Investigating Genetic Algorithms for Scheduling

Hsiao-Lan Fang

MSc Dissertation
Department of Artificial Intelligence
University of Edinburgh
1992

Abstract

This project investigates the use of Genetic Algorithms on the problem of scheduling exam timetables for the AI/CS MSc exams at the University of Edinburgh, with the constraints that the timetables should be as fair as possible on students (avoiding clashes, three exams in a day, and consecutive exams). This problem is very hard to solve because of the extremely large number of possible timetables, and the difficulty of simultaneously satisfying a large number of students. Human experts take many iterations at this task, until over several days of trial and error and consultation with students a timetable is arranged. This project shows that a Genetic Algorithm system can perform this job quickly (in about 5 minutes) and satisfactorily (producing timetables which are measurably better than those produced by the human schedulers). The project describes Genetic Algorithms, the representation used for exam timetables, and also discusses experiments in which ways are studied to improve performance. We find that an *elitist* strategy using *inverse square pressure*, variation of operator rates (crossover and mutation) and fixed-point uniform crossover is a measurably better strategy than traditional Genetic Algorithms on this problem. The system described can be easily adapted for use in similar timetabling problems.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Scheduling Problems

There are many kinds of problems in the domain of Expert Systems: Diagnosis, Monitoring, Interpretation, and Scheduling etc... Scheduling problems are particularly challenging for artificial intelligence and expert systems. They are problems of time-based planning and combinational optimization which tend to be solved with a cooperation of search and heuristics to get optimal or near optimal solutions.
Why are scheduling problems so difficult?

1. They are computationally NP-complete problems[E. G. Coffman 76][Davis 91]. This means that optimal solutions (or near optimal solutions) cannot be obtained within polynomial time. Also, the space of possible solutions is far too large for brute-force or undirected search methods to be feasibly applied.

2. Advanced search techniques using various heuristics to prune the search space will not be guaranteed to find an optimal (or near optimal) solution.

3. Scheduling problems are often complicated by the details of a particular scheduling task. Algorithmic consideration of these specific constraints must often be embodied in domain-specific knowledge.

4. Real world scheduling problems often involve constraints that cannot precisely be represented or even precisely stated.

1.2 Genetic Algorithms

Genetic Algorithms (GAs) are a group of methods which solve problems using algorithms inspired by the processes of Darwinian evolution. In Genetic Algorithms, the performance of a set of candidate solutions to a problem (called 'chromosomes') are evaluated and ordered, then new candidate solutions are produced by using the best candidates as 'parents' and applying mutation or crossover (combining bits of two parents to produce one or more children) operators. The new set of candidates is then evaluated, and this cycle continues until an adequate solution is found.

1.3 Applying Genetic Algorithms to Scheduling Problems

Scheduling problems are highly computationally complex problems which often have highly domain-dependent features (ie: a particular timetabling problem will be complicated by details of the particular domain). They thus tend to need highly domain specific optimization algorithms. Genetic Algorithms, however, provide a way of separating the optimization algorithm from domain knowledge which has led to promising and useful results in the areas of scheduling and timetabling, as well as several other areas. The trick Genetic Algorithms use is to separate a roughly domain independent and very powerful optimization technique (mutation and crossover operators applied to populations of chromosomes) from the domain-specific aspects of a problem (the evaluation function for the chromosomes).
A timetabling problem is a kind of scheduling problem, in which events (exams, classes etc...) have to be arranged into a number of time slots, subject to various constraints. The need for powerful methods for solving a large timetabling problem is plain by considering the simple fact that with, say, e exams to be fitted into t time slots, there are t^e possible candidate timetables, which vary in optimality according to what exams particular students are taking in any particular year. In the 91/92 AI/CS MSc exam timetabling problem, this is 28^{44}, which is approxi-

mately 10^{46}. The performance of random search methods on such a problem will typically be a function of how much of the space of solutions are actually *good* solutions; typically, this ratio is extremely low, so that looking randomly for a good timetable is like looking for a needle in a haystack. Classical heuristic search-based techniques often perform adequately on scheduling/timetabling problems, but in a problem like this one the size of the search space of solutions is such that we can expect classical heuristic search-based techniques to be very time consuming. Classical heuristic search can be seen as very similar to the method used by human experts. In same way, it can be expected to find local minima often. The main aim of this project was to investigate the powerful techniques of Genetic Algorithms on this sort of problem.

1.4 Scope

The aim of this project is to investigate the use of Genetic Algorithms in a hard timetable scheduling problem. The particular timetabling problem suggested is that of arranging an exam timetable, subject to the constraints that an individual student should not have more than two exams in one day, if possible (and certainly not more than one exam at the same time!) and should have exams spread out as much as possible, etc...

In this project, we used real AI/CS MSc exam data to test the program. Furthermore, the structure of the final implementation makes it simple to use for similar kinds of problem, such as scheduling lecture courses, or exam timetables for different sets of students in different department. No changes need be made to the program in these cases; the user only needs to modify certain data files, as described in chapter 5.

1.5 Methodology

The project proceeded in roughly the following way:

1. The essential details of the AI/CS timetabling problem were constructed, aided by correspondence with Drs Robert Fisher and Alan Smaill.

2. Methods were investigated and relevant literature was examined in order to determine an effective scheme for representing timetables as 'chromosomes' for use in a genetic algorithm.

3. A scaled down version of the problem was implemented in *Prolog*[1] to examine the feasibility of the project idea.

4. A large scale implementation was written in *C*, having the ability to experiment with various parameters and aspects of Genetic Algorithms.

5. A large set of experiments were performed in order to:

   (a) assess the performance of a straightforward Genetic Algorithms on real MSc exam timetabling data (for the years 90/91 and 91/92).

   (b) examine ways of improving this performance (by varying aspects of the evaluation function, varying the type of crossover operator used, and varying the selection method for generating new populations).

   (c) gain insight into how the Genetic Algorithms should be set up, in relation to different sizes and types of timetabling problem.

[1]This program is included in appendix J.

1.6 Environment

Implementation will work in the following environment:

- UNIX operating system
- C compiler
- X windows(optional)
- LaTeX(optional)

1.7 Result

This project is a general investigation on the application of Genetic Algorithms to a hard scheduling problem. There are two main aspects of the results achieved in this investigation:

- It has been shown that straightforward Genetic Algorithms can solve the AI/CS MSc exam timetabling problem quickly and satisfactorily. The program which does this is easy to run and it is easy to encode the exam and student data for any particular year, and so one result of this project is a useful, usable tool for generating good exam timetables. As we will see in chapter 7 , the timetables generated(quickly) by the system are better than those produced by the AI/CS MSc course organizers for the 90/91 and 91/92 timetabling problems. The latter were typically the result of weeks of revision and redrafting.

- Experiments with different Genetic Algorithm parameters and crossover operators lead to results which show how best to set up and control the Genetic Algorithm given a particular timetabling problem. We find in particular for the AI/CS MSc exam timetabling problem that the use of fixed-point uniform crossover and an elitist strategy using inverse square pressure and variation of operator rates is much better than simple Genetic Algorithms on this problem.

1.8 Reader's Guide

The contents of the project have been arranged to be read chapter by chapter. However if the reader is already familiar with Genetic Algorithms, he or she can skip chapter 2 and chapter 3 and go to chapter 4 directly. The contents of each chapter are as follows:

- Chapter 1 gives a general introduction of the project.
- Chapter 2 introduces Genetic Algorithms, including a worked sample example of Genetic Algorithms on a timetable scheduling problem.
- Chapter 3 describes the theoretical foundation of Genetic Algorithms and presents the mathematical results which indicate why Genetic Algorithms are so powerful.
- Chapter 4 describes a real case of an MSc exam timetable problem.
- Chapter 5 provides a user manual of how to use the program.
- Chapter 6 provides details of the implementation.
- Chapter 7 describes experiments with different parameters.
- Chapter 8 provides conclusions and suggestions for further research.

Chapter 2

Introduction to Genetic Algorithms

Every living thing consists of one or more cells, but beyond that, there are exceptions to just about every rule in biology. Nevertheless, certain characteristics of heredity and evolution seem to be universal or nearly so. Darvin champions the principle of evolution through natural selection. The following characteristics are the essence of "Survival of the Fittest".[Winston 92]

1. Each individual tends to pass on its traits to its offspring
2. Nevertheless, nature produces individuals with different traits
3. The fittest individuals – those with the most favorable traits – tend to have more offspring than do those with unfavorable traits. This drives the population as a whole towards favorable traits.
4. Over a long period, variation can accumulate, producing entirely new species whose traits make them especially suited to particular ecological niches.

In the nature of genetics, genes which adapt better are selected for producing their children. As a result, the children of these better parents inherit genetic traits from both of their parents. Genes of the new generation which adapt well are also selected for mating and reproducing. The evolutionary cycle continues from generation to generation. Therefore, poor chromosomes disappear in the environment and better chromosomes reproduce good children and the children perform better too. Finally, the survival of the fittest population will have adapted optimally or near optimally to the environment.

2.1 What are Genetic Algorithms?

The best description of Genetic Algorithms I have ever found is in Goldberg's "Genetic Algorithms in Search, Optimization & Machine Learning"[Goldberg 89], who says:(P1)

> Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine "Survival of the fittest" among string structures with a structure yet randomize information exchange to form a search algorithm with some of the innovative flair of human search. In every generation, a new set of artificial creatures (strings) is created using bits and pieces of the fittest of the old; an occasional new part is tried for good measure. While randomized, genetic algorithms are no simple random walk. They efficiently exploit historical information to speculate on new search points with expected improved performance.

How are Genetic Algorithms different from other optimization and search methods?
Again [Goldberg 89] says they are different in four ways:(P7)

- Genetic Algorithms work with a coding of the parameter set, not the parameters themselves.
- Genetic Algorithms search from a population of points, not the parameters themselves.
- Genetic Algorithms use payoff (objective function) information, not derivation or other auxiliary knowledge.
- Genetic Algorithms use probabilistic transition rules, not deterministic rules.

2.2 History of Genetic Algorithms

The initiator of research in Genetic algorithms was John Holland. He published the book "Adaptation in Natural and Artificial Systems"[Holland 75] and from then on some dissertations and papers began to be published by different researchers. From 1985 the general approach began to receive wide attention. Two formal conferences on Genetic Algorithms: "**Proceeding of the International Conference on Genetic Algorithms and their Applications**"[Grefenstette 85][Grefenstette 87] [Schaffer 89][Belew & Booker 91] are held every two years from 1985 and the theoretical conference of "**Foundations of Genetic Algorithms and Classifier Systems**[1]"[Grefory 91] are held every two years from 1990. There are also many papers published in Machine Learning, Neural Networks and other Artificial Intelligence conferences and magazines.

2.3 Outline of Genetic Algorithms

Genetic Algorithms solve a problem by, roughly, generating, changing and evaluating candidate solutions to that problem. A candidate solution to a problem is called a *chromosome*, and a chromosome is usually a bit string or some other encoding or representation of a solution. Eg, an eight bit binary string can be a chromosome representing numbers(from 1 to 128).
Initially, a random population of such chromosomes is generated. Changing chromosomes is done by mutation and/or crossover operators(described below), while chromosomes(candidate solutions) are evaluated by way of a domain dependent fitness function, which first decodes the chromosome and then evaluates its optimality as a solution to the particular problem being addressed.

The outline of traditional Genetic algorithms is as follows:

[1]Defn: Classifier System is a Genetic-based machine learning system[Holland *et al* 86] [Davis 87][Goldberg 89][Shavlik & Dietterich 90][Holland 92].

1. Initialize and encode a random population of chromosomes.

2. Decode and evaluate each chromosome's fitness in the population.

3. Reproduce a new generation by stochastically selecting current chromosomes as parents according to fitness to generate new children.

4. Apply crossover and mutation operators to the new chromosomes.

5. Repeat 2-4 until an adequate solution is found.

Different researchers may implement the algorithm in slightly different ways but the outline is almost always the same. The difference between Genetic Algorithms and other search algorithms is that they separate domain-specific knowledge from domain independent knowledge. Unlike other algorithms, with Genetic Algorithms only the encoding, decoding method and evaluation function are domain dependent; whereas the search technique itself (crossover and mutation operators) is domain independent. Therefore, the system designers can concentrate their time and intelligence on the domain-dependent parts and let the Genetic Algorithms cope with the other parts. It can save people much effort compared with other optimization and search methods.

2.4 Initialization

There are many ways to initialize and encode the initial generation. Binary vs non-binary, fixed vs variable length strings. The traditional Holland's encoding method is as a binary-fixed length string. At the initial stage, the system just randomly generates bits of each chromosome and encodes all the chromosomes as a population, then decodes and evaluates the chromosomes for using in reproduction and selection of the first generation.

2.5 Reproduction and Selection

The most common method for parents selection is Roulette Wheel Parent Selection. The algorithm of Roulette Wheel Parent Selection is as follows:

1. sum the fitness of all the population members. For example: if we have chromosomes c1, c2, c3 and c4, with fitness 10, 4, 9 and 7 respectively, then their sum is 30.

2. generate a random number between 0 and 1, and multiply by the sum of the fitnesses. For example: we might generate the number 0.7, and multiply by 30 to get 21.

3. return the first population member whose fitness, accumulated from the first fitness of the population, is greater than or equal to the random number generated by the second step. For example: making the cumulative sum of fitness, we first pass the number 21 when we add on c3 (c1+c2=14, c1+c2+c3=23), we therefore return c3 as a chosen parent for the next generation.

The effect of Roulette Wheel Parent selection is to return a randomly selected parent. Although this selection procedure is random, it does not imply Genetic Algorithms is a directionless search. The chance of each parent being selected is directly proportional to its fitness.

2.6 The Crossover Operators

The crossover operator is the most important operator in Genetic Algorithms. Crossover is a process yielding recombination of bit strings via an exchange of segments between pairs of chromosomes. There are many kinds of crossovers, here we just illustrate the concepts of four kind of crossovers, and the examples will be given in section 2.9.3 of this chapter.

2.6.1 One-point Crossover

The procedure of one-point crossover is as follows:

1. randomly generate a number(less than or equal to the chromosome length) as the crossover position.

2. keep the bits before the number unchanged and swap the bits after the crossover position between the two parents.

One-point crossover is inspired by biological processes.

2.6.2 Two-point Crossover

Two point crossover is similar to one-point crossover except that we must select 2 positions and only the bits between the two positions are swapped. This crossover can preserve the first and last parts of a chromosome and just swap the second part. This can improve the performance of one-point crossover because sometimes we want to keep head and tail unchanged and only swap the middle parts. In one-point crossover situation, if we do the crossover operation, the head and tail should be placed in different children because one crossover only have two parts and the two parts will be separated after crossover.

2.6.3 N-point Crossover

N point crossover is similar to one-point and two-point crossover except that we must select n positions and only the bits between odd and even crossover positions are swapped. The bits between even and odd crossovers are unchanged.

2.6.4 Uniform Crossover

The procedure of uniform crossover is as follows:

1. randomly generate a number(n) indicating how many positions to swap.

2. randomly generate n numbers say P_1, P_2 ... P_n indicating which n positions to swap.

3. for each P_i, swap the P_ith bit of one parent with the P_ith bit of the other parent, thus generating two new children.

2.7 The Inversion Operator

In Holland(1975) there are three operators for causing children to be different from their parents: *crossover*, *mutation*, and *inversion*. Inversion operates as a kind of reordering technique. It operates on a single chromosome and inverts the order of the elements between two randomly chosen points on the chromosome. While this operator was inspired by a biological process, it requires additional overhead. Though there are some researchers who have investigated the operator, it has not produced good results in Genetic Algorithms in practice. Therefore, we don't use this operator in our timetable scheduling problem.

2.8 The Mutation Operator

Mutation is used to insure that all possible chromosomes are reachable. For example, if the first position in a chromosome can be any number from one to twenty, it may happen that in the initial population there is no chromosome with "15" in the first position. With only crossover operators, it is impossible to generate such a chromosome. Even with inversion operators, the search is constrained to alleles (possible variants of genes) which exist in the initial population. This ensures that the selection process does not get caught in a local minimum because it is possible that use of crossover and the inversion operator will only produce a set of bit strings that are better than all local neighbors but not optimal in a global sense. This can happen since crossover and inversion may not be able to produce undiscovered bits. The mutation operator can overcome this by simply randomly selecting any bit position in a string and changing it. There is a slight difference between Goldberg and Davis in the definition of this operator.

Student	*Examination*
$s1$	e1,e2,e4,e6
$s2$	e1,e2,e4,e5
$s3$	e1,e2,e4,e5
$s4$	e1,e2,e5
$s5$	e4
$s6$	e4,e5
$s7$	e2,e4
$s8$	e2,e4,e6
$s9$	e3,e4,e6
$s10$	e3,e4,e6

Table 2–1: Example of student-exam data

In [Goldberg 89], once we decide to apply the mutation operator , the bit will unconditionally change from 0 to 1 or vice versa. But in [Davis 91], the changed value will depend on a random bit generator, so it is possible that there will be no effective change and it could be form 0 to 0 or 1 to 1.

2.9 A Simple Example to Illustrate Genetic Operators

Let's assume that there are ten students(s1..s10), six exams(e1..e6) and two days (altogether eight time slots(t1..t8)) for all the six exams. Each day has four time slots with two in the morning and two in the afternoon. Each student takes different numbers of exams as in Table 2–1 where student 1 takes exams 1, 2, 4, 6, student 2 takes exams 1, 2, 4, 5 and so on. The timetable slots before scheduling are as in Table 2–2:

The constraints are as follows:

1. The same student can't take two different exams at the same time slot.

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
day1	t1	t2	t3	t4
day2	t5	t6	t7	t8

Table 2–2: Example of empty time slots

day	*AM*		*PM*	
	time1	time2	time3	time4
day1			e1	e3
day2		e5	e6	e2,e4

Table 2–3: Assigning exams to time slots

2. The same student can't take three different exams on the same day.

3. We must strongly avoid the same student taking two different exams in the morning or in the afternoon of the same day.

4. We must avoid the same student taking two different exams consecutively in the morning and in the afternoon of the same day.

In the following sub-sections, we will see how a Genetic Algorithm solves this simple timetable scheduling problem:

2.9.1 Initialization

We use a non-binary bit string representation to represent the chromosomes here, because it is easy to understand and represent. Our example has six positions representing six exams with each position's value as the time slot assigned to the exam. We can generate the population randomly to assign each exam a time slot. If we randomly generate six number 3, 8, 4, 8, 6, 7 as six time slots for e1–e6, then the chromosome is *3 8 4 8 6 7* and the timetable is as in Table 2–3.

Index	*Fitness*	*Chromosome*	*Accu. fitness*
1	0.005	3 8 4 8 6 7	0.005
2	0.062	*7 3 7 6 1 3*	0.067*
3	0.006	5 3 5 5 5 8	0.073
4	0.020	7 6 7 7 2 2	0.093
5	0.040	*1 7 4 5 2 2*	0.133*

Table 2–4: Initial population and fitness

2.9.2 Reproduction and Selection

In Table 2–4, if we randomly generate a number 0.5 and multiply by the sum of fitnesses 0.133, then get an accumulated fitness 0.0665 which is what we expect to select among our parents. We consult the table above to get the accumulated fitness 0.0665 is between the first accumulated fitness 0.005 and the second accumulated fitness 0.067, so we select the second chromosome *7 3 7 6 1 3* as our first parent. Similarly using the same method to get another accumulated fitness 0.095, we select the fifth chromosome *1 7 4 5 2 2* as our second parent.

2.9.3 Crossover

one-point crossover: With the two parents selected above, we randomly generate a number 2 as the crossover position:

parent1: 7 3 $_\wedge$ 7 6 1 3
parent2: 1 7 $_\wedge$ 4 5 2 2

then we get two children:

child1 : 7 3 ||4 5 2 2
child2 : 1 7 ||7 6 1 3

where the first child has the first part from parent1 and the second part from parent2 and the second child has the first part from parent2 and the second part from parent1.

two-point crossover: With the two parents above, we randomly generate two numbers 2 and 4 as crossover positions:

parent1: 7 3 $_\wedge$ 7 6 $_\wedge$ 1 3
parent2: 1 7 $_\wedge$ 4 5 $_\wedge$ 2 2

after crossover, we get two children:

child1 : 7 3 ||4 5|| 1 3
child2 : 1 7 ||7 6|| 2 2

uniform crossover: If we randomly generate a number 3, then randomly generate 3 numbers 2, 3, 5 as crossover positions:

parent1: 7 $_\wedge$3 $_\wedge$7 6 $_\wedge$1 3
parent2: 1 $_\wedge$7 $_\wedge$4 5 $_\wedge$2 2

after crossover, we get two children:

child1 : 7 7* 4* 6 2* 3
child2 : 1 3* 7* 5 1* 2

2.9.4 Inversion

Here we just give a simple example of pure inversion:
If we randomly choose two positions 2, 5 and apply the inversion operator:

$$3 _\wedge 8\ 4\ 8\ 6 _\wedge 7$$

then we get the new string:

3 6 8 4 8 7
 * * * *

The inversion operator only has one parent.

2.9.5 Mutation

Assume we have already used crossover to get a new string:

7 3 4 5 1 3
↑

Assume the mutation rate is 0.001(usually a small value). Next, for the first bit 7, we randomly generate a number between 0 and 1. If the number is less than the mutation rate(0.001), then the first bit 7 needs to mutate. We generate another number between 1 and the maximum time slot(8), and get a number(for example 2). Now the first bit mutates to 2. We repeat the same procedure for the other bits. In our example, if only the first bit mutates, and the rest of the bits don't mutate, then we will get a new chromosome as below:

2 3 4 5 1 3
*

Because the mutation rate is very small, each chromosome only has a 0.001*6 = 0.006 possibility of mutation. It takes on average about 167 chromosomes to have one bit mutation.

Chapter 3

The Fundamental Theorem of Genetic Algorithms

In this chapter we describe the fundamental theorem of Genetic Algorithms, called the schema theorem. It is not a part of this project to investigate this theorem as such, or do any kind of formal analysis of Genetic Algorithms.(rather, this project is about on implementation and experimental analysis of the use of Genetic Algorithms). However, this project(like many others) demonstrates the extraordinary and surprising power of Genetic Algorithms, where simple, genetically-inspired search operators seem to quickly solve many difficult problems. The schema theorem, and associated mathematical results about Genetic Algorithms, particularly the mathematical analysis of building blocks and implicit parallelism, go a long way along the route of describing just why Genetic Algorithms are so powerful. I therefore include this chapter in my report, to help explain why the methods and experiments described work so well. The mathematical descriptions of Genetic Algorithms that we use in this chapter is adapted from [Spiessens 88].

3.1 Implicit Parallelism

A schema (Holland 75) is a similarity template describing a subset of strings with similarities at certain string positions. Why are schemata important for Genetic Algorithms? [Goldberg 89] says:

> "In some sense we are no longer interested in strings as strings alone. Since important similarities among highly fit strings can help guide a search, we question how one string can be similar to its fellow strings. Specifically we ask, in what ways is a string a representative of other string classes with similarities at certain string positions? The frame of schema provides the tool to answer these question."

A schema is a string of the following format:

$$\# \; 1 \; 0 \; 1 \; \# \; 0$$

with each position a 1, 0 or #[1]. The more # it has, the less specific it becomes. In a binary string, if there are n '#', then a schema can describe *2^n strings*. In our example there are 2 '#', therefore it can describe four strings:

$$\begin{array}{l} \mathit{0}\;1\;0\;1\;\mathit{0}\;0 \\ \mathit{0}\;1\;0\;1\;\mathit{1}\;0 \\ \mathit{1}\;1\;0\;1\;\mathit{0}\;0 \\ \mathit{1}\;1\;0\;1\;\mathit{1}\;0 \end{array}$$

The total number of different *schemata* of length n is $(2+1)^n$, because each of the n positions can be 0, 1 or #. The total search space remains 2^n since

---

[1]Defn: # is a don't care symbol, can be 1 or 0

the '#' symbols are not involved in the real string processing; they only serve as a tool to describe similarity templates. How many schemata are processed usefully in each generation? Despite the processing of only, say, c chromosomes each generation, Genetic Algorithms process something like c^3 schema. Holland calls this *implicit parallelism* and the observation is a major part of the explanation of the robustness of Genetic Algorithms. Even though we perform computation proportional to the size of the population, we get useful processing of something like c^3 schema in parallel with no special book keeping or memory other than the population itself.(for the detailed proof of c^3 see [Goldberg 89] p40-41)

3.2 Reproduction and Schemata

The frequency m(H,t) of a schema H at generation t will change at generation $t+1$ proportionally to the selection probability of strings for reproduction. There are two steps to consider. The first step is the selection step, in which we choose from the generation at time t the chromosomes that will be the parents for the generation at time $t+1$. Let $\mathrm{m}^P(H,\mathrm{t})$ represent the frequency of schema H among the *parents* selected to produce generation $t+1$. In the variation step, which we will consider later, we crossover these parents with each other to produce the new generation $t+1$.

Let f(H) be the average fitness of a string containing schema H in generation t and $\bar{f}$ represent the average fitness of the entire population. We can derive the following formula in equation 3.1

$$m^P(H,t) = m(H,t) * f(H)/\bar{f} \tag{3.1}$$

This simply states that the number of instances of schema H which will be parents of the generation $t+1$ is equal to the number of instances of schema H in the generation t, multiplied by how much more fit schema H is than the average fitness. The equals sign is not actually correct, but because the selection step is a stochastic process we can say that the equals sign becomes more and more true as the size of the population gets larger.

It is clear from this simple first analysis that good schemas, or building blocks, become more and more frequent in later generations. In a similar way, bad schemas die off. Implicitly, a genetic algorithm processes about c^3 schemata of this sort all the time (where c is the number of chromosomes in a generation).

We must now look at the effect of the variation step. Without the variation step (crossover, and mutation operators), we would only get an algorithm which continues to work, according to equation 3.1, until all of the chromosomes are the same, which is the one that happened to be the best one in the first generation. Without the variation step, we never get new chromosomes, and so no new schemata which were not in the first generation anyway. However, does the variation step badly affect the property of the selection step? In other words, the selection step ensures that schemata with above average fitness increase in number. The variation step (crossover) is necessary in order to have new possibly good schemata, and to combine together previous good schemata. However, does the variation step *destroy* too many good schemata, so that the effect of equation 3.1 is decreased?

In the following section, we will consider the crossover operation, and study the variation step to see if this is true.

3.3 Crossover and Schemata

In the variation step, we can assume that when a schema H is crossover with another schema H, the schema H is not destroyed. We will therefore just consider how often it might happen that a schema H is crossover with a schema different from H. This only means that H is destroyed (probably) when the crossover position cuts the schema (eg: if 111111 is crossover with 000000 at position 3, then the result is the chromosomes 111000 and 000111. But if 11#### is a good schema, then we can see that this schema is not destroyed. However, the schema 1111## is destroyed, because the crossover position cuts into this schema). So we must find the probability that an instance of schema H will be crossover with a chromosome that will destroy the schema. This is the combination of the probability of crossover, the probability of another chromosome not having an instance of schema H, and the probability that the crossover position cuts into schema H.

If P_c is the probability of a crossover, and the defining length $\delta(H)$ is the distance between the first and last specific schema positions of H, and l is the total length of a string then the probability of a schema being disrupted because of crossover is as in equation 3.2

$$P_c * \delta(H)/(l-1) \tag{3.2}$$

The probability that an instance of schema H is crossed with a schema *different* from H is in equation3.3:

$$1 - m^P(H,t)/c \tag{3.3}$$

where c is the number of chromosomes in the population. If we multiply the probabilities from equation 3.2 and equation 3.3, we get the probability that schema H will be destroyed by the crossover operation. This is as in equation 3.4

$$(1 - m^P(H,t)/c) * P_c * \delta(H)/(l-1) \tag{3.4}$$

This is, however, usually a small number. Eg: P_c will be typically about 0.7, while, with a good schema H, the number in the population which are not this schema may be about 0.4, and the probability of cutting the schema will be typically about, say, 0.3 – so the number in equation 3.4 might be usually a small number less than 0.1. Let's call this number ϵ. We can now show equation 3.5(developed from this recent discussion and equation 3.1), which simply gives the number of instances of schema H that we expect in the new population:

$$m(H,t+1) \geq (1-\epsilon) * m(H,t) * f(H)/\bar{f} \tag{3.5}$$

The greater-than-or-equal sign is there because we actually overestimate the amount of schemas H that will be destroyed. This is because it is still possible to produce instances of schema H by crossover of schema H with a chromosome which has no schema H even if the crossover position cuts the schema. Also, it is possible to produce instances of schema H even if not one of the parent chromosomes contains schema *H*. As we can see, since (1 - ϵ) is near to 1, the variation step does not harm the selection step too much. This means we can still be sure of having good schemas increasing in future generations, and bad ones die out, at the same time as producing new good schemas by crossover.

3.4 Mutation and Schemata

The mutation rate is usually very small in normal situations. If the probability of a mutation is P_m and the order of a schema *H*, denoted by o(*H*), is simply the number of fixed positions(number of 0's and 1's in binary string) present in the template, then the probability of a schema being disrupted because of mutation is as in equation 3.6

$$o(H) * P_m \tag{3.6}$$

This is Goldberg-style mutation, for example: if P_m is 0.001 and

H1: # # 1 # 0 #
H2: 0 1 # 1 # 0

we know o(*H1*) is 2 and o(*H2*) is 4. So, there is 0.001*2=0.002 probability of disruption of schema *H1* and 0.001*4=0.004 probability of disruption of schema *H2*. Therefore we know that short order schemata are more likely to survive than long order schemata. Now, we must modify our schema formula to:

$$m(H, t+1) \geq (1-\epsilon) * (1 - o(H) * P_m) * m(H,t) * f(H)/\bar{f} \tag{3.7}$$

So, the effect of the selection step is only affected by the amount (1 - ϵ)*(1 - o(*H*)*P_m), which is an overestimate of how much crossover and mutation destroy instances of schema *H*. Now, we can conclude that highly fit, short defining length, lower order schemata are strongly propagated generation to generation.

This propagation of good schemata is driven by the stochastic selection according to high fitness, and the above equation 3.7 means that we can crossover and mutate, and so produce new and possibly much better schemata, with only small change to this effect (because the amount of destruction of good schema is usually very small). This is a very important conclusion in Genetic Algorithms and it has a special name : *the Schema Theorem* or *the Fundamental Theorem of Genetic Algorithms*.

The Schema Theorem indicates some necessary conditions for utilizing the implicit parallelism. For fully using the schema information, disruption of above average fitness schemata should be minimized. Highly fit schemata of low defining length and low order are so important in the field of Genetic Algorithms that they have a special term: *Building Blocks*. Genetic Algorithms seek near optimal performance through the evolution of short, low-order, high-performance schemata, or Building Blocks.

We can conclude that the Genetic Algorithms used in this project work so well because, as with all Genetic Algorithms, there are good short building blocks in the chromosomes population which are exploited by the Genetic Algorithm. For Example, a building block might be a schema which positions four large exams on the same day which have no students in common, or one which puts the three most popular exams, which nearly all students take, on different days.

Chapter 4

Domain Analysis

4.1 Problem Description

A general timetabling problem is one where events(e1,e2...) have to be performed at specific times(t1,t2...). In a particular problem, there are constraints on whether certain events can appear at the same time, close together, etc... In the AI/CS MSc exam timetabling problem, the events are exams, and the times are 24 or 28 separate time slots(four per day for six days in 90/91 and seven days in 91/92). We apply Genetic Algorithms to the yearly MSc exam timetable scheduling problem, optimizing a timetable so as to avoid clashes and maximize individual students' rest period between exams.

4.2 Problem Constraints

The format of actual time slots allocated is shown in Table 4–1.

There are five days in a week for the exams and each day has four time slots, two in the morning and two in the afternoon. The interval within the same half day is half an hour, however the interval between morning and afternoon is one hour. If there are seven days for the exams, then the time slots will be 28 and

TIMETABLE	*AM*		*PM*	
	09:30-11:00	11:30-13:00	14:00-15:30	16:00-17:30
day of week				

Table 4–1: Actual time slots allocation

they must occupy at least two different weeks(if the first day is Friday of the first week, then the last day should be Monday of the third week). The constraints are as follows:

STRONG CONSTRAINTS:

1. The same student cannot take two differents exams at the same time. In other words, the exams cannot clash for any student.

2. Very strongly prefer no more than 2 exams per day for a given student.

WEAK CONSTRAINTS:

1. Strongly prefer not to have exams consecutive on the same half day for the same student.

2. Prefer not to have exams consecutive on different half days of the same day for the same student.

OTHER CONSTRAINTS:

1. Try to spread out the exams across the full period as much as possible.

2. Some lecturers specify that some exams may not happen on specified days or afternoons when they are unavailable. We might try to fill them in specific time slots.

3. The real timetable also includes CS MSc students and AI4 CS4 students. There are also Met. students.

4.3 Problem Representation

When using Genetic Algorithms to solve an optimization problem, the first important step is to choose how to represent a solution of that problem as a chromosome, which can then submitted meaningfully to crossover/mutation operators. The choice I made here is for a chromosome to be an ordered list of numbers, for example, there are ten exams to be scheduled, and eight time slots, thus a chromosome in this problem is a list of length ten(each position in this list represents a particular exam), where each gene in its list can be a number from one to eight, for example, the chromosome: *3 7 2 1 7 7 4 6 5 8*, represents a timetable where exam 1 is in time slot 3, exam 2 is in time slot 7, etc...
Other work on timetabling problems has used a different chromosome representation where, more like the actual time table itself, positions are time slots and the values that a 'position gene' can take are sets of exams that might occur at that time. This kind of representation however leads to a problem called the *label replacement* problem[Abramson & Abela 91]; the problem is that crossover operator can easily lead to producing timetables which leave out many of the exams that need to be scheduled. An algorithm must be used to then modify the chromosome to include all the necessary exams. The analogous situation in our representation is that a chromosome may represent timetables where not all time slots are filled: this is, however, a perfect, legal, and useful timetable. Therefore, using this representation, the Genetic Algorithm does not need to be modified with a computationally expensive label replacement algorithm.

There are a lot of modules during the exams including Artificial Intelligence department and Computer Science department modules. The module numbers, abbreviations and full names in 91/92 are as in table 4–2 and those in 90/91 are included in appendices D, G.

number	*abbrev.*	*course name*
1	aied	AI and Education
2	cvis	Computational Vision
3	ias	Intelligent Assembly Systems
4	isc	Intelligent Sensing and Control
5	es	Expert Systems
6	kri1	Knowledge Representation and Inference I
7	kri2	Knowledge Representation and Inference II
8	lisp	Programming in Lisp
9	mvis	Machine Vision
10	mthr	Mathematical Reasoning
11	nlp	Natural Language Processing
12	prol	Programming in Prolog
13	essp	Special Expert Systems Project
14	clg1	Computational Linguistics 1
15	clg2	Computational Linguistics 2
16	lng1	Linguistics 1
17	lng2	Linguistics 2
18	fsm1	Formal Semantics 1
19	nnet	Neural Networks
20	sp1	Speech Processing 1
21	sp2	Speech Processing 2
22	os	Operating Systems
23	db	Database Systems
24	cpin	Computability and Intractibility
25	swl	Software for Parallel Computers

Table 4–2: MSc 91/92 module information

number	*abbrev.*	*course name*
26	grph	Computer Graphics
27	cad	Computer Aided Design
28	ccom	Computer Communications
29	ssc	Software System Concepts
30	fprg	Formal Programming Language Semantics
31	algc	Algebraic Complexity
32	cca	Concurrent Computer Architecture
33	ds	Distributed Systems
34	cc	Communication and Concurrency
35	aps	Applicative Programming and Specification
36	cafr	Computer Aided Formal Reasoning
37	bsv	Behavioural Specification and Verification
38	si	System Integration
39	asicd	ASIC design
40	spd	Systolic Processing Design
41	ccomp	Computational Complexity
42	rs1	Remote Sensing 1
43	rs2	Remote Sensing 2
44	cp	Cognitive Psychology

Table 4–2: MSc 91/92 module information–continued

Chapter 5

User Interface

In this chapter, we will demonstrate how to use the timetable scheduling system, how to give parameters the system needs and what the output report and timetable look like.

5.1 Program Usage

The program must be used under the UNIX operating system. It was tested on an SUN SPARC workstation, but should be portable without any change to any workstation under UNIX. By the way, after slight changes, it can be run in any operating system providing different kinds of C compiler.

5.2 Data Files

In order to run this program, we must prepare a template file, a student-exams file, and a modules abbreviation file. The format of each file is as follows:

5.2.1 Template of Modules

We can pre-define some exams to be held in certain slots, just by filling in the pre-defined slot number for the relative exam. For example: if we have 44 modules with the *fifth* module being set in time slot *8* and the *seventh* module being set in time slot *3*, then the template file is as follows:

0 0 0 0 *8* 0 *3* 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0

with un-predefined modules filled with '0' and each number separated by a space.
A module template file is simply a file containing such a list of numbers.

5.2.2 Data of Students-Exams

If we just want to schedule AI or CS department timetables one at a time, then we divide the data into two separate files. Otherwise, we merge these two files into one file leaving no blank line between them.
A students-Exams file is simply a file containing 8 numbers in each row to represent a certain student's eight modules.

AI Department Students: Each row represents a student's eight exam modules, using a space as delimiter.

8 12 6 7 1 5 22 28
8 12 6 7 9 4 3 2
8 6 7 16 14 15 5 23
8 12 6 7 10 19 11 1
8 6 7 16 14 15 5 23
8 12 6 7 10 23 24 29
....
....
....

The complete list is shown in appendix A.

CS Department Students: Each row represents a student's eight exam modules, using a space as delimiter, part time students' modules must be filled with '0' up to eight modules.

28 32 25 33 29 22 23 26
34 30 35 36 31 24 5 12
32 28 25 33 37 35 26 38
....
....
28 32 33 25 0 0 0 0
....
....

The complete list is shown in appendix B.

5.2.3 Data of Modules Abbreviation

A modules abbreviation file is simply a file containing the abbreviation of a module in each row and the order of the abbreviations must be the same as those in the template file described above.

aied
cvis
ias
isc
es
kri1
kri2
lisp
mvis

mthr

nlp

prol

...

...

...

The complete list is shown in appendix C.

5.3 Input Parameters

The user should prepare three data files as described in the last few sections and some parameters must be provided interactively with the system, for example:

- What kind of crossover operator?
- Template file name.
- Student - Examination data file name.
- The number of the time slots(equal to 4 times the number of exam days).
- How many generations during this run?
- How many populations in each generation?
- Any number as a random number seed for this run.
- Modules abbreviation file name.
- Which weekday is the first day of exams?
- LaTeX timetable output file name(for example, xxx.tex).

The procedure to start is very simple, you just type **ga** and hit the **Return** key, then the following menu will show up on the screen:

Exams Timetable Scheduling

==================

Genetic Algorithms crossover selection:

1 One-point crossover.

2 Two-point crossover.

3 Uniform crossover.

4 Fixed-point uniform crossover.

Please type a number here: **1**

Then you choose an appropriate *number* and hit **Return**.
The following questions should be answered one by one interactively. Output from the system is shown here in NORMAL type, and input from the user in **BOLD** type.

Please enter the pre-defined template file: **tprand.91**

0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*

0*0*0*0*0*0*0*0*0*0*0*0*0*0* 44 exams

please enter the student-exams data file: **91.ac**

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........
/......../......../......../ 93 students

How many time slots during the exams: **28**
how many generations : **300**
how many populations in a generation: **50**
Please enter a number as seed: **6**

Now, the system begins to run as shown in the next section.

5.4 Output Report

The output report contains several abbreviations in the header:
the meaning of these are as follows:

- *Index*: the identification of chromosome.
- *err*: the total punishment of this individual.
  - *Type1*: the number of times there are exams consecutive on different half days of the same day. The score of each punishment is 1.
  - *Type3*: the number of times there are exams consecutive on the same half day. The score of each punishment is 3.
  - *Type10*: the number of times there are 3 exams per day for the same student. The score of each punishment is 10.
  - *Type30*: the number of times there are exams happened at the same time, in other words, the exam clashes. The score of each punishment is 30.
- *fitness*: the fitness of this individual calculated by the objective function which will be described in section 6.4.2.

- *pa1+pa2 $_\wedge$cro1$_\wedge$2*: two selected parents(pa1, pa2) and their crossover positions(cro1, cro2)[1]

- *1. 2. 3. 4. 5. 6. 7. 8.*.........: each number represents the position of the chromosome.

After finishing each generation, we produce a temporal report recording statistical information on each chromosome and the maximum, minimum and average fitness of current generation. The format of the reports is as follows:

5.4.1 Initial Generation

After all the parameters are input, the initial generation will be randomly produced as follows:

```
One-point crossover generation 0 begin:
-------------------------------------------------------------.........
Index err fitness    pa1+pa2 ^cro1^2: 1. 2. 3. 4. 5. 6. 7. 8..........
-------------------------------------------------------------.........
  50 3232 0.00000010   0+0    ^ 0^ 0= 25 23 10 13  7 21 26 23..........
        Type1=26     Type3=92       Type10=14   Type30=93
  49 3217 0.00000010   0+0    ^ 0^ 0= 20 18  4 19 21  2  5  4..........
        Type1=37     Type3=70       Type10=18   Type30=93
  48 5262 0.00000004   0+0    ^ 0^ 0= 22  5 13 17 20 13 25 10..........
        Type1=38     Type3=108      Type10=40   Type30=150
......................................................................
......................................................................
......................................................................
-------------------------------------------------------------.........
Index err fitness    pa1+pa2 ^cro1^2: 1. 2. 3. 4. 5. 6. 7. 8. ........
-------------------------------------------------------------.........
One-point crossover generation 0 end!
        min conflicts= 2083     max fitness= 0.00000023
```

[1]one-point crossover:cro1, two-point crossover:cro1, cro2, uniform crossover:* directly above the crossover positions of the chromosome.

```
                                average fitness= 0.00000011
```

The cycle will continue from generation to generation until maximum generation 300 with typically increasing performance in each new generation.

5.4.2 Last Generation:

the last generation of this run is as follows:

```
One-point crossover generation 300 begin:
-------------------------------------------------------------.........
Index err fitness    pa1+pa2 ^cro1^2: 1. 2. 3. 4. 5. 6. 7. 8..........
-------------------------------------------------------------.........
......................................................................
......................................................................
......................................................................
   3    3 0.06250000   2+2   ^44^ 0=  9  3 13 19 25 17  8 16..........
          Type1=0      Type3=1        Type10=0    Type30=0
   2    3 0.06250000   2+2   ^44^ 0=  9  3 13 19 25 17  8 16..........
          Type1=0      Type3=1        Type10=0    Type30=0
   1    3 0.06250000   0+0   ^ 0^ 0=  9  3 13 19 25 17  8 16..........
          Type1=0      Type3=1        Type10=0    Type30=0
-------------------------------------------------------------.........
Index err fitness    pa1+pa2 ^cro1^2: 1. 2. 3. 4. 5. 6. 7. 8..........
-------------------------------------------------------------.........
One-point crossover generation 300 end!
        min conflicts= 3        max fitness= 0.06250000
                                average fitness= 0.04633500
```

Do you want to see the recommended solution? (y/n) **y**

After finishing running all the generations, the system will ask the user if he wants to see the recommended timetable or not. If yes, then a recommended report will be generated with no clashes for any student and, if possible, no student taking three examinations on the same day. If we cannot find any satisfactory solution in this run, the system will recommend the user to try another run using a different seed or parameters. The recommended timetables for this example are as follows:

5.4.3 Recommended Timetables

The recommended timetables are as follows:

```
One-point Crossover optimal Timetable Scheduling begin:
--------------------------------------------------------------.........
Index err fitness    pa1+pa2 ^cro1^2: 1. 2. 3. 4. 5. 6. 7. 8..........
--------------------------------------------------------------.........
.......................................................................
.......................................................................
  36    3 0.06250000  41+50  ^44^ 0=  9  3 13 19 25 17  8 16..........
        Type1=0        Type3=1        Type10=0    Type30=0
.......................................................................
.......................................................................
  44   16 0.00346021  16+22  ^22^ 0=  9  3 13 19 25 17  8 16..........
        Type1=0        Type3=2        Type10=1    Type30=0
.......................................................................
.......................................................................
--------------------------------------------------------------.........
Index err fitness    pa1+pa2 ^cro1^2: 1. 2. 3. 4. 5. 6. 7. 8..........
--------------------------------------------------------------.........
```

Do you want to see the recommended solution? (y/n) **n**

If we decide to use the 36th solution above, then the meaning is that the first module is set in slot 9, the second module set in slot 3 etc... Furthermore, we know that no students take two consecutive examinations one in the morning and the other in the afternoon of the same day or three examinations on the same day, and no students take two examinations at the same time but one student can not avoid taking two consecutive examinations on the same half day.

5.4.4 Preview Timetable on the Screen

We can also preview the timetable on the screen and save it in a file in order to print from a printer later if we answer *y* to the question "Do you want to print the recommanded solution? (y/n)". The procedures are as follows:

Do you want to print the recommanded solution? (y/n) **y**

Please enter the modules abbreviation file: **91.abb**

Please input the solution number: **36**
9 3 13 19 25 17 8 16 12 14 24 21 11 12 2 24 19 26 27 14 13 20 10 5 6 24
23 1 4 20 12 16 8 13 28 18 17 12 25 3 2 18 21 22

```
28 0 0 0 0      15 41 0 0 0     2 40 0 0 0      29 0 0 0 0
24 0 0 0 0      25 0 0 0 0      0 0 0 0 0       7 33 0 0 0
1 0 0 0 0       23 0 0 0 0      13 0 0 0 0      9 14 31 38 0
3 21 34 0 0     10 20 0 0 0     0 0 0 0 0       8 32 0 0 0
6 37 0 0 0      36 42 0 0 0     4 17 0 0 0      22 30 0 0 0
12 43 0 0 0     44 0 0 0 0      27 0 0 0 0      11 16 26 0 0
5 39 0 0 0      18 0 0 0 0      19 0 0 0 0      35 0 0 0 0
```

Please enter the output file: **91.tex**

Which weekday is the first exam day(1-5): **1**

Now, we can preview the timetable as shown in table 5–1 with *Monday* as the first day of the timetable.

After leaving the timetable system, we can use LaTeX to compile the file **91.tex** and then print the timetable from a printer.

	930-1100	1130-1300	1400-1530	1600-1730
Monday	ccom	clg2 ccomp	cvis spd	ssc
Tuesday	cpin	swl		kri2 ds
Wednesday	aied	db	essp	mvis clg1 algc si
Thursday	ias sp2 cc	mthr sp1		lisp cca
Friday	kri1 bsv	cafr rs1	isc lng2	os fprg
Monday	prol rs2	cp	cad	nlp lng1 grph
Tuesday	es asicd	fsm1	nnet	aps

Table 5–1: GAs timetable in MSc 91/92

5.5 Another Sample Run

Test data sets for MSc 90/91 are included in appendices E, F, G.

We can run the program using similar procedures as described above, first, type **ga**, then answer the following questions to save a specific timetable in a user-defined output file and preview the LaTeX recommended timetable as follows:

Exams Timetable Scheduling

=================

Genetic Algorithms crossover selection:

1 One-point crossover.

2 Two-point crossover.

3 Uniform crossover.

4 Fixed-point uniform crossover.

Please type a number here: **4**

How many positions do you want to swap? **19**

Please enter the pre-defined template file: **tprand.90**

0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0*0

0*0*0*0*0*0*0* 38 exams

please enter the student-exams data file: **90.ac**

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../......../......../........

/......../......../......../......../......../......../......../ 47 students

How many time slots during the exams: **24**

how many generations : **300**

how many populations in a generation: **50**

Please enter a number as seed: **4**

... ... waiting ... ...

Do you want to see the recommended solution? (y/n) **n**

Do you want to print the recommended solution? (y/n) **y**

Please enter the modules abbreviation file: **90.abb**

Please input the solution number: **44**

Please enter the output file: **90.tex**

Which weekday is the first exam day(1-5): **2**

Now, we can preview the timetable as shown in table 5–2 with *Tuesday* as the first day of the timetable.

	930-1100	1130-1300	1400-1530	1600-1730
Tuesday	kri2	ds	algc	cvis swl
Wednesday	kri1	si	lng1	mthr acarc
Thursday	its grph	mvis db		prol clg1 asicd fprg
Friday	lisp ccomp		nlp	robsn lng2 csys
Monday	ias nnet cpin	cafr		cad swt aps
Tuesday	os bsv	isc clg2	spd ccom	es

Table 5–2: GAs timetable in MSc 90/91

Chapter 6

The Implementation Details

In this chapter we will describe the implementation issues of the system. The data used in this chapter is the AI/CS MSc 91/92 student and module data, which is in appendices A, B, C.

6.1 Random Number Generator

We produce two random number generator functions which are used very often in Genetic Algorithms. Let's briefly describe how to use them.

- int *flip(probability)*: returns 1 if random() is less than probability[1], otherwise returns 0.

- unsigned long *rnd(range)*: returns an integer between 1 and range.

[1]Defn: a value between 0.0 and 1.0

6.2 Data Structure for an Individual

An individual include a chromosome and other information. The data structure of an individual is as follows:

```
/* datastru.h */

struct   individual
         {
           int       chrom[100];       /* nobinary bit string */
           int       count;            /* conflicts accumulator */
           float     fitness;          /* object function value */
           int       parent1;          /* the first parent */
           int       parent2;          /* the second parent */
           int       crossoverpos1;    /* crossover position 1 */
           int       crossoverpos2;    /* crossover position 2 */
           char      crossarray[100];  /* position-based crossover */
           int       type1;            /* punishment=1 */
           int       type3;            /* punishment=3 */
           int       type10;           /* punishment=10 */
           int       type30;           /* punishment=30 */
         };

typedef  struct individual *population;
```

where *chrom* is an array for storing our chromosome as a list string, *count* is the sum of type1, type3, type10, type30 punishment counts, *fitness* is the objective function value for this chromosome, *parent1*, *parent2* are the parents selected for producing this chromosome, *crossoverpos1* is one-point crossover position and *crossover1*, *crossover2* are used for two-point crossover, *crossarray* is used for uniform crossover, which is filled with * if the positions need to be swapped, else filled with blanks in elements of the array. *type1* is the punishment for two consecutive exams with one in the morning and the other in the afternoon for the same student on the same day. *type3* is the punishment for two consecutive exams on the same half day for the same student. *type10* is the punishment for three exams held on the same day for the same student. *type30* is the punishment for two exams clashes for the same student. The numerical punishments for type1, type3, type10, type30 are 1, 3, 10, 30 respectively. These numbers are picked for no precise reason, but they seem to agree with a feeling of how important are the different punishment types.

In the following sections we will use pseudocode to describe the key functions in this system.

6.3 Domain-Independent Implementation

In this section we will outline the domain-independent functions used in the system.

6.3.1 Main Function

Most of the flow in this system is controlled by the big loop in the main function. By the way, we also set the initial probability of crossover=0.7 and probability of mutation=0.003. The main function is described as follows:

```
BEGIN
        LOAD DATA and ask PARAMETERS
        initialize the INITIAL generation

        WHILE generation less than maximum+1 generation
        {
                adjust crossover and mutation rate
                SELECTION and REPRODUCTION
                apply pre-defined TEMPLATE
                DECODE the chromosomes and EVALUATE their fitness
```

```
                STATISTICS and REPORT
                copy NEW population to OLD population
                increase generation counter
        }
END
```

The crossover and mutation rate could be fixed values but we adjust them linearly in this system, as experiments show in Chapter 7 that this is better. Because at the beginning, the chromosomes are very diverse, crossover can mate better chromosomes to produce better children effectively. When the generation number increases, the exploration[2] happens less and less, because high-fitness schemata are growing rapidly, and becoming most of the population. However, exploitation[3] happens more and more. This means a problem because in order to prevent converging to a local minimum we would like to boost exploration as generations increase. For example, we might have after 300 generations all of the chromosomes being the same. In order to break away from this possible local minimum we need to have more exploration, which is done by increasing the possibility of mutation operator. So, we gradually increase mutation and decrease crossover probability as the generations increase in time. Furthermore, they must have limits in order not to let the exploration and exploitation disappear. We set 0.2 as the lower bound of crossover and 0.01 as the upper bound of mutation. If we don't set the lower bound on crossover, then the system will not produce children using mating after some time because the crossover rate is too small to let crossover happen. Therefore, it will take a long time to find the optimal solution. If we don't set the higher bound on mutation, then there will be too many new variations because the mutation rate is too high. Therefore, the higher fitness

---

[2]Defn: meaning to make search as much of the chromosome space as possible

[3]Defn: meaning to make maximum use of the good schemas and building blocks in the population

chromosomes among them will be very easily to be destroyed by mutation and it will become a random search system.[4]

6.3.2 Initialization

The initialization function is described as follows:

```
BEGIN
        randomly GENERATE and ENCODE the data into non-binary chromosomes
        apply pre-defined TEMPLATES
        DECODE the chromosomes and EVALUATE their fitness
        STATISTICS and REPORT
END
```

The initial generation is different from all other generations because all the chromosomes in this generation come not through crossover and mutation but are randomly generated. However, it is also necessary to be adjusted by the TEMPLATE as is done in the other generations.

6.3.3 Reproduction and Selection

The reproduction and selection function is described as follows:

```
BEGIN
        let Number_of_population equal to 0
        choose the HIGHEST FITNESS PARENT as our first child
        Number_of_population = Number_of_population + 1
        WHILE Number_of_population < three quarters of Maximum_population
        {
                SELECT 2 parents using Roulette Wheel Selection
```

[4]Note: in GA1 (described in Chapter 7) we do not change the mutation and crossover probabilities from 0.003 and 0.7 since GA1 is a traditional simple Genetic Algorithm.

```
            apply CROSSOVER and MUTATION operator
            REPRODUCE two children
            Number_of_population = Number_of_population + 2
        }
        WHILE Number_of_population < Maximum_population
        {
            randomly GENERATE a chromosome as our child
            Number_of_population = Number_of_population + 1
        }
END
```

This pseudocode shows the Elitist + one quarter new population strategy (see Chapter 7). It is simple to see how it can be changed to do the other strategies. In order to guarantee that the best parent will survive from generation to generation, we copy the highest fitness parent as our first child. This is called *Elitist Selection* in Genetic Algorithms. Then we use Roulette Wheel Parent Selection to select three quarters of the population. Finally, we generate one quarter of the population randomly in order to keep on introducing new chromosomes. This is different from the standard *Generational Replacement* which replaces the whole generation just using mating through crossover and mutation operators.

6.3.4 Crossover

The crossover operator is the most important operator in Genetic Algorithms. In one-point crossover, we first use flip(crossover_rate) to decide whether to apply crossover or not, then call rnd(exams-1) to choose the crossover position, finally swapping the last part of two parents because there are only two parts divided by one crossover. The following procedure is used in one-point crossover:

```
BEGIN
        IF      within crossover posibility
        THEN
                randomly generate a CROSSOVER POSITION between
                1 and MAXIMUM_EXAMINATION - 1
        ELSE
```

```
        set the CROSSOVER POSITION equal to MAXIMUM_EXAMINATION

    initialize two chromosome CHILD1,CHILD2

    FOR ALL positions < CROSSOVER POSITION
    {
            copy bit from PARENT1 to CHILD1
            apply MUTATION
            copy bit from PARENT2 to CHILD2
            apply MUTATION
    }

    FOR ALL positions >= CROSSOVER POSITION
    {
            copy bit from PARENT1 to CHILD2
            apply MUTATION
            copy bit from PARENT2 to CHILD1
            apply MUTATION
    }
END
```

In a two-point crossover situation, we must call rnd(exams-1) twice instead of once to choose two crossover positions. The middle part of the two selected parents needs to be swapped instead of just the last part as in one-point crossover. In uniform crossover, the system will randomly generate a number for each chromosome to decide how many bits need to be swapped. In fixed-point uniform crossover, the user himself must specify how many points are to be swapped in each chromosome. Then the system uses rnd(exams) to swap those bits of both parents until the number of times specified above is used up. Therefore each position has a chance to be swapped with each other to get more schemata but it is also more time-consuming because we must throw more dice and swap more.

6.3.5 Mutation

The mutation operator is not so important in the beginning. But it is important gradually as time progresses. It uses flip(mutation_rate) to decide if we want to change the current bit or not and use rnd(slots) to generate the new value. It is a good way to escape local minima when crossover couldn't introduce a new schema again. It's procedure is described as follows:

```
BEGIN
        IF      within mutation posibility
        THEN
                randomly generate a MUTATING VALUE between
                1 and MAXIMUM_SLOT
        ELSE
                keep the bit UNCHANGED
END
```

Though mutation is an important operator especially near the end of the run, it should not be too high. Otherwise, better schemata will be very easily destroyed because each bit will have a large possibility of being changed. Therefore, we need to set an upper bound to the mutation rate.

6.3.6 Template

The function of templates is for some lecturers hoping their modules are set in a certain slot or some module wanting to be set before another module or other special requirement. It's procedure is described as follows:

```
BEGIN
        FOR ALL new population
        {
                FOR ALL exams
                {
                        IF    already pre-define value in TEMPLATE
```

```
                        THEN
                              ADJUST the exam to pre-defined value
                        ELSE
                              keep the value unchanged
                }
         }
END
```

The algorithm is very simple. After all the children are produced, we overwrite the pre-defined slots to their relative modules to ensure that each chromosome's pre-defined bits are not changed by the mutation or crossover. Then we can begin to decode and evaluate the adjusted chromosomes to get their fitness.

6.4 Evaluation Technique

The evaluation function is a domain-dependent function, it is also the most difficult part of Genetic Algorithms. It includes two functions: Decode and Objective Function.

6.4.1 Decode

The Decode function is a bottleneck and the most time-consuming part of our system. We must consider whether each student, each module and each schedule is compatible to each other or not. It's procedure is described as follows:

```
BEGIN
     FOR ALL INDIVIDUAL
     {
        FOR ALL STUDENTS
        {
            FOR h = 1st to 7th COURSES
```

```
            {
                FOR i = 2nd to 8th COURSES
                {
                    IF     two exams CLASH
                    THEN
                           PUNISHMENT = PUNISHMENT + 30
                           continue

                    IF     i less than 7
                    THEN
                           FOR j = (i+1)th to 8th COURSES
                           {
                               IF    THREE courses on the same day
                               THEN
                                     PUNISHMENT = PUNISHMENT + 10
                           }
                           continue

                    IF     two CONTINUOUS exams on the SAME day
                    THEN
                           PUNISHMENT = PUNISHMENT + 3

                    IF     two CONTINUOUS exams one in the morning and
                           the other in the afternoon of the SAME day
                    THEN
                           PUNISHMENT = PUNISHMENT - 2
                 }
             }
         }
     }
END
```

We use five nested for-loops to calculate the punishment. For each student, we must consider whether his first course clashes with his second to eighth courses or

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
dayX	e1	e2		
dayX	e2	e1		
dayX			e1	e2
dayX			e2	e1

Table 6–1: Two consecutive exams on the same half day

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
dayX		e1	e2	
dayX		e2	e1	

Table 6–2: Two consecutive exams on different half of the same day

not and his second course clashes with his third to eighth courses or not and so on. Because it is the most serious constraint, if it is happens, then the schedule will be considered a failure. Therefore the punishment is *Thirty*.
Furthermore, we must also prevent a student from taking consecutive exams. There are three combinations of consecutive exams in our timetable and each combination has two permutations in our real case.

In Table 6–1, two exams are set consecutively on the same half day(e1&e2) for the same student. It is a weak constraint of our system that we must weakly avoid this situation. Therefore, the punishment is *Three*.

In Table 6–2, two exams are set consecutively for the same student. But one is in the morning and the other is in the afternoon. We must avoid this situation because it is also a weak constraint. The interval between morning and afternoon is longer than those in the same half day in above situation, therefore the punishment is just *One*.

In Table 6–3, two exams are set consecutively but on different days for the same student. It is allowable in our system. Therefore, there is no punishment for this. If necessary, it would be easy to have a punishment for this situation.

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
dayX				e1
dayX+1	e2			
dayX				e2
dayX+1	e1			

Table 6–3: Two consecutive exams on different day

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
dayX	e1	e2	e3	
dayX	e1	e3	e2	
dayX	e2	e1	e3	
dayX	e2	e3	e1	
dayX	e3	e1	e2	
dayX	e3	e2	e1	

Table 6–4: Three exams on one day (case I)

The most complex and time-consuming part is to test how not to let a student take 3 courses within one day. There are four kinds of combinations and each combination has six permutations as shown in Table 6–4, 6–5, 6–6, 6–7.

We must test three courses at a time, briefly speaking for the first student we must test the (first, second, third) pairs in the 24 permutations. Then test (first, second, fourth) pairs in above 24 situations and so on until (sixth, seventh, eighth) pairs. Then we begin to test the second student up to the last student using the same method as for the first student. Because it is a strong constraint in our application, therefore the punishment is *Ten*. It would be possible to use information from the result of testing for consecutive exams; for example: if a student has no consecutive exams on a particular day, then there is no need to test for three-on-one-day. However, if we modified the decode function to use this information (which would speed things up a little), we would lose some modularity.

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
dayX	e1	e2		e3
dayX	e1	e3		e2
dayX	e2	e1		e3
dayX	e2	e3		e1
dayX	e3	e1		e2
dayX	e3	e2		e1

Table 6–5: Three exams on one day (case II)

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
dayX	e1		e2	e3
dayX	e1		e3	e2
dayX	e2		e1	e3
dayX	e2		e3	e1
dayX	e3		e1	e2
dayX	e3		e2	e1

Table 6–6: Three exams on one day (case III)

day	*AM*		*PM*	
	4n+1	4n+2	4n+3	4n+4
dayX		e1	e2	e3
dayX		e1	e3	e2
dayX		e2	e1	e3
dayX		e2	e3	e1
dayX		e3	e1	e2
dayX		e3	e2	e1

Table 6–7: Three exams on one day (case IV)

That is, some exam schedulers might not be worried about consecutive exams, but still want to avoid three-on-one-day.

6.4.2 Objective Function

The objective function is the function that holds the relationship between the punishment, produced by the evaluation function and the previous function, and the fitness score. The fitness score is an important score which guides how likely a chromosome will be used for reproduction for the next generation. The pseudocode of the objective function is as follows:

```
BEGIN
        FOR ALL INDIVIDUALS
        {
            Punishment = Count + 1

            Build the relation between Fitness and Punishment
            as specified by User
        }
END
```

In this function, we provide an opportunity to let the user define his own objection function using a certain format. For example: if the user wants the relation between fitness and the punishment is inverse linear then the formula will be *fitness=1.0/punishment* or inverse square then the formula might be changed from:

fitness=1.0/punishment

to:

*fitness=1.0/(punishment*punishment)*

and leave the other codes unchanged(the C source code of objective function is in appendix I), then type

make

under the UNIX operating system, then the make file will automatically recompile the objective function source code and link it with the other domain-independent Genetic Algorithms code.

The format of the makefile is included in appendix H.

Chapter 7

Experiments with Genetic Algorithms for Timetable Scheduling

In this chapter we will present some results of experiments with different kinds of Genetic Algorithms, ranging from a traditional Genetic Algorithm (GA1), to other versions with different parameters and aspects. We first describe Genetic Algorithms versions in which we alter aspects of the evaluation function, crossover and mutation probabilities, and aspects of how the new generation is made from the old generation. Later, we examine these techniques in respect of different kinds of crossover operator. The programs used in this chapter are:

- *GA1*: traditional Genetic Algorithm.
- *GA2-1*: GA1 + one quarter new population each generation. That is, as one possible way of eliminating premature convergence to a local minimum, this version places new, randomly generated chromosomes in every generation. One quarter of each generation are such chromosomes.
- *GA2-2*: GA1 + Inverse Square Pressure. Here, this differs from GA1 only in that the fitness function is *squared*; this increases the pressure on chromosome performance.
- *GA3*: GA2-2 + one quarter new population each generation.
- *GA4*: GA2-2 + Elitism. In *elitism*, we always ensure that the best chromosome from a population is placed in the new population. With no elitism, in

traditional GA1 for example, this can only happen if the best chromosome mates with a replica of itself, and this only has a probability of happening, not a certainty.

- *GA5*: GA4 + interpolating operator rates for crossover and mutation operators. Here, rather than keep crossover and mutation with fixed probabilities, we make crossover gradually less likely and mutation gradually more likely (though still small probability) as the generations increase. At the beginning, early in the operation of the Genetic Algorithms, the search space is being *explored* by crossover and mutation, in order to find good schemata and good combinations of schemata; as generations go on, this turns more into *exploitation*, as most of the population contains lots of high-fitness schemata, and these are *exploited* as crossover operators keep on combining the same chromosomes without much exploration of other possibilities. This has the possibility of falling into a local minimum before the search space has been explored enough. Therefore, we want to achieve the best tradeoff between exploration and exploitation, and we do this by gradually lowering the probability of crossover as generations go on, and raising the probability of mutation.

Because Genetic Algorithms use selection, crossover, mutation stochastically, a single run may not properly represent the whole performance. Therefore, we do each experiment ten times, using different random seeds, and then work out the average of the best results of the ten experiments as our final result. In other words, the punishment shown in our figures for each generation represents the average of the best chromosome's punishment over ten experiments.

The performance graph for each of these GA versions, using one-point crossover with population size 50 and 300 generations, is shown in figure 7–1. In the rest of this chapter, we discuss the performance of each version.

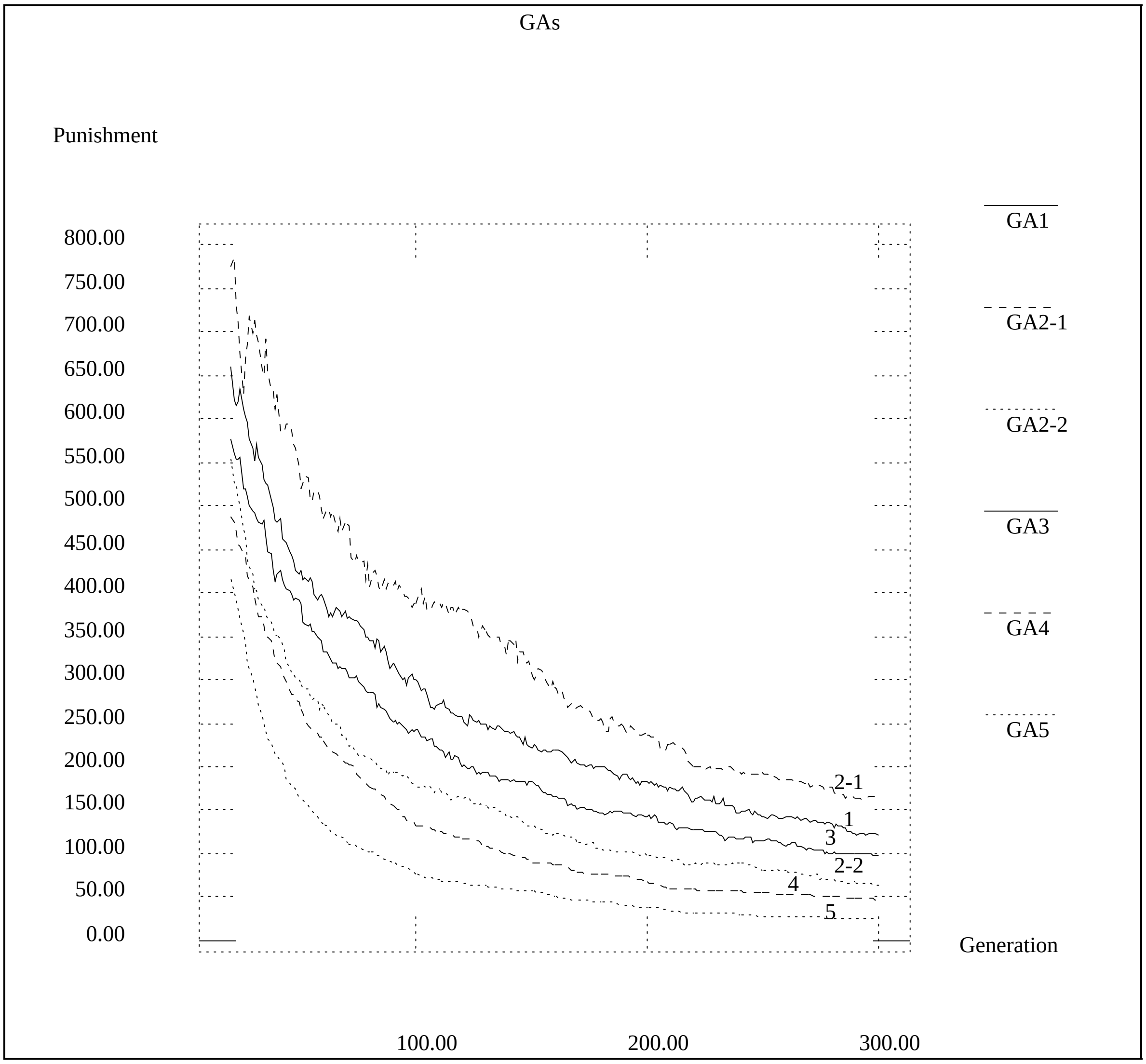

Figure 7–1: Performance graph for GA1, 2-1, 2-2, 3, 4, 5

7.1 Traditional Genetic Algorithms

Traditional Genetic Algorithms have the following characteristics:

- *binary string*: using 0, 1 to represent the chromosomes[1].
- *generational reproduction*: replacing all the populations in the parent's generation to produce the child's generation.
- *one-point crossover*: simulating biological genetic crossover.

In figure 7–1, GA1 is the traditional Genetic Algorithm, the others are some extensions or modification of GA1. We use non-binary bits string to represent the chromosomes for it is more clear and efficient than a binary bit string representation because each non-binary bit represents a time slot. As a result, we don't need to transfer from non-binary bits to binary bits and then from binary bits to non-binary bits. The fitness in GA1 is inversely proportional to the punishment of each chromosome. Therefore, the larger the punishment, the smaller the fitness. The difference between two fitness is therefore smaller at the beginning generations(for example, 1/1000 and 1/1001) but will be larger and larger as the generations increase(for example, 1/10 and 1/11). This can satisfy our expectation because the chromosomes are very random at the start and the fitness is very low at the beginning, we don't want to put too much pressure in the evaluation. As the generation increasing, the chromosomes are more and more uniform. Therefore, we need to put more pressure on them in order to choose better chromosomes from the very competitive ones. In other words, small differences in fitness become more and more important as the chromosomes' average fitness increases; the probability of choosing a chromosome of fitness 1/10 rather than one of fitness 1/11 is more than of choosing a chromosome of fitness 1/1000 over one of fitness 1/1001. This is important, and GA2-2 (see section 7.3) examines the effects of enhancing this effect by squaring the punishment values.

[1]Note: In our system, we used non-binary string to represent the chromosomes.

From our experimentation, we draw the performance graph in figure 7–1 from which we see that not every modification from traditional Genetic Algorithms is useful in improving the performance. We will discuss the experiment results in the following few subsections.

7.2 New Population

In this version, GA2, one quarter of each new generation are random chromosomes. This is a strategy designed to prevent premature convergence to a local minimum. We thought that local minima would be very possible in such a problem; imagine if three large exams with one triple student clash are put in the same time slot — this could be a good building block, leaving free time slots which lets us easily schedule the other exams, but it is an unacceptable situation which we must avoid. The strategy of introducing a quarter of each new population randomly is therefore tried in order to let the GA1 always have new chromosomes to use. From our experiments, we know that this strategy can help just a little in the first few generations if at all. This is because in the beginning, the chromosomes are very random and more new population can help introduce more better chromosomes into the system. But it quickly becomes a load on the system, because selection and crossover soon mate and produce good chromosomes for us compared to the new introduced chromosomes. Therefore, the one quarter new chromosomes become useless for their randomly low fitness, but each generation the system has one quarter less possibility to mate with other chromosome because the total number of chromosomes is constant. The results of this experiment are shown in figure 7–1 GA2-1. Though we introduce a quarter of new population in each generation, the performance in GA2-1 is much poorer than with the traditional Genetic Algorithm GA1. Perhaps such a GA2-1 strategy might be better for a different kind of problem, where the local minima are more widespread. We will do other experiments in section 7.4 using GA3 to re-confirm this observation that this strategy is not much help here.

7.3 Inverse Square Pressure

[Davis 91] mentioned about two normalization techniques to improve the performance of traditional Genetic Algorithms; both of these adjust the evaluation function in order to somehow improve performance:

- *Windowing*: Find the chromosome with the lowest fitness in the population. Assign each chromosome a fitness equal to the amount that it exceeds this minimum. Optionally, a minimum amount greater than the minimum value may be created as a guard against the lowest chromosomes having no chance at reproduction.
- *Linear Normalization*: Order the chromosomes by decreasing evaluation. Create fitnesses that begin with a constant value and decrease linearly. The constant value and the rate of decrement are parameters of this technique.

In the project, we introduce *Inverse Square Pressure* as another attempt to modify the evaluation function in a way that might improve performance. Usually, at the end of our timetable scheduling, there are a few candidate competitors and their fitness are very similar. Therefore, we increase the selection pressure to further increase the chances of better solutions to mate with other chromosomes. The inverse square pressure decreases the possibility of choosing less better chromosomes because of the stochastic selection characteristics of Genetic Algorithms by using an inverse square instead of inverse linear punishment function as used in our traditional GA1 to help distinguish between two competitive chromosomes.
We can see from figure 7–1, GA2-2 differs from GA1 just in using inverse square pressure, but the performance in GA2-2 is much better than GA1. In the AI/CS MSc exam timetabling domain, it therefore seems that inverse square pressure is effective. It is however not clear how the effectiveness of inverse square pressure depends on the shape of the evaluation function space. More experiments are needed to help find this out.

7.4 Hybrid

In this section, we use a program GA3 which is a hybrid of Inverse Square Pressure and one quarter new population to do the experiments again to test the conclusions we made in section 7.2. We produce GA3 by combining the methods used in GA2-1 and GA2-2. As a result, GA3 is a traditional Genetic Algorithms with the extension of using a quarter of new population and Inverse Square Pressure. From figure 7–1, GA3 is poorer than GA2-2 because of the introduction of one quarter new population but better than GA1 because of *Inverse Square Pressure*. Now, we can say more definitely that the introduction of new populations is harmful because we have less possibility to mate with other chromosomes. But it is also interesting that the improvement given by Inverse Square Pressure is robust, and thus seems to be a strong effect on improving traditional GA performance on this kind of problem.

7.5 Elitism

The *elitist* strategy copies the best fitness chromosome from the parent's generation to the child's generation in order to guarantee that the best chromosome in each generation won't disappear because of the stochastic selection. In our program, we just choose the best *one* to copy to the next generation because if we copy too many best chromosomes to next generation, it will be very easy to fall into premature convergence. This strategy improves the performance in our system a lot though there is a trade-off between quickly finding a satisfactory solution and falling into the local minimum. In our system, most of the recommended timetables are better than the actual timetable. Therefore, we don't need to worry about falling into a local minimum but we do care more about how fast the system can give us a satisfactory solution.

We can see from figure 7–1 that the performance of GA4 is better than GA2 because of the *Elitism* strategy.

operator	*start*	*step*	*stop*
Crossover	0.7	-0.002	0.2
Mutation	0.003	0.0003	0.01

Table 7–1: Parameters for crossover and mutation

7.6 Interpolating Operator Rates

As we mentioned in section 6.3.1, crossover is important in the beginning and as the generations increase, its importance will be less, as mutation becomes more important, though crossover is still very much the most important. We also mentioned why they must have bound values. In GA5, we set the parameters for the probability of crossover and mutation as in table 7–1. The initial value of crossover is best large(larger than 0.5), and the initial value of mutation should be small(less than 0.005). The step value of both operators should be as small as possible in order to get a smooth variation. Finally, the stop value of crossover should not be too near 0 to let crossover disappear and the stop value of mutation should not be so near 1 as to yield a random search system.

We can see from figure 7–1, GA5 is enhanced from GA4 by interpolating the operator rates. The performance of GA5 is improved again.

7.7 Different Crossover Operators

Till now, all of the operators of our programs shown in figure 7–1 use one-point crossover. It is a traditional crossover inspired by natural evolution. In this section we experiment with different kinds of crossovers to test the performance. The crossovers used in this section are two-point crossover, uniform crossover, and fixed-point uniform crossover. In uniform crossover, we randomly generate a number less than or equal to the maximum module 44 to specify how many positions needed to be swapped. In fixed-point uniform crossover, we use the mean number 22 of the numbers to specify the number of positions to be swapped. If the number of positions swapped in uniform crossover is larger than the mean length of a chromosome, then this introduces extra computational load which is unnecessary, because, for example, swapping 30 positions between two length 44 chromosomes can be expected to have the same effect as swapping 14 positions. The latter strategy means much less computational load of generating random numbers and swapping. So, we do not want the fixed point of fixed-point uniform crossover to be more than 22. But, we would expect a fixed point of 22 to be more effective than a fixed point of less than 22, because this gives us more access to more potentially good schemata. That is, a fixed point of 10, say, only directly manipulates schemata of length 10 or less. But a fixed point of 22 manipulates all of these schemata and more. Hence we choose 22 as the fixed point for fixed-point uniform crossover in this case. Also, we can predict that fixed-point uniform crossover will perform better than uniform crossover because the latter can be seen as mixing fixed point 22 with less optimal points.

The performance graph for our four types of crossover using GA1 and GA5 are in figure 7–2 and figure 7–3. In GA1, the performance of two-point crossover is better than other crossovers. In GA5, all of the performance is very good, it is hard to say which one is really best, but fixed-point uniform crossover seems slightly better than the others. In both situations, uniform crossover is on average never better than fixed-point uniform crossover as we predicted above. Therefore, we can improve the performance by using other kinds of crossovers instead of one-point crossover, for example using two-point crossover and (fixed-point) uniform crossover.

7.8 Different Population Size

In this section, We try GA1 and GA5 using population size 100 instead of 50 in each generation and using 150 generations instead of 300 generations. The total populations are the same as we discussed above equal to 15,000(100 times 150 or 50 times 300). We average our four crossover results in our final results and the experiments are shown in figure 7–4. The performance in the population size 100 situation of both programs is not better than in the population size 50 situation. Therefore, we know that using a small population in each generation and running

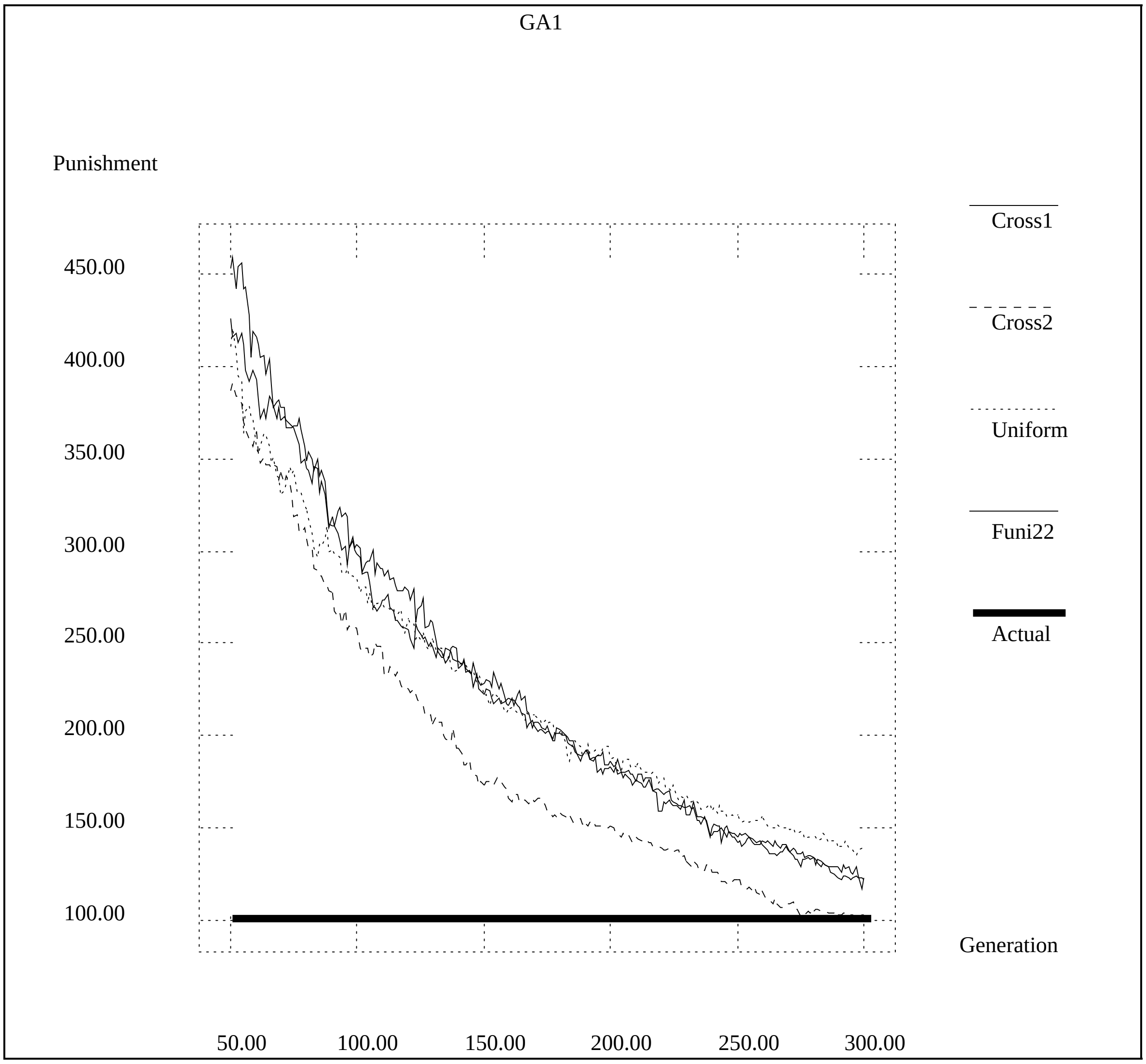

Figure 7–2: Performance graph for GA1 compared with actual timetable

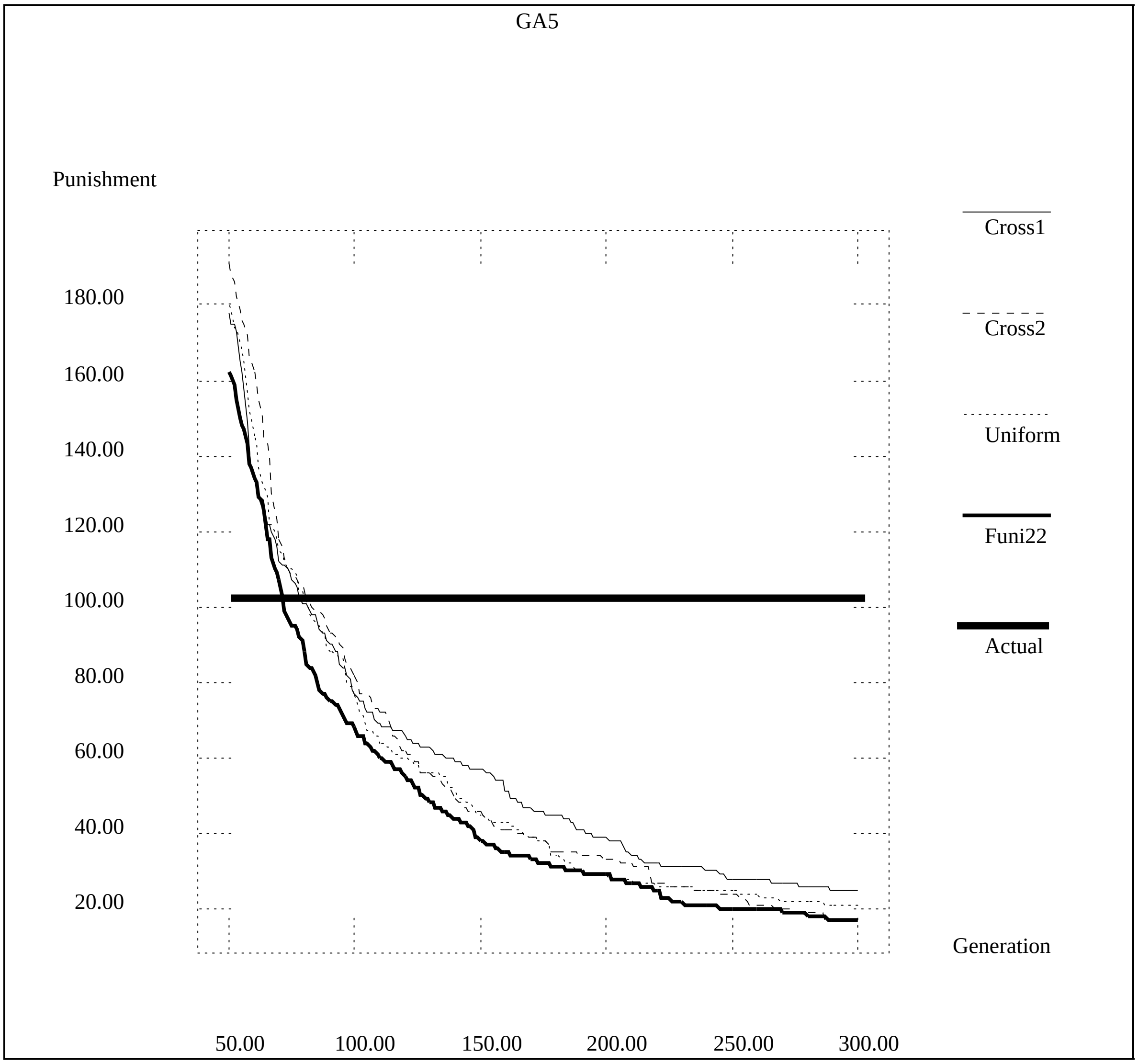

Figure 7–3: Performance graph for GA5 compared with actual timetable

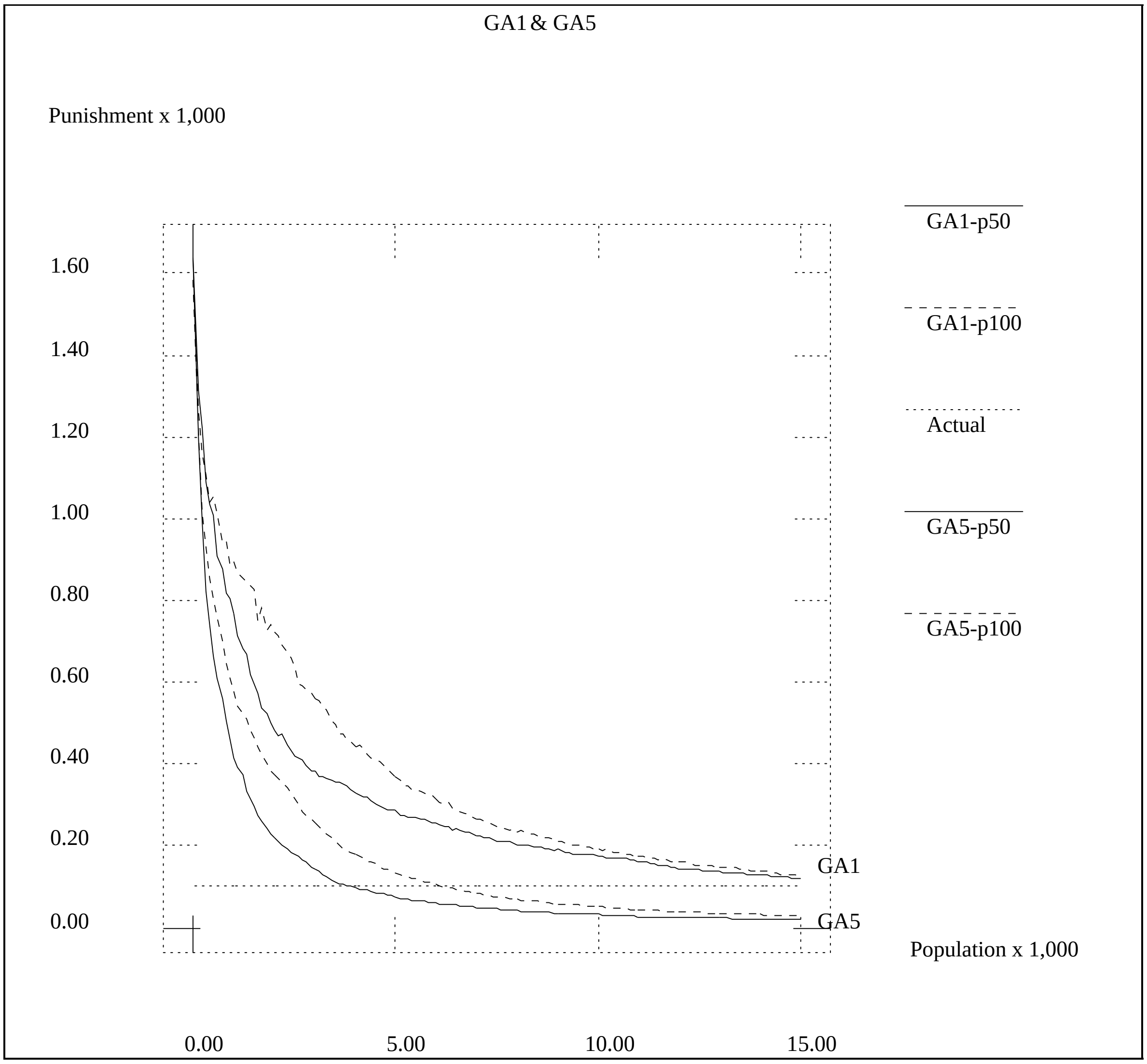

Figure 7–4: Performance graph for GA1, GA5 using population size 50 and 100

more generations is better than using a large population in each generation and running fewer generations except when the population size in each generation is too small to produce good chromosomes. It would be interesting to experiment, or mathematically analyze, to find the optimal population size. This should probably be a function of the size of the chromosome space.

7.9 Comparison with Actual Timetable

We can also compare the actual human-generated timetable in figure 7–2 and figure 7–3. The horizontal black bar is the actual timetable punishment 101 for MSc 91/92. We can see that the performance of GA1 is slight poorer than the actual timetable scheduled by the domain experts but GA5 is much better than the actual timetable. Within one hundred generations in GA5, we can get a satisfactory timetables and within three hundred generations in GA5, we can get the near optimal timetable[2].

In the next section, we present a clear comparison between the actual timetables produced by human experts and the best timetables produced by the system.

7.10 Summary of Genetic Algorithms Performance vs. Human Performance

In this section, we outline the results of the best performance within ten runs of the Genetic Algorithms programs on the AI/CS MSc 90/91 and 91/92 exam timetable problems, compared with the actual timetables prepared by Drs Robert Fisher and Alan Smaill.

7.10.1 AI/CS MSc 90/91

Actual Timetable. The actual human-produced timetable for AI/CS MSc 90/91 is shown in table 7–2 with the punishment=37 (Type1=4, Type3=11, Type10=0 and Type30=0). In other words, there are 4 occurrences of two consecutive exams on the different half days of the same day and 11 occurrences of two consecutive exams on the same half days of the same day. This was the result of a first draft

---

[2]Defn: a optimal timetable is that the punishment of the chromosome is zero.

which took about an hour to produce, followed by repeated iterative feedback, and more drafts, as students complained and the timetable was revised.

The Best GA-generated Timetable. The best GA-generated timetable for AI/CS MSc 90/91 within ten runs using fixed-point uniform crossover operator is shown in table 5–2 with the punishment=0 (Type1=0, Type3=0, Type10=0 and Type30=0). In other words, it is an *optimal* solution, in respect of the identified constraints. Furthermore, ten out of the ten runs are better than actual timetable. Therefore, it is clearly better than actual human-produced timetable, and takes about 5 minutes to produce.

7.10.2 AI/CS MSc 91/92

Actual Timetable. The actual human-produced timetable for AI/CS MSc 91/92 is shown in table 7–3 with the punishment=101 (Type1=33, Type3=16, Type10=2 and Type30=0). In other words, there are 33 occurrences of two consecutive exams on the different half days of the same day and 16 occurrences of two consecutive exams on the same half days of the same day. In addition, there are 2 occurrences of three exams on the same day.

The Best GA-generated Timetable. The Best GA-generated timetable for AI/CS MSc 91/92 within ten runs using one-point crossover operator is shown in table 5–1 with the punishment=3 (Type1=0, Type3=1, Type10=0 and Type30=0). In other words, there is only *one* occurrence of two consecutive exams on the same half days of the same day. Furthermore, ten out of the ten runs are better than actual timetable. Therefore, it is much better than actual human-produced timetable.

The Best GA-generated Timetable in Six Days. The Best GA-generated timetable for AI/CS MSc 91/92 within ten runs using uniform crossover operator in *six* days is shown in table 7–4 with the punishment=52 (Type1=3, Type3=13, Type10=1 and Type30=0). In other words, there are 3 occurrences of two consecutive exams on the different half days of the same day and 13 occurrences of

	930-1100	1130-1300	1400-1530	1600-1730
Tuesday	its asicd	isc cpin	clg1 nnet	mvis cafr
Wednesday		prol ccomp	spd	es acarc
Thursday	lisp bsv	cad	lng2 ds	robsn grph
Friday	swl algc	mthr lng1		nlp os fprg
Monday	kri1 si		cvis clg2 db	csys
Tuesday	kri2 ccom	ias	swt aps	

Table 7–2: Human-produced timetable in MSc 90/91

	930-1100	1130-1300	1400-1530	1600-1730
Monday	cca	kri1 ccomp	clg1 spd	mvis fprg
Tuesday	cvis fgm1 ssc	cad	lisp	algc
Wednesday	isc lng1 cp	grph	kri2 cafr	ccom
Thursday	aied clg2	ds	db rs1	prol lng2 sp1
Friday	es sp2	asicd	os cpin rs2	ias bsv
Monday	swl	nlp		nnet
Tuesday	(essp) aps	cc	mthr	si

Table 7–3: Human-produced timetable in MSc 91/92

two consecutive exams on the same half days of the same day. In addition, there is 1 occurrence of three exams on the same day. Furthermore, seven out of the ten runs are better than actual timetable.

Another Best GA-generated timetable for AI/CS MSc 91/92 within ten runs using fixed-point uniform crossover operator in *six* days is shown in table 7–5 with the punishment=45 (Type1=2, Type3=11, Type10=1 and Type30=0). In other words, there are 2 occurrences of two consecutive exams on the different half days of the same day and 11 occurrences of two consecutive exams on the same half days of the same day. In addition, there is 1 occurrence of three exams on the same day. Furthermore, ten out of the ten runs are better than actual timetable.

Therefore, it is on average still better than actual human-produced timetable for *seven* days.

	930-1100	1130-1300	1400-1530	1600-1730
Monday	es ds		swl	kri2
Tuesday	ccom	kri1		ias mthr cc asicd
Wednesday	nnet aps	ssc	lng1	mvis cpin si
Thursday	cvis essp clg1 cad	cafr	lisp spd rs1	grph
Friday	aied clg2 fprg bsv rs2	cp	nlp db	sp1 sp2 ccomp
Monday	os	prol fsm1	algc	isc lng2 cca

Table 7–4: GAs timetable in MSc 91/92 within six days - uniform crossover

	930-1100	1130-1300	1400-1530	1600-1730
Monday	isc lng1 algc	ds	kri2	ccom
Tuesday	lisp cc asicd	rs1	cafr si	mvis
Wednesday	db	kri1	cca	prol cp
Thursday	ias mthr clg2 aps		fsm1 ccomp	aied sp2 grph
Friday	es bsv		essp os	nnet fprg rs2
Monday	sp1 cpin swl	nlp lng2 cad	ssc	cvis clg1 spd

Table 7–5: GAs timetable in MSc 91/92 within six days - fixed-point uniform crossover

Chapter 8

Conclusions and Future Research

8.1 Conclusions

There are two aspects of the results and conclusions of this work that can be described. First: evaluating the project in terms of the usefulness of the programs written as a tool for exam timetable schedulers. Second: using the experiments and experience of the project to gain knowledge about the effectiveness of different varieties of Genetic Algorithms.

8.1.1 Evaluation as an Exam Timetable Scheduling Tool

The ins and outs of our Genetic Algorithms timetable scheduling and the human expert timetable scheduling are as follows:

- *Clashes.* This is the only necessary constraint. Any timetable which violates this constraint is an illegal timetable. Both human experts and our Genetic Algorithms timetable scheduling can handle this constraint very well. Timetable scheduling by human experts would use heuristics such as to arrange the most popular modules first to avoid most potential clashes, while Genetic Algorithms solve it by setting a high punishment on clashes to get rid of clash situations effectively; that is Genetic Algorithms put strong pressure on avoiding clashes becauses clashes have a very low fitness. Unless the domain is very awkward, the Genetic Algorithm is guaranteed to avoid clashes, but these may still happen with human timetable scheduling so that timetables need to be revised.

- *Three-Exams-on-One-Day*. This is also one of our two strong constraints that we should not let it happen, though it is not so serious as the one described above. The timetable should be able to spread the student's load as much as possible. Usually, the exams last for more than one week, therefore it is not very difficult to prevent each student taking three exams on the same day. This situation is one that happens often when human experts do the scheduling, and is the main cause of complaints from students which leads to revising of the timetable. Genetic Algorithms solve this problem by setting a middle punishment to prevent this situation.

- *Two-Exams-on-Same-Half-Day*. Because the interval between two consecutive exams on the same half day is just half an hour, there is not enough time to let a student move from one place to the other place, if this was necessary. More important, this kind of situation is unfair on a student, not allowing enough time to relax and prepare for the next exam. We must avoid this situation especially when the two consecutive modules are at different places. That is to say, although moving from place to place is not a consideration in the AI/CS MSc exam situation (where all exams are usually in the same room), punishing consecutive exams makes sense for both reasons of unfairness to the student, and also moving from place to place if this is necessary. This might be necessary in exams scheduling in other departments.
  After a human expert has tried to schedule to avoid clashes, and then to avoid three exams in a day for every student, this situation, and the next one, are difficult and awkward to find out and to deal with. The problem is very complex, so that if a change is made to avoid consecutive exams for one student, then this might lead to new problems for other students. This is another main reason why human timetable scheduling needs iterations over many days. Genetic Algorithms solve this problem by setting the punishment between middle and low value to specify the importance of this constraint and let the system to handle the priority of the constraint. It is with this and the next constraint in particular that the Genetic Algorithm wins out over the human expert, because this constraint is handled easily by the Genetic Algorithm, producing timetables which avoid all of the previous constraints and this one.

- *Two-Exams-on-Different-Half-Day.* Though the interval between morning and afternoon lasts for one hour, it seems long enough for a student to have lunch, rest or move from one place to the other place, we still hope to do our best to avoid this situation to be a more considerate system, but the punishment for this constraint does not need to be as bad as for the previous constraint. As above, this constraint is a complex and difficult one for human expert to deal with, but simple for our Genetic Algorithm.

The first timetable draft will take a human expert about one hour and the Genetic Algorithm about five minutes. After scheduling the first version timetable draft, we can send the timetable to each student to see whether the timetable is suitable or not. Usually in the situation of human timetable scheduling, we must modify the timetable five or six times until every student is happy with the timetable. But in the Genetic Algorithm timetable, these modifications will be unnecessary, because, as our results show, there are likely to be no or very little problems for students in the first draft. If there are problems, this is very likely that these problems cannot be avoided anyway. As long as student data does not change between Genetic Algorithm scheduling and showing the timetable to students, further rescheduling will not be needed. If some big problem occurs though, say that for some reason some exam cannot happen in a certain timeslot, we don't need to run the timetable from the beginning, we just set the previous version timetable as our *template* and put a *0* for the modules we have to reschedule and then let the system run again. It will take no more than about one third of the time of the first draft by using smaller populations size and fewer generations.

8.1.2 Results and Generalizations from Genetic Algorithms Experiments

We also do some experiment with the traditional Genetic Algorithms and get some results as follows:

- *One Quarter New Population.* The one quarter new chromosomes in each generation just help a little if at all. But it quickly becomes a load on the system, because selection and crossover soon mate and produce good chromosomes compared to the new introduced chromosomes.

- *Inverse Square Pressure.* The inverse square pressure decreases the possibility of choosing less better chromosomes because of the stochastic selection characteristics of Genetic Algorithms by using an inverse square instead of inverse linear punishment function to help distinguish between two competitive chromosomes. This has been shown to do well with our particular problem.

- *Elitism.* This elitism strategy improves the performance in our system a lot though there is a trade-off between quickly finding a satisfactory solution and falling into the local minimum. It seems that what we might call *conservative elitism*, where we assure only that one copy of the best chromosome is put in the new generation, is a good position in this tradeoff. We could do other experiments to test this more.

- *Interpolating Operator Rates.* The initial value of crossover is best large(larger than 0.5), and the initial value of mutation should be small(less than 0.005). The step value of both operators should be as small as possible in order to get a smooth variation. The stop value of crossover should not be too near 0 to let crossover disappear and the stop value of mutation should not be so near 1 as to yield a random search system.

- *Different Crossover Operators.* The performance of two-point crossover and (fixed-point) uniform crossover are on average better than one-point crossover. Therefore, we can improve the performance by using these crossovers instead of one-point crossover.

- *Different Population Size.* Using a small population in each generation and running more generations is better than using a large population in each generation and running fewer generations except when the population size in each generation is too small to produce good chromosomes.

In a word, the advantage of Genetic Algorithms is that they save much precious time of human expert to do the boring routines every year but let the system do the trial-and-error for the human and if we want, we can open many windows to run many tasks at the same time with each task using different seeds to get many feasible timetable, then choose one or two best timetable as our timetable. The advantage of a human expert is that he is more flexible and can cope with some other less important additional requirement, for example try to spread out the exams across the full period as much as possible or have the exams with the most students at the earlier part of the exam period to give the lecturer more time to mark. The Genetic Algorithm system in this project can help by taking care of the most important constraints, and so generate possible timetables which the human expert can choose from with use of other constraints.

8.2 Future Research

Although the system can schedule timetables very well, some improvements are still possible, for example:

- *Adding Classroom Considerations.* Though there are enough classrooms at this moment, no one can guarantee that there will be enough some years later. If we can also consider the exam–student-classroom pairs in advance of this system, it will be a more perfect or complete timetable. One common constraint for including the classroom is that no two exams held at the same time can be set in the same classroom. But there are still some exceptions. In large classrooms, more than two exams can be held at the same place and at the same time if the classroom is large enough and no students are taking these exams at the same time.

- *Using Distributed or Parallel Techniques.* In order to improve the performance, we can remote login to different computers and run several programs using different seeds at the same time. Another approach not yet implement by us is to divide the large population into several small sub-populations, let the several sub-populations run in parallel and then periodically select some individuals and exchange them with other sub-populations; this is know as *migration* [Spiessens 88] [Tanese 89].

- *Seeding the Initial Population.* Genetic Algorithms are a way of solving highly complex optimization problems. They usually start with a random population of chromosomes, but their performance may be improved by introducing into the initial population a number of chromosomes which are derived from previously found solutions to similar problems. For example, one could have one tenth of the initial population for the 1992 timetabling problem be copies of solutions to the 1991 problem. There may be different numbers of exams, or different numbers of time slots in the two years, but part of the work will be to determine how to 'convert' a solution so that it fits with the new problem. (for example, if there are more exams in 1992, then simply add the extra exams randomly to the 1991 timetable.)

- *Preference of the Sequence.* Usually some lecturers are not so serious to let their modules to be set in a certain time slot. They just say that they want module A set before module B because of their dependency or more generally they want their module set in the afternoon or their module set in the first three days or in the first week of the exams, other lecturers specify that some exams may not happen on specified days or morning when they are unavailable. In our system, we can just use *Template* to set a certain module at certain time slot to set the various of preference at this moment. We might try a random exclusion day for each module and some other non-structured constraint in our later system.

- *Different Lengths Exams.* The Genetic Algorithms Scheduling system in this project can be used for any exam scheduling problem by simply changing the data files, so long as all exams are of the same length. In some places, students might have a mixture of 90 minute exams and two-hour exams, for example. We can easily modify the system to allow this, but we need to change the evaluation function. For example, chromosomes will still be the same, but checking for consecutive exams and so on will be a little more complicated. The system is modular so that we only need to change the evaluation function.

In a word, the algorithms and techniques used in the system can be very easily modified to some other similar kinds of scheduling problems (for example:

school courses timetable scheduling, or scheduling athletics events in a stadium). And the more complicated *Flow Shop* and *Job Shop Scheduling*[Conway *et al* 67] [E. G. Coffman 76] can also use the exam timetable scheduling techniques we use here as their foundation and inspiration to modify and extend the domain-specific parts such as the representation and evaluation.

We have shown that a Genetic Algorithm approach is very effective and useful in the AI/CS MSc exam scheduling problem. Using this program might lead to timetables in future which are fairer to students and so produce better results in the end, and also give the human experts more time to do other things. In this problem, it looks like Elitism, Interpolating Operator Rates and Inverse Square Pressure are good things to add to the Genetic algorithm. The reason why is possibly very complex, and to do with the structure of this particular problem and the evaluation function. Extensions to this work include many further experiments to test different kinds of objective function, population numbers, and also to analyze how the performance changes according to how many exams have to be scheduled for how many students.

Bibliography

[Abramson & Abela 91] D. Abramson and J. Abela. A parallel genetic algorithms for solving the school timetabling problem. Technical report, Division of Information Technology, C.S.I.R.O., April 1991.

[Belew & Booker 91] R.K. Belew and L.B. Booker. *Proceedings of the Fourth International Conference on Genetic Algorithms.* San Mateo: Morgan Kaufmann, 1991.

[Conway *et al* 67] Richard W. Conway, William L. Maxwell, and Louis W. Miller. *Theory of Scheduling.* Addison Wesley Publishing Company, 1967.

[Davis 87] L. Davis. *Genetic Algorithms and Simulated Annealing.* San Mateo: Morgan Kaufmann, 1987.

[Davis 91] L. Davis. *Handbook of Genetic Algorithms.* New York: Van Nostrand Reinhold, 1991.

[E. G. Coffman 76] JR. E. G. Coffman. *Computer and Job-Shop Scheduling Theory.* John Wiley and Sons, Inc., 1976.

[Goldberg 89] David E. Goldberg. *Genetic Algorithms in Search, Optimization & Machine Learning.* Reading: Addison Wesley, 1989.

[Grefenstette 85] J. J. Grefenstette. *Proceedings of the First International Conference on Genetic Algorithms and their Applications.* San Mateo: Morgan Kaufmann, 1985.

[Grefenstette 87] J. J. Grefenstette. *Proceedings of the Second International Conference on Genetic Algorithms and their Applications.* San Mateo: Morgan Kaufmann, 1987.

[Grefory 91] J. E. Grefory. *Foundations of Genetic Algorithms.* San Mateo: Morgan Kaufmann, 1991.

[Holland 75] John H. Holland. *Adaptation in Natural and Artificial Systems.* Ann Arbor: The University of Michigan Press, 1975.

[Holland 92] John H. Holland. *Adaptation in Natural and Artificial Systems.* Cambridge: Massachusetts: M.I.T. Press, 1992.

[Holland *et al* 86] J.H. Holland, K.J. Holyoak, R.E. Nisbeet, and P.R. Thagard. *Induction: Processes of Inference, Learning, and Discovery.* Cambridge: Massachusetts: M.I.T. Press, 1986.

[Schaffer 89] J. D. Schaffer. *Proceedings of the Third International Conference on Genetic Algorithms and their Applications.* San Mateo: Morgan Kaufmann, 1989.

[Shavlik & Dietterich 90] Jude W. Shavlik and Thomas G. Dietterich. *Readings in Machine Learning.* San Mateo: Morgan Kaufmann, 1990.

[Spiessens 88] P. Spiessens. Genetic algorithms: Introduction, applications and extensions. Technical report, Artificial Intelligence Laboratory, Vrije Universiteit, Brussel, 1988.

[Tanese 89] Reiko Tanese. *Distributed Genetic Algorithms for Function Optimization.* Unpublished PhD thesis, Dept. of Computer Science and Engineering, University of Michigan, 1989.

[Winston 92] Patrick Winston. *Artificial Intelligence 3rd edition.* Addison Wesley Publishing Company, 1992.

Appendix A

AI Student Exams File-91/92

```
8 12 6 7 1 5 22 28
8 12 6 7 9 4 3 2
8 6 7 16 14 15 5 23
8 12 6 7 10 19 11 1
8 6 7 16 14 15 5 23
8 12 6 7 10 23 24 29
8 12 6 7 9 2 3 4
8 12 6 7 10 1 5 27
8 12 6 7 4 19 25 5
8 6 9 4 3 19 27 25
8 12 6 7 11 1 5 29
8 12 6 7 4 19 1 2
8 6 9 4 3 2 23 19
8 6 16 14 15 19 7 17
8 12 6 7 19 5 11 10
8 12 6 7 4 10 19 2
8 12 6 7 9 10 11 5
8 12 6 7 9 11 1 5
8 12 6 4 19 3 2 9
8 12 6 7 13 10 1 5
8 12 6 7 4 19 3 5
8 6 7 14 15 16 5 19
8 12 6 7 9 10 5 2
8 12 6 7 5 9 11 1
8 12 6 7 10 11 1 25
8 12 6 9 4 19 3 2
8 6 7 16 14 15 18 17
8 12 6 7 10 1 11 5
8 12 6 7 10 11 1 5
8 12 6 9 4 19 3 2
8 6 7 16 14 15 21 19
8 6 7 9 4 19 3 2
8 12 6 9 4 11 3 2
8 12 6 7 9 10 5 2
8 12 6 7 10 24 30 31
8 12 6 7 9 10 11 5
```

```
8 12 6 7 9 4 11 5
8 6 7 16 14 15 5 20
8 12 6 7 9 10 5 26
8 6 7 14 15 16 19 5
8 12 6 7 4 10 19 2
8 12 6 7 10 19 11 1
8 12 6 9 4 3 2 19
8 12 6 7 5 9 4 19
8 12 6 7 14 15 16 5
8 12 6 7 4 11 5 29
8 12 6 7 4 19 1 2
8 12 6 7 9 11 1 5
8 6 7 14 15 16 19 5
8 12 6 7 10 11 1 5
8 12 6 7 9 10 5 2
8 6 9 4 3 5 2 19
8 12 6 7 4 19 1 5
8 6 7 16 14 5 15 23
8 12 6 7 10 19 11 2
8 12 6 7 10 1 11 5
8 12 6 7 11 1 5 13
8 12 6 7 10 19 1 25
8 12 6 7 11 5 9 1
8 12 6 7 10 19 2 9
8 12 6 7 10 19 11 5
```

Appendix B

CS Student Exams File-91/92

```
28 32 25 33 29 22 23 26
34 30 35 36 31 24 5 12
32 28 25 33 37 35 26 38
36 30 35 34 24 26 33 23
32 28 33 25 26 23 29 19
37 39 27 38 22 35 23 12
32 28 33 25 29 9 42 43
32 28 33 25 9 2 19 26
32 28 25 33 29 34 26 38
28 32 25 33 22 23 39 38
36 30 35 31 32 25 40 23
28 32 33 25 0 0 0 0
32 28 25 33 29 22 26 38
25 33 0 0 0 0 0 0
32 28 25 33 29 22 38 23
32 25 28 33 22 26 38 23
32 28 25 33 22 26 40 19
28 32 25 33 29 22 23 26
35 30 36 34 26 23 33 25
32 28 25 33 30 36 34 35
28 32 33 25 29 22 39 26
36 34 30 35 41 31 33 24
31 24 30 36 34 26 23 29
36 34 35 31 25 6 5 2
37 39 27 38 25 35 33 32
39 37 27 40 31 19 28 23
32 28 25 33 37 35 38 23
32 28 25 33 40 34 38 19
34 35 36 24 23 33 44 18
36 35 31 34 22 29 23 26
39 37 40 38 32 28 25 33
32 28 25 33 22 29 35 23
```

Appendix C

Modules Abbreviation File-91/92

```
aied
cvis
ias
isc
es
kri1
kri2
lisp
mvis
mthr
nlp
prol
essp
clg1
clg2
lng1
lng2
fsm1
nnet
sp1
sp2
os
db
cpin
swl
grph
cad
ccom
ssc
fprg
algc
cca
ds
cc
aps
cafr
```

```
bsv
si
asicd
spd
ccomp
rs1
rs2
cp
```

Appendix D

Modules List-90/91

1> AI and Education
2> Computational Vision
3> Intelligent Assembly Systems
4> Intelligent Sensing and Control
5> Expert Systems
6> Knowledge Representation and Inference I
7> Knowledge Representation and Inference II
8> Programming in Lisp
9> Machine Vision
10> Mathematical Reasoning
11> Natural Language Processing
12> Programming in Prolog
13> Robot Sensor Engineering
14> Computational Linguistics 1
15> Computational Linguistics 2
16> Linguistics 1
17> Linguistics 2
18> Neural Networks
19> System Integration
20> ASIC design
21> Systolic Processing Design
22> Operating Systems
23> Database Systems
24> Computability and Intractibility
25> Software for Parallel Computers
26> Computer Graphics
27> Computer Aided Design
28> Computer Communications
29> Programming techniques and software tools
30> Formal Programming Language Semantics
31> Algebraic Complexity
32> Advanced Computer Architecture
33> Distributed Systems
34> Communicating Systems
35> Applicative Programming and Specification
36> Computer Aided Formal Reasoning

37> Behavioural Specification and Verification
38> Computational Complexity

Appendix E

AI Student Exams File-90/91

```
8 6 7 16 17 14 15 1
8 12 6 7 4 10 5 1
8 12 6 7 11 5 23 24
8 12 6 7 1 5 29 22
8 12 6 7 18 5 10 25
8 12 6 9 4 5 3 13
8 12 6 7 4 5 29 25
8 12 6 9 4 3 5 2
8 12 6 7 10 11 5 29
8 6 7 9 4 3 13 25
8 12 6 7 10 18 11 5
8 12 6 7 9 18 5 2
8 12 6 7 4 18 11 2
8 12 6 9 4 3 13 25
8 12 6 7 10 18 11 5
8 12 6 7 18 11 1 2
8 12 6 9 4 3 13 25
8 12 6 7 10 18 11 5
8 6 7 16 17 14 15 1
8 12 6 9 4 11 3 2
8 12 6 7 4 1 3 2
8 12 6 7 9 10 11 5
8 12 6 7 4 10 11 5
8 12 6 7 10 18 25 26
8 12 6 7 4 3 32 25
8 12 6 9 4 11 3 2
8 6 7 16 17 14 15 5
8 12 6 9 4 3 2 13
8 12 6 7 18 5 25 26
8 12 6 7 10 5 11 18
8 12 6 7 4 18 11 5
8 12 6 7 4 5 29 25
8 12 6 7 10 11 5 26
```

Appendix F

CS Student Exams File-90/91

```
32 35 36 34 28 33 30 25
32 36 28 33 26 29 25 6
32 28 23 33 22 29 25 8
32 20 27 33 22 25 19 21
32 20 37 34 25 19 21 9
32 28 23 33 22 29 25 18
32 28 33 22 29 25 8 9
32 38 24 28 33 30 29 25
32 28 33 25 0 0 0 0
32 28 23 33 22 29 25 6
32 20 28 33 22 29 25 19
32 36 28 33 22 29 25 12
35 20 37 36 34 26 19 21
31 35 37 36 38 34 30 21
```

Appendix G

Modules Abbreviation File-90/91

```
its
cvis
ias
isc
es
kri1
kri2
lisp
mvis
mthr
nlp
prol
robsn
clg1
clg2
lng1
lng2
nnet
si
asicd
spd
os
db
cpin
swl
grph
cad
ccom
swt
fprg
algc
acarc
ds
csys
aps
cafr
```

```
bsv
ccomp
```

Appendix H

The Make File

```
ga : ga.o objf.o ran.o kernal.o cross.o reports.o utility.o
cc ga.o objf.o ran.o kernal.o cross.o reports.o utility.o -lm -o ga
ga.o : ga.c datastru.h global.h
cc -O -c ga.c
objf.o : objf.c datastru.h
cc -O -c objf.c
ran.o : ran.c
cc -O -c ran.c
kernal.o : kernal.c datastru.h global.h
cc -O -c kernal.c
cross.o : cross.c datastru.h global.h
cc -O -c cross.c
reports.o : reports.c datastru.h global.h
cc -O -c reports.c
utility.o : utility.c datastru.h global.h
cc -O -c utility.c
```

Appendix I

Objective Function

```
/* objf.c */

#include "datastru.h"
#include "global.h"

objfunc(pop)
population pop;
{
        int i,punishment;

        for(i=0;i<popsize;i++)
        {
           punishment =pop[i].count+1;

           /* user can define the objective function here */
           /* pop[i].fitness=1.0/punishment; */
           pop[i].fitness=1.0/punishment/punishment;
        }
}
```

Appendix J

GAs Timetable Scheduling Prototype Using Prolog

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%   Title  : GAs in scheduling prototype using one-point crossover
%   Author : Hsiao-Lan Fang
%   Date   : 15, June 1992
%   Description :
%                 Scheduling/timetabling problems are highly
%           computationally complex problems which often have
%           highly domain-dependent features (ie: a particular
%           timetabling problem will be complicated by details
%           of the particular domain) and thus tend to need
%           highly domain specific optimisation algorithms.
%           GAs, however, provide a way of separating the
%           optimisation algorithm from domain knowledge which
%           has led to promising and useful results in the areas
%           of scheduling and timetabling, as well as several
%           other areas. The trick GAs use is to separate a roughly
%           domain independent optimisation technique (mutation
%           and crossover operators applied to populations of
%           chromosomes) from the domain-specific aspects of a
%           problem (the evaluation function for the chromosomes).
%   Data : as shown in Table 2-1.
%   Remarks : 1. two exams not at the same time for same student.
%             2. not continuous two exams for same student.
%   Sample Run: gene(6,8).   % 6 exams and 8 slots
%              Enter seed (0.0 --- 1.0) > 0.5.
%              Enter Population size (1 --- 250) > 20.
%              Enter Generation size (1 --- 10) > 10.
%   Reference: [Holland 75], [Goldberg 89], [Davis 91]
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
%%%%%%%%%%%%%%%%%%%
%  INITIALIZATION
%%%%%%%%%%%%%%%%%%%

/*
 *  gene(+Exams,+Slots).
 *     Function: main predicate of genetic algorithm.
 */
gene(Exams,Slots):-
        initdata,
        randomize,
        initialize(Gensize,Popsize,Exams,Pop,Sta,Fit,Totalfitness,Slots),
        generation(Pop,_,Exams,Gensize,Popsize,Totalfitness,Fit,Sta,Slots).


/*
 *  initialize(-Gensize,-Popsize,+Lchrom,-Pop,-Sta,-Fit,-Totalfitness,+Slots).
 *     Function: create three array -- Pop, Sta, Fit and initialize
 *               zero generation then decode and compute fitness.
 */
initialize(Gensize,Popsize,Lchrom,Pop,Sta,Fit,Totalfitness,Slots):-
        repeat,
        nl,write('Enter Population size (1 --- 250) > '),ttyflush,
        read(Popsize),
        ((Popsize >= 1,
          Popsize =< 250,!) ;
         fail),
        repeat,
        nl,write('Enter Generation size (1 --- 10) > '),ttyflush,
        read(Gensize),
        ((Gensize >= 1,
          Gensize =< 10,!) ;
         fail),
        nl,write('initializating...'), ttyflush,
        create_array(Popsize,Lchrom,Pop,chrom),
        create_array(Popsize,1,Sta,conflict),
        create_array(Popsize,1,Fit,fitness),
        initialpop(Popsize,Lchrom,Pop,Slots),
        decode(Popsize,Slots,Lchrom,Pop,Sta,Totalconflict),
        objfunc(Popsize,1,Sta,Fit,Totalconflict,Totalfitness),
        nl,nl,pr(Pop,Sta,Fit,Popsize,1),
        nl,write('Generation 0'),
        Averagefitness is Totalfitness / Popsize,
        write('     ***** Average fitness: '),
        write(Averagefitness),write(' *****'),nl,nl,
        write('processing...'),nl,nl.


/*
 *  initialpop(+Popsize,+Lchrom,+Pop,Slots)
 *     Function: set random number in the population of zero generation.
 */
```

```
initialpop(Popsize,Lchrom,Pop,Slots):- !,
        initialpop(Popsize,1,Lchrom,1,Pop,1,Slots).

initialpop(Popsize,Np,_,_,_,_,_):-
        Np > Popsize,!.
initialpop(Popsize,Np,Lchrom,NL,Pop,I,Slots):-
        NL > Lchrom,!,
        Np1 is Np + 1,
        initialpop(Popsize,Np1,Lchrom,1,Pop,I,Slots).
initialpop(Popsize,Np,Lchrom,NL,Pop,I,Slots):-
        rnd(1,Slots,Val,I),
        set_array(Pop,Np,NL,Val),
        NL1 is NL + 1,
        I1 is I + 1,
        initialpop(Popsize,Np,Lchrom,NL1,Pop,I1,Slots).



%%%%%%%%%%%%%%%%%%%%%%%%%
%  CROSSOVER & MUTATION
%%%%%%%%%%%%%%%%%%%%%%%%%

/*
 *  generation(+Pop,-Pop1,+Lchrom,+Gensize,+Popsize,+Totalfitness,+Fit,
 *             +Sta,+Slots).
 *     Function: create new population Pop1 and assign suitable initial
 *               value.
 */
generation(Pop,Pop1,Lchrom,Gensize,Popsize,Totalfitness,Fit,Sta,Slots):-
        create_array(Popsize,Lchrom,Pop1,chrom),
        generation(Pop,Pop1,Lchrom,Gensize,1,Popsize,1,Totalfitness,1,Fit,
                   Sta,Slots).

/*
 *  generation(+Pop,+Pop1,+Lchrom,+Gensize,+Ng,+Popsize,+Np,+Totalfitness,
 *             +I,+Fit,+Sta,Slots).
 *     Function: 1. assign initial value of Generation, Population and Index
 *                  and create new Sta1 and Fit1.
 *               2. select two parents then corssover and mutate from old to
 *                  new generation.
 *               3. decode and compute new statistic data and fitness.
 */
generation(_,_,_,Gensize,Ng,_,_,_,_,_,_,_):-
        Ng > Gensize, !,
        retractall(database(_)),
        retractall(exam_stu(_,_)).
generation(_,Pop1,Lchrom,Gensize,Ng,Popsize,Np,_,I,_,_,Slots):-
        Np > Popsize, !,
        create_array(Popsize,1,Sta1,conflict),
        create_array(Popsize,1,Fit1,fitness),
        decode(Popsize,Slots,Lchrom,Pop1,Sta1,Totalconflict1),
        objfunc(Popsize,1,Sta1,Fit1,Totalconflict1,Totalfitness1),
```

```
        nl,nl,pr(Pop1,Sta1,Fit1,Popsize,1),
        nl,write('Generation '),write(Ng),
        Averagefitness is Totalfitness1 / Popsize,
        write('     ***** '),write(' Average fitness: '),
        write(Averagefitness),write(' *****'),nl,nl,
%       unix(system(clear)),
        write('processing...'),nl,nl,
        Ng1 is Ng + 1, nl,
        create_array(Popsize,Lchrom,Pop2,chrom),
        generation(Pop1,Pop2,Lchrom,Gensize,Ng1,Popsize,1,Totalfitness1,I,Fit1,
                   Sta1,Slots).
generation(Pop,Pop1,Lchrom,Gensize,Ng,Popsize,Np,Totalfitness,I,Fit,Sta,Slots):-
        select(Fit,Popsize,Totalfitness,Parent1,I),
        I1 is I + 1,
        select(Fit,Popsize,Totalfitness,Parent2,I1),
        I2 is I1 + 1,
        write('  '),write(Parent1),write('*'),write(Parent2),
        crossover(Pop,Parent1,Parent2,Pop1,Np,Lchrom,0.6,0.01,I2,Slots),
        I3 is I2 + 1, Np1 is Np + 2,
        generation(Pop,Pop1,Lchrom,Gensize,Ng,Popsize,Np1,Totalfitness,I3,Fit,
                   Sta,Slots).


/*
 * crossover(+Pop,+Parent1,+Parent2,+Pop1,+Np,+Lchrom,+Pcross,+Pmutation,+I,
            +Slots).
 *     Function: 1. filp then decide the position to crossover
 *               2. mutation the front parts and the back parts.
 */
crossover(Pop,Parent1,Parent2,Pop1,Np,Lchrom,Pcross,Pmutation,I,Slots):-
        flip(Pcross,Boolean,I),
        ((Boolean == 1, Lchrom1 is Lchrom - 1, I1 is I + 1,
          rnd(1,Lchrom1,Jcross,I1), !) ;
         Jcross is Lchrom, I1 is I + 1 ),
        I2 is I1 + 1,
        mutation(Pop,Parent1,Pop1,Np,1,Jcross,Pmutation,I2,Slots),
        I3 is I2 + 1, Np1 is Np + 1,
        mutation(Pop,Parent2,Pop1,Np1,1,Jcross,Pmutation,I3,Slots),
        ((Jcross < Lchrom, Jcross1 is Jcross + 1,
          I4 is I3 + 1,
          mutation(Pop,Parent2,Pop1,Np,Jcross1,Lchrom,Pmutation,I4,Slots),
          I5 is I4 + 1,
          mutation(Pop,Parent1,Pop1,Np1,Jcross1,Lchrom,Pmutation,I5,Slots),!) ;
        write('') ),
        write('^'), write(Jcross).



/*
 *  mutation(+Pop,+Parent,+Pop1,+Np,+Start,+End,+Pmutation,+I,+Slots).
 *     Function: 1. filp then decide which position to mutate.
 *               2. randomize and decide increase(+1) or decrease(-1)
```

```
 *                  the value of the position.
 */
mutation(_,_,_,_,Start,End,_,_,_):-
        Start > End, !.
mutation(Pop,Parent,Pop1,Np,Start,End,Pmutation,I,Slots):-
        flip(Pmutation,Boolean,I),
        I1 is I + 1,
        ((Boolean == 1, random(I1,Number),
          get_array(Pop,Parent,Start,Val),
          transfer(Number,Val,Newval,Slots),
          set_array(Pop1,Np,Start,Newval), !) ;
         (get_array(Pop,Parent,Start,Val),
          set_array(Pop1,Np,Start,Val)) ),
        I2 is I1 + 1, Newstart is Start + 1,
        mutation(Pop,Parent,Pop1,Np,Newstart,End,Pmutation,I2,Slots).


/*
 *  transfer(+Number,+Val,-Newval,+Slots).
 *     Function: 1. if Number > 0.5 then increase else decrease.
 *               2. if > Slots then let new value equal 1,
 *                  else if < 1 then let new value equal Slots,
 *                  else let new value unchanged.
 */
transfer(Number,Val,Newval,Slots):-
        ((Number > 0.5, Numtemp is Val + 1, !) ;
         Numtemp is Val - 1 ),
        ((Numtemp > Slots, Newval is 1, !,write(Newval),nl) ;
         (Numtemp < 1, Newval is Slots, !) ;
         Newval is Numtemp ).


/*
 *  select(+Fit,+Popsize,+Totalfitness,-Parent,+I).
 *     Function: 1. random generate a number and multiple total fitness
 *                  to get a value to select.
 *               2. use Roulette wheel with slots method.
 *               3  decide which parent to select.
 */
select(Fit,Popsize,Totalfitness,Parent,I):-
        random(I,Number),
        Rand is Number * Totalfitness,
        accumulate(Fit,Popsize,Rand,0,Parent,0).


/*
 *  accumulate(+Fit,+Popsize,+Rand,+Partsum,-Parent,+I).
 *     Function: 1. accumulate fitness until large enough.
 *               2. simple reproductin allocates offsprings using a
 *                  roulette wheel with slots sized according to fitness.
 */
accumulate(_,Popsize,_,_,Popsize,Popsize):- !.
```

```
accumulate(_,_,Rand,Partsum,I,I):-
        Partsum >= Rand, !.
accumulate(Fit,Popsize,Rand,Partsum,Parent,Sofarparent):-
        Sofarparent1 is Sofarparent + 1,
        get_array(Fit,Sofarparent1,1,Fitness),
        Partsum1 is Partsum + Fitness,
        accumulate(Fit,Popsize,Rand,Partsum1,Parent,Sofarparent1).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  DECODE & OBJECT FUNCTION
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
 *  decode(+Popsize,+Date,+Lchrom,+Pop,+Sta,-Totalconflict).
 *     Function: 1. extract the first chrom from array to list.
 *               2. set initial values and continue decode.
 */
decode(Popsize,Date,Lchrom,Pop,Sta,Totalconflict):- !,
        extract(2,1,Lchrom,Pop,List),
        decode(Popsize,1,Date,1,Lchrom,Pop,Sta,0,List,0,Totalconflict).

/*
 *  decode(+Popsize,+Np,+Date,+Nd,+Lchrom,+Pop,+Sta,+Count,+List,+Subtot,
 *         -Totalconflict).
 *     Function: 1. check at least which two exams at time slot N.
 *               2. check is there any student take the exams at the
 *                  same time slot.
 *               3. if 2. is yes, then increase counter.
 *               4. mate neighbour pairs.
 *               5. increase time slot and go to 1.
 */
decode(Popsize,Np,_,_,_,_,_,_,_,Totalconflict,Totalconflict):-
        Np > Popsize, !.
decode(Popsize,Np,Date,Nd,Lchrom,Pop,Sta,Count,_,Subtot,Totalconflict):-
        Nd > Date, !,
        set_array(Sta,Np,1,Count),
        Subtotal is Subtot + Count,
        Np1 is Np + 1,
        extract(Popsize,Np1,Lchrom,Pop,List1),
        decode(Popsize,Np1,Date,1,Lchrom,Pop,Sta,0,List1,Subtotal,
               Totalconflict).
decode(Popsize,Np,Date,Nd,Lchrom,Pop,Sta,Count,List,Subtot,Totalconflict):-
        memberpos(Nd,List,Memberposlist1),
        length(Memberposlist1,N),
        Nd1 is Nd + 1,
        memberpos(Nd1,List,Memberposlist2),
        ((N > 1,
          findall( E1/E2 ,permutation(Memberposlist1,[E1,E2|_]),Perm),
          unique(Perm,[],Permlist),
          countness(Permlist,Count,Count1), !) ;
```

```
         Count1 is Count ),
        neighbour(Memberposlist1,Memberposlist2,Mixpairs),
        countness(Mixpairs,Count1,Count2),
        decode(Popsize,Np,Date,Nd1,Lchrom,Pop,Sta,Count2,List,Subtot,
               Totalconflict).


/*
 *  objfunc(+Popsize,+Np,+Sta,+Fit,+Totalconflict,-Totalfitness).
 *     Function: set suitable initial values
 */
objfunc(Popsize,Np,Sta,Fit,Totalconflict,Totalfitness):- !,
        objfunc(Popsize,Np,Sta,Fit,Totalconflict,0,Totalfitness).

/*
 *  objfunc(+Popsize,+Np,+Sta,+Fit,+Totalconflict,+Subtotfitness,-Totalfitness).
 *     Function: objection function  =
 *                     [ (20 - Conflict) / 20]  ** 10
 */
objfunc(Popsize,Np,_,_,_,Totalfitness,Totalfitness):-
        Np > Popsize, !.
objfunc(Popsize,Np,Sta,Fit,Totalconflict,Subtotfitness,Totalfitness):-
        get_array(Sta,Np,1,Conflict),
        Count1 is 20 - Conflict,
        ((Count1 < 0,
          Fitness is 0, !) ;
         (Fitness1 is Count1 / 20,
          power(Fitness1,10,Fitness)) ),
        Totfitness is Subtotfitness + Fitness,
        set_array(Fit,Np,1,Fitness),
        Np1 is Np + 1,
        objfunc(Popsize,Np1,Sta,Fit,Totalconflict,Totfitness,Totalfitness).


/*
 *  countnes(+Permlist,+Count,-Count1).
 *     Function: check how many counts that the same student take two
 *               exams at the same time.
 */
countness([],Count,Count):- !.
countness([ E1/E2 |T],Count,Count1):-
          exam_stu(E1,S),
          exam_stu(E2,S),
          Counttemp is Count + 1,
          countness(T,Counttemp,Count1).
countness([_|T],Count,Count1):-
          Counttemp is Count,
          countness(T,Counttemp,Count1).


/*
 *  unique(+Perm,+Emptylist,-Permlist),
```

```
 *     Function: take off duplicate in the Perm
 */
unique([],L,L) :- !.
unique([H|T],L,U) :-
        membership(H,L),!,unique(T,L,U).
unique([H|T],L,U) :- !,
        unique(T,[H|L],U).


/*
 *  extract(+Popsize,+Np,+NL,+Pop,-List).
 *     Function: extract chrom from array to list.
 */
extract(Popsize,Np,NL,Pop,List):- !,
        extract(Popsize,Np,NL,Pop,[],List).

extract(Popsize,Np,_,_,_,_):-
        Np > Popsize, !.
extract(_,_,NL,_,Y,Y):-
        NL < 1, !.
extract(Popsize,Np,NL,Pop,X,Z):-
        get_array(Pop,Np,NL,Val),
        append([Val],X,Y),
        NL1 is NL - 1,
        extract(Popsize,Np,NL1,Pop,Y,Z).


/*
 *  memberpos(+Nd,+List,+Memberposlist).
 *     Function: 1. extract member position to a list.
 *               2. set suitable initial values.
 */
memberpos(Nd,List,Memberposlist):- !,
        memberpos(Nd,List,1,[],Memberposlist).

/*
 *  memberpos(+Nd,+List,+Memberposlist).
 *     Function: compute the member position.
 */
memberpos(_,[],_,X,X):- !.
memberpos(H,[H|T],Num,X,Y):- !,
        Num1 is Num + 1,
        memberpos(H,T,Num1,[Num|X],Y).
memberpos(H,[_|T],Num,X,Y):-
        Num1 is Num + 1,
        memberpos(H,T,Num1,X,Y).


/*
 *  neighbour(+LIst1,+List2,-List3).
 *     Function: mate each element of List1 with List2 then put the
 *               pairs in List3.
```

```
 */
neighbour(_,[],[]):- !.
neighbour([],_,[]):- !.
neighbour(H1,[H2|T2],[H1/H2|T3]):-
        number(H1), !,
        neighbour(H1,T2,T3).
neighbour([H1|T1],List2,List3):-
        neighbour(H1,List2,SofarH),
        neighbour(T1,List2,SofarT),
        append(SofarH,SofarT,List3).


/*
 *  pr(+Pop,+Sta,+Fit,+popsize,+I)
 *     Function: print report
 */
pr(_,_,_,Popsize,I):-
        I > Popsize, !,nl.
pr(Pop,Sta,Fit,Popsize,I):-
        nl,
        ((I < 10,
          write('  '), !) ;
         (I < 100,
          write(' '), !) ;
          write('') ),
        write(I),
        write(' '),arg(I,Sta,Y),write(Y),
        write(' '), arg(I,Pop,X),write(X),
%       write(' '),arg(I,Fit,Z),write(Z),
        I1 is I + 1,
        pr(Pop,Sta,Fit,Popsize,I1).



%%%%%%%%%%%%%%%%%%%%%%%
%  array manipulation
%%%%%%%%%%%%%%%%%%%%%%%

/*
 *  create_array(+Row,+Column,+Array,+Name).
 *     Function: create a Row * Column dimension array.
 */
create_array(R,C,Array,Name):-
        functor(Array,Name,R),
        init_array(Array,1,R,C,Name).


/*
 *  init_array(+Array,+CurrentRow,+Row,+Column).
 *     Function: initialize all elements of the array into arguments of
 *               functor.
 */
```

```
init_array(_,CurrentR,R,_,_):-
        CurrentR > R,!.
init_array(Array,CurrentR,R,C,Name):-
        functor(F,Name,C),
        arg(CurrentR,Array,F),
        NewR is CurrentR + 1,
        init_array(Array,NewR,R,C,Name).


/*
 *  set_array(+Array,+Row,+Column,+Value).
 *     Function: set the value of the array.
 */
set_array(Array,R,C,Value):-
        arg(R,Array,F),
        arg(C,F,Value).


/*
 *  get_array(+Array,+Row,+Column,-Value).
 *     Function: extract the value of the array.
 */
get_array(Array,R,C,Value):-
        arg(R,Array,F),
        arg(C,F,Value).



%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%  Random number generator
%%%%%%%%%%%%%%%%%%%%%%%%%%%%

/*
 *  random(+Index,-Number).
 *     Function: return a real pseudorandom number between 0 and 1.
 */
random(Index,Number):-
        I is Index mod 55,
%        retract(database(X)),
        database(X),
        ((I = 1,Index > 1,
          retract(database(X)),
          create_array(1,55,New,newrandom),
          advance_random(X,New), get_array(New,1,1,Number),
          assert(database(New)), !) ;
         (I == 0, get_array(X,1,55,Number)
%           , assert(database(X))
          , !) ;
         (get_array(X,1,I,Number)
%           , assert(database(X))
         ) ).
```

```
/*
 *  flip(+Probability,-Boolean,+I).
 *     Function: return a Boolean value with a specified probability.
 */
flip(1,1,_):-!.
flip(Probability,Boolean,I):-
        ((random(I,Number),
          Number =< Probability,
          Boolean is 1, !) ;
         Boolean is 0 ).


/*
 *  rnd(+Low,+High,-Val,+I).
 *     Function: return an integer selected uniformly and pseudorandomly
 *               between upper and lower limits.
 */
rnd(Low,High,Low,_):-
        Low >= High,!.
rnd(Low,High,Val,I):-
        J is High - Low + 1,
        random(I,Number),
        JJ is Number * J + Low,
        Trimint is JJ // 1,
        ((Trimint > High,
          Val is High, !) ;
         Val is Trimint ).


/*
 *  randomize.
 *     Function: queries terminal for user-specified random seed and
 *               initializes random predicate.
 */
randomize:-
        unix(system(clear)),
        repeat,
        write('Enter seed (0.0 --- 1.0) > '),ttyflush,
        read(Randomseed),
        ((Randomseed > 0.0,
          Randomseed < 1.0,!) ;
         fail),
        warmup_random(Randomseed).


/*
 *  advance_random(+Oldrandom,-Newrandom).
 *     Function: set suitable initial values to advance_random.
 */
advance_random(Oldrandom,Newrandom):-
        advarand(24,1,Oldrandom,Newrandom),
```

```
        advarand(55,25,Oldrandom,Newrandom).

/*
 *  advarand(+Count,+CurrentCount,+Oldrandom,-Newrandom).
 *     Function: retrive a new batch of pseudorandom numbers.
 */
advarand(Count,CurrentCount,_,_):-
        CurrentCount > Count, !.
advarand(Count,CurrentCount,Oldrandom,Newrandom):-
        get_array(Oldrandom,1,CurrentCount,Temp1),
        ((Count<25,I1 is CurrentCount+25,get_array(Oldrandom,1,I1,Temp2),!);
         (I1 is CurrentCount - 24, get_array(Newrandom,1,I1,Temp2)) ),
        Newrandom1 is Temp1 - Temp2,
        ((Newrandom1 < 0.0, Newrandom2 is Newrandom1 + 1.0, !) ;
         Newrandom2 is Newrandom1),
        set_array(Newrandom,1,CurrentCount,Newrandom2),
        Nc is CurrentCount + 1,
        advarand(Count,Nc,Oldrandom,Newrandom).


/*
 *  warmup_random(+Randomseed).
 *     Function: initialize random number generator.
 */
warmup_random(Randomseed):-
        create_array(1,55,Oldrandom,oldrandom),
        set_array(Oldrandom,1,55,Randomseed),
        Newrandom is 0.000000009,
        Prevrandom is Randomseed,
        initrand(54,1,Oldrandom,Newrandom,Prevrandom),
        create_array(1,55,New1,new1),
        advance_random(Oldrandom,New1),
        create_array(1,55,New2,new2),
        advance_random(New1,New2),
        create_array(1,55,New,newrandom),
        advance_random(New2,New),
        assert(database(New)).


/*
 *  initrand(+Count,+CurrentCount,+Oldrandom,-Newrandom,+Prevrandom):-
 *     Function: compute new values for random number.
 */
initrand(Count,CurrentCount,_,_,_):-
        CurrentCount > Count, !.
initrand(Count,CurrentCount,Oldrandom,Newrandom,Prevrandom):-
        Temp is 21 * CurrentCount,
        I1 is Temp mod 55,
        set_array(Oldrandom,1,I1,Newrandom),
        Newrandom1 is Prevrandom - Newrandom,
        ((Newrandom1 < 0.0, Newrandom2 is Newrandom1 + 1.0, !) ;
         Newrandom2 is Newrandom1),
```

```
        get_array(Oldrandom,1,I1,Prevrandom1),
        Nc is CurrentCount + 1,
        initrand(Count,Nc,Oldrandom,Newrandom2,Prevrandom1).



%%%%%%%%%%%%%%%%%%%%%
%  UTILITY PREDICAE
%%%%%%%%%%%%%%%%%%%%%

/*
 *  append(+LeftList,+RightList,-ResultList)
 *     Function: attaches RightList to the end of LeftList, returning
 *               the resulting list in ResultList.
 */
append([],L,L).
append([H|T],L,[H|U]) :-
        append(T,L,U).


/*
 *  permutation(+List,-Permulist).
 *     Function: permute a list to all possible permutation.
 */
permutation([],[]).
permutation([X|L],P):-
        permutation(L,L1),
        insert(X,L1,P).


/*
 *  insert(+Atom,?List,?BiggerList).
 *     Function: insert an atom to a list at any possible position.
 */
insert(X,List,BiggerList):-
        del(X,BiggerList,List).


/*
 *  delete(+Atom,?BiggerList,?List).
 *     Function: delete an atom from a biggerList.
 */
del(X,[X|Tail],Tail).
del(X,[Y|Tail],[Y|Tail1]):-
        del(X,Tail,Tail1).


/*
 *  membership(+Item,+List)
 *     Function: check if an atom or atom1/atom2 pair is a member of List.
 */
membership(H,[H|_]).
```

```
membership(E1/E2,[E2/E1|_]).
membership(I,[_|T]) :-
        membership(I,T).


/*
 *  power(+X,+Y,-R)
 *     Function: X to the power of Y is R.
 */
power(0,0,0):- !.
power(_,0,1):- !.
power(X,1,X):- !.
power(X,Y,R):-
        power(X,Y,1,R).

power(_,0,Final,Final):- !.
power(A,P,Accum,Final):-
        Np is P - 1,
        Newaccum is Accum * A,
        power(A,Np,Newaccum,Final).



%%%%%%%%%%%%%%
%   Database
%%%%%%%%%%%%%%

/*
 *  initdata.
 *     Function: add all exam & student pairs into database.
 */
initdata:-
        student_exams( 1,[1,2,4,6]),
        student_exams( 2,[1,2,4,6]),
        student_exams( 3,[1,2,4,5]),
        student_exams( 4,[1,4,5]),
        student_exams( 5,[4]),
        student_exams( 6,[4,5]),
        student_exams( 7,[2,4]),
        student_exams( 8,[2,4,6]),
        student_exams( 9,[3,4,6]),
        student_exams(10,[3,4,6]).


student_exams(_,[]):- !.
student_exams(Student_Num,[Exam_Head|Exam_Tail]):-
        assertz(exam_stu(Exam_Head,Student_Num)),
        student_exams(Student_Num,Exam_Tail).
```