# Timetable Scheduling via Genetic Algorithm

Final Year Thesis 2019

Andrew Reid East

National University of Ireland, Galway

Computer Science and Information Technology

# Timetable scheduling via genetic algorithm

Final year thesis by

Andrew Reid East

In partial fulfilment of the Requirements
for the Degree of
Bachelor of Science

April 2019

National University of Ireland, Galway

Computer Science and Information Technology

## Abstract

A genetic algorithm machine learning technique is utilised to create course timetables for a university. Outputted timetables do not have any conflicting scheduled modules for lecturers, students, or venues, and the algorithm can improve timetables with user preferences if feasible within a schedule's constraints. The algorithm runs efficiently on an enterprise Java application server running on the cloud, presented over a web API to control the algorithm's execution. All entities to be scheduled into a timetable are stored within a database to be retrieved to be scheduled. A service-oriented architecture was designed to contain the algorithm and separate it from the database service, which has its own API, along with a user interface running as a modern web app. This final year thesis describes genetic algorithms' usage in the context of the problem domain and covers the implementation details of this application.

# Table of Contents

# List of Figures

# 1.0 Introduction and Literature Review

## 1.1 History

While learning machines influenced by the principles of evolution, such as heredity, mutation, and natural selection, were first proposed by Alan Turing in 1950 [1], the structured ideas for algorithmic operations that form the basis for genetic algorithms were first popularised by John Holland in 1975 [2] as a way to model the biological processes of evolution, as well as by his students, such as Kenneth De Jong [3], who helped to devise practical engineering applications on increasingly powerful hardware and bring genetic algorithms into their modern era of being applied to real machine learning problems. Genetic algorithms have a long history of academic study and many novel industrial applications, where they have been used in optimisation problems to generate sometimes surprising effect. Because of their strength in producing optimal solutions when the total number of possible solutions of a problem is dauntingly large or otherwise difficult to capture in its entirety, genetic algorithms will continue to be a useful approach to solving classes of problems in many applicable domains.

A genetic algorithm is a learning algorithm capable of searching for an optimised solution based upon creating smaller building blocks that guide towards and make up an ideal solution. This contrasts with search techniques that require being initialised with *a priori* knowledge and a model of an archetypal, optimal solution. The foundation of genetic algorithm techniques is the natural processes of evolution of species via sexual reproduction of individuals and gene mutation of DNA, but a genetic algorithm is an abstraction and application rather than a strict representation of biological concepts. As a computer science algorithm, this technique has sound mathematical roots. [4], [5], [6]

## 1.2 Description of a Genetic Algorithm

Genetic algorithms are guided by the principle that a more fit individual has a reproductive advantage to produce more offspring. This was proposed by Charles Darwin: individuals in a population with the forms best adapted to their specific environment, no matter how minor the advantage, would be more likely to live longer and "rear more young, which would tend to inherit these slight peculiarities" [7, p. 6296]. This concept has been translated by analogy to the computer science field of AI search as genetic algorithms. The analogy relates biological individuals in a population to possible solutions of a search and relates being more fit to being ranked closer to the global optimum value of the search. A genetic algorithm creates a population of individual candidate solutions, represented by virtualised strands of DNA, called chromosomes, and simulates evolution over time by subjecting these mock-genetic structures to analogues of biological genetic processes. A single candidate solution is broken up into pieces of the solution, termed genes, and the suitability of that solution is judged by interpreting these genes altogether within the constraints of the problem domain, done by a fitness function. The output of this function for an individual is the numeric analogue to Darwin's biological fitness. As with genetic selection in the natural world, genetic algorithms do not require any outside-imposed order, direction, or foreknowledge of the exact nature of the optimal solution [4].

Further, genetic algorithms operate well in large and complex domains, where enough knowledge about the domain is unknown that it is impossible to realistically develop a complete heuristic to support traditional search methods [3]. A genetic algorithm has the advantage of building up a complex solution from smaller pieces of knowledge; it can work from the basis of not having complete domain knowledge. A genetic algorithm may discover essential characteristics of the problem itself while the algorithm is in progress [4].

This is the primary advantage of utilising a genetic algorithm for certain search applications: building on only a partial set of the domain knowledge of the complete system (enough to compare an individual candidate solution to another), they can create novel, optimal results that could not be predicted during the design phase, and yet they still can be implemented with a straightforward algorithm [4]. Further, even though they depend on randomness in the core operation and are in the class of probabilistic search algorithms, because they have elements of directed and stochastic searches, genetic algorithms can robustly converge on solutions that would be inefficient for other methods [4].

## 1.3 Flow of the Algorithm

The structure of a genetic algorithm itself is a linear loop (Figure 1.1 [4]). The procedure may be summarised as: over many generations, individuals compete based on their fitness to be selected more often in the next generation, sometimes creating new individuals based upon existing ones through crossover and mutation, repeating the generations until an individual emerges as the desired optimal solution. Uncomplicated as the algorithm may be, having individual candidate solutions compete over many iterations yields the optimal solution emerging from the population. This process is stochastic but well-founded: its operations have been demonstrated to work based on mathematical principles of probability [4].

Beyond this basic summary, to implement a genetic algorithm there are several more details that must be determined [4]:

1. Representation of candidate solutions (chromosomes) as data structures
2. Creation of an initial population of chromosomes
3. Evaluation of the fitness of a chromosome
4. Genetic operators that work upon chromosomes
5. Parameters for the above details, such as rates of applying genetic operators and for how many generations to run
6. When to terminate the algorithm

These parameters will be covered in detail, in the context of the available literature on genetic algorithms.

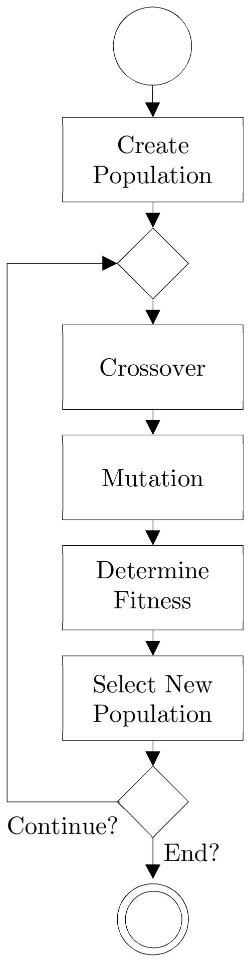


Figure 1.1: A genetic algorithm's main loop

## 1.4 Data Structures

To operate, a genetic algorithm must devise a representation of any possible candidate solution to the problem as a data structure that is analogous to a chromosome, with a linear series of individual attributes of the solution, or genes [4]. To support genetic crossover, each gene must have some aspects that can be swapped with other genes while still maintaining the validity as a candidate solution. Each gene should be able to be distorted in a random way to implement the idea of genetic mutation. Finally, whatever representative data structure is chosen to hold a candidate solution, data must be decoded to have its fitness judged. A traditional choice for representing chromosomes is a bit string, either where each individual bit corresponds to a gene or several bits are grouped to make up the various aspects of a gene, and each of these sets of bits in series makes up the whole bit string chromosome. Some problem domains can also be represented by more complex strings, such as hexadecimal strings, floating point arrays, or graph/tree/vertex/edge encodings [4]. Bit strings such as this support the abstracted genetic operators well, but also require a tight coupling between the genetic operators and the details of a candidate solution.

While bit strings are a traditional choice for representing genes in chromosomes in a genetic algorithm, mathematical analysis has shown that it is not the use of abstracted bit strings themselves that makes this search technique successful [3]. Rather, it is important to ensure that whichever internal representation of data is used allows for the formation of building block to still occur and thus allow better solutions to tend to emerge via crossover (see §1.9.2 below). Research has demonstrated that utilising object-oriented techniques in the design of a genetic algorithm program can not only result in feasible run-times but also provides for the decoupling of the details of each gene from the chromosome itself [8], [9], [10]. This decoupling gives a flexibility that more abstract solutions, such as bit strings, do not. When programming with object-oriented data structures, it is possible to implement detailed interaction between genes within chromosomes. This loses sight of the fact that genetic algorithms' mathematical operation depends on abstracted genes to be independently recombined to create objective, probabilistic searching [4, p. 463]. Therefore, when programming each of the genetic operators these complex objects should be are encapsulated and treated as abstract genes with no properties.

## 1.5 Creating a Population

With an appropriate representation created, the first step for the execution of the algorithm is to create a population of individuals. Maintaining many such potential candidate solutions at once, grouped into a population, is they key idea behind a genetic algorithm being able to perform a simultaneous, multidirectional search [4]. The size of a population has often been recommended to be in the range of 30 to 60 individuals, summarised from the results of thirteen individual papers by Kazimipour *et al.* [11], but results of a genetic algorithm might be improved by working on a much larger population, such as 300, while increasing processing time only linearly [11].

With a size chosen, the population must be stocked by an initial collection valid (if not optimised) chromosomes, which are typically initialised randomly. This randomness stochastically distributes candidate solutions evenly across the set of all possible solutions, or search space, which prevents solutions from converging on good (but not the best) solutions too quickly, as discussed further in §1.9.3 below.

## 1.6 Fitness Function

Now that it is known how chromosomes will be represented in data structures and initialised, the most important implementation detail of a genetic algorithm can be examined. A fitness function is a goal-oriented evaluation of an individual chromosome [3]. Biological fitness is judged by how well an individual can compete for scarce resources, and in analogy, a genetic algorithm abstracts this concept to numerical comparison of the output of a fitness function for each chromosome. The input to the fitness function is one individual with all its genes, and its output is a fitness value with which to rank that individual. Fitness must be a value with some natural ordering, and positive integers are often utilised.

The fitness function is the core of any genetic algorithm, as it is the only place to encode domain knowledge into the system. The fitness function usually determines fitness by examining the properties of each gene as it relates to other genes in the individual, using domain knowledge to output a value for that comparison, and creating a sum of its attributes. In some implementations, the fitness function instead runs a simulation to create a simulated entity. This entity is subjected to a model which runs a simulation to eventually output a score. The fitness function will be run on every individual in every generation to determine which candidates are worth preserving as a population evolves.

Since it is repeated so often this leads naturally on to careful consideration of the algorithmic complexity and computational cost of executing the fitness function. Low processing time for each invocation is required since over 1,000 generations are often needed [3]. Since the complexity of internal representations and of comparisons needed between each increases along with the complexity of the search space, more complex problem domains require more optimisation of the fitness function. This is especially applicable when the number of attributes in an individual is large e.g. the number of genes per chromosomes: each gene will be compared against every other gene in the chromosome, and thus each invocation of the fitness function has quadratic run time complexity. If too many comparisons are required for a certain problem domain, a genetic algorithm may not be a suitable choice for a search algorithm.

## 1.7 Genetic Operators

Through different operators that are analogous to biological genetic processes, the population in a genetic algorithm experiences simulated evolution, where relatively good solutions thrive, and relatively bad solutions die out. These functions are collectively called genetic operators [4].

### 1.7.1 Crossover

Crossover is the genetic operator analogous to the biological process of DNA recombination, where two parent individuals contribute a portion of their genes during sexual reproduction to make a filial individual, or offspring. This offspring will be different from either parent and may inherit a unique combination of traits which make it superior to either parent. In the genetic algorithm equivalent to this, the most commonly used crossover method is to take half the genes from each chosen parent candidate solution and stitch them together into a new individual, called binary crossover [4]. This may be done at exactly the halfway point in the array of genes. However, breaking a chromosome at the halfway point may end up always breaking apart good building blocks in the individual, or always including bad building blocks with good.

Since the algorithm has no knowledge of which sets of genes in the chromosome are good building blocks or not, it must rely on randomness to stochastically choose above-average blocks. This may be done by choosing a crossover point at random from which to splice and mix genes, choosing multiple points to mix from, choosing random genes but maintaining their relative order, or based on a heuristic to attempt to get intelligent crossovers. These methods have been shown to have novel efficiency gains in certain search domains [4]. Many of these result in varying proportion of each parent being utilised rather than exactly half, but all successful crossover methods share the common result of preserving the structure of those smaller pieces in offspring.

One observation of the different crossover methods is that they select a contiguous range of genes at random from parent chromosomes to cross over. Like the biological recombination it mimics, crossover can create worse solutions the parents, resulting in poor offspring. These will likely be selected against during subsequent generation, which in inherent in the nature of the multi-generational algorithm. However, the alternative can also happen: a beneficial crossover. This occurs when two structured components, one from each parent and each component individually has above-average fitness, are brought together into an offspring. The resultant offspring possesses something that neither parent has: two good components. This is leads to the building block hypothesis, which is discussed further in §1.9.2 below. Therefore, implementations of the crossover operator must have a probability of preserving these structured components, which vary in size by the domain of the problem. No matter the representation utilised, a less complex crossover method is more likely to preserve good building blocks and pass them on to the offspring [4].

The rate at which the crossover operator is applied must also be considered. If increased, the population may converge towards optimal solutions faster, but has if it is happening too often, potentially good candidate solutions will not have the generations they need to develop because they are crowded out by an overabundance of crossed-over offspring. A good starting value to perform crossover with probability at 0.10 [4], but could be has high as 0.9 [12].

### 1.7.2 Mutation

Any solution that has higher fitness than all its closely related peers is a local maximum within the search space, and one of these is the global maximum. In a population of candidate solutions, fitness and selection pick out optimal individuals, and thus members of the population will probabilistically converge on the best solution currently represented within the population. The population will not tend to move itself away from one of the maxima that have emerged and will not search around to find a different maximum. If the only solution represented is merely one of the local maxima, then the algorithm will never locate the global maximum in the search space.

The purpose of mutation is to add diversity within a population with the goal of preventing the algorithm from converging on a local rather than global maximum. Introducing a mutated gene into a chromosome has the chance that an above-average building block will be created out of genes in that chromosome. There is a chance that the new individual is close enough to the global maximum that subsequent generations begin to be pulled towards that maximum. Without mutation, the above-average building blocks in the chromosomes will be likely to be shared among individuals until every individual in a population has reached a local maximum and becomes stagnant.

The most basic mutation method is to randomly select a gene within a chromosome and then to randomly change that gene to any other acceptable value.

This concept can be expanded by instead selecting a random number of randomly selected genes to mutate simultaneously in a chromosome. Other ideas include shuffling a subset of genes in a chromosome, inverting the order or a range of genes, swapping two randomly selected genes, or bit shifting the entire chromosome. Ideas have also been proposed for heuristic mutation, where all possible permutations of any given mutation are performed, their fitness calculated, and the best example chosen as the mutated offspring [4].

Choosing a mutation operator must be done with consideration of maintaining the validity of a candidate solution. For example, if a chromosome must always be made up of every value from set of values with no repetition, then randomising one gene is likely to make a duplicate value and thus an invalid solution. In this situation, genetic diversity comes from the order of those values, thus a swapping or shuffling method will maintain validity while still providing a novel result with a different order of values.

Finally, the rate of mutation can be chosen to be high, which causes the population to be more likely to diverge from local maxima. If good solutions are being discarded randomly too often, then the rate of mutation should be lowered. A good starting value is to perform a mutation with probability 0.01 [4], but some problem domains have seen increased convergence speed using a mutation rate between 0.2 and 0.3 [12].

### 1.7.3 Selection

Once the new individuals have been possibly created through crossover or mutation and the fitness of each individual candidate solution in a population has been judged by the fitness function, the next biological step would be for the strong to increase in number, the weak to die off, and the moderates to merely survive. This is most commonly done via a stochastic roulette-wheel selection, where each candidate solution is first assigned a probability of selection equal to the proportion of their fitness to the population's total fitness [2]. One individual is selected randomly using these probabilities, cloned, and added to a new population. This is repeated until the new population is of the desired size, often equal to the previous number of individuals, but could very generation-over-generation [4]. This new population of candidate solutions is then used to run the next generation of the algorithm.

Many alternatives to this basic selection procedure have been proposed. Some methods dispense with the roulette wheel and discard individuals with the lowest fitness, passing on all others to the new generation [4]. The selection function may first guarantee that the elite, top-fitness chromosomes are passed into the new generation, and then the remaining population is selected with the roulette wheel method, which has been shown to lead to faster convergence on optimal solutions (but extra precaution must be taken to avoid settling on local maxima) [12]. There are different methods of scaling the survival probabilities, with the goal of fixing the number of offspring or evenly scattering the fitness values for more moderate offspring. These modified selection methods have been shown mathematically to produce different effects in population diversity, but the necessary criteria for choosing a selection method is to ensure that the most fit individuals have a higher probability of being passed on to the next generation [4].

As explained in §1.7.1 and §1.7.2 above respectively, the crossover and mutation operators create brand new individuals. A final selection consideration is at which rate to select from the pool of these offspring versus the existing pool of parent individuals. Beginning from the most biologically inspired methods [2], different reproductive schemes have been proposed to ensure that recombined offspring tend to replace their parent chromosomes in new populations. Some methods propose an enlarged sampling space to make two separate selection processes, first selecting from the original population and then selecting from the new offspring, bringing the two together to make a population of the desired size. A final method greatly simplifies the process by selecting from a combined pool of parents and offspring to construct a new population. This will still have the effect of causing poorly-performing parents to be quickly replaced by superior offspring, but it reduces the risk of losing high-quality parent chromosomes due to a strict reproduction plan that follows biological models too closely [4].

## 1.8 Generations

A genetic algorithm will proceed through the above genetic operators cyclically, doing crossover and mutation before selecting a new population on which to repeat the process. Each of these repetitions is called a generation. Working over a long series of generations allows the population to gradually increase in fitness. Consideration must then be given to when to stop the algorithm. The most uncomplicated approach is to stop after a certain number of cycles, with the ideal number determined empirically from running the algorithm on sample input data.

If time and resources allow for long-running operations, a genetic algorithm best practice to overestimate the number of generations so the best solution is more likely to be found, and less likely to have settled into a local maximum [4]. A high number of generations, such as between 500-1,000, is a recommended starting point to observe if solutions emerge [3].

The alternative is to decide when to stop the algorithm by detecting that a reasonable solution has been found. In some domains, candidate solutions are binary: acceptable or not, and finding a single valid solution is enough to terminate the algorithm. In others, all solutions have some validity, and the goal is to find the optimal from these. The algorithm should be provided with criteria, synthesised using domain knowledge, to differentiate a merely adequate solution from a true candidate for the global maximum, such that the algorithm continues until long enough to possibly find that best solution. There may also be hybrid problems, where some solutions are invalid, but the set of valid solutions could be optimised further to get the best solution. These problems may detect when a valid solution is found but then allow the algorithm to run for a set number of generations afterward, giving it a chance to optimise the solution. (In this case, the probabilistic nature of a genetic algorithm requires guarding against the valid solutions being lost in the meantime, however.)

All of these require a problem domain where the validity of a solution is detectable programmatically and efficiently enough to be run every generation. There are many domains where the validity of a solution requires a human's judgement. The implementation of a genetic algorithm in this search space would have to terminate after a straightforward generation count. This follows on to the usefulness of a hybrid method, where the algorithm is paused, a human is requested to behold a solution to judge its acceptableness as a complete solution. After the processing of generations is halted (either by counter or detecting a possible solution), an operator is asked to inspect the solution and determine if the algorithm should be allowed to run for more generations. This method is discussed by Burke *et al.* [13], where the user interface allows for inspecting a population of candidate timetable solutions.

## 1.9 Internal Workings of a Genetic Algorithm

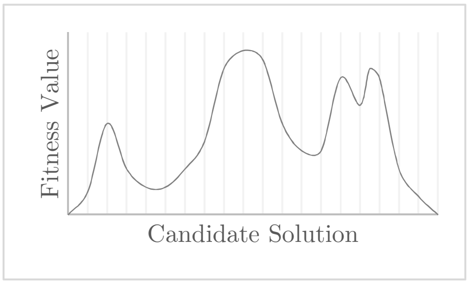
Without delving into the pure mathematics that demonstrate a genetic algorithm's operation, the ability of a genetic algorithm to find an optimal solution without being programmed as to full nature of the solution can be explained through several foundational concepts.

### 1.9.1 The Fitness Landscape

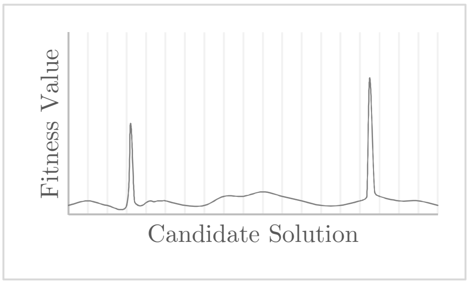
For a genetic algorithm to work within a given problem domain, that search problem must demonstrate a proper fitness landscape, which is a conceptual illustration of the suitability of all possible candidate solutions at once. Determining if a solution can be found with a genetic algorithm for a given search and optimisation problem domain depends on the characteristics of a fitness landscape.

In the study of search and optimisation algorithms, the search space is defined as the range of all possible candidate solutions. A genetic algorithm's search space is a finite set of possible permutations of chromosomes. In a fitness landscape, each possible candidate solution is a value along the x-axis, and they are plotted against their fitness values on the y-axis. The resulting chart is called a fitness landscape, owing to its resemblance to mountainous terrain, showing peaks corresponding to local maxima, as well as the global maximum as the highest peak, and troughs for poor solutions. A search domain with a proper fitness landscape (Figure 1.2a) has the desired characteristic: optimal solutions being sought out (maxima) have a smooth slope going upward towards them. Genetic algorithms need this gradual increase in fitness to work, exhibiting a hill climbing effect from areas of low fitness to areas of high fitness.

An improper fitness landscape for a genetic algorithm (Figure 1.2b) will not allow this hill climbing effect. Such a search space has members separated into two disjoint sets: those with very good fitness and those with very poor fitness. There are no moderate candidate solutions, which would allow gradually finding the very good solutions. A genetic algorithm running a search in this fitness landscape still



(a) Proper fitness landscape



(b) Unsuitable for a genetic algorithm

Figure 1.2: Mock fitness landscapes

does have a non-zero probability through mutation of randomly finding one of the candidate solutions with very good fitness. However, because it does not have the benefit of the gradual hill-climbing mechanism, it will not work towards an optimal solution. The question remaining is how this hill climbing effect takes place during the algorithm's operation.

### 1.9.2 Building Block Hypothesis

The mechanism through which a genetic algorithm follows a hill climbing effect can be explained through the building block hypothesis [6]. A candidate solution can be thought of as being composed of a series of structure elements, or building blocks, where each building block is a set of genes. When crossover happens, these elements will be passed between the two parent chromosomes to create offspring with structured elements from both.

When the fitness of a solution is evaluated in the fitness function, it is done by essentially examining each of the structured elements in turn. When more of the elements in that solution are above the average of the entire population, the solution will be ranked above others, and it will be more likely to be crossed-over with other high-fitness individuals. When crossover happens, both individuals' superior structured elements have a chance of being recombined intact into the offspring, resulting in an offspring with superior blocks from both parents, and possibly better fitness than either parent (and of course, possibly worse). If the offspring does happen to end up with high fitness, it is likely that it, along with those superior building blocks, will be propagated into future generations. The process will be repeated, and more and more copies of the superior building blocks will exist within the population. As this happens, more individuals with high fitness are selected to be in the population, and it will slowly build up to better solutions, thus exhibiting a hill climbing effect (Figure 1.4).

As previously discussed in §1.9.1, improper fitness landscapes occur where optimum solutions are all-or-nothing, with no gradual slope from poor solutions to great ones. This occurs in problem domains where building blocks cannot be interchanged usefully. In these domains a great solution must have all its pieces interlocked together, and no one component structured element would be judged as superior when considered individually or crossed over to another solution. Candidate solutions here do not show high fitness unless all the right blocks have come together perfectly and taking away any of those blocks would make it a poor solution again. This is the opposite of the building block hypothesis, and such problems will not work with a genetic algorithm. If a good piece of the solution cannot be taken out and recombined with others, then a genetic algorithm cannot develop good solutions into better solutions.

While the building block hypothesis is a valid explanation and illustration of the internal operation of a genetic algorithm, more detailed and concrete mathematical work has upheld the solid foundation of the of the workings of a genetic algorithm [4], [5], [6]

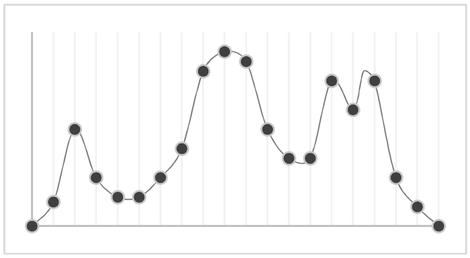
### 1.9.3 A Multipoint Search to Avoid Local Maxima

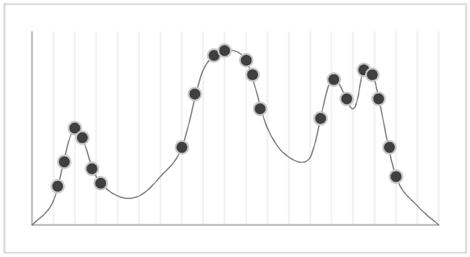
A genetic algorithm considers many points in the search space at once by maintaining a population of candidate solutions [4]. The benefits of a multipoint search are threefold: (1) the search may converge faster in parallel, (2) there may exist backup candidate solutions if a promising subspace in the search ends up being sub-optimal, and (3) having many candidates can prevent converging on local maxima. The most novel of these benefits is the genetic algorithm's ability to avoid premature convergence on local maxima, and it follows from two components of the algorithm.

First, when creating the initial population, individuals should ideally be distributed evenly across the entire search space, such that some individuals will be near any maxima in the fitness landscape, and ideally with some near the global maxima. This uniform distribution can be achieved satisfactorily by initialising the population randomly and having enough individuals to stochastically achieve uniform distribution. However, as the generations proceed, this effect can be cancelled out if the population is culled too judiciously; if only the best individuals are selected to form the next generation, lower-fitness individuals will be lost entirely. An individual that is beginning to hill-climb towards the global maximum, but which is currently less fit than those near a local maximum, will be removed by such a culling, and thus the best solution will not be found. This should be accounted for in the selection operator at the start of every generation, such as with a roulette-wheel selection that gives all individual some chance of being selected [4].

Second, whenever the algorithm allows a mutation to occur, the mutated offspring has some chance of moving a large distance away on the fitness landscape. This may be closer to the global maximum, and that would have the effect of perturbing a mature population away from a local maximum and closer to the global best (Figure 1.3).

Despite these features built into the algorithm to help avoid it, genetic algorithms are not guaranteed to return the globally best solution. In some problem domains, they may sometimes return one of the local maxima instead. An implementation may not account for parts of the problem domain, and it may even tend to return sub-optimal solutions when it runs, terminating before the global maximum is found. If the problem domain is especially unexplored, the creator may not know the characteristics of the best solution, and since a genetic algorithm can still return with subprime solutions in this scenario, the user will not be able to judge if the solution produced is the best. Genetic algorithms are a valid method for solving optimisation problems but require special consideration to judge the worthiness of their solutions.


(a) Initial population


(b) After several generations

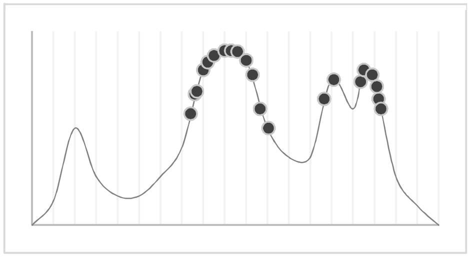
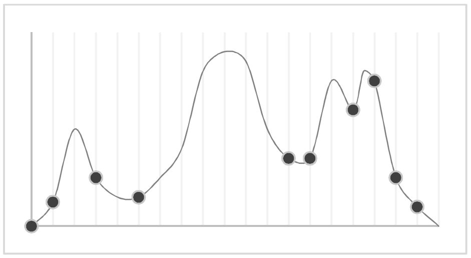
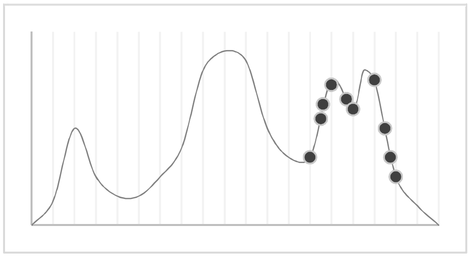

(c) A mature population

Figure 1.4: The building block hypothesis


(a) Poorly distributed initial pop.


(b) Converging on a local maximum

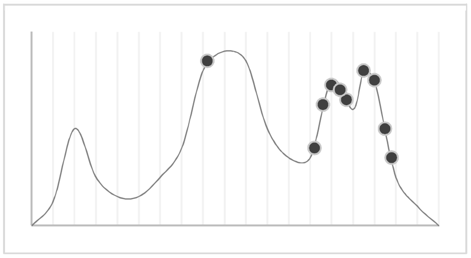

(c) Mutation finds the global maximum

Figure 1.3: Benefits of mutation

# 2.0 Technical Review

The motivation and development of this project was influence both by the study of genetic algorithms, as detailed above, and by previous application of machine learning techniques to the problem domain, along with the strengths and limitations of the current system for creating university timetables

## 2.1 Scheduling Timetables Using a Genetic Algorithm

Now that the mechanical details of genetic algorithms have been described, this class of optimisation algorithm can be analysed in how useful they may be to the given problem domain. Using a genetic algorithm to automatically deal with the complex, interlocking problem of scheduling groups of people with finite resources has been well established in the literature of machine learning.

Genetic algorithms have been used to schedule patients, doctors, and therapeutic devices in the highly-constrained environment of a hospital, establishing each object as abstract resources, with domain knowledge of the system captured as specific rules [14]. A traditional and much-studied problem to utilise genetic algorithms is in exam timetabling, such as by Nie [10] and by Burke *et al.* [13]. A published example of using a genetic algorithm to schedule usage of a truck loading dock provided practical evidence of high convergence speed by utilising a binary crossover at high crossover rate with roulette-wheel selection modified to always select several of the elite, most-fit candidate solutions each generation [12]. An example of utilising a genetic algorithm to schedule university courses with professors into lecture venues has been published by Jacobson and Kanber [9], proposed in a manner akin to the goals of this project and utilising the Java language.

### 2.1.1 Alternative Machine Learning Approaches

There are other search and optimisation algorithms that may be appropriate for scheduling a student timetable, such as a graph colouring algorithm [15], [16]. These algorithms require constructing a globally relevant model, resulting in an NP-complete problem [17]. There have been hybridised approaches to scheduling, where a modified graph colouring algorithm is used to create a population of feasible, but not optimised, timetables, and then utilising a genetic algorithm to select for an optimal timetable [13]. This combined algorithm approach is functional, but a pure genetic algorithm approach is nearly as efficient, and by starting with randomised, invalid timetables and using the fitness function to select for both more feasible and more optimised individuals, a solution may be reached nearly as quickly.

It has been noted by De Jong [3] that if the required domain knowledge is completely known, and it is possible to use this to implement a traditional search algorithm, such an approach should be used. A traditional algorithm can better exploit a surplus of knowledge than a genetic algorithm could.

Because of the flexibility of the fitness function to distil complex notions of real-world domain knowledge of the problem down into local comparisons, a genetic algorithm approach is a superior methodology for scheduling a timetable compared to the presented alternatives.

### 2.1.2 Timetable Scheduler User Interface

Burke *et al.* discuss the non-functional requirements of a genetic algorithm-based exam timetable scheduling application, commenting that the only true test of a quality implementation is if administrators in the institution will use it. Their research discusses developing an application GUI to support the underlying algorithm, including the feature of allowing direct user input to modify the schedule. This enhances the search, filling in where the knowledge encapsulated by the programmed fitness function was not adequate. Proposed is a modified spreadsheet interface, supporting drag-and-drop to manually schedule courses. The GUI idiom of week-view calendar element is widespread and familiar to modern app users, so a calendar interface will be a suitable extension of a spreadsheet.

## 2.2 An Unfulfilled Need in the Problem Domain

The existing published solutions within this problem domain have well covered many technical aspects of the problem of scheduling a timetable [9], [10], [12], [13], [14]. Many of these implementations were created to explore some academic or technical aspect of genetic algorithms, and others have practical benefits. One gap in this domain is a complete system that can meet the end-to-end needs of a large institution, such as a university. One university was studied, and it was found that timetabling is the responsibility of each individual academic discipline, and that the IT system that is available to manage the usage of campus facilities was not designed to keep up with the growth of the institution. A timetabling system can leverage computers and the processing logic of machine learning with a genetic algorithm at its core for the difficult, and currently manually performed, task of fitting courses into the constrained resources of a weekly schedule and a finite number of lecture venues.

Such a system should have a robust, well-tested, enterprise-grade backend and a usable and feature-rich frontend user interface, providing a highly available, modern progressive web and mobile application. The genetic algorithm application must not only be able to create a valid timetable that makes efficient use of university resources, but it must also be able to be configured for the current requirements of courses and professors. Without these features, creating a genetic algorithm to make timetables scheduling remains an academic exercise, and the burden of timetabling will continue to be done as a labour-intensive, manual task.

## 2.3 Current University Timetabling: Case Study

The technologies in current use for the problem domain of creating timetables at a university were studied. The whole-campus facilities department manages which venues are slated for use during each time slot, but this information is gathered from individual disciplines, each of which is responsible for their own timetables.

One academic department was studied for their current timetabling solutions. The discipline administrator has created a spreadsheet-based system to organise the 11 courses in the discipline and their composite modules plus lecturers and place them into time slots and potential venues. The spreadsheet is backed by well-developed scripting and formula-type automation, taking away a lot of the busywork of filling out a schedule. Finally, this automation spreadsheet generates formatted output, suitable for distribution to staff and students.

### 2.3.1 Requirements for a New System

The key deduction from this analysis is that the process requires manual human work is to fit all modules into timeslots and to find venues with appropriate number of seats that the facilities department has made available at those times. Much of the effort year-over-year can be carried over from previous years' timetables, and only new or modified modules need be considered, but there is still a substantial amount of human effort required every year when a new timetable must be produced. The time required for this aspect of the problem domain may be replaced by automation. This requires a genetic algorithm that can modify an existing timetable with minimal turbulence to existing modules, minimising perturbations among staff and students.

An acceptable new solution for timetabling must also be able to export the results in a usable form for distribution. The communication between the facilities department and the disciplines may be augmented by an enterprise timetable management application. Such a system can further enhance the process by allowing communication between disciplines, such as to coordinate scheduling a cross-discipline module. This will require a usable data entry interface in the application, since every single course, module, lecture, and venue must be inputted by their departments, and modified when needed.

## 2.4 Fitness Function for University Timetabling

To define the search specifications for this problem domain, a genetic algorithm must have the domain knowledge of scheduling university timetables distilled into a fitness function.

### 2.4.1 Hard Constraints

The most aspect of the fitness of a candidate timetable solution is any hard constraints, which would prevent a module from being taught if not fulfilled. These include:

1. No two modules are scheduled in the same venue during the same time slot
2. No two modules in the same course are scheduled during the same time slot
3. No lecturer is scheduled to teach two modules during the same hour
4. A module will be taught in the appropriate venue type: lecture hall, computer suite, science laboratory, lecture hall which has a chemistry demonstration desk, etc.
5. A module will be taught in a venue that has adequate seating for the number of students required to be in attendance
6. Other hard constraints, such as accommodations for those with disabilities

Totalling the potential hard constraints that need to be met multiplied by the number of modules that must be scheduled gives a minimum numeric fitness value that a candidate solution must exceed in order to be a valid solution.

### 2.4.2 Soft Constraints

Within the task of making a timetable, there are many parameters that a skilled administrator might consider when scheduling modules. These soft constraints include:

1. Department's preferences to hold lectures in their discipline's building
2. Lecturer's preference for venue amenities, such as desiring a chalkboard versus whiteboard versus transparency projector versus digital projector
3. Distance to travel between venues when a lecturer or student cohort will attend back-to-back modules
4. Time preferences, such as late evenings being less desirable
5. Preference to scheduling the weekly sessions of a module during subsequent hours versus on separated hours versus on different days
6. Ensuring that a venue is the appropriate size for a module, namely that a small module isn't scheduled in an expansive lecture theatre
7. Existing schedule: a module that has previously been scheduled should be left in its time and place if possible
8. Changing another discipline's already-scheduled modules should be less preferential to modifying one's own modules
9. Changing the venue of a module is preferable to rescheduling its time
10. Other soft constraints, which might exist for a given university, discipline, or even individual lecturer, and will be discovered as the completed application is presented to potential end users

As a candidate solution meets more soft constraints, it becomes more like the timetable that an experienced administrator would be able to create manually and is a better solution to be presented to a human operator for approval.

The existence of hard constraints versus soft constraints in this problem domain reveals a definitive stop condition for the algorithm: once a candidate solution emerges in the population which has all hard constraints met and is thus valid, the algorithm could stop. Alternatively, the algorithm might continue for a set number of additional generations, allowing more soft constraints to be met, stopping later if the solution still is valid. This dual conditional even allows soft versus hard constraints to be grouped equivalently, with hard constraints simply being worth a higher fitness value, and a Boolean flag being emitted by the fitness function to indicate that a candidate solution contains a valid schedule.

## 2.5 Validating the Efficacy of the Genetic Algorithm

In order to be confident that the implemented application is outputting correct results, it may be subjected to testing. The first test required is to visually inspect the results: have all modules been place on the timetable in valid venues and are there any overlapping scheduled modules?

Second, an analysis of the algorithm itself may be undertaken. By gathering data for the fitness values of each chromosome in the population, and sampling this data as the generations progress, the population over time can be examined. It should exhibit a general upward trend, as individual chromosomes get better. It should also be characterised by the population becoming homogenous as the above-average building blocks existing in the population tend to be shared between individuals (this is the stagnation mentioned in §1.7.2 above). Finally, there should be random spikes in the fitness of the population as mutations happen—some that bring the population higher and some that cause the fitness in some individuals to be lower. If the population shows this behaviour over time, then the algorithm's success will be validated. The fitness data will be sampled by the genetic algorithm service and logged.

Finally, the efficiency of fine tuning the parameters of the algorithm can be inspected. In Cekała *et al.* [12], parameters of a genetic algorithm were adjusted, and the results were judged by the number of generations the algorithm took to converge on a valid solution. A similar testing methodology will be used: the genetic algorithm service application programming interface (API) allows for optionally passing parameters to it to modify the genetic algorithm, and it outputs the number of generations taken. This can be done repeatedly for an appropriate number of replicates, and the parameters used plus generation run length will be logged in a comma separated format to be analysed. The parameters studied will be mutation rate, crossover rate, elite individuals, and population size.

# 3.0 Technical Implementation

The genetic algorithm was implemented to above requirements as an enterprise web service with a secured API to both run the genetic algorithm and to provide access to timetable data, along with a user interface as a progressive web app.

## 3.1 Programming Language Choices

As explained in §1.4 above, object-oriented techniques—used carefully—offer some advantages when implementing a genetic algorithm. Java is an idea language for running a genetic algorithm under such requirements. It offers structured and formal object-orientation as a core language feature. While it does make compromises in raw execution efficiency to provide platform portability, Java is a low enough level language that it does not have the extra layers of abstraction of many modern, high-level languages.

There are lower-level and faster programming languages available, which would be able to execute a genetic algorithm quicker, but this must be balanced with the language's ability to be integrated with the wider functional requirements of the application, such as delivering a web service as an interface. As a mature language, Java has been optimised over many versions to work as efficiently as possible. Java 8 is a suitable version of choice because it provides many modern features, such as lambdas and streams for compact code syntax.

The frontend web application is written in JavaScript, as is required for a web browser-based service.

## 3.2 The Genetic Algorithm in Java

The genetic algorithm is implemented as a standalone service within the application architecture (Figure 3.1). Through its interface, the service is sent the parameters needed to create a timetable along with preferences to fine tune the algorithm. The execution is controlled by a genetic algorithm Job class, which has a lifecycle of one scheduling request from a user. The Job first creates a Population of

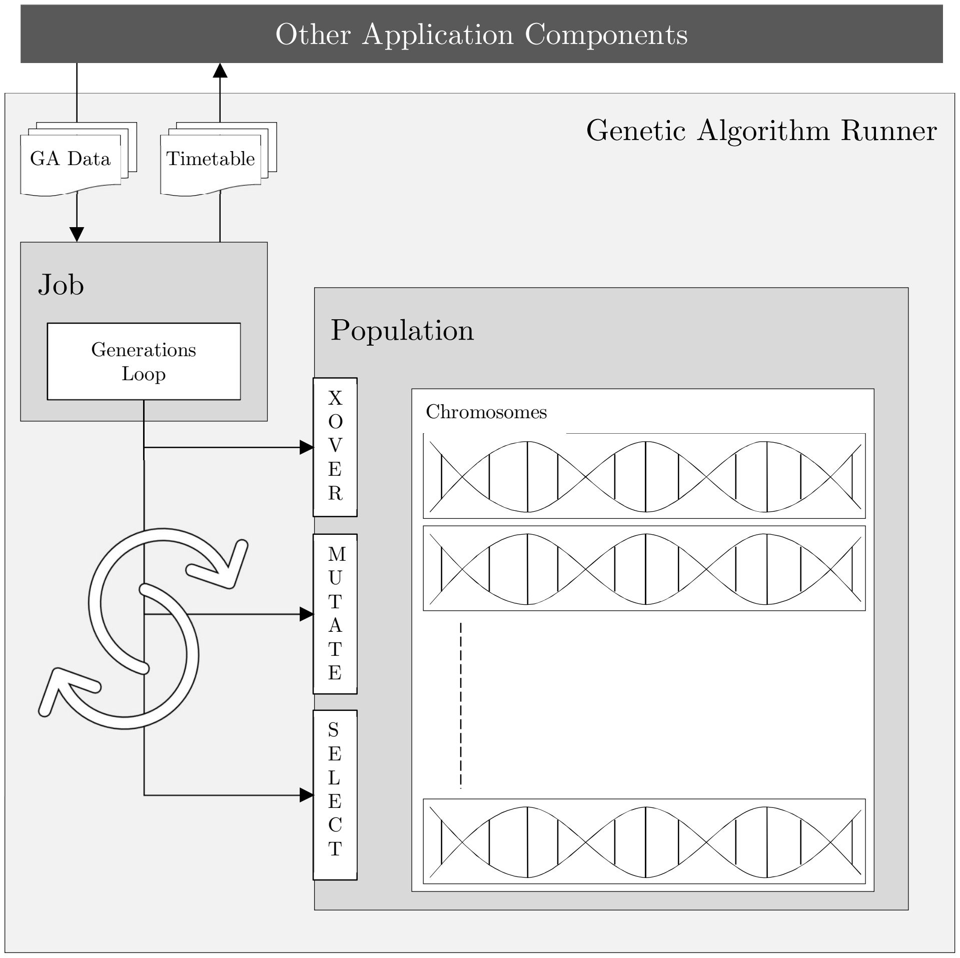


Figure 3.1: The genetic algorithm in Java

individuals, which is responsible for managing all operation on those chromosomes. The Job runs as a loop, one for each generation. It terminates either after a maximum number of generations have transpired or after a candidate solution with no hard conflicts has emerged. In the latter case, it also then runs for some extra generations, proportional to the number already executed, in order to optimise any potential soft constraints. Each loop, the Job signals the Population object to mutate, crossover, and select the next generation of individuals, passing configuration parameters to it. When the algorithm terminates, the Job bundles the resultant schedule and sends it back to the rest of the application to be saved.

The Population object keeps track of one generation at a time of candidate solutions and has responsibility over all individuals within the population. When a mutation or crossover is requested, the Population selects the individuals to be affected—randomly, and according to the algorithm parameters—but the individuals themselves are responsible for performing the operator. Mutation is performed with p=0.05 and crossover is performed with p=0.20 each generation. Newly crossed or mutated offspring are kept in the list of individuals temporarily, but the population size is kept constant when each new generation is selected. When requested to be selected, the Population first selects two elite, top-fitness individuals and then uses roulette wheel selection to randomly select from all individuals ranked by their fitness values. As generations progress, the Job can query the Population if a valid candidate solution has emerged. Finally, when the algorithm terminates, the Population object is asked to pick out the most fit individual in the out of the final population.

A population consists of individual Chromosomes, array-backed objects, each one of which is responsible for possible mutation of itself, possible crossover with another Chromosome passed to it, and calculating its own fitness. Mutation is done by randomly selecting a new timeslot and venue for between one and 20 random component genes, or, if one of those chosen genes already is in a suitable venue, with probability p=0.50 a heuristic mutation is used to only mutate the timeslot. Crossover is done by either binary crossover splitting the two chromosomes at one point, double point crossover splitting the chromosome into three pieces, or circular crossover where the gene is split into four pieces. In order to give an equal chance of any potential above-average building blocks of unknown sizes being passed between individuals, one of these three methods performed, chosen with equal probability each generation.

A Chromosome's final responsibility is to calculate its own fitness, which it does so when it is created or whenever any of its genes have change from a genetic operator. Despite technically being new objects, individuals selected to be transferred to the next generation are created through a customised cloning function that avoids having to recalculate fitness. Through this, the cached fitness values can be propagated far, and much extra calculation is avoided. The fitness function itself must examine each component gene for its own fitness, such as if a module is slotted into a venue that can seat all its students, and it must also compare each of its genes against all others, such as to find if two modules are scheduled in the same time and place. This is an $O(n^2)$ operation, where n is the number of modules being scheduled across all courses. While the fitness function itself is not threaded, functions that create or modify many Chromosomes are parallelised, and a thread pool is utilised to spawn child threads.

The final data structure used to run the genetic algorithm is a Gene, which is each module scheduled into a timeslot and venue. Genes only have responsibility to report on their own fitness when requested, such as if their module is in an appropriate venue, or to generate themselves with random timeslots and venues in a factory-like pattern. The data within a Gene is made of Module, Timeslot, and Venue objects. Each contain all information needed for fitness calculation, which is gathered up during algorithm initialisation from databased joins on relational tables, such as module sizes or timeslot hours. As they are heavyweight objects, they are never cloned generation-over-generation, and instead are used by reference between all the Genes and Chromosomes in the Job.

## 3.3 Architecture

The backend of the project is designed in the Service-Oriented Architecture style to support separating the features of the app into distinct services [18]. Each service has well-defined, self-contained concerns to meet the separate real-world requirements to run the project: The Data Service, the Genetic Algorithm Service, and the Web Service (Figure 3.2). These services will be described in this section.

### 3.3.1 Framework

The backend of the project is a web service running on the Tomcat application server backed by the Spring Boot framework. This framework brings in many enterprise concepts, such beans, data contexts, and managed component lifecycles. These features will be used to provide robustness to the application, as well as abstracting many implementation details, such as making a database connection pool, server sockets, or login security. Spring's design paradigm is to be preconfigured with sensible defaults for a web service and allow further customisation through annotations in the source code instead of with verbose XML. Because it can help create a project with enterprise-grade features with less developer effort, Spring is a good choice for a student project.

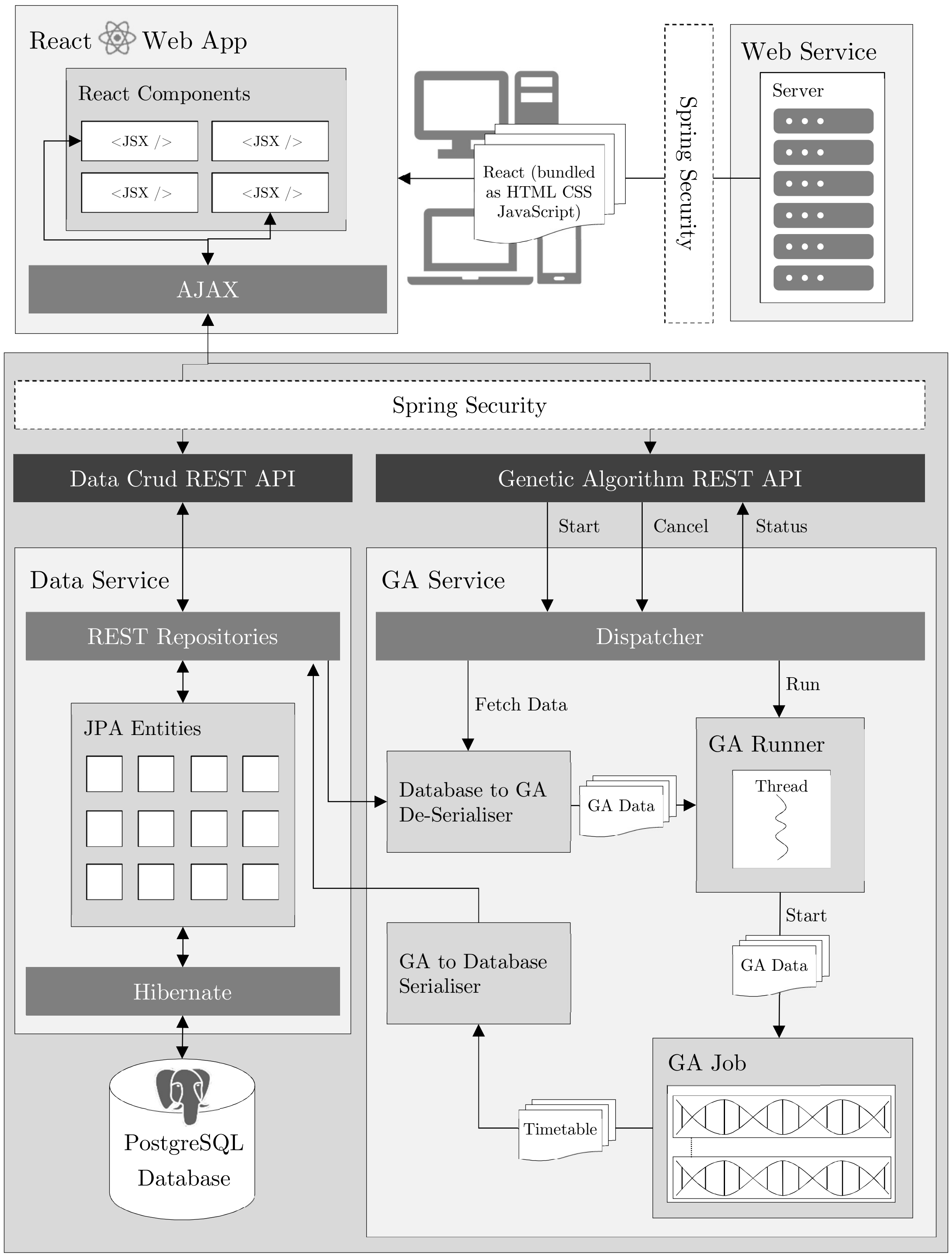


Figure 3.2: Architecture of the genetic algorithm application, showing the three separated services

## 3.4 The Data Service

The Data Service manages data access, data transport, and interface to the database. The Entity framework is used to associate Java objects with database tables, and the Java Persistence API implemented by Hibernate manages data reading, writing, and searching. Database connection details are stored in configuration files that are not checked into source control. They are read out of this file into environment variables to be accessed by JPA either by a Gradle script for local development or sent to the cloud server via an config file for production. The database runs on the Amazon Relational Database Service, but for development a local database server is run. Switching connection is done by swapping the active configuration file.

### 3.4.1 The Database

The database is implemented in PostgreSQL, chosen for being an open source solution that is widely used and supported. The database can be separated into three types of data tables (Figure 3.3). First, there are tables that hold all the pieces of a university's timetable, such as venues, buildings, modules, lecturers, and timeslots. Each item has the entity's data, such as course name or venue seating capacity, and is linked through others through relations of appropriate arity. Secondly, there are tables to support the operation of the web application. These are the Users table with usernames and hashed, salted passwords plus the Job table to keep track of jobs currently dispatched to the genetic algorithm, and they have relations to other data, such as the timetables a user manages.

Finally, the Schedules and Scheduled Modules tables represent timetables that have been created by the genetic algorithm. Once the algorithm completes, a Schedule will have a one-to-many link to a series of Scheduled Modules, one for each Module in the database. Each of these will link to a single record in the module, timeslot, and venue tables. Other details can be reached through database joins, such as which building a venue is located in. Stored this way, the core concern of the entire application can be stored, accessed and modified later, and shared with other users.

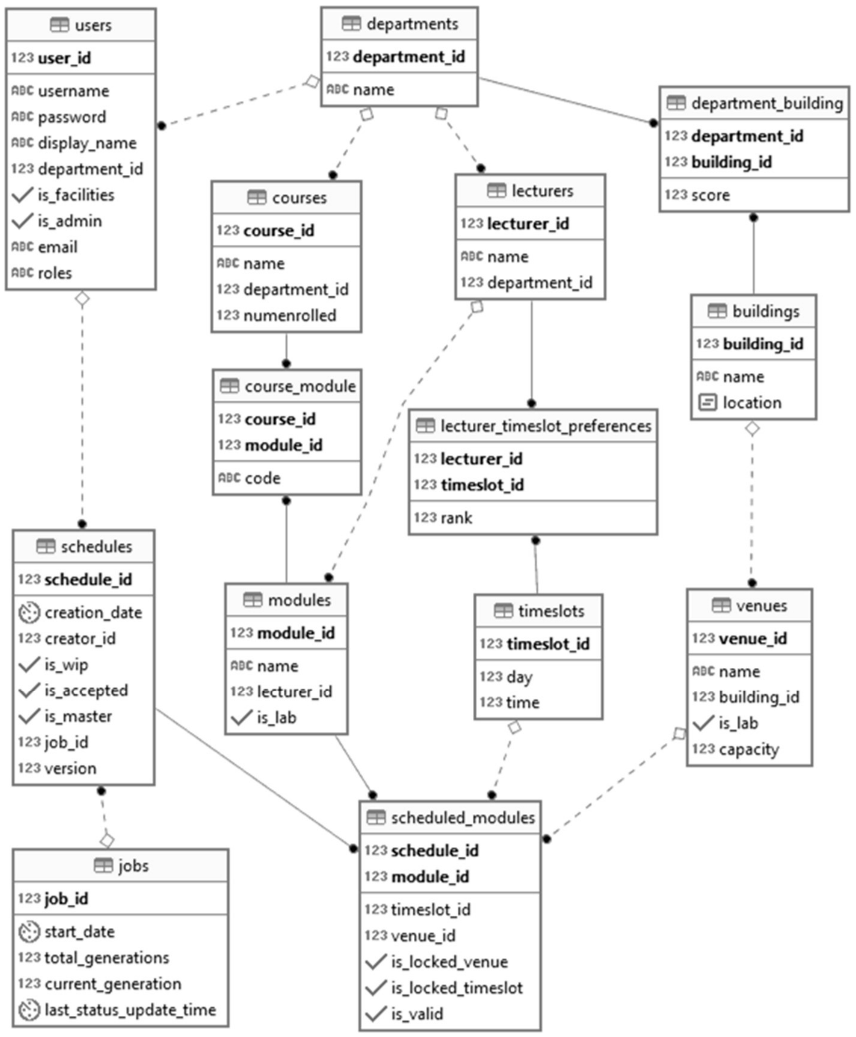


Figure 3.3: Entity-Relationship diagram of the PostgreSQL database

### 3.4.2 The Data Service's API

The Data Service exposes an API to the web app front end. The API is driven by the Spring Data REST framework, which provides annotation-based configuration of a RESTful-type service with sensible defaults. It automatically transforms database entities into API endpoints and can exclude data that is private or irrelevant to outside consumption, such as passwords. It uses a hypermedia-driven paradigm for REST services [19], allowing discoverability of all services from the base URL of the server. This paradigm is useful to expose the relations of a relational database.

## 3.5 The Genetic Algorithm Service

Separated from database concerns is the second architectural component of the backend, the Genetic Algorithm Service. This service fulfils the main functional requirement of this project. When a new job is requested by the frontend web app, the service first must get the details of the job from the Data Service. It does this through a Deserialiser class, which contains the business logic that understands how to translate between database entities and in-memory objects suitable to being used by the genetic algorithm. This component makes a bundle of genetic algorithm objects.

It can then fork an instance of the genetic algorithm as a computational thread, saving the job details to the database. The data objects are passed to the newly forked worker thread. Details of the algorithm's operations have been described in §3.2. The genetic algorithm then runs until a solution is found, a maximum number of generations are elapsed, or the job is cancelled by an outside API call. Afterwards, the Serialiser class will write the generated solution back to the database and push a message to the frontend web app over a Web Socket.

The Genetic Algorithm Service is completely independent to the Data Service and Frontend Web Service. It has its own API to start a job, stop a job, and query job status. The two backend services only interact at well-defined interface points to exchange data.

## 3.6 The Frontend Web Service

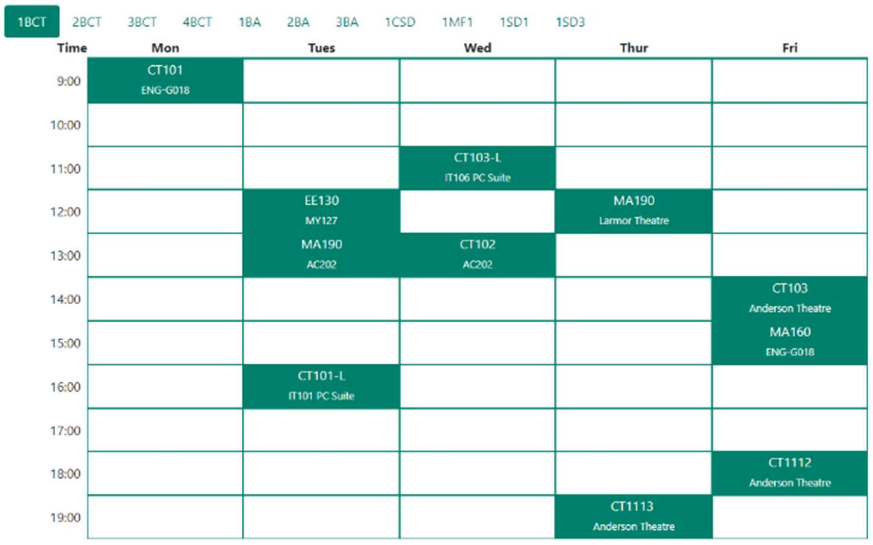
The third service is the web server. The frontend of the application is served from the Tomcat application server, which provides authentication services through Spring Security to the database Users table to authorise user before it serves any content or allows API queries. The Web Service's main purpose is to serve the bundle of HTML, CSS, and JavaScript that makes up a progressive web app. In server terms, this means the entire frontend is served as static content, and only becomes dynamic when it runs within the client's web browser. The real frontend web application is thus completely contained in an independent React JavaScript application.

### 3.6.1 React

React is a declarative, component-based JavaScript framework to make interactive web applications. It allows writing self-contained components to encapsulate each quantised bit of functionality of GUI. These are organised in a hierarchical manner. Each component has a state, which can be updated by its parent, and children can pass messages back upward to their direct parents, such as when a button is pressed to start a genetic algorithm job.

An example of the state hierarchy is the Timetable component of the GUI: A Timetable is instantiated with the complete dataset of the results of a genetic algorithm scheduling job, which it sorts into timetables for each university course. For each course, it makes a week calendar. It then creates a Row component, to split up the week into hours, and gives each Row only the modules which have been scheduled for that hour. The Row then breaks itself down into columns, making a Cell component for each. A Cell's state is whether a module has been scheduled for that day/hour. If so, its data is displayed, else the spot on the schedule is left blank. No logic is needed at the top level to explicitly decide which cells must be filled in, since responsibility is left to the lowest objects in the component hierarchy. When a scheduled timetable is updated after a new run of the genetic algorithm, the updated data is passed to the Timetable by updating its state object. In turn, each child Row and Cell is updated. A dynamic user interface is thus created.

Running on an end user's computer, the front end communicates with the application services over the REST APIs. It uses asynchronous HTTP requests (AJAX) to get user data, request job runs, or update timetables in the GUI.



| Time | Mon | Tues | Wed | Thur | Fri |
|---|---|---|---|---|---|
| 9:00 | CT101 ENG-G018 | | | | |
| 10:00 | | | | | |
| 11:00 | | | CT103-L IT106 PC Suite | | |
| 12:00 | | EE130 MY127 | | MA190 Larmor Theatre | |
| 13:00 | | MA190 AC202 | CT102 AC202 | | |
| 14:00 | | | | | CT103 Anderson Theatre |
| 15:00 | | | | | MA160 ENG-G018 |
| 16:00 | | CT101-L IT101 PC Suite | | | |
| 17:00 | | | | | |
| 18:00 | | | | | CT1112 Anderson Theatre |
| 19:00 | | | | CT1113 Anderson Theatre | |

Figure 3.4: The Timetable component in the React GUI

### 3.6.2 Functionality of the Web App

In the frontend web app, a user may:

- Log in
- View a list of schedules they have created
- Explore an existing schedule, looking at details of all schedule modules for each course separately
- Start a genetic algorithm job to create a new schedule or modify an existing
- View the updated results of a job
- Explore the available REST API endpoints through an interactive interface

## 3.7 Cloud Application Deployment

The application is run with the platform-agnostic Spring framework, and the platform of choice to deploy onto is the Amazon Web Services (AWS) cloud servers. A virtual machine was provisioned with Linux and Java preconfigured. A Tomcat application server was initialised, and the packaged application was pushed directly to the cloud virtual machine via AWS's command line tools. A PostgreSQL database was provisioned on Amazon's RDS service.

Security of the AWS cluster was managed according to Amazon best practices. The cloud servers were grouped under a Virtual Private Cloud (VPC), creating a single security entity, and the VPC edge firewall was configured to only allow HTTP traffic to the web server and API endpoints. Communication between the web server and database server was only allowed for IP addresses within the VPC, protecting these uncommon ports. Rather than each service with full root privileges, multiple users were generated: creation and deletion of resources was managed by a less privileged Administrator user, and command line deployment was authenticated by separate credentials from in-application database connection strings, each one assigned least privileges. The cloud server's details were used to configure DNS on a web domain, giving the frontend a user-friendly URL.

## 3.8 Build Toolchain

To manage the build of the front end, backend, and genetic algorithm components of the application, the Gradle build tool is used to automate and manage the development toolchain. For an enterprise project, a build manager is essential to create stable, consistent compiled versions of the application. An application of this size has inputs of not only the source code, but also any dependencies, such as frameworks' code or configuration files. A build tool can fetch these dependencies, use them to successfully compile the source code, process configuration files, and put all this together in a specified way into a useful output.

The Gradle build tool provides a modern methodology to do this. It is based upon the convention-over-configuration paradigm of the Maven build tool, but it uses a procedural build script more akin to the Ant build tool, a style that fits programmer procedure-oriented mindset better than the verbosity of a machine-readable XML configuration file. A Gradle build script contains declarative syntax for identifying project dependencies and basic properties, but it also allows for arbitrary script execution within the configuration itself for any needed task automation. For example, before the application server is started, a short script parses database details from a configuration file and made available to the JVM's environment.

The backend and genetic algorithm components are packaged with Gradle into a Java archive (JAR). The front-end components will be packaged and tested with the Webpack JavaScript build tool, managed as Gradle tasks. Through another Gradle task, a local Tomcat application server can be launched for testing the entire application. For production deployment, the application's JAR is packaged with all libraries and dependencies using Gradle, which can then be deployed on an application server running on cloud server platforms.

# 4.0 Genetic Algorithm Results

## 4.1 Implemented Algorithm Goals

The genetic algorithm runs successfully within the application architecture, taking input from the database, running the algorithm, and writing a table of scheduled modules back to the database. All of the hard constraints detailed in §2.4 above were able to be implemented. Any date and time conflicts wherein a venue, lecturer, or student cohort is double-booked created lower fitness of a candidate solution. The fitness function also detected when a module was scheduled for in a venue that was not large enough for the number of students or not the correct venue type, lecture hall versus computer lab.

The soft constraints of attempting to put a department's modules into a building they prefer was implemented, along with trying to schedule a lecturer's modules during their preferred time slots. Soft constraints on distance to travel between classes, or any other constraints that depended on two different modules at the same time, were not implemented or made part of the database's design. Any of the soft constraints that depended on comparisons to a previous version of an existing schedule were not modelled.

For soft constraints, the numeric fitness worth of each one was dynamically calculated to restrict the total possible sum of fitness values for all fulfilled soft constraints to be equal to the value of just *one* hard constraint (minus 1). In other words, if every soft constraint for a candidate solution is fulfilled, but another candidate solution, which happens to have very poor soft constraints, has even one fulfilled hard constraint, then the second chromosome will be more likely to be selected for. The result of this is that soft constraints can be improved as the algorithm proceeds, but it will never be done at the expense of better hard constraint fulfilment. This is must be done dynamically since the number of objects that need to be compared comes from the database and can change when university parameters are modified.

Being able to fine-time the parameters of the genetic algorithm was implemented and can be modified whenever a new genetic algorithm is started via arguments to the launch-job API endpoint. Rates of mutation and crossover, population size, number of elite chromosomes to select each generation, and the maximum generation limit may be changed. Statistics about the job are outputted to log suitable for analysis, if the application is run in verbose debugging mode.

The feature of modifying an existing schedule was implemented. The job runner detects from the database that this Schedule record has been processed before and reads the pre-existing data into the genetic algorithm rather than starting from scratch. However, this is of limited usefulness in the current version of the project, since the GUI requires a more in-dept fitness function and more features, such as manually moving or locking in modules to timeslots/venues or being able to change the parameters stored in the database via the interface.

The stop condition of a genetic algorithm generation-running loop is determined dynamically. First, it is never allowed to go above the maximum generation parameter. If a valid candidate solution, which has no hard constraint violations, emerges from the population, then the loop will be run for additional generations equal to 20% of the current generation count. This gives soft constraints a chance to be improved when there is less selection pressure against them. After the additional generations, the random nature of the genetic algorithm caused the valid solution to be lost, then it continues as if the valid solution had never existed. However, if there is still a good solution, the algorithm will terminate early.

## 4.2 Correctness of Solutions

The previous section details the which of the functional requirements of the algorithm were programmed. To validate this, the algorithm's results were inspected. The genetic algorithm, given inputs of the courses, modules, and lecturers of the discipline of information technology, along with available venues in university buildings, does successfully slot all lectures and labs into appropriate venues in a weekly timetable with no time conflicts for a lecturer, a course cohort of students, or a double-booked venue.

Soft constraints were also examined for correctness. Given valid solutions, timetables were observed to tend to choose venues in buildings that the department has been set up to have preferred. Further, some lecturers were set to prefer morning classes, some were set to prefer a day of the week with no classes, and others had no preferences. Lecturers did not tend to have their modules scheduled into undesirable times slots.

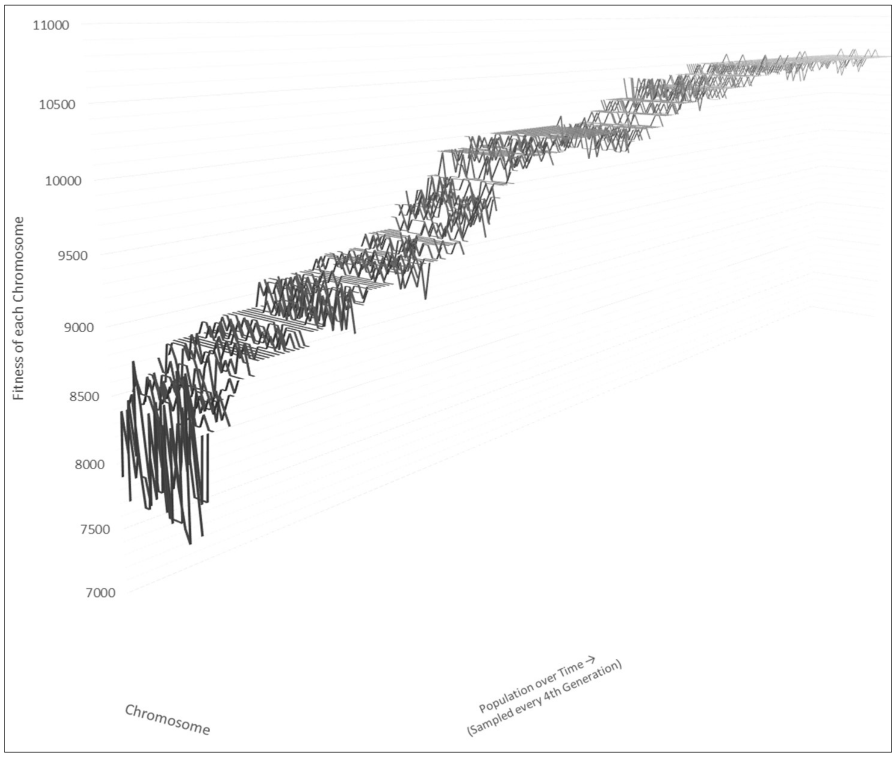
## 4.3 Observation of the Genetic Algorithm in Operation

In order to demonstrate that this implementation of a genetic algorithm search exhibits a proper fitness landscape, as detailed above in §1.9.1, the fitness value of each gene within each chromosome at the beginning of every generation was outputted to a CSV file by the application. This was then charted to show the progression of the population over time.

The numerical fitness values in this chart are not important, but rather the shape of it as the generations progressed. A few characteristic behaviours can be identified. The population is initialised randomly, so each chromosome is very different from each other in the first few generations, creating a chaotic zig-zag shape. Mutation will tend to make individuals of very different fitness, which are the spikes above (and below) other individuals in generations. Selection tends to normalise exceptional individuals over time, either by eliminating poor specimens or by bringing up others to meet positively-mutated new individuals.

This also demonstrates the building block hypothesis in action, as good mutations in one gene ends up being shared as blocks to other genes, which causes fitness to rise and then plateau as the building blocks are shared widely and the population becomes homogenous. These phases of population stability indicate the population is grouped around one of the local maxima in the search space but has not yet found a valid solution. Such plateaus become very long towards the end of the run, as any further improvements to the candidate solutions are less likely and take more generations to happen upon. Overall fitness of the population increases as time goes on, although not at an even rate. Just like in biological evolution, the genetic algorithm is characterised by long periods of little change and then brief periods of extreme disruption.

Because these disruptions tend towards increasing overall fitness, it can be confidently stated that the timetabling problem as a genetic algorithm has been implemented correctly and that it exhibits a proper fitness landscape: individuals climb the metaphorical hills towards valid solutions (maxima) within the search space. Candidate solutions which are given above-average building blocks, either through mutation or crossover, tend to be selected. These better solutions are closer to a maximum, or peak of a hill in the fitness landscape. The process continues cyclically, until the global maximum is found.



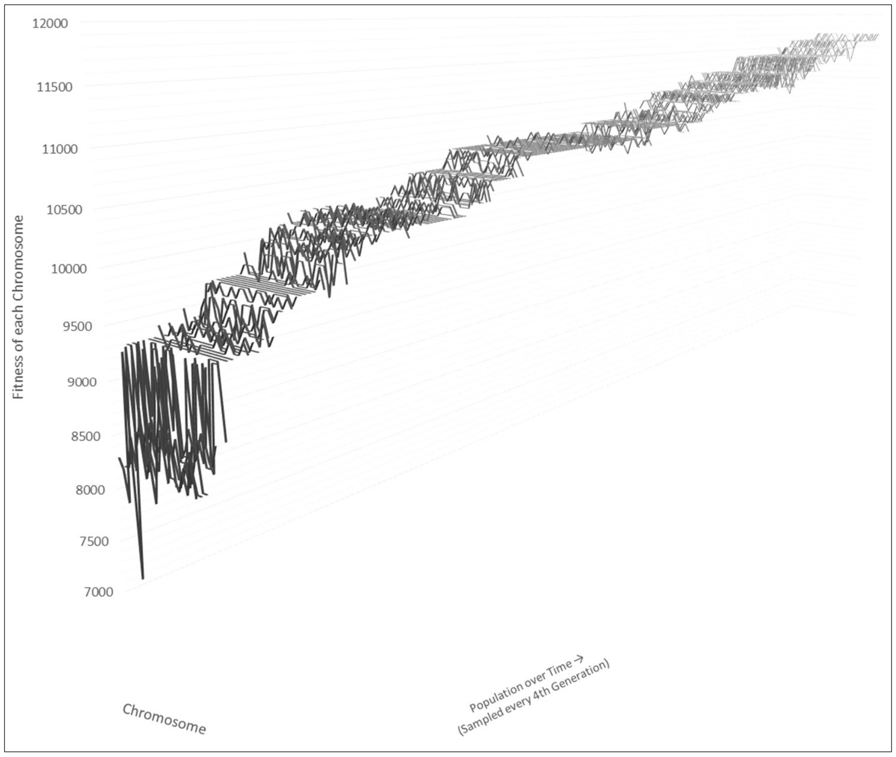


Figure 4.1 (a) and (b): Fitness of chromosomes over generations of two executions of the genetic algorithm

## 4.4 Genetic Algorithm Parameters

To provide evidence for the parameters chosen, the genetic algorithm was run repeatedly while input parameters were varied, and data was gathered on how many generations the algorithm took to converge, as discussed in §2.5. For each set of inputs, expressed as rows in Figure 4.2, five replicates were run, and their mean and standard deviation taken. Also reported is how many, if any, of the five executions failed to converge on a valid solution within a generation limit of 10,000. The baseline configuration was chosen based on suggestions found in the literature on genetic algorithms. Each parameter was then varied up and down from the baseline, and results reported.

A casual analysis of this data reveals a positive impact from increasing the population size and from including more top-fitness elite individuals during each selection. There is a negative impact from doing the opposite of those two. This leads to the conclusion that a larger population size and more elite survivors is significant. The genetic operator probability parameters for crossover and mutation do not appear to have significant impact.

Deeper analysis may give more confidence in these conclusions. The standard deviations from these results are very large, which is not surprising given the random nature of genetic algorithms but might be lessened with more than five replicates. Further, these results could benefit from repeating this procedure with more replicates, more variation in the independent variables, cross-analysis of independent variables, and deeper statistical analysis performed to determine if there is a significant difference in these parameters and their interactions.

| Variant | Population Size | Crossover Probability | Mutate Probability | No. Elite Survivors | Failed | Mean generations to converge (with std. dev.) |
|---|---|---|---|---|---|---|
| Baseline | 60 | 0.1 | 0.05 | 2 | - | $3083 \pm 1901$ |
| Crossover -0.05 | 60 | 0.05 | 0.05 | 2 | - | $6338 \pm 2618$ |
| Crossover +0.05 | 60 | 0.15 | 0.05 | 2 | - | $4160 \pm 1535$ |
| Mutation -0.02 | 60 | 0.1 | 0.03 | 2 | - | $3269 \pm 1484$ |
| Mutation +0.05 | 60 | 0.1 | 0.1 | 2 | - | $4067 \pm 1807$ |
| Elite -1 | 60 | 0.1 | 0.05 | 1 | 3 | $8387 \pm 2323$ |
| Elite +2 | 60 | 0.1 | 0.05 | 4 | - | $2149 \pm 1043$ |
| Population -40 | 20 | 0.1 | 0.05 | 2 | 4 | $9322 \pm 1516$ |
| Population +40 | 100 | 0.1 | 0.05 | 2 | - | $2354 \pm 909$ |

Figure 4.2: Summary table indicating the suitability of values of parameters to the genetic algorithm

# 5.0 Final Conclusions

Overall, the implementation of this project was a success. A genetic algorithm that converges upon the desired solution was implemented, and the service was run within a web service as an enterprise-grade software architecture. The results of a genetic algorithm scheduling job are presented in a pleasant, human-readable fashion in a frontend web application, a GUI exists to control launching such jobs.

## 5.1 Implementation Improvements & Future Work

### 5.1.1 Improving the Application & Architecture

Further improvements could include more robustness in the software, such as automated unit tests and end-to-end tests using a mock server and API. An experience Java engineer would utilise the Java profiler tool to discover if more memory or CPU resources needed to be assigned to the JVM for more performant genetic algorithm executions. The profiler could also identify bottlenecks in the algorithm implementation, which could be optimised in Java, or translated to a compiled language for speed.

Because of their independence within the architecture, each service could be separated into their own JVM, or even run on their own server or potentially a cluster for parallelisation. The Genetic Algorithm Service was designed to be run externally and is meant to communicate with the database via the Data Service's API. A future goal would be to make the genetic algorithm into a microservice. Further, the stateless nature of the genetic algorithm means it could potentially be run on a so-called serverless compute platform like AWS Lambda or Open Lambda.

### 5.1.2 Improving the Genetic Algorithm

Beyond the fitness function hard and soft constraints that were not able to be implemented, covered in §4.1, there are several more real-world aspects of university scheduling that could be added to the genetic algorithm to improve the quality of generated timetables, but were not included mainly due to the complexity that they would add to that data model.

It is assumed that each student in a course is enrolled in the same modules, giving a simple total used for the capacity of classrooms that are needed for an entire course. This does not allow for students to choose various modules within their course but implementing it would require storing data on every student in this database (adding a data privacy issue on top of the complexity). This would also allow for the algorithm to take in preferences from students, such as timeslots they prefer, through a mobile app interface.

A few other real-world intricacies include that real module might be taught by more than one lecturer, either simultaneously or in sequence with half a semester each. Each module is only modelled with time slot; many university modules have multiple hours of lectures every week. Adding this complexity to the database would additionally allow for soft constraints on those modules, such as lecturers who prefer to teach back to back vs on different days. Finally, labs associated with a module are simply unconnected modules, with no interaction that could be modelled, and only one type of lab venue is included.

### 5.1.3 Front End

The user interface is progressive web app that allows for starting scheduler jobs, viewing a resultant timetable and its details. This core functionality allows interaction with genetic algorithm, but more GUI features could be implemented.

In its current state, the web app is designed to operate on a desktop or laptop PC, but also has enough responsiveness built-in that it scales well visually to work mobile web browsers, allowing for viewing timetables on the go. This could be extended to port the web app to a native Android or IOS app for further usability.

### 5.1.4 Improving the User Experience

The genetic algorithm application is intended to be presented via a user interface designed to distil the human knowledge required for a good schedule, and to fulfil this goal the interface should have high awareness of user experience (UX) needs for its use by college discipline administrators. A future step in this process is to undertake a user study with the cooperation of potential end users to analyse how application will meet the needs of such participants in the university timetable creation process. Rather than making assumptions of the best interface layouts and usage patterns, techniques such as user polls and A-B studies can be used to guide improvements to the final design. A final product will not only be able to fulfil the functional requires of designing a timetable but also will have high enough usability that the target users would *choose* to use it along with their existing timetable construction process.

## 5.2 Context of the Project

Designed with input from university administrators to help understand the needs being unfulfilled with current timetabling solutions, this completed project would be able to be used by such professionals, on this university campus and beyond. Because of the specialised nature of its design, from the database on up, the application itself would not be able to be transferred easily to other types of scheduling, even something as closely related as employee shift scheduling, bus stop timetables, or the other challenging domains in the literature (§2.1). However, the experience gained in implementing this project could be applied to using genetic algorithms in scheduling in these domains and beyond.

Since it is designed to automate a task, this application takes away some of the responsibilities of human workers. However, the knowledge of an experienced department administrator is key to making well-suited course timetables. A machine learning algorithm like this can find a solution to the difficult puzzle of scheduling, but only a human can validate that is a quality schedule. Further, without the organisational skills of a university administrator, the preferences and properties of university entities, such as lecturers, cannot be distilled from deep domain knowledge into machine-usable parameters. This genetic algorithm is a tool for the task of creating a timetable, not a replacement for the entire process.

Further, as an enterprise web application, this project could help solve problems in this domain that require cross-department, campus wide coordination between all users logging in. Disciplines could share or make use of spaces and resources that they previously did not know about. It would allow the facilities department to manage allocation of classrooms between all disciplines in a seamless manner. The application could also allow for collecting statistics on campus facility utilisation, which could lead to better funding and justification for better campus programmes. The scheduling application could play a part in assisting persons with disabilities by automatically allocating venues that facilitate needs.

## 5.3 Closing

This project could be improved to be a more complete piece of software, and there are valuable additional features to be added. However, the goals of implementing a functioning genetic algorithm to create university timetables, and having it execute in a robust, service-based architecture, were fulfilled.

# References

[1] A. M. Turing, “Computing machinery and intelligence,” *Mind,* vol. LIX, no. 236, p. 433–460, 1 October 1950.

[2] J. H. Holland, Adaptation in Natural and Artificial Systems, Ann Arbor: University of Michigan Press, 1975.

[3] K. De Jong, “Learning with genetic algorithms: an overview,” *Machine Learning,* vol. 3, no. 2-3, p. 121–138, October 1988.

[4] N. P. Padhy, “Genetic algorithms and evolutionary programming,” in *Artificial Intelligence and Intelligent Systems*, Oxford, Oxford University Press, 2005, pp. 459-527.

[5] D. Whitley, “A genetic algorithm tutorial,” *Statistics and Computing,* vol. 4, no. 2, pp. 65-85, June 1994.

[6] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, Boston: Addison-Wesley Longman Publishing Co., Inc., 1989.

[7] C. Darwin and A. Wallace, “Three papers on the tendency of species to form varieties; and on the perpetuation of varieties and species by natural means of selection,” *Journal of the Proceedings of the Linnean Society of London,* vol. 16, pp. 6293-6308, 1858.

[8] F. M. Stefanini and A. Camussi, “Aplogen: an object-oriented genetic algorithm performing monte carlo optimization,” *Bioinformatics,* vol. 9, no. 6, pp. 695-700, 1 December 1993.

[9] L. Jacobson and B. Kanber, Genetic Algorithms in Java Basics: Solve Classical Problems like The Travelling Salesman with GA, New York: Apress, 2015.

[10] S. Z. Nie, “A Java EE platform test system based on improved genetic algorithm,” *Applied Mechanics and Materials,* Vols. 556-562, pp. 2581-2585, 2014.

[11] B. Kazimipour, X. Li and Q. A. K., “Initialization methods for large scale global optimization,” in *IEEE Congress on Evolutionary Computation*, Cancun, Mexico, 2013.

[12] T. Cekała, Z. Telec and B. Trawiński, “Truck loading schedule optimization using genetic algorithm for yard management,” in *ACIIDS 2015: Intelligent Information and Database Systems*, Bali, Indonesia, 2015.

[13] E. Burke, D. Elliman and R. Weare, “A genetic algorithm for university timetabling,” in *AISB Workshop on Evolutionary Computing*, Leeds, 1994.

[14] V. Podgorelec and P. Kokol, “Genetic algorithm based system for patient scheduling in highly constrained situations,” *Journal of Medical Systems,* vol. 21, no. 6, p. 417–427, December 1997.

[15] A. J. Cole, “The preparation of examination time-tables using a small-store computer,” *The Computer Journal,* vol. 7, no. 2, p. 117–121, 1 January 1964.

[16] S. Ahmed, “Applications of graph coloring in modern computer science,” *International Journal of Computer and Information Technology,* vol. 3, no. 2, January 2013.

[17] R. M. Karp, “Reducibility among combinatorial problems,” in *Complexity of Computer Computations*, Boston, MA: Springer, 1972, pp. 85-103.

[18] Open Group SOA Workgroup, “Service-oriented architecture,” in *SOA Source Book*, Van Haren, 2009.

[19] M. Fowler, “Richardson Maturity Model,” 18 March 2010. [Online]. Available: https://martinfowler.com/articles/richardsonMaturityModel.html. [Accessed 17 March 2019].

[20] J. Grefenstette, R. Gopal, B. J. Rosmaita and D. Van Gucht, “Genetic algorithms for the traveling salesman problem,” in *Proceedings of the 1st International Conference on Genetic Algorithms*, Pittsburgh, 1985.