 18: Best solution found by Hill Climbing Algorithm**

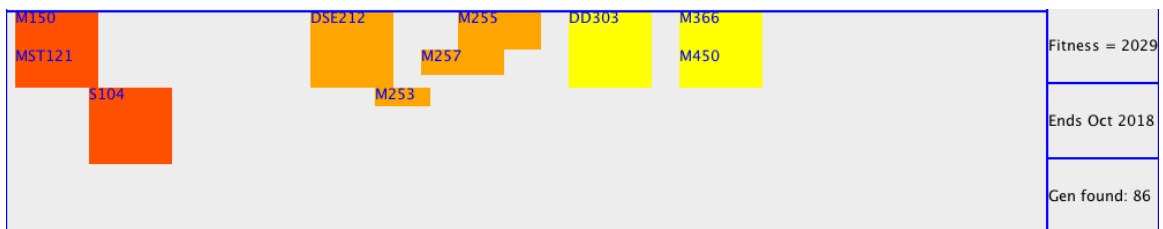
Why was the GA struggling more now that level order was a factor? Based on further experimentation and my study of the literature, three observations struck me from three particular solutions suggested by the system. Consider figure 19.



**Figure 19: A Schedule that would benefit from a tweak to the way shortness is judged.**

This schedule could have been better if M366 and M450 were both shunted forward a year. Yet there was no way for the fitness function to select for this: if a mutation happened to occur that brought one of them forward a year, this mutated solution would not have been any fitter because the last active month is still the same, thus the new Schedule is awarded the same shortness score. And as there are probably many copies of the unmutated version in the population, the new version would be unlikely to get much chance to reproduce and perhaps mutate again so that the last module might also move forwards. This made me think that perhaps when judging length I ought not only to penalize based purely on the *date* of the last active month, but on the *weight* of activity for the last active month. This would encourage even just one of a pair of modules at the end of a schedule to move to an earlier position.

The second observation relates to the idea of repair operators as discussed by Gröbner and Wilke (2001). Consider figure 20.

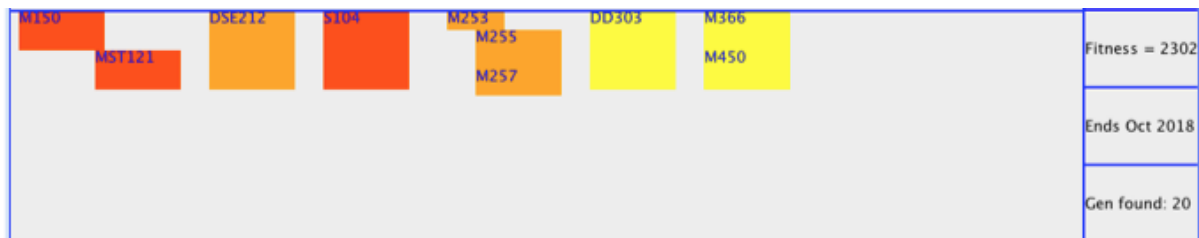


**Figure 20: A Schedule that would benefit from a gap eliminating operator.**



This Schedule made me realize that by introducing level order into the fitness function, I had made it much harder for modules to move around to fill in gaps. For instance here, if level were not a factor, the last two modules would probably have moved themselves into that gap between S104 and DSE212 before long. But with level order “on”, the price for doing this is too high, because the level 3 modules would be highly penalized for appearing before the level 2 modules. For the level 3 modules to move left without destroying the order, the level 2 modules would have to move left to create some space, and there is no motivation for them to do so, as the schedule does not become shorter and therefore fitter unless the L3 modules move simultaneously – a vanishingly unlikely evolution. It is therefore almost impossible for this schedule to improve. However, I realised that could add a “gap-eliminating repair operator” that runs at the end of a generation, analyses schedules for gaps like the one above, and shunts all the subsequent modules forward while retaining their order. Such an operator would also be a different way to fix the problem in Figure 19.

Similarly, in figure 21 we see that S104 and DSE212 are in the reverse of ideal order, but they are unable to swap because they would have to both evolve into each other’s position at the same instant, which is very improbable.



*Figure 21: A Schedule that would benefit from a ‘swap’ operator (genetic or repair operator)*

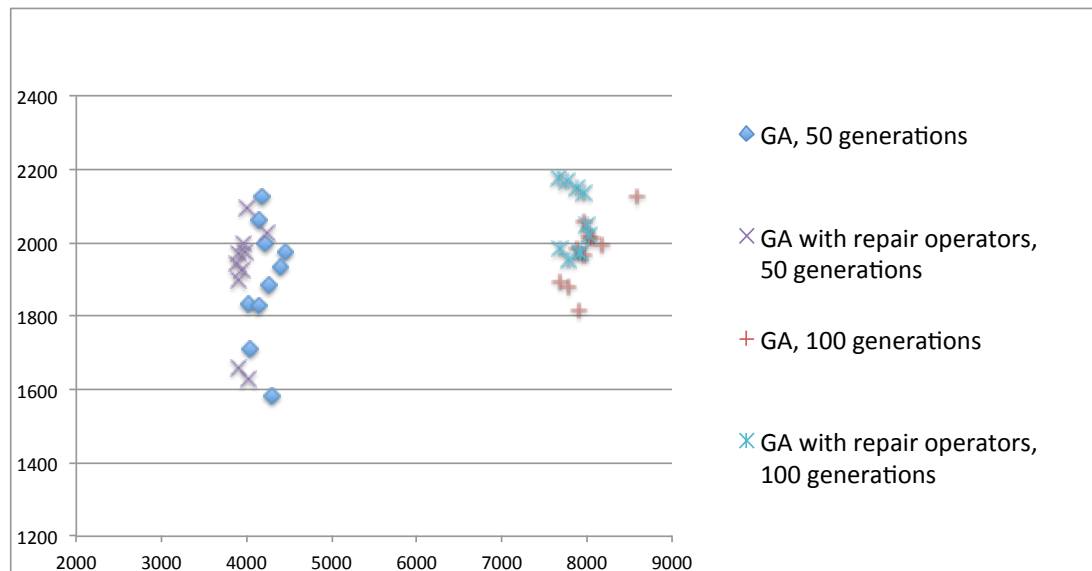
Many GAs for scheduling problems, including Nammuni et al (2002), use a genetic operator *swap* that swaps two random units according to some probability. Such an operator would make it quite likely that the improved solution would evolve. Alternatively, swap could be implemented as a Repair operator, applied locally after genetic operators in specific cases such as the above where swapping would increase fitness.

Given my limited time at this point in the project, I decided to focus on the gap-eliminating repair operator, as this had the potential to solve two of the three problems I had identified. Implementing it meant adding some new core functionality, to allow schedules to re-encode their improvements into the genome to achieve the “Lamarckian evolution” described by Gröbner and Wilke (2001). This new core functionality would, I hoped, also have a secondary benefit: beyond improving solution quality, it might also improve speed, because it would allow me to permanently re-encode valid starting months to the genome, instead of having to repeat, in every generation, the task of incrementing the encoded month until a valid starting month was found.

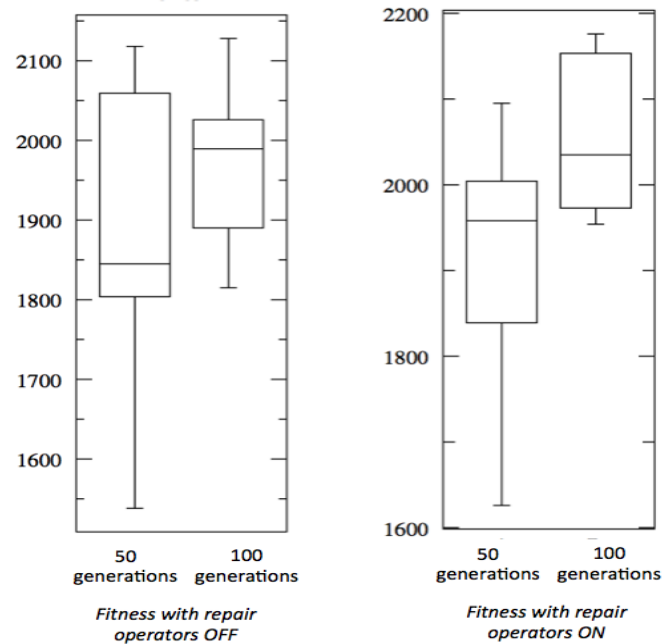
### *Do repair operators improve the GA’s performance?*

Figure 22 below shows how repair operators impacted performance and completion time of the Genetic Algorithm. At the standard 50 generations that I had been using throughout the project, repair operators did not produce a significant improvement in performance (Student T-test,  $t = 0.263$ ,  $sd = 158$ ,  $df = 18$ ,  $p > 0.05$ ). However, there was a slight but highly significant reduction in completion time ( $t = 4.38$ ,  $sd = 122$ ,  $df = 18$ ,  $p < 0.001$ ). This supports the claim of Gröbner and Wilke (2001) that “Lamarckian strategies can speed up the search process”.

Interestingly, I found that with repair operators on, solutions took longer to converge. I therefore tried extending the runs to 100 generations, and found that this allowed significantly fitter solutions to be found. Without repair operators, there had been no significant difference in the fitness found after 100 generations versus that found after 50 ( $t = 1.37$ ,  $sd = 132$ ,  $df = 18$ ,  $p = 0.19$ ). However, with repair operators on, the extra generations did yield a fitness improvement significant at the 95% level ( $t = 2.60$ ,  $sd = 125$ ,  $df = 18$ ,  $p = 0.018$ ). See figure 23.

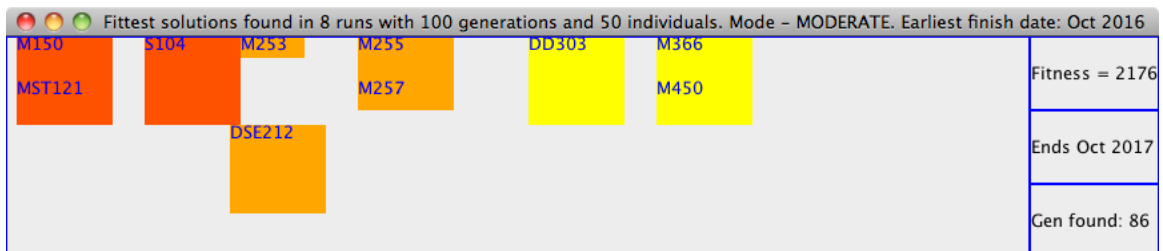


**Figure 22: A scatterplot of results comparing the GA with and without repair operators over 50 and 100 generations.**



**Figure 23: Boxplot comparing the GA with 50 and 100 generations, with and without repair operators.**

Clearly the completion time becomes much longer with 100 generations: an average of 7864 ms versus just 3408 ms for the Hill Climbing algorithm running for the equivalent of 100 generations. However, the best solutions found by the GA with repair operators at 100 generations really were quite good (see figure 24 below), and, as can be seen from figures 22 and 23 above, the high quality of the results was fairly consistent.



**Figure 24: Best solution found by Genetic Algorithm with repair operators after 100 generations**

If we therefore take the 100-generation GA with repair operators to be our best GA option, and return to the question of whether a GA performs better than a Hill Climbing algorithm, we are left with no definitive answer. Much depends on our definition of “better”. The Hill Climbing algorithm is much faster, but is less able to produce consistently good solutions. In order to decide which would be better for my system to use in Student mode, I made several comparisons of four runs of the Hill Climbing algorithm with a single run on the GA with 100 generations and repair operators (both of which took about the same time to complete) and in two out of three comparisons, the single run of the GA produced a fitter solution than any of the Hill Climbers were able to find. I have therefore opted to use this GA in my

system, although it remains disappointing that the GA hardly scores better than a Hill Climbing algorithm. Following are some reflections on why this might be.

**Possibility 1: In my problem, the definition of the ‘best’ solution is too subjective**

When discussing scheduling problems, Cartwright (1994) stipulates that the quality of any candidate solution should be able to be easily determined. He gives as an example the Travelling Salesman Problem, in which the best solution is unambiguously the one with the lowest number of miles. In flowshop scheduling problems, the schedule with the earliest completion time is clearly the best. By contrast, the study schedules produced by my system are more subjective. Even though each has a quantified fitness score, the real judgement of its quality is qualitative and will vary from student to student. Because of this, even the solutions produced by 100 generations of the GA with repair operators, which statistically are significantly better than the hill climbing solutions, are not always manifestly better to the human eye (and perhaps not better enough to warrant waiting several times longer to get them).

**Possibility 2: There are many “good-enough” solutions at local maxima in this problem’s search space**

Cartwright (1994) states that the search space of scheduling problems is generally highly corrugated, without long-range order, and that large numbers of local minima and maxima prevent hill climbing algorithms from being effective search techniques. Mitchell (1996) points out that GAs seem to be good at quickly finding promising regions of the search space, while hill climbers are often good at zeroing in on optimal individuals in those regions. The success of the Hill Climbing algorithm with my problem may therefore suggest that the search space for my problem (although large) is rather regular, with lots of hills all at similar and respectable heights. This makes sense when we consider that there lots of different ways to produce a pretty good schedule, as compared to, say, the Travelling Salesman Problem, where just a few high quality solutions, and one optimal one, are hidden somewhere in the vast search space, amongst many local maxima. With the TSP, if you start in the wrong place and search only locally, you have no hope of a good result, so GAs significantly improve performance by allowing you to start in several places. In my problem, it appears that whichever region you start in, you can usually climb from there to a hill that will suffice, so a hill climbing algorithm will do the job.

**Possibility 3: The problem lacks the global constraints that GAs are particularly good at handling**

Most of the scheduling and timetabling problems tackled in the literature contain global constraints regarding finite overall resources, such as the number of machines available on which jobs can be processed, or the number of students who can be taught at any one time. Gröbner and Wilke (2001) have a global constraint on their employee roster of “fairness”, which stipulates that shifts, especially night and weekend shifts, should be distributed uniformly among employees, and they justify their use of a GA partly on the basis it can deal with global constraints much better than most other techniques. In my problem, however, one schedule applies to one individual student only, and no global resources are considered. Mine is therefore an inherently simpler problem than most scheduling problems, and this may explain why a Hill Climbing algorithm is sufficient for finding reasonable solutions. If we

were to re-scope the problem to introduce a global constraint such as “a maximum of 50 students may study a module during any given presentation” and require the system to formulate schedules for hundreds of students and consider their combined as well as their individual merit, it seems likely a Hill Climbing algorithm would cease to be as effective (presumably because of the increasing corrugation of the search space) and a GA may well emerge as a more powerful tool.

## Review Stage of Development

I am very pleased to have achieved a working system that could perhaps be of some (limited) use to an OU student in its current form. Although some of the aspects of the project that I had originally anticipated including had to be taken out of scope as the project progressed and issues arose, I was able in the end to meet all the requirements of my reduced scope, so in that sense the project was successful.

There remain, however, some major limitations. The system holds data for only 25 modules. To be of any genuine use in the real world, the system would need to import data for a much wider range of modules, as previously discussed. The system is not robust, with very little exception handling, no input validation, no unit testing, and not enough protection of data privacy. The system is slow, taking something close to 25 seconds to deliver three schedules, and that is without performing multiple runs and choosing the best, which it probably ought to do to avoid the risk of giving students a poor schedule. Perhaps if this system were to be made available for public use, output could be delivered by email rather than in real time to avoid keeping users waiting. Alternatively, a more expert programmer than myself could probably find ways to improve the program’s speed. Making it multi-threaded would probably help enormously, but unfortunately this was beyond my skills at the beginning of the course, and later (when I had learnt more about threads) I found that the way I had designed the classes was not conducive to implementing them.

There are so many other ways my system could be improved. One would be a proper GUI that helps students select the modules they wish to include by displaying useful information about each module such as its title and description, and allows students to search and sort through the options. Another improvement would be to take into account more factors on which a student may wish to base their decisions, such as whether or not modules have an exam, and even what the chances are of passing or of obtaining a specific desired grade for a module (based on results published by the OU in Sesame magazine).

I would very much have liked to implement a system that not only schedules given modules but actually selects the best modules to take from a longer student-chosen list of possible modules to take. These could be chosen on a basis of pure efficiency (choosing the modules which were capable of being combined into the fittest schedule) or more sophisticated criteria could be introduced, such as a student’s relative declared interest in taking each module, or even the likelihood of successful completion of those modules in that order, given how each would enhance the student’s skills. This last is the ontology-based system I would have liked to develop, building on the work of Nammuni et al (2002), with the student’s

skillset being updated across time on the assumption of successful study of each module, and with fitness penalties for modules scheduled at points where the student's skillset was insufficient to take it on. This would be considerably more useful than my rather crude reliance simply on level order, but as stated earlier it would require more knowledge about course pre-requisites and assumed skills than the OU currently makes available.

## Review Project Management

Although this was not a “messy” project in terms of stakeholders, it was quite open-ended and exploratory. I had some ideas that I wanted to start experimenting with quickly, without having to study a large amount of literature first. I saw studying the literature as an on-going activity parallel to my own efforts at solving the problem, with the two coming together during periods of reflection after each cycle of work. As such, the Structured Case model worked quite well for me. When I began, I had some hunches about how to tackle my problem, but conceptually I did not even really know how to classify it. Through on-going scrutiny of the literature, and through concurrent practical work, as this life cycle model dictates, I gradually arrived at a clearer understanding of my problem. The Structured Case model also helped me to hone my scope, and to evolve my understanding of what constitutes a solution.

The flexibility of Structured Case has been helpful, then, but there were a few downsides. One is that it did not enforce the strict stages of software development that I would have had with a more rigid approach such as the waterfall model. As a result, I pursued software development in a rather haphazard way, which led to some sub-optimal design and code which I could not later change. However, I think this was due more to the limitations of my software development skills than to the particular model I was using (see *Review of Personal Development* section) and in any case, the waterfall approach would have been completely inappropriate given that the problem was neither routine, familiar, nor well-defined.

Another downside of Structured Case is that because each cycle was quite long and not fully defined in advance, the model did not do much to help me stay on track with my timing plan. I sometimes found myself spending much longer than anticipated on something that my reading revealed to be far more complex than I had realised. However, with a few changes to the way I was planning my time – such as including lots of “buffer” weeks in case I got waylaid and delayed – I managed to continue using Structured Case successfully.

Lastly, the “parallel path” approach of practical work and literature definitely has its drawbacks. There are several areas in which I would have made a different decision if I had read the relevant literature at the beginning of the process (eg integer over binary encoding). On the other hand, if I had tried to read all the literature first, I'm not sure I would have got as far with my practical work as I did – and indeed much of the literature only began to make sense to me as I pursued my practical work and become properly acquainted with the issues being discussed.

## Review of Personal Development

The autonomy of this project has been immensely satisfying, and motivated me to acquire new skills. Most dramatic has been the improvement in my Java and software development skills. Prior to this project I had not written a program larger than perhaps half a dozen simple classes, and I certainly had not taken a systematic approach to software development before. Studying M256 alongside this module was very useful in teaching me the importance of good design – I only wish I had completed it beforehand so that I could have put the principles into practice from the beginning. Having said that, I tend to be rather action-oriented by nature, and find it hard to engage in lengthy planning before getting started on practical work. This has definitely caused me some problems and cost me time, for instance in having to quite substantially redesign classes upon realizing I needed them to behave or collaborate in some way I had not anticipated. For instance, I originally passed Module objects around as strings of their codes, but later realised it would be better to pass a reference to the actual object so that other fields could be accessed, such as their startable dates. Being reluctant to fiddle with working code, I had to build some ugly workarounds. However, as the project went on, I became better at using class diagrams to visualise and think through the relationships between different parts of the system, and this helped me avoid rushing into implementation too hastily. This also helped me identify some changes that improved cohesion, such as having Genome objects calculate their own fitness as part of their construction, rather than have the Population object do it for each Genome.

I believe I have also got better at making appropriate use of literature. This is not easy for me, as I tend to want to do everything myself, and usually find practical work more rewarding than reading the work of others. With this project, this problem was exacerbated by the fact that I found it quite difficult to find source material at the right level. Many of the academic papers my initial searches uncovered were written at a level so far beyond my knowledge that I could not even understand enough to decide if they were relevant. The most recent papers especially contained advanced maths, which tended to drive me back to my coding. I developed several strategies that helped me get more out of literature. One was to be more ruthless about abstracts, and not download a paper unless I was really sure it was going to be useful. This helped me avoid the disheartening situation of trying to tackle a thick pile of impenetrable and barely relevant papers. Another strategy was to locate and read only one paper at a time, and then follow up the most promising reference from that paper as my next “lead”. Again, this avoided overwhelming myself with content. Yet another strategy was simply to prefer slightly older papers over the most recent ones. While this meant I was excluding myself from the cutting edge of the field, it did save me from “running before I could walk”. Having reached a better understanding of scheduling problems and GAs in general through this project, I now feel better equipped to take on some of those papers that were previously beyond me.

After an over-optimistic start, I have got better at realistically assessing what I can achieve in a given time. My enthusiasm often causes me to attempt to take on too much, and my original scope and timing plan were certainly too ambitious. I tend to get absorbed in coding

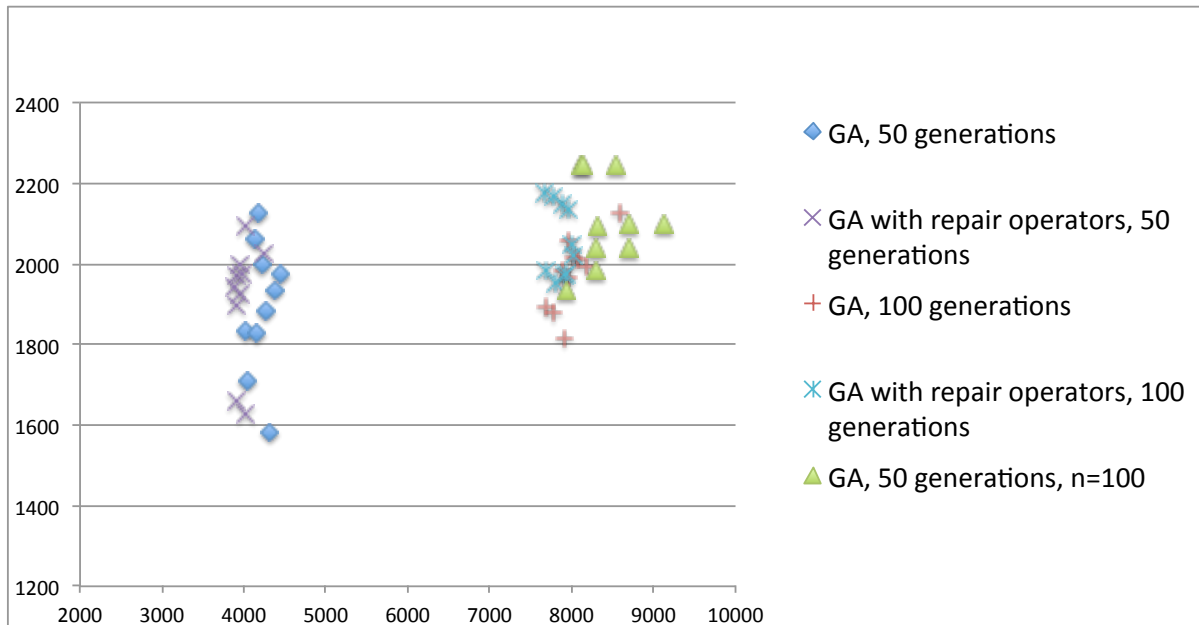


and spend much more time than I intended trying to debug “just one more little thing”. A missed TMA deadline taught me a lesson, and forced me to reduce my scope and devise a timing plan with far more empty space than my instincts called for. Yet this more conservative approach turned out to offer plenty of work while remaining achievable. I have learnt, therefore, to override my instincts when it comes to scope and timings, and err on the side of caution.

To further develop the project, I would need some more technical skills, such as an understanding of databases. It is frustrating that I lacked the database skills and the SPARQL knowledge to hook into the OU’s open data system (although as previously discussed, some critical information is missing from that system). This project has opened my eyes to this aspect of computing which has not previously interested me, and I am now intending to take M359. I believe the system could also have been much faster if I had had the ability to design and code it as a multi-threaded system, so this is another technical skill I would like to extend.

## Epilogue

During some last minute tweaking, I tried increasing the number of individuals from 50 to 100, and found that this markedly improved results (see figure 22b). I had not observed this when experimenting with number of individuals before repair operators were implemented.



*Figure 22b: An update of figure 22 to show the superior results of the GA with 100 individuals*

Although the results across 10 runs were not significantly fitter than those for the GA with 50 individuals and 100 generations (Student t Test,  $t = 1.02$ ,  $sd = 101$ ,  $df = 18$ ,  $p = 0.32$ ), this



GA did manage to find a solution of higher fitness than I had ever seen from the system before (see figure 28), and to find it on three runs out of the ten.

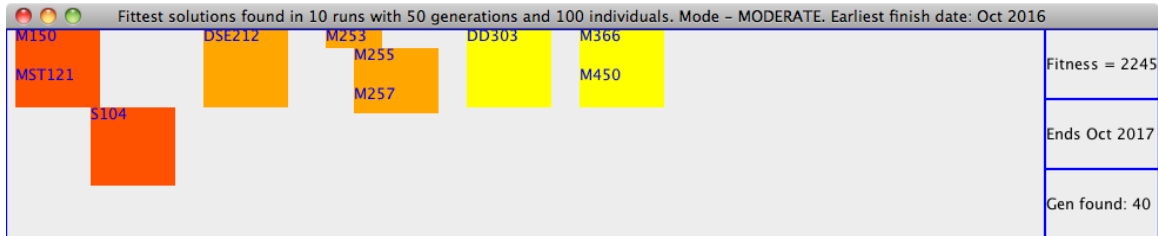


Figure 28: The fittest solution ever found by my system, found using a GA with  $n=100$  and 50 generations.

I believe that perhaps what had been happening is this: when I implemented repair operators, they were actually improving the algorithm’s potential more than I realised, but much of this potential improvement was being wiped out by the side-effect of rendering the search local. This is something Gröbner and Wilke (2001) warn about, and indeed they suggest introducing repair operators only gradually to avoid this problem. I had not heeded this advice, because repair operators seemed to improve my results (if only marginally) so I had assumed this was not happening in my case. However, the fact that doubling the number of individuals leads to such fitness improvement suggests to me that the extra diversity was sorely needed, and that it is successfully helping the algorithm avoid local minima. Against this new benchmark, we can now claim with 99% confidence that a GA finds significantly better solutions than a Hill Climbing algorithm (Student t Test,  $t = 3.86$ ,  $sd = 135$ ,  $df = 18$ ,  $p = 0.0012$ ), and indeed with the improvement being an average of 232 points (around 12%), this may well be considered a difference worth waiting for.

As a result of this finding, I have changed the default settings in my code so that when the system runs in “student mode”, the 100 individual, 50 generation GA is used. As per my tutor’s request, all my code is available – to run or examine – as a Netbeans project, downloadable from [www.lizwade.com/m450/module\\_scheduler\\_netbeans.zip](http://www.lizwade.com/m450/module_scheduler_netbeans.zip)

## References

aathishankaran (2008) “Recursion in Java”, *Java Samples* [online], <http://www.java-samples.com/showtutorial.php?tutorialid=151> (Accessed 2 Aug 2011).

Bloch, Joshua (2008) *Effective Java (2nd Edition)*, Prentice Hall

Brailsford, S.C., Potts, C.N. and Smith, B.M., (1998) *Constraint Satisfaction Problems: Algorithms and Applications*, Working Papers, University of Southampton - Department of Accounting and Management Science.

Carey, M.R. and Johnson, D.S. (1979) “Computers and intractability – a guide to the theory of NP-completeness” cited by Cartwright (1994)

Cartwright, Hugh (1994) “Getting the Timing Right – The Use of Genetic Algorithms in Scheduling” in *Applications of Modern Heuristics* (1994) Ed V.J.Rayward-Smith

Fang, Hsiao-Lan (1994) *Genetic Algorithms in Timetabling and Scheduling* (PhD Thesis), Department of Artificial Intelligence, University of Edinburgh

Gen, M. and Cheng, R. (2000) *Genetic Algorithm and Engineering Optimisation*, New York, US: John Wiley and Sons.

Gröbner, Matthias and Wilke, Peter (2001) “Optimizing Employee Schedules by a Hybrid Genetic Algorithm” in *Proceedings of the EvoWorkshops on Applications of Evolutionary Computing*, Egbert J. W. Boers, Jens Gottlieb, Pier Luca Lanzi, Robert E. Smith, Stefano Cagnoni, Emma Hart, Gunther R. Raidl, and H. Tijink (Eds.). Springer-Verlag, pp.463-472.

Mitchell, Melanie (1996) *An Introduction to Genetic Algorithms*, Harvard, US: MIT

Murakami, K., Tasan, S., Gen, M., Oyabu, T (2010) ‘A solution of human resource allocation problem in a case of hotel management’, *Computers and Industrial Engineering (CIE)*, 2010 40th International Conference on , vol., no., pp.1-6, 25-28 July 2010

Muthusamy, K, Chin Sung, S, Vlach, M, & Ishii, H (2003) 'Scheduling with fuzzy delays and fuzzy precedences', *Fuzzy Sets & Systems*, 134, 3, p. 387, Academic Search Complete, EBSCOhost, viewed 28 February 2011.

Nammuni, K, Levine, J. and Kingston, J. (2002) ‘Skill-based Resource Allocation using Genetic Algorithms and Ontologies.’ Proceedings of the International Workshop on Intelligent Knowledge Management Techniques (I-KOMAT 2002) held at the 6th International Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES 2002), September 2002, IOS Press, Amsterdam.

Open University (2007) M366 *Natural and Artificial Intelligence*, Block 5, ‘Evolutionary Computing’, Milton Keynes, The Open University.

Open University (2011) *How much time does it take?* [online], <http://www8.open.ac.uk/study/explained/study-explained/building-your-qualification/how-much-time-does-it-take> (Accessed 15 Sep 2011).

## Appendix 1:

## Javadoc – Class Summary

*Package m450*

<b>Class Summary</b>	
<b>Breeder</b>	Performs genetic operations on Genome objects
<b>FitnessJudger</b>	Judges the Schedule object produced from a Genome object, and assigns an appropriate fitness score to the Genome object
<b>GeneticAlgorithmRunner</b>	Drives a Genetic Algorithm to generate a solution
<b>Genome</b>	Simulates a chromosome, which encodes a possible Schedule
<b>HillClimbingAlgorithmRunner</b>	Drives a Hill Climbing algorithm to generate a solution
<b>Initializer</b>	Initializes the system for a student user, with default parameters for the default GA
<b>Main</b>	Starts the program in either Scientist Mode or Student Mode.
<b>Module</b>	Models an OU Module
<b>ModuleBank</b>	Reads in relevant data for modules offered by the OU, and uses it to create and hold Module objects
<b>Output</b>	Displays the recommended schedules for the student user
<b>PopRecord</b>	Holds a record of the key statistics for a population at a particular stage, including which generation the population is at, how many milliseconds have elapsed since the algorithm began, the fitness score and the bitString of the fittest member, and the mean fitness score of the entire population.
<b>Population</b>	Holds all the Genomes in one generation
<b>PopulationVisualiser</b>	Provides an on-screen visual representation of a Population object
<b>RouletteSelector</b>	Responsible for creating the next generation of genomes, based on the previous generation, using a biased roulette

	wheel
<b>Schedule</b>	A possible solution, decoded from the Genome object into a series of months, some of which hold start dates for modules
<b>ScheduleVisualiser</b>	Provides a visual on-screen representation of a Schedule object
<b>StatCollector</b>	Performs runs and collects statistics
<b>StatVisualiser</b>	Provides a visual on-screen representation of a the results of a series of runs
<b>Student</b>	Models the preferences of a student, such as which modules they wish to study and the max monthly hours they want to spend studying It is used by the system to create and evaluate Schedules
<b>StudyMonth</b>	Models a calendar month, in which a module may start or be in progress
<b>TournamentSelector</b>	Responsible for creating the next generation of genomes, based on the previous generation, using Tournament selection

## Enum Summary

<b>AlgorithmType</b>	
<b>Mode</b>	
<b>SelectionType</b>	
<b>VisType</b>	

## Package UI

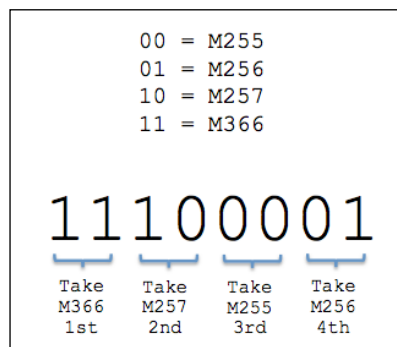
## Class Summary

<b>AlgorithmChooser</b>	A simple interface that lets the computer scientist specify the parameters she wants to use.
<b>StudentPreferenceChooser</b>	A simple interface that lets the student specify her preferences for modules, workload and mode(s) for schedule

## Appendix 2:

### Rejected plan for encoding the problem for use with a GA

My first thought was to assign a binary number to each of the modules to be taken, and then assemble them in a bit string that represented a chronology of modules. So for instance, if there were four modules to be taken, M255, M256, M257, M366, they could be represented as 00, 01, 10 and 11 respectively. The bit string 11100001 would therefore represent a schedule where M366 was taken first, followed by M257, followed by M255, and lastly M256 (see figure 25). However, I quickly realized that this would not work, because changes to the bit string would be very likely to introduce duplicate modules, eg a mutation in the example bit string at bit 7 would produce a schedule where M366 was taken twice and M255 was not taken at all. Of course, chromosomes representing schedules with duplicate modules could be penalized or removed, but as there would be many of them, this would be computationally expensive. Additionally, a chronological order for the modules is not really specific enough. For instance, some short modules are offered several times a year. If a chromosome represented a schedule where such a module appeared between two modules with annual February starts, we would not know exactly when it should be taken.

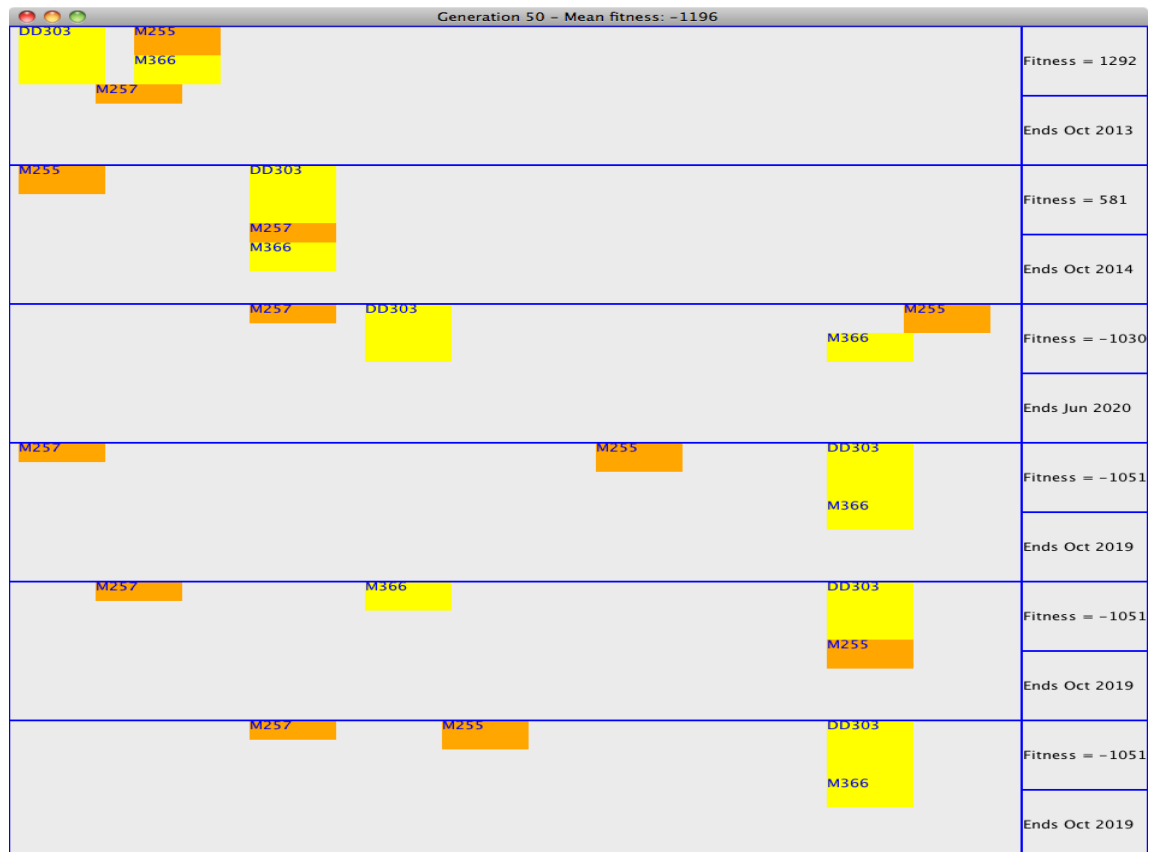


**Figure 25: First (abandoned) plan for GA encoding**

## Appendix 3:

### Fixing my “reverse evolution” bug

In my first test of the system. I used four modules, 50 individuals, and 50 generations. The results were disappointing and confusing. Figure 26 shows the “fittest” 6 members of the population after 50 generations.



*Fig 26: The “fittest” 6 individuals after 50 generations*

As figure 26 shows, the only genomes with non-negative fitness scores are the two preserved by elitism, which in fact were randomly generated in the first generation! It took me longer than it should have done to realize what the problem was. Obviously, I had noticed that genomes had negative fitness, but, reasoning that a negative number is still a perfectly good description of their relative fitness vs other genomes in the population, I didn't worry about it. Eventually it dawned on me that negative numbers were biasing the Roulette Wheel in the wrong direction. The wheel assigns slices according to the genome's fitness divided by the total fitness. This means that if the total fitness is negative, genomes with non-negative fitness get a “negative slice” of the Roulette Wheel (effectively, none), while those with negative fitness get a slice proportional to how bad they are! This explains why the

population became less fit with every generation. Clearly, the proximate cause of the error was that the arbitrarily chosen value of 1000, from which the large length penalty was being subtracted, was too small to keep the scores positive. Rather than just pick another arbitrary number, which might resolve this situation but cause a bug in the next one, I altered the code to ensure that a fitness score could not be negative (any fitness score calculated as negative is now recorded as 1). This is not ideal, as it classes a great number of low fitness individuals equally, failing to preserve their relative fitness. A better long term solution would be to change the system so that the fitness score is represented as the number of penalty points, and individuals with lower scores are considered to be more fit (as used in many studies, eg Gröbner and Wilke 2001, and Nammuni et al 2002).

## Appendix 4:

### Raw data for GA versus Hill Climbing Algorithm

Genetic Algorithm		
run	ms	fitness
1	5322	1222
2	5095	1211
3	4911	1244
4	4503	1306 *
5	4738	1190
6	4812	1240
7	4865	1216
8	4518	1242
9	4871	1256
10	4886	1295
<b>AVE</b>	<b>4852</b>	<b>1242</b>

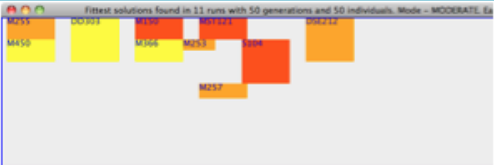


Table 1: Results from 10 runs of the Genetic Algorithm. The fittest solution found is marked by an asterisk and displayed below the data.

Hill Climbing Algorithm		
run	ms	fitness
1	2135	1211
2	1868	1172
3	1812	1172
4	1786	1249
5	1751	1122
6	1764	1184
7	1758	1223
8	1754	1242
9	1759	1267 *
10	1763	1176
<b>AVE</b>	<b>1815</b>	<b>1202</b>




Table 2: Results from 10 runs of the Hill Climbing Algorithm. The fittest solution found is marked by an asterisk and displayed below the data.