Genetic Algorithm Time Tabling

final project for "Introduction to Artificial Intelligence"
Shimon
HUJI March 2010

Abstract

This project is an experimental attempt at solving Time Tabling Problems through the use of a genetic algorithm.

The Time Tabling Problem is a well-known combinatorial search problem that consists of scheduling events over a period of time. The events being scheduled usually require resources such as rooms and people and are usually subject to certain constraints (such as: a person cannot attend more than one event at the same time).

The problem often arises in real-life situations, solving these instances of the problem from scratch is way beyond humans (at HUJI no software is used to come up with time-tables, each year's time-table is a slight variation of the preceding year's one).

Practically all available time-tabling research and software products concern themselves strictly with finding good feasible solutions. Trying to find an optimal solution is too hard of a problem, such attempts are simply not justified by practical considerations - searching for non-optimal solutions often yields worthy results and incurs relatively low costs.

The variant of the problem this project deals with is coming up with time-tables for courses being taught at a university.

Genetic algorithms are a search technique inspired by evolutionary biology. A population of solutions is maintained and managed through the use of genetic operators – selecting some solutions that are better than others, combining them and modifying them. The process is executed in a continuous iterative manner, and as time progresses better solutions are found.

Genetic algorithms are often used when dealing with hard optimization problems and are widely applied with the time-tabling problem, mainly due to their ability to traverse sparse search spaces, the likes of which the TT problem has.

The underlying purpose of this project is to explore, from the grounds up, the behaviors that genetic algorithms exhibit and the phenomena that manifest during the evolutionary process.

For that purpose, a Java application was designed that lets the user observe the different states of the population over the course of the evolutionary process. It is possible to halt the process, change the algorithm's configuration (namely – the genetic operators being used) and continue execution. This enables a better understanding of the tendencies that are present and of the effects the genetic operators have on these.

Commonly used genetic operators were implemented, and several unique ones were designed as well. This paper reviews the strengths and weaknesses of these operators, as well as the genetic algorithm as a whole.
1 Overview

Section 2 "Quick application usage review" is present in order to introduce the application's functionality, so that the reader may use it alongside reading this document.

Section 3 "Problem model" contains a formalization of the chosen time-tabling problem.

Section 4 "Genetic representation" surveys a core part of each genetic algorithm – the encoding used to represent instances of problem solutions.

Section 5 "Fitness" covers another fundamental aspect of genetic algorithms – the way solutions are scored.

Section 6 "Evolution and the basic genetic operators" deals with the mechanics of the genetic algorithm implemented, well-known genetic operators are described.

Section 7 "Advanced genetic operators" reviews the genetic operators that were designed specifically for the problem at hand.

Section 8 "Test cases and observations" introduces 4 test cases that were used in order to examine the available genetic operators. Observations about these and general observations about the genetic process are included.

Section 9 "Conclusions and possibilities" sums up the main conclusions this project has brought forth, as well as the possible future developments that logically follow from these.

Section 10 "References" lists the most helpful resources that were used in preparing this project, some of these are included in the submission.
# Quick application usage review

This section introduces the main parts of the application and their functionality.
It is advised to read this alongside experimenting with the application.

- The left 3 tabs of the application window – *Classrooms*, *Teachers*, and *Courses* are used for creating and editing the database entities this application deals with. Interaction is quite intuitive, some nitty gritty details that are not obvious are specified in the next section.

- The *Solver* tab is where the algorithm is configured and ran from. The application comes with several variants of most of the genetic operators that are a part of the evolutionary process. You can set an initial configuration for the algorithm's run and change it in the midst of the process:
  - Try opening "01 – Easy.ttp" and pressing "Start" with the default configuration set.
  - Press "Pause" after a few hundreds of generations.
  - Change the Selection type to "Tournament" and press "Resume".
  - The effect of the change on the average fitness and standard deviation is apparent.
  - In order to stop the run and unfreeze the application, press "Finish" (the best solution found will be applied over the database).

- The *Schedule* tab is used in order to survey solutions. You can view the resulting time-table for each classroom\teacher\course. A time-slot in the classroom and teacher views has a background color that corresponds to the entity's availability at that time-slot (see next section for details). The backgrounds of time-slots belonging to lectures that have a manually fixed classroom & time are grey.
  - You can click a time-slot in order to get detailed information about it. The cell itself includes informative icons.

Figure 1 explained:
- The top row shows icons related to the lecture and to the course it's a part of:
  - The *X* icon means that another lecture that belongs to a conflicting course is taught during this time-slot. The *A* icon means that this lecture is taught at a room that does not have all the required study-aids. The *S* icon means that the room doesn't have enough seats to host all of the lecture’s expected participants.
  - The middle row shows icons related to the lecture's teacher:
    - The *X* icon means the teacher has another lecture assigned at this time-slot. The blank red square means that the teacher is unavailable for work at that time-slot.
  - The bottom row shows icons related to the lecture's classroom:
    - The *X* icon means another lecture is being taught at the classroom at that time-slot.
    - The blank red square means that the classroom is unavailable at that time-slot.
3 Problem model

This section describes the formal structure of the time-tableing problem instance this project focuses on – creating a feasible\optimal time-table for the courses of a university. It is advised to read this alongside inspecting the left three tabs of the application: classrooms, teachers, and courses.

A time-tableing problem, as considered in this project, is composed of:

- A set of classrooms, where each classroom has a certain capacity, certain available study aids and an availability time-table (see Figure 2). The availability of a classroom in each hour of the week may be one of:
  - not available (e.g. due to cleaning)
  - prefer (e.g. preferred hours of a lab's staff)
  - avoid (hours that are preferably not to be scheduled for use)
  - none of the above
  Each classroom also has a unique id and a unique name.

- A set of teachers, where each teacher has an availability time-table (the same as mentioned above), a unique id and a unique name.

- A set of courses. Each course is comprised of a set of lectures. The scheduling times for lectures that belong to the same course must not overlap, it is intended for students of the course to participate in them all. On top of that, a course has a (possibly empty) set of conflicting courses. The lectures of these courses must not overlap with the course's lectures. (E.g. the times for Linear Algebra and Infinitesimal Calculus cannot overlap). Each course also has a unique id, a unique name and a unique abbreviated name (for display purposes).

![Figure 2 – an availability time-table](image)
Each lecture has the following manually set properties: a teacher, a duration, an expected number of participants and a set of required study-aids.

Each lecture also has a classroom and a time. These last two properties may be assigned manually in advance, or marked as (auto), in order to let the solver come up with a scheduling.

Each lecture also has a type (either lecture, recitation, or lab), and a serial identifier.

A couple of notes:
- The conflicting course relation is symmetric, i.e. when you define course A as a conflicting course of course B, course B will automatically be defined as a conflicting course of course A.
- Concerning time, HUJI's practices were applied: a week is treated as having five study-days, each of which consists of twelve periods (8:00-20:00).
- If a vast course exists for which we actually wish to let lectures overlap (e.g. Infi2 has many lectures, and a student can choose which he wishes to go to), the course should be split, and entered into the system as more than one course to begin with.
- If you really want, you can make a certain lecture invalid by deleting its teacher. The algorithm will disregard such lectures and a scheduling will not be made for them.

A solution to a problem instance is a mapping: \( \text{Lectures} \rightarrow \text{Classrooms} \& \text{Times} \)

Where Lectures is the set of all lectures (of all courses) for which either a classroom, a time, or both were not manually fixed.

A valid solution is a solution that does not entail any impossibilities (see Section 5). The algorithm's solution should be valid and should satisfy the preferences of teachers\'classrooms as much as possible, as well as lectures' needed study-aids, and needed number of seats.

3.1 Observations about search space

The size of the search space grows exponentially with the size of the input. The time-tabling problem has been proven to an NP-Complete one.

When regarding the fitness of a candidate solution as a function of very many parameters, we observe that this function is anything but continuous – excellent solutions may lie next to terrible ones. Further more, the variables this function takes are dependent on one another, their values cannot be found separately.

The search spaces of most realistic time-tabling problems are sparse with feasible solutions – a search procedure traversing such spaces should be able to bridge gaps of low fitness between high fitness subspaces.

These characteristics make the stochastic multi-directional nature of genetic algorithms suitable for the problem at hand.
4 Genetic representation

In choosing what encoding to use when representing solutions (chromosomes), there are generally three approaches one can take:

1. **Binary representation** – the commonly termed "Classical Genetic Algorithm" (Holland, 1975) represents a chromosome by a string of ones and zeros. There is a two-way translation procedure between these strings and an actual domain-specific solution (in our case – classrooms & times for all unfixed lectures). As discussed in [1] the main advantage of using this type of representation (besides its domain-independence) is the possible soundness of the search-space sampling that the algorithm can attain.

E.g. if we have chromosomes of length 4, we may wish that the subspace "1***" (where the asterisks stand for wildcards – the subspace of all solutions that have 1 at the first bit position) be represented by an eighth of the population at any given moment. Such guarantees enable performing the search in a more systematic, mathematical manner.

2. **Implicit representation** – as discussed in [2, 10] a solution's representation must not be an absolute valuation to the problem's variables. It can be a higher-level construct that bounds\guides a range of possible solutions.

E.g. a solution instance of the form "ABCD" can mean: "use heuristic A to schedule first event, use a heuristic B to schedule second event" and so forth.

The main advantage of using such encodings is that the search may be narrowed down to smaller and much fitter subspaces.

3. **Direct representation** – the representation type which is the most common for the TT Problem, and the one which is used by this implementation. We represent a solution by a simple valuation of a classroom & a time for each unfixed lecture the problem has. More precisely:

A **Gene** is an unfixed lecture (whose classroom\time\both the algorithm needs to assign).
An **Allele** consists of a classroom for the lecture to take place in, and of a time for it to start at – a day and an hour. A **Chromosome** is an array of such alleles for the given problem's genes.

The straight forward advantages of using such a representation are mainly simplicity, and the ease of applying domain-dependent genetic operators, which this implementation has – no translation or interpretation of the encoding is needed.
5 Fitness

This section deals with a fundamental design decision one has to make when constructing a genetic algorithm – how to score devised solutions in order to differentiate between good and bad ones. Calculating fitness for solutions is one of the phases of the evolutionary process (which is what the next section is about), it is reviewed here because of its preliminary importance.

The fitness operator is applied on a candidate solution (a chromosome) and yields a positive number in the (0,2) range. This number represents the solution’s absolute fitness it determines how good a chromosome is (higher is better), and is the sole factor affecting its chances to reproduce and spread its genes. (Later on, the selection process uses this absolute fitness in order to come up with a relative fitness, that is dependent upon other members of the population).

As suggested in [3] the fitness operator this implementation facilitates differentiates between two kinds of constrains:

- **Hard constraints** – the violation of which result in a logical impossibility (e.g. a teacher cannot attend two lectures simultaneously) or an infringement of the problem's bounds of validity (e.g. if a classroom was set to be unavailable at a certain time, it is treated as such – no valid solution can have a lecture scheduled for it at that time).
  Other hard constraints: a teacher may be unavailable at a certain time, no more than one lecture can be scheduled for a classroom at any given time, lectures of conflicting courses cannot overlap.

- **Soft constraints** – the violation of which does not constitute lack of feasibility for a candidate solution, but it should be avoided as much as possible.
  These include: missing classroom seats, missing classroom study-aids, classroom availability (prefer, avoid, or none – unavailable is considered a hard constraint) and teacher availability (the same).

In order to come up with a fitness score for a chromosome, the fitness operator goes over all the hours of all the lectures the solution entails and sums up penalties for violated constraints, using the following formula:

\[
 f(\text{chromosome}) = \frac{1}{1 + |\{\text{hard}\}|} + \frac{\delta}{1 + \sum_{s \in \{\text{soft}\}} w(s)}
\]
Where \{\text{hard}\} is the set of all violated hard constraints encountered, and \{\text{soft}\} is the set of all soft ones, \(w(s)\) is the weight associated with a soft constraint \(s\) (see note below), and \(\delta\) is zero until \{\text{hard}\} is empty.

Meaning: we do not account for soft constraints until all hard ones have been satisfied. This mechanism ensures satisfying many soft constraints never comes at the expense of violating even a single hard one.

**A note about soft constraints**

The project's problem definition includes "positive" preferences of entities — teachers and rooms can prefer certain times of the week. One approach in order to account for this would be: awarding chromosomes that satisfy such preferences by subtracting from their sum of penalties seen so far.

However, this would lead to negative sums under certain circumstances (great chromosomes that do not violate any hard or soft constraints, rather only satisfy preferences). This could be amended by normalizing the fitnesses of the population's members, it would be enough for the evolutionary mechanism to work (the selection process can operate perfectly well on a generation of chromosomes supplied only with their relative fitnesses in respect to one another).

One major disadvantage this approach has: we would lose the ability to give absolute scores to chromosomes that will remain relevant in future generations as well, there will be no common ground for scoring. This would somewhat harm the possibility of designing an intelligent evolutionary process that takes into account past states of the population and calibrates itself accordingly (the design of which is one of the conclusions of this project, see Section 9).

Instead, what is practiced in this implementation is penalizing all soft constraints (even the "positive" prefer ones) – at different extents. The algorithm is wired to penalize avoid by 1, none by 0.5 and prefer by 0.333. This overcomes the issue described above, and also sets a power-balance between types of soft constraints: The algorithm will prefer (1 avoid, 3 prefer's) just a bit better than (4 none's), i.e. we agree to assign an avoid if it gets us 3 prefer's.
6 Evolution and the basic genetic operators

This section briefly reviews the evolutionary process, as implemented in this project, and presents the classical genetic operators the application is equipped with. It is advised to read this alongside reviewing the application's Solver tab. The following figure illustrates the evolutionary state-machine, a discussion about each phase follows.

![Evolutionary State-Machine](image)

6.1 Spawn

The spawn operation occurs a single time in each evolutionary process. It is used to create the initial population of chromosomes, before commencing the long cycle of evolution. Spawn operators:

- **Random** – randomly assigns classrooms and times.
- **Jump start** – in order to speed up the initial phase of the evolutionary process (in which the chromosomes fitnesses are usually quite terrible) the algorithm will attempt to spawn valid solutions i.e. ones that do not break any hard-constraints. This is done by randomly choosing a classroom and a time for each lecture in a way that does not conflict with assignments done so far. A limited number of attempts are performed before giving up.

6.2 Selection

The selection operation occurs at the start of each evolutionary phase. $N$ pairs of chromosomes are selected for breeding, where $N$ is the chosen size of the population.
All selection operators select chromosomes, one way or another, according to their fitness (with Random being the only exception). The "Deny incest" option is available with all selection operators, if enabled then mating chromosomes must not be identical. Enabling this helps to keep the population diverse, and to avoid premature convergence. Basic selection operators:

- **Random** – randomly selects couples for mating.

- **Roulette** – we calculate a relative fitness for each chromosome – the fraction of the chromosome’s absolute fitness out of the sum of absolute fitnesses of all members of the population. This fraction then serves each chromosome as its probability of being chosen for reproduction. If the "exponential" checkbox is enabled then before applying the roulette process we take the square root on the absolute fitness of each chromosome. Enabling this acts as another counter measure against premature convergence – the evolution process may have a snow-ball effect where an individual almost doubles its presence in the population with each generation, taking the square root relaxes this possible tendency.

- **Linear Ranking** – a selection method that does not take into account chromosomes’ absolute fitnesses, nor their relative ones, only their rank is considered (in a population of 10 chromosomes the best one has rank 10, the worst one rank 1). Basic ranking would have simply treated these ranks as new fitnesses i.e. with a 10-sized population the sum of fitnesses would have been 1+2+...+10=55, and the probability of the best chromosome to reproduce would have been 10/55, the second best 9/55, etc.

Linear ranking allows you to control the slope of the graph of these new rank-based fitnesses using the "selective pressure" numerical parameter – this controls the slope of the graph – the steeper the graph (the higher the parameter is), the more favorable the selection process will be towards chromosomes with high fitnesses. The formula used to compose fitness according to rank is a slight variation of the one described in [7]:

$$new\ fitness(rank\ i) = 1 - sp + \frac{2*sp*i}{\text{popSize} - 1}$$

Where sp is the selective pressure and popSize is the size of the population. The following graph shows the effect the selective pressure parameter has on the graph of new fitnesses:

![Figure 5 – Linear ranking: blue line for sp 0.2, green line for sp 0.8](image)
• **Tournament** – another selection method that lets you control how harsh the natural selection process is. In order to choose each and every parent chromosome for mating, a tournament is held: a certain number of individuals are randomly picked out of the population and the highest fitness one is chosen. The "tournament size" parameter lets you specify the size of the tournament as a fraction of the population size. As with the selective pressure parameter, the higher this parameter is the more favorable the selection process is towards higher fitnesses.

### 6.3 Crossover

The crossover (or: recombination) process combines each previously chosen pair of parent chromosomes into a pair of offspring chromosomes. Basic crossover operators:

• **One point** – a random position is chosen along the genes array. All alleles up to that point are copied from one parent, all alleles after that are copied from the other. The second offspring is created as the complement of the first one.

• **Uniform** – for each and every gene of the first offspring a coin is tossed in order to decide from which parent to acquire the allele. The second offspring is created as the complement of the first one.

### 6.4 Mutation

The mutation operation is primarily responsible for introducing new genetic material into the population i.e. it is a major part of the Genetic Algorithm's explorative nature. After offsprings are created by the crossover operation, each allele of each offspring undergoes mutation with the probability set by the "Mutation rate" control. Basic mutation operators:

• **Random** – if an allele is to be mutated, a coin is tossed in order to decide what to mutate (either changing the classroom, or changing the time). If the classroom is to be mutated, a random one is chosen blindly. If the lecture's time is to be mutated, another coin is tossed in order to decide whether to change the day or the hour. For either one, a new one is chosen blindly.

• **Gentle Random** – the same as above, only that we consider the changing of a room as a major change, the changing of a lecture's day as a less drastic one, and the changing of a lecture's hour as a mild one.
The more massive the change type is, the less frequently we perform it: for every 12 hour changes a single day change is performed, for every 5 day changes a single room change is performed.

This mutation operator is meant to be less coarse then the basic random mutation one, and to generally apply more subtle changes in order not to ruin already established chromosomes.

6.5 Replacement

The replacement (or: re-insertion) operation is responsible for composing the next generation of chromosomes out of the previous generation and its offsprings. The "Elitists fraction" control specifies how many of the previous generation's best chromosomes automatically survive to participate in the next generation (a feature termed elitism). This mechanism ensures the search will never lose its best solution seen so far, there is a minimum of one elitist (even if you supply zero for this parameter). The parameter is given in terms of a fraction of the population size. Besides elitists, the next generation is composed of the offsprings of the previous one. Replacement operators:

- **Both offsprings** – when re-inserting chromosomes into the population, both offsprings created by the crossover process out of a pair of parent chromosomes are taken into account.

- **Best offspring** – out of each pair offsprings created from a pair of parents only the best one is allowed to participate in the next generation. This option acts as another counter measure against the hopefully avoided possibility of dominance by a few high fitness chromosomes.

6.6 Measure

This phase is not actually a core part of the genetic process, it does not affect the population in any way, it only performs statistical measurement of it – what it the best fitness, the average one, the standard deviation of the population. Using these measurements (which the application's GUI presents) one can realize the state of the population – how good are members of it, and how diverse is it.

Additionally, every 500 generations a check is performed to see if all members of the population are identical. If so, the algorithm stops with an informative message. This scenario is expected to happen only for very small problems, generally the algorithm does not stop until the user presses the "Pause" button.
7 Advanced genetic operators

This section reviews the genetic operators that are unique to this project. In a way, these operators make the final genetic algorithm presented here a basic hybrid algorithm – an algorithm that combines search methods (the other search method, besides the genetic one, being a form of greedy search).

7.1 The DFS Crossover operator

This operator is an extended version of the Uniform Crossover operator - when two parent chromosomes are to be mated each allele (except for a few) is randomly chosen to be taken either from the first parent, or from the second. As for the few other alleles (which are randomly chosen), all possible combinations are tried out for them. The best combination and the second best are each placed in the genomes of the pair of offsprings.

Checking all possible combinations is done with a simple DFS recursion. The "Recursion depth" control parameter specifies for how many alleles are the combinations checked, which is the same as specifying the recursion's depth. Possible values for this parameter are \([0,10]\), anything above 6 causes the algorithm to be too slow.

This operator is meant to be a means against the possibly destructive force of random recombination. It's effect is mild, but definitely there.

7.2 The Greedy Mutation operator

This operator is of a Lamarckian type – the individual is actively improving. If an allele is to be mutated, we flip a coin in order to decide whether to change its room or time. Whichever one we choose, we compose a list of all possible values for the variable that will not produce any new impossibilities in the time-table represented by the allele's chromosome. We then randomly pick out a value for the allele's variable out of the list.

This operator is meant to help members of the population achieve validity, it resolves many situations in which the basic genetic operators simply can't find a feasible solution.

7.3 The Permute Mutation operator

This operator is meant to be used on already strongly established solutions, that require further strengthening. Half of the time this operator acts as the Random Mutation operator, at other times it switches between the classroom & time (allele values) of two lectures (genes) already present in the about-to-be-mutated chromosome.

If an allele is to be mutated, the room\time values of all other alleles are inspected and a list of candidate alleles for switching is composed (an allele is accepted to the list if a switch
between the alleles' values will not incur a validity breach). A random allele is chosen from the list, and the room\time values between the two alleles are switched. A side effect of applying this operator is that the population's average fitness goes down, this is part and parcel of the exploration taken.

7.4 The Fluctuate Mutation operator

A combination of the Greedy Mutation operator and the Permute Mutation operator. Applies greedy mutation for 400 generations, and then permute mutation for a hundred. This is a basic attempt to get the best of both worlds – the greedy mutation may get stuck at local maxima, the permute mutation often resolves this and allows further advancement. A more sophisticated mutation operator, of an adaptive type, may be developed (see Section 9).
8 Test cases and observations

This section goes over a few test cases in order to review the application’s performance, some relevant observations are introduced.

A general point which should be made concerns what was termed in this paper the "Basic Genetic Operators" – for each basic selection or crossover operator there exist some circumstances in which it does a bit better than others. However, the differences are usually subtle enough to be ignored. Generally speaking, where One-Point Crossover fails – so does Uniform Selection, where Roulette Selection fails so does Linear Ranking Selection and Tournament Selection fail (when the "selection pressure" and the "tournament size" parameters are fixed).

Failure, in this context, is usually getting stuck on a local maxima of fitness – either converging prematurely into a single solution (hardly ever happens), or constantly surveying solutions in a certain subspace, when a better solution exists outside this subspace.

It is important to note that The Linear Ranking Selection and Tournament Selection operators do actually hold great potential because they are parameterized. One of the conclusions of this project is that further advance can be made by designing an adaptive genetic algorithm, such a device can use these two selection operators.

Another point concerns the chosen population size – unless extremely small or large sizes are chosen, it seems that the same evolutionary phenomena occur. This leads to the conclusion that the population size should be proportional (in some way) to the size of the search space being explored. Too small of a population will never reach a good solution, a population too large would needlessly incur lengthy computations and will not improve to the final result.

Following are the four supplied test cases.
8.1 Test case 1: "01 - Easy.ttp"

This test case consists of 4 classrooms, 3 teachers and 27 lectures. There are very many valid solutions and many good ones where preferences of entities are satisfied – the problem is not constrained in any way. All possible configurations of the Solver reach a valid solution quickly. However, even in this simple case we can observe the flaw that exists in the classical genetic operators – they do not optimize. Different combinations of classical selection\crossover operators usually yield a score of ~1.04 whereas when using either one of Greedy, Permute or Fluctuate mutations we go over the 1.05 threshold.

8.2 Test case 2: "02 – Big Picture.ttp"

This test case consists of a very simple scenario, it is included here in order to more clearly demonstrate the pitfalls that a genetic search may encounter. There is a single classroom with only 8 available study periods. There is 1 course with 8 lectures, each taught by a different teacher. Each one of the teachers prefers to teach in one of the available time slots the classrooms has (they all prefer different time slots). Again, all configurations of the algorithms reach validity in a split second. However, the basic genetic operators seem to miss the "big picture" – they assign at best 3-4 teachers to their desired time-slots, and quickly converge to a single solution. The mechanics of large instances of time-tabling problems are fundamentally the same as in this problem – previously assigned classroom times may prevent reaching an optimal solution, the search contains dead-ends. With this in mind the Permute mutation operator was designed, and indeed it finds the optimum quickly after spawning.
8.3 Test case 3: "03 – Local Maxima.ttp"

This test case consists of a plausible setup, but has many dead-ends in its search-space. It is like test case 2 in the sense that there exist an orderings of assignments which will yield a desired result, but they are fewer.

The classical genetic operators stagnated at a local maximum of 1/2 or 1/3 in about half of the trials (Figure 7A).

The simple greedy character of the DFS Selection and Greedy Mutation yields a valid solution quite quickly, and even optimizes it a bit in later generations (Figure 7B).

The combination of these two operators together makes sense – the Greedy Mutation operator establishes valid solutions, and the DFS Selection combines them in a way that (over many generations) does not ruin their validity.

![Figure 7A](image1.png) – a common stagnation case for Roulette\Linear Ranking\Tournament selection combined with One-Point\Uniform crossover and Random mutation

![Figure 7B](image2.png) – combining DFS Selection and Greedy Mutation reaches validity and optimization
8.4 Test case 4: "04 – Constrained.ttp"

This test case would probably be considered an easy real-life scenario by advanced timetabling systems. For the submitted application, the algorithm is pushed to its best. The problem consists of 3 Rooms with a total of 100 available hours, 12 teachers with no availability or preferences and 10 courses comprising 39 lectures and a total of 80 needed hours.

The lectures have varying lengths, required study-aids and student attendance. The rooms have many non-available hours and the ones that are available are far from being continuous. Additionally, each room has a different capacity and a different array of available study aids, that suit only part of the lectures. The problem was designed so that optimal solutions exist, but only a few of them.

The classical genetic operators hardly ever reach validity.

The combination used for Test case 3 – the DFS Crossover & Greedy Mutation – usually does not attain validity, it stagnates on a plateau (Figure 8A). Even if validity is reached, the optimal fitness of 1.0235 is hardly ever reached.

The Fluctuate Mutation operator solves some of the dead-ends the above combination reaches (Figure 8B), given enough time it will solve them all, in the 1000 generation range 3 out of every 4 trials succeed. A major plus is that after validity is reached – optimality is attained by the Permute Mutation part of the Fluctuate Mutation operator.

![Figure 8A – common case for DFS crossover + Greedy Mutation](image)

![Figure 8B – common case for the Fluctuate Mutation operation. The advance around generation 430 is due to the Permute Mutation part of the Fluctuate Mutation operator.](image)
9 Conclusions and possibilities

The first most striking fact that reveals itself when attempting to solve a time-tabling problem that is anything over trivial, is the helplessness a naive genetic algorithm may posses. Without some fine-tuning and parameter tweaking the algorithm usually lets its convergent nature rule out its explorative one.

Even when fine-tuning is performed for a certain problem, it quickly becomes obvious that it is relevant only for a certain type of similar problems. The inescapable conclusion is that either we develop domain-specific genetic operators that will produce a sound genetic search for many types of problem configurations, or we remain domain-independent and design an adaptive genetic mechanism that tweaks its own parameters according to the state of the population (e.g. increasing mutation when variance in the population is low). A third option, and probably the best one, would be a compromise between the two.

This implementation includes basic forms of the above mentioned designs:

The Greedy Mutation operator is a domain-specific operator and applying it, as mentioned in this paper, is certainly useful. However, in order to deal with harder problems this operator is certainly not enough. The first advance that could be made in extending this operator is to let it know which genes are causing troubles, and have it work on them. (Currently the Greedy Mutation operator picks out random genes to work on).

More generally, we may design an algorithm that recognizes bottlenecks (e.g. a few specific time-slots of a certain classroom that are highly desired by many lectures), and revolves the search around them – when a certain lecture gets these time-slots a subspace is defined, the algorithm can knowingly travel these subspaces and switch between them when a good solution doesn’t seem to be anywhere near.

It would also be possible to design a corrective mutation operator that tries to improve a solution from a different perspective than the one used by Greedy Mutation – for example, trying to fill up the time-slots of classrooms as tightly as possible, trying not to assign lectures to classrooms that can host many more students than the lecture requires, etc. – the bottom line being an attempt to keep the search as least constrained as possible.

The general approach expressed here may seem as deviating too far from the classical genetic one. The fact of the matter is that many research papers today and certainly most commercial products are guided by this sort of hybrid design ideas.

Concerning an adaptive genetic algorithm, the Fluctuate Mutation operator shows how a dynamic genetic algorithm can greatly resolve issues that arise with the regular non-dynamic algorithm. This operator is by no means adaptive, it changes the algorithm's operation.
periodically in order to arouse it from slumbers of local maxima. But it is enough to demonstrate the potential that a truly adaptive algorithm holds.

And lastly, the topic discussed in [10] (that usually gets little or no attention) – it may be possible to design an intelligent crossover operator. The classical genetic algorithm relies mostly on selection in order to rule out bad offsprings that a naive crossover procedure produces. However, achieving intelligent crossover may help the search both with gaining feasible solutions and with optimizing them.

The **DFS Crossover** is a basic step towards this direction and it is productive mainly for small problems. The obvious problem with this idea is the sheer complexity involved with deciding how to crossover a pair of parent chromosomes, but one may use domain-specific heuristics in order to overcome this.
10 References

Darrell Whitely, Colorado State University

[2] Evolutionary Timetabling: Practice, Prospects and Work in Progress
Dave Corne, Peter Ross, Hsiao-Lan Fang, University of Edinburgh

[3] An Evolutionary Algorithm for solving the School Time-Tabling Problem
Calogero Di Stefano, Andrea G. B. Tettamanzi

[4] Comparison of fitness scaling functions in genetic algorithms with applications to optical processing
Farzad A. Sadjadi, University of Minnesota

[5] Introduction to Genetic Algorithms, Marek Obitko

http://www.rennard.org/alife/english/gavintrgb.html

[7] GEATbx
http://www.geatbx.com/docu/algindex.html

Knut-Edvart Ellingsenl, Manuel Penaloza

Wilhelm Erbnen, Jurgen Keppler

[10] Specialised Recombinative Operators for Timetabling Problems
Edmund Burke, David Elliman, Rupert Weare, University of Nottingham