Scalability of a Genetic Algorithm that solves a University Course Scheduling Problem Inspired by KTH

HIROYUKI VINCENT YAMAZAKI
JOHN PERTOFT

Degree Project in Computer Science, DD143X
Supervisor: Pawel Herman
Examiner: Örjan Ekeberg

CSC, KTH 2014-04-13
Abstract

Scheduling university courses is a combinatorial optimization problem where a set of events such as lectures, seminars and labs have to be scheduled into certain timeslots while taking various constraints into account. The problem is considered difficult by conventional methods and the amount of required computations usually increases exponentially with the size of the problem. However, for universities such as KTH, Royal Institute of Technology in Stockholm, this is a task that has to be dealt with regardless of its complexity. In this report, Genetic Algorithms (GAs) were applied to solve the course scheduling problem for multiple data samples inspired by KTH. The sizes of the input data were carefully designed to study the scalability of the GA. The results indicate an exponential growth of the running time measured not in actual time but number of iterations. The scalability is highly dependent on the definition of a high quality solution as well as algorithm parameters but also different methods being used in the GA.

Keywords - Scheduling, Timetabling, KTH, Royal Institute of Technology, Scalability, Genetic Algorithm, GA, Constraints, Fitness, Selection, Crossover, Mutation, Repair, Extinction, Generation
1 Introduction

Creating a course schedule for a university is a problem called timetabling which is one of the most common scheduling problems. Timetabling can be described as allocating a set of events and resources to different timeslots while taking various constraints into account. The goal is then to minimize the number of violations of constraints while using as few resources as possible. The timetabling problem has no known polynomial time algorithm for solving optimally and is for that reason seen as an NP-hard combinatorial optimization problem. However, the real difficulty lies in actually achieving a complete set of constraints and to define criteria for evaluating the quality of the solution. This makes each timetabling problems unique, depending on the constraints that are considered.

Previous research in this field has led to a wide array of different approaches. Most of these can be described as metaheuristic algorithms which are algorithms that are commonly applied to optimization problems. The basic idea of those algorithms are to work as a guiding strategy for designing the underlying heuristics in order to avoid getting stuck at local optima [1]. Many metaheuristic algorithms are inspired by nature and some examples of such algorithms are evolutionary algorithms, ant colony optimization and simulated annealing [2]. Among evolutionary algorithms we also find the Genetic Algorithms (GA).

GA became a well known methodology the 1970s with the release of the book Adaption in Natural and Artificial Systems, written by John Henry Holland and his student at the University of Michigan [3]. At that time, GA remained mostly as a theoretical approach due to its heavy computational cost and remained so until the desktop computer experienced its drastic improvement in the 1980s. Since then, GAs have been used to solve combinatorial optimization problems with many constraints and variables where no analytical solutions could be applied. One such problem was the university course scheduling problem which this report focuses on.

This paper is organized as follows. Section 2 states the problem and explains the scope of the report. Section 3 describes the general procedure of GA and the background of the university course scheduling problem. In section 4, the methodology for how the experiment was carried out is explained; input data, data structures, parameters and configurations for the GA as well as specific implementation details. Section 5 presents the results, the required computations for each input file and the scalability of the GA. Section 6 analyzes the results and discusses the scalability. Section 7 provides the conclusions to this report. The Appendices lists the input data files followed by the GA source code written in the programming language Java.

2 Problem Statement

In this paper, a GA is implemented and benchmarked to solve the course scheduling problem for various data samples inspired by KTH, Royal Institute of Technology in Stockholm. The goal is to schedule the classes into available timeslots using as few resources as possible without violating the constraints,
while studying the scalability of the algorithm on the various data. The problem domain is a scaled down and only certain constraints and properties from KTH will be taken into account. Hence, the result will not be directly applicable to the actual problem that KTH is facing but could yield guidance in further research.

3 Background

3.1 The University Course Scheduling

The university course scheduling is a combinatorial optimization problem. Briefly explained, it can be described as allocating a set of events such as lectures and labs to different timeslots while taking various constraints of the university into account. The goal is then to minimize the number of violations of constraints while utilizing facilities effectively and efficiently. This problem is NP-hard which means that the amount of required computations to find the optimal solution increases exponentially with the size of the problem [4]. Previous research in this field has led to different approaches to the problem. Solving it using different types of local search algorithms such as Tabu search or simulated annealing has been commonly used methods [4]. Today, there are software available that handles the course scheduling such as the Mimosa Scheduling Software from Mimosa Software Ltd. and Comprehensive University Timetabling System from UniTime LLC which utilizes local search [5, 6].

3.2 The Genetic Algorithm

The GAs are search heuristics that take inspiration from nature’s evolution and is suited for solving multidimensional combinatorial problems.

The algorithm begins with an initial set of random solutions to the problem. But because of the crudeness of these initial solutions, the quality of them are not guaranteed at this stage [7]. In the context of GA, each solution is called a chromosome and a set of chromosomes form a population. Each chromosome has a number of genes which can be of different values and correspond to certain properties in the solution. Using these genes, the quality of the chromosome is evaluated using a fitness function. In order to generate a qualitative solution to the problem, parent chromosomes are selected from the population based on their fitnesses and crossed to create new offspring chromosomes. These offspring are then randomly mutated to expand the search space. This procedure is repeated as long as the population gradually improves. The goal is to eventually generate an individual that meets a fitness condition.

The pseudocode in Algorithm 1 below gives a basic outline of the GA. The following subsections describes each of the GA actions.
ALGORITHM 1: Genetic Algorithm

create random population and evaluate fitness of its chromosomes

while most fit individual is not fit enough do
  while offspring population is not full do
    select two parent chromosomes
    crossover parent chromosomes to create offspring chromosomes
    mutate offspring chromosomes
    evaluate fitness of offspring chromosomes
    add offspring chromosomes to offspring population
  end
  merge the populations and take the best chromosomes
end

return most fit chromosome from population

3.2.1 Selection

When creating a new generation of chromosomes, certain chromosomes must be selected from the current population as parents to create new offspring chromosomes. This is the first action in the inner loop of the GA algorithm described in Algorithm 1. Some methods of how to select the crossover parents are described below.

Elitism Selection

Only the chromosomes with the highest fitnesses are allowed to create the offspring chromosomes. The number of parent chromosomes to use may vary but the remaining chromosomes will be disregarded.

Roulette-wheel selection

Parent chromosomes are randomly selected, where chromosomes of higher fitnesses are more likely to be selected. The probability distribution is proportional to the fitness values for each chromosome in relation to the total sum of all fitnesses among the whole population. As seen in Figure 1, similar to a roulette-wheel, each chromosome has its own segment where the size of the segment depends on its fitness.
Tournament selection
Parent chromosomes are selected from tournament pools consisting of a certain number of randomly chosen candidates. The size of the pool may vary. The chromosome with the highest fitness from each tournament pool is chosen as the chromosome to be used for the crossover.

3.2.2 Crossover
Crossover is the act of combining two parent chromosomes to create new offspring chromosomes. Different crossover methods vary in how and which genes are carried over from each parent. Some crossover methods are described below.

Single point crossover
A single gene index is randomly chosen. The offspring genes will consist of all genes up to that randomized index from one of the parents and the rest of the genes from the second parent. Either one or two offsprings may be created from the chosen parent chromosomes. In the Figure 2 below, two offspring chromosomes are created from two parent chromosomes.

Two point crossover
Similar to the single point crossover but uses two randomized indices.

Uniform crossover
Each gene is copied from either the first or the second parent with equal probability.
3.2.3 Mutation

Mutation occurs randomly, with a certain probability, on each gene when an offspring is created. This is the second action in the inner loop of the GA described in Figure 1 which changes the properties of the gene and expands the search space in order to not get stuck in a local optima causing premature convergence. If a gene is mutated, its property is swapped with another gene property from the same chromosome [7].

3.2.4 Repair

For an infeasible chromosome that is outside the search space, the repair algorithm alters the genes that causes the infeasibility to turn the chromosome into a feasible solution [4, 7]. This action is not included in the GA description in Algorithm 1 due to the fact that it is optional depending on the GA application. If applied, a chromosome should be repaired after the mutation.

3.2.5 Extinction

Some chromosomes have to be wiped out from the population, either prior to creating the offsprings or after a certain number of offsprings have been created. This is to make sure that the population size doesn’t grow but instead sticks to one size in each algorithm iteration. This is the last action performed in the GA before starting the creation of the next generation.

3.2.6 Generation

Each iteration of the algorithm creates a set of chromosomes which is called a generation. The number of generations is one way of measuring the effectiveness of the algorithm.

4 Method

This section describes the input data followed by the specific implementation of the GA for this report and the procedure for testing the scalability of the GA.

4.1 The University Course Scheduling Inspired by KTH

4.1.1 Input Data

The input data inspired by KTH are formatted as seen in Figure 3, containing all available rooms and lecturers as well as student groups, their required courses and classes during a one week period. For this experiment, four input files with various sizes were created to study the scalability of the GA. The beginning of each new section is marked with an ‘#’, followed by all entries within that section. The first section contains all available rooms, their capacity of students and also their type. The types of the rooms are lecture, lesson and lab and are represented with integers 0, 1, 2 respectively. Next section lists all courses with their corresponding course name, number of lectures, lessons and labs given during one week. The third section lists all lecturers with their names and a list of arbitrary length with all courses that they may teach. The last section contains
all of the student groups, their names (names of the programmes), number of students and a list of arbitrary length with courses that they need to attend.

A week is divided into five weekdays, Monday to Friday and a weekday is divided into four timeslots of two hours each. This means that every room has a total of 20 timeslots.

Figure 3: Input Data File Format

# ROOMS
RoomName RoomCapacity RoomType
# COURSES
CourseName NumOfLectures NumOfLessons NumOfLabs
# LECTURERS
LecturerName CourseA CourseB...
# student groupS
student groupName NumOfStudents CourseA CourseB...

The four input data files vary in the number of student groups and hence the required courses and classes that needs to be allocated. When the input files were created, the number of events to available timeslots density was taken into account to not differ to an too large extent. Following Table 1 lists the details input data specifications.

<table>
<thead>
<tr>
<th>Input Data File</th>
<th>kth_smallest</th>
<th>kth_small</th>
<th>kth_medium</th>
<th>kth_large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lecture Rooms</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Lesson Rooms</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>Lab Rooms</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>Courses</td>
<td>6</td>
<td>12</td>
<td>15</td>
<td>21</td>
</tr>
<tr>
<td>Lecturers</td>
<td>4</td>
<td>9</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>Student Groups</td>
<td>3</td>
<td>6</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>Total Events</td>
<td>41</td>
<td>70</td>
<td>115</td>
<td>159</td>
</tr>
<tr>
<td>Total Timeslots</td>
<td>100</td>
<td>160</td>
<td>240</td>
<td>320</td>
</tr>
<tr>
<td>Event Density</td>
<td>0.41</td>
<td>0.44</td>
<td>0.48</td>
<td>0.50</td>
</tr>
</tbody>
</table>

All student groups have two courses where each course has one to three event that needs to be scheduled. The event density is the number of events divided by number of timeslots.

4.1.2 Data Structures

From the input files, objects for each room, course, lecturer, and student group are created. Each room object stores information of the size and type of the room, each course object stores information of the number of lectures, lessons and labs that needs to be scheduled for each week, each lecturer object store which courses that lecturer may teach, and the student group objects store the
size of that group and what courses they attend.

From the data, all the events that needs to be scheduled are created. An event stores information of the number of people attending that event, the room type needed for that event, the student group that attends it, the course, and the lecturer of it. For every course for every student group, different events of all three types are created. Each event is given a unique id. The lab events have a max size of 25 to match the size of the lab rooms, the lessons events have a max size of 40 and the lecture events are the full student group size. It is assumed that a student group is never of a size larger than the largest lecture halls. Since the student groups are generally larger than the max size of the labs and lessons, student groups form several events from one lesson or lab. They are given an event group id in order to be able to group those events together.

Pseudocode for the event creation follows.

**Algorithm 2: Event Creation**

```plaintext
nextGroupID ← 0

foreach studentgroup do
    foreach course of that studentgroup do
        for i ← 1 to course.numLectures do
            Create Lecture event
        end
        for i ← 1 to course.numLessons do
            size ← studentgroup.size
            groupID ← nextGroupID + +
            while size > 0 do
                eventSize ← min(maxLessonSize, size)
                Create Lesson event of eventSize with groupID
                size ← size − eventSize
            end
        end
        for i ← 1 to course.numLabs do
            size ← studentgroup.size
            groupID ← nextGroupID + +
            while size > 0 do
                eventSize ← min(maxLabSize, size)
                Create Lab event of eventSize with groupID
                size ← size − eventSize
            end
        end
    end
end
```

All the data described so far is persistent during the execution of the rest of the program. The GA merely changes the timeslots that the event ids are placed in.

A timetable for KTH consists of a list of the timetables for each room. A timetable for a single room is represented as an integer matrix where each row represents a certain timeslot of a day and each column represents the weekdays. The integer values that are written to this matrix are the IDs of the events that are being scheduled.
4.2 Solving the University Course Scheduling Problem Inspired by KTH Using a Genetic Algorithm

For each of the input data files, the problem is considered solved when all events are assigned a timeslot and at the same time not breaching any of the following constraints. Those constraints are hard, which means that if not met would mean that the schedule is unapplicable. The scalability of the GA is measured based on the total number of required generations.

List of Hard Constraints
- All courses need to have their required hours scheduled.
- An invalid timeslot may not be used.
- A student or an instructor can not be at two places at the same time.
- A room can not hold more students or instructors than its capacity.
- A room can not be used for more than one lecture at the same time.
- A lecture with special equipment requirements needs to be scheduled to a room that accommodates those requirements.

One may also consider soft constraints but those are not taken into account for this report. Soft constraints are the type of constraints that if met would make that schedule more desirable but if not, the schedule would still be feasible. Some typical soft constraints are that the distance between two rooms in which a person attends sequentially should not be too far away from each other, and that there should be few unused time slots in between classes for both students and instructors.

4.2.1 The Implementation of the Genetic Algorithm

The GA runs a main-loop until one feasible solution is found. There are several steps in this loop. The first steps include creating a new population of offsprings. This is done by running a loop to fill the new population where in each iteration two parents are chosen with the roulette-wheel selection technique and a new chromosome is generated from the two parents using a single-point crossover. The offspring genes are then randomly mutated with a certain probability and then repaired if it turns out to be outside the search space. Before proceeding to the next step, the fitness is evaluated for each offspring chromosome using the fitness function. The fitness function weights all constraint breaches and generates an negative integer as long as the timetable chromosome is not feasible. The second step is to merge the two populations in order to create a new one, exticting unfit chromosomes. This is done by simply choosing the best chromosomes from the old population and the child population until the next population is at full size. The merge first sorts the child population which means the next population remains sorted. This creates the new generation.
ALGORITHM 3: Actual Implementation of Genetic Algorithm

create random population and evaluate fitness of its chromosomes

while most fit individual is not fit enough do
  while offspring population is not full do
    select two parent chromosomes with roulette selection
    perform single point crossover with the two parent chromosomes
    mutate offspring chromosome
    repair offspring chromosome
    evaluate fitness of offspring chromosome
    add offspring chromosome to offspring population
  end
  merge the parent and offspring populations
  delete the rest of the chromosomes
end
return most fit chromosome from population

4.2.2 Fitness

The fitness for each chromosome, or timetable is evaluated by calling a set of constraints functions listed below. For a timetable, every breached hard constraint gives a negative addition to the fitness value and when all events are assigned a timeslot without breaching any hard constraint, the fitness reaches the value of zero. The fitness function uses a linear weighting model for the different constraints and for this implementation certain hard constraints were weighted higher than others. The reason for this is that with the specific input data that was used, the constraints with the higher weights were more common during early testing. By punishing those constraints harder, the algorithm was more stable.

\[
  \text{fitness}(\text{timetable}) = -(2s\text{gd}(\text{timetable}) + l\text{db}(\text{timetable}) + 4r\text{cb}(\text{timetable}) + 4r\text{tb}(\text{timetable}))
\]  

\(sgdb\) returns the number of student group double bookings
\(ldb\) returns the number lecturer double bookings
\(rcb\) returns the number of room capacity breaches
\(rtb\) returns the number of room type breaches
ALGORITHM 4: StudentGroupDoubleBookings

numBreaches ← 0

foreach timeslot do
    foreach studentgroup do
        eventGroupCounts ← emptymap
        foreach roomtimetable do
            if timeslot is booked and the event belongs to this student group then
                eventGroupID ← groupID of this event
                eventGroupCounts[eventGroupID] + +
            end
        end
        sum ← sum of all eventGroup sizes
        biggestGroupSize ← size of the biggest event group
        numBreaches ← numBreaches + sum − biggestGroupSize
    end
end

return numBreaches

ALGORITHM 5: LecturerDoubleBookings

numBreaches ← 0

foreach lecturer do
    foreach timeslot do
        numBookings ← 0
        foreach roomtimetable do
            if timeslot is booked and event is of Lecture type then
                if event.lecturer = lecturer then
                    numBookings + +
                end
            end
        end
        if numBookings > 1 then
            numBreaches ← numBreaches + numBookings − 1
        end
    end
end

return numBreaches

ALGORITHM 6: RoomCapacityBreaches

numBreaches ← 0

foreach roomtimetable do
    foreach timeslot do
        if timeslot is booked then
            if event.size > roomsize then
                numBreaches + +
            end
        end
    end
end

return numBreaches
ALGORITHM 7: RoomTypeBreaches

\[
\text{numBreaches} \leftarrow 0 \\
\text{foreach } \text{roomtimetable do} \\
\text{foreach } \text{timeslot do} \\
\quad \text{if timeslot is booked then} \\
\quad \quad \text{if event.type} \neq \text{roomtype then} \\
\quad \quad \quad \text{numBreaches}++ \\
\quad \text{end} \\
\text{end} \\
\text{end} \\
\text{return } \text{numBreaches}
\]

The data structure makes it impossible to violate the constraints Invalid timeslots may not be used and A room can not be used for more than one lecture at a time since the valid timeslots are the only ones available and each cell of the matrices cannot hold more than one event id.

4.2.3 Population Size

The full size of the population was set to 100. A larger population size allowed for finding a solution with fewer number of generations. However, each generation required longer computation times.

4.2.4 Selection

Roulette-wheel selection was chosen as the method of selecting the parent chromosomes to create an offspring. Tournament selection and elitism selection were also tested on the input data files but the roulette-wheel selection could find a solution with the least number of generations. 100 offspring chromosomes were created in each generation prior to the extinction.

4.2.5 Crossover

Single point crossover was chosen as the crossover method since it was significantly faster than the uniform crossover. In the single point crossover all events booked to timeslot-indices up to that point are copied from the first parent and the rest from the second parent as described in the Algorithm 8.
4.2.6 Mutation

The mutation algorithm goes through every timeslot in every room. With probability given by the mutation rate the current timeslot is mutated. This is done by taking a random other timeslot in the same room and swapping the booked events between those timeslots. The mutation rate for each gene was set to 6%. Higher mutation rates had a tendency to decrease the fitness improvement per generation during the early stage while lower mutation rates sometimes got stuck in local optimum before reaching the fitness of zero. This was observed during early testing.

4.2.7 Repair

After the crossover and/or mutation it is possible that a certain event has been booked to more than one slot or that it is not booked at all. The repair algorithm goes through all the available timeslots and keeps track of all the slots that each event has been booked to as well as all the unbooked time slots. For each event that has been booked to more than one slot, the repair algorithm removes bookings of that event randomly until there is only one booking left. The freed slots are added to the unused slots. The unbooked events are then put in the unused slots randomly. After the repair, each event is supposed to be booked to exactly one timeslot.
**ALGORITHM 9: Repair Timetable**

```plaintext
locations ← empty map of lists
unusedSlots ← empty list

foreach roomtimetable do
    foreach timeslot do
        if timeslot is not booked then
            add this timeslot-room to unusedSlots
        else
            add this timeslot-room to locations[eventID]
        end
    end
end

unbookedEvents ← empty list

foreach eventID do
    if locations[eventID] is empty then
        add eventID to unbookedEvents
    else if locations.size > 1 then
        slots ← locations[eventID]
        shuffle slots
        while slots.size > 1 do
            remove first of slots
            add the removed timeslot-room to unusedSlots
        end
        remove the corresponding booking from the room schedule
    end
end

shuffle unusedSlots

foreach eventID in unbookedEvents do
    take first from unusedSlots and book eventID to it
end
```

### 4.3 Testing the Scalability of the Genetic Algorithm

Since a timetable is either high qualitative with a fitness of zero or not applicable at all (negative fitness), the scalability of the GA was evaluated based on the total number of required generations observed over several runs. The standard deviations was also computed.

For all input data files, 20 runs were performed. Each run was terminated when the problem was considered solved, i.e. when a high quality solution with the fitness of zero was found. In each run, the number of generations was recorded and then later used to evaluate the average number of required generations for each of the four input data files. Those numbers were used to study the scalability of the GA along with the standard deviations. Once the average number of generations was evaluated, several runs were also made per input data file until one with similar number of generations as the average number was recorded. The change of the fitness with respect to the generation for those runs were used to create the graphs for each input data file listed in the result section.

The GA was able to always solve the problem within 60 seconds on an Intel Core2 Quad CPU Q9550 @ 2.83GHz processor equipped computer with 4GB
memory running on Ubuntu 12.04 LTS 64-bit.

5 Results

Running the GA 20 times with a population size of 100 timetables and a mutation rate of 6% on four input data files, the results turned out as shown in the following figures.

Figure 4 shows the average number of generations needed to solve the problem as well as the standard deviation for each input file.

Figure 5 shows the same data as Figure 4 but plotted in a graph where the y-axis represents the number of required generations and the x-axis the size of the input data, precisely being the number of events.

Figure 6 shows the GA results for each of the 20 runs for each input data file.

![Figure 4: Number of Generations and Standard Deviation](image)
5.1 Results for Smallest

The average number of generations was 57.9 with the standard deviation of 12.79. A sample run with 56 generations is shown in Figure 7.
5.2 Results for Small

The average number of generations was 126.8 with the standard deviation of 38.56. A sample run with 125 generations is shown in Figure 8.

5.3 Results for Medium

The average number of generations was 830.1 with the standard deviation of 271.44. A sample run with 873 generations is shown in Figure 9.
5.4 Results for Large

The average number of generations was 3276.2 with the standard deviation of 2133.53. A sample run with 2860 generations is shown in Figure 10.

6 Discussion

Figure 4 and Figure 6 showed a seemingly exponential increase of both the number of generations needed to reach a solution as well as the standard deviation. Due to the number of dimensions in the input data, defining an actual size of each of them was a problem. In Figure 5, the x-axis representing the size of the data was plotted with respect to the number of events for each of the four
input data files. Since the variance in event density was relatively low for all of the input data files, the number of events was considered the most suited parameter for representing the problem size. It is worth noticing that variations in the other input parameters such as number of available lecturers per course could yield a different scaling result. One such scenario could be produced by allowing all lecturers to teach all courses instead of a subset.

As seen in Figure 4, the fitness rapidly increased during the early generations and almost stagnated near zero. These characteristics are evident in the larger input data files such as kth_medium and kth_large. Lowering the fitness requirements for a high quality solution would therefore yield a better scalability of the GA although it might not be of interest since a timetable with a negative fitness is not applicable in practice. However, there are possibilities for altering the strategy when reaching a near zero fitness to e.g. local search since such an algorithm can address the timeslots that breaches the constraints with better accuracy. For this reason it could be interesting to investigate the scalability of the first $N$ generations. In that case the scalability could instead be defined as a percentage of how much the fitness has increased from the initial fitness to the fitness after those $N$ generations.

Part of the reason for the exponentially increasing standard deviation could be explained with the GA parameter settings. The parameters were the same for each input file which is most likely suboptimal as the size of the space of appropriate parameter values seems to decrease as the input size increases. This means that the parameters that were used may have been sufficient for the smaller input files but worse for the larger ones which in turn could have caused higher deviations that would have been seen if the parameter settings were better fitted for each specific input file. This problem could most likely be improved upon by applying some of the improvements to the algorithm discussed below.

With a relatively low probability, the measurements of the number of generations needed to reach the desired fitness were significantly higher than the average which is due to the stochastic nature of GAs. Worst case for the largest input data required 13463 generations before reaching a fitness of zero and 2159 generations for kth_small, which is almost 20 times the average. This can also be seen from the increase of the standard deviation with the input data size. If the input data would have been even larger this would be more problematic, which could mean that this relatively simple approach to using a GA to solve this problem would be a bit too simple. In order to improve the algorithm a number of different techniques could be explored but the general idea would be to make the algorithm more adaptive to the current state of the population. These numbers were not taken into account when creating the figures in the result section but was replaced with an additional run.

**Improvement I, Adaptive Fitness Weighting Model**

For the results from the testing, the algorithm used a specific weighting of the constraint breaches in the fitness function in order to punish certain errors more. The way these weights were devised was highly dependent on the input data that was used. In order to make this more general and at the same time more adaptive, one improvement could be to change the weights on different constraints.
during runtime. This could be done by keeping track of how the number of constraint breaches for each type of constraint change and then increasing the weight for a certain constraint if it does not improve for some time or lower it in the opposite case. This would make it easier for the algorithm to move away from regions in the search space that does not lead to a solution and change trajectory towards better regions.

**Improvement II, Adaptive Selection Method**

The different selection methods have different properties. The advantage of the roulette-wheel selection is that it gives at least a small chance for the less fit parents to create offspring which in turn could mean a higher genetic variation in the population. This is most likely best to use in the earlier generations in order to explore a larger part of the search space. When the population has a higher average fitness it could be wise to increase the rate at which the better fit individuals procreate and lessen the same rate for the less fit individuals. To simply change selection method at a certain fitness level could however lead the population to getting stuck at local optima. Instead the choice of selection method should also be adaptive with regards to the change in fitness over many generations.

**Improvement III, Adaptive Mutation Rate**

Another way to increase the genetic variation in the population when the fitness becomes stagnant is to increase the mutation rate and lower it in the opposite case.

**Improvement IV, Using Domain Knowledge in Repair Algorithm**

The implementation of the repair algorithm in this report is rather simple as it only makes sure all events are accounted for. If more domain knowledge were to be included in this algorithm solutions could be found faster. Domain knowledge in this case refers to paying attention to the type of events and rooms and make smarter choices with regards to that and the constraints specified. If, for instance, the number of breaches of a certain hard constraint could be minimised in the repair algorithm the explored search space could be much smaller.

**Improvement V, Adaptive Crossover Algorithm**

Different crossover algorithms have different properties which means that the choice of crossover technique could also be adaptive. The uniform crossover brings in more genetic variance but is also more destructive to good solutions and the point-crossover is more prone to maintain blocks of eventbookings that work well together but creates at the same time a less diverge population. For this reason it could be wise to use different crossover algorithms during different stages.

**Improvement VI, Identifying Building Blocks**

According to Goldbergs Building Block Hypothesis on exactly why GAs work, a large part of it is due to the constructive effects of crossover and mutation which find and combine building blocks (schemata) of solutions [7]. The timetabling problem, in the way it was chosen to be represented in this report, is a sort of permutation problem, unique events need to be placed in a certain order. The building blocks in this case refers to certain combinations of assignments
of events to certain slots that work well together, that is they do not breach constraints. At the same time the crossover and mutation also have destructive effects as they might destroy good building blocks. If the destructive effects could be decreased and the constructive effects increased a good solution could be found faster. A lot of research has been put into better identifying building blocks, such as the messy GA [8]. Solving this problem with a messy GA would mean a complete rewrite of the program though. It might be possible to find other techniques to identify these building blocks better though.

Improvement VII, Using Genetic Algorithms as Input to Another Optimizing Algorithm

The results shown in Figure 10 show a fairly rapid increase of fitness at the start and then most of the time is spent fixing the last few errors. For this reason, another idea could be to run the GA for a set number of generations and then use the resulting solution as input to another search algorithm to find a better solution [9]. By doing this it might be possible to benefit from the quick fitness increase of GA while avoiding the slow final part of it.

Improvement VIII, Parallelisation

GAs in general are very well suited for parallel computing since the fitnesses of each chromosome can be calculated separately. The crossovers during the creation of the next population can also be done in parallel.

For this implementation little time was invested in choosing good parameter settings for the GA. This would have been less of a problem if they were more adaptive as discussed above, but when the parameter settings are static the algorithm may be more sensitive. One (meta-) way of doing this could be to use another GA to find the best parameter settings. The parameter settings for the meta-GA would not matter as much since it would only need to run once.

7 Conclusions

The results in this paper show that the GA rapidly increases the quality of a timetable during the early stage of the algorithm for all sizes of input data. Smaller input data have a tendency to then quickly find a high quality solution that doesn’t breach any of the hard constraints while larger data require a significantly higher number of generations. By defining the feasible solution to be a timetable breaching no constraints at all, the scalability of the GA is seemingly exponential. Future work may be aimed to optimize the parameter settings and altering various methods used in the GA to create a more dynamic and adaptive algorithm.

References


Appendices

A. Java Source Code to Genetic Algorithm
https://github.com/hvy/kthscheduling/tree/cleaned_up

B. kth_smallest

# ROOMS
D1 200 0
D45 40 1
E35 40 1
SPEL 40 2
SPOR 30 2

# COURSES
CALC 2 1 0
JAVA 1 0 1
MULT 2 0 1
CTEC 1 2 0
CSEC 0 1 1
SCON 1 1 1

# LECTURERS
SVEN CALC MULT
BERT JAVA SCON OOPC
KARL CSEC
GUNN CTEC

# student groups
COMP.1 200 CALC JAVA
COMP.2 120 MULT CTEC
COMP.3 70 CSEC SCON

C. kth_small

# ROOMS
D1 200 0
D3 50 1
D45 40 1
E1 350 0
E35 40 1
SPEL 40 2
SPOR 30 2
MUSI 40 2

# COURSES
CALC 2 1 0
JAVA 1 0 1
MULT 2 0 1
CTEC 1 2 0
CSEC 0 1 1
SCON 1 1 1
DIGI 1 0 1
ENCM 1 0 1
ALGD 1 1 0

24
ELEC 1 0 0
PROB 1 0 1
OPER 1 1 0
# LECTURERS
SVEN CALC MULT
BERT JAVA SCON OOPC
KARL CSEC
GUNN CTEC
BERI DIGI
SARA OPER
OLLE ENGM ELEC
BENG ALGD
PELL PROB
# student groups
COMP_1 200 CALC JAVA
COMP_2 120 MULT CTEC
COMP_3 70 CSEC SCON
INFO_1 200 DIGI ENGM
INFO_2 100 ALGD ELEC
INFO_3 50 PROB OPER

C. kth medium

# ROOMS
D1 200 0
D2 50 1
D3 50 1
D45 40 1
D46 40 1
E1 350 0
E35 40 1
SPEL 40 2
SPOR 30 2
MUSI 40 2
ROD 30 2
ORA 30 2
# COURSES
CALC 2 1 0
JAVA 1 0 1
MULT 2 0 1
CTEC 1 2 0
CSEC 0 1 1
SCON 1 1 1
DIGI 1 0 1
ENGM 1 0 1
ALGD 1 1 0
ELEC 1 0 0
PROB 1 0 1
OPER 1 1 0
TERM 2 0 1
DIFF 2 1 0
MECH 0 1 2
OOPC 1 1 1
REAC 1 0 2
# LECTURERS
SVEN CALC MULT
BERT JAVA SCON OOPC
KARL CSEC
GUNN CTEC
BERI DIGI
ERIK DIFF
SARA OPER
OLLE ENGM ELEC
BENG ALGD
JUDI TERM REAC
MANS MECH
PELL PROB
# student groups
COMP 1 200 CALC JAVA
COMP 2 120 MULT CTEC
COMP 3 70 CSEC SCON
INFO 1 200 DIGI ENGM
INFO 2 100 ALGD ELEC
INFO 3 50 PROB OPER
PHYS 1 200 CALC TERM
PHYS 2 180 DIFF MECH
C. kth large
# ROOMS
D1 200 0
D2 50 1
D3 50 1
D45 40 1
D46 40 1
E1 350 0
E35 40 1
E36 40 1
F1 300 0
SPEL 40 2
SPOR 30 2
MUSI 40 2
ROD 30 2
ORA 30 2
VIO 40 2
GRA 30 2
# COURSES
CALC 2 1 0
JAVA 1 0 1
MULT 2 0 1
CTEC 1 2 0
CSEC 0 1 1
SCON 1 1 1
DIGI 1 0 1
ENGM 1 0 1
ALGD 1 1 0
ELEC 1 0 0
PROB 1 0 1
OPER 1 1 0
TERM 2 0 1
DIFF 2 1 0
MECH 0 1 2
QUAN 1 1 0
OOPC 1 1 1
TCHE 2 1 0
PERS 1 0 0
REAC 1 0 2
POLY 1 1 0

# LECTURERS
SVEN CALC MULT
BERT JAVA SCON OOPC
KARL CSEC
GUNN CTEC
BERI DIGI
ERIK DIFF
SARA OPER
OLLE ENGM ELEC
BENG ALGD
JUDI TERM REAC
MANS MECH
MICH QUAN
PELL PROB
DARI TCHE POLY
MORT PERS

# student groups
COMP 1 200 CALC JAVA
COMP 2 120 MULT CTEC
COMP 3 70 CSEC SCON
INFO 1 200 DIGI ENGM
INFO 2 100 ALGD ELEC
INFO 3 50 PROB OPER
PHYS 1 200 CALC TERM
PHYS 2 180 DIFF MECH
PHYS 3 100 QUAN OOPC
CHEM 1 150 CALC TCHE
CHEM 2 130 PERS DIFF
CHEM 3 100 REAC POLY