DEGREE PROJECT, IN COMPUTER SCIENCE , FIRST LEVEL
STOCKHOLM, SWEDEN 2015

A comparative analysis of a Tabu Search and a Genetic Algorithm for solving a University Course Timetabling Problem

CASPER RENMAN AND HAMPUS FRISTEDT

KTH ROYAL INSTITUTE OF TECHNOLOGY

CSC SCHOOL

Abstract

This work implements a Tabu search (TS) algorithm for solving an instance of the University Course Timetabling Problem (UCTP). The algorithm is compared to a Genetic algorithm (GA) implemented by Yamazaki and Pertoft (2014) that solves the same instance. The purpose of the work is to explore how the approaches differ for this particular instance, and specifically investigate whether a TS algorithm alone is sufficient, considering the small size of the UCTP instance. The TS was based on the OpenTS library (Harder, 2001) and reuses parts from Yamazaki and Pertoft's (2014) GA. The results show that the TS alone is too slow. Even in a small instance of the UCTP, the search space is too big for the TS to be fast. This confirms the state of the art, that a combination of TS and GA would perform better than either of them individually. Further improvements on the TS would involve reducing the number of moves generated each iteration.

Sammanfattning

Detta arbete implementerar en *Tabu search*-algoritm (TS) för att lösa en instans av *University Course Timetabling Problem* (UCTP). Algoritmen jämförs med en *Genetic algoritm* (GA) implementerad av Yamazaki och Pertoft (2014) som löser samma instans. Syftet med arbetet är att utforska hur angreppssätten skiljer sig för denna instans, och specifikt undersöka om det räcker att endast använda en TS-algoritm, eftersom instansen är liten. TS-algoritmen baserades på *OpenTS*-biblioteket (Harder, 2001) och återanvänder delar av Yamazaki och Pertofs (2014) GA. Resultaten visar att TS-algoritmen är för långsam. Även på en liten instans av UCTP är sökrymden för stor för att TS-algoritmen ska vara snabb. Detta bekräftar aktuell forskning som säger att en kombination av TS och GA skulle ge bättre resultat än algoritmerna var för sig. Vidare förbättringar av TS-algoritmen skulle involvera att minska antalet drag som genereras varje iteration.

Contents

Chapter 1

Introduction

Most universities face the challenge of having to provide schedules for the teachers and students at the university. A naive solution would be to let every course responsible teacher choose the schedule for its course, without taking any other course into account. This would make it easy for the teacher, but create a lot of collisions in the students' schedules. The given approach is thus to try to avoid these collisions, while trying to satisfy everybody's needs. Solving this manually would require a significant amount of time, depending on the size of the university. Therefore, researchers have tried to make the process easier by automating it, using various kinds of algorithms and techniques. The problem goes by the name *University Course Timetabling Problem* (UCTP).

UCTP is a difficult combinatorial problem, partly because there are a lot of variables involved such as teachers and students with constraints between them, and partly because every university has its own variables and constraints. A UCTP involves assigning course events, students and lecturers to a time and place while violating as few constraints as possible.

Dorigo and Stützle (2004) defined the UCTP as follows:

> *"one is given a set of time slots, a set of events, a set of rooms, a set of features, a set of students, and two types of constraints: hard and soft constraints. Hard constraints have to be satisfied by any feasible solution, while soft constraints do not concern the feasibility of a solution but determine its quality. The goal is to assign the events to the time slots and to the rooms, so that all the hard constraints are satisfied and an objective function whose value depends on the number of violated soft constraints, is optimized".*

1.1 Previous research

Research on UCTP has been done since the 1960's and the problem is considered to be NP-complete (Awad et al., 2011). Various kinds of algorithms have been used over the years, including algorithms based on Integer programming, Network flow and Graph coloring as mentioned by Schaerf (1999). Lewis (2008) finds that metaheuristic-based techniques are the most appropriate techniques for approximating solutions to the UCTP problem.

Metaheuristics are heuristics for solving combinatorial optimization problems. These problems are to find the best solution from a set of possible solutions. The solution could be a graph, an integer, a permutation, a subset or a structure specific to the problem. In the UCTP a solution is a timetable. A loose definition of metaheuristics is (Bianchi et al., 2009):

> Given a finite set S of feasible solutions x, and a real valued cost function $G(x)$, find $min_{x \in S}\ G(x)$. The set S is called the search space.

Al-Betar and Khader (2012) explain that the metaheuristic algorithms are commonly divided into two groups. One group is local search-based which starts with a solution (that could be randomized) and iteratively tries to improve by using a fitness function while trying to satisfy constraints. This goes on until an optimal solution is found. Local search algorithms check the immediate search space for improved solutions, and hence a drawback is that the search can get stuck in a certain area of the search space if no neighbouring solutions are improvments. Tabu search (TS) is a local-search based algorithm.

The other group of metaheuristic algorithms is population-based algorithms. These algorithms start with more than one solution and search a bigger area of the search space by combining current solutions to obtain new ones. Population-based solutions may be worse than the ones produced by local search because they lack precision in the search space (Al-Betar and Khader, 2012). However, population-based algorithms get stuck in a bad part of the search space less often than local search algorithms for the very same reason.

The most popular population-based algorithms are evolutionary algorithms (Ross et al. (1998); Burke et al. (1994); Colorni et al. (1991); Erben (2001)). With evolutionary algorithms there is a risk of premature convergence (Al-Betar and Khader, 2012) where the algorithm ends up with a loss of genetic variation, meaning the algorithm cannot find offspring that is superior to its parents. A genetic algorithm (GA) is an evolutionary algorithm.

1.2 Problem motivation

Al-Betar and Khader (2012) state that in light of the aforementioned, researchers have started focusing more on local search-based algorithms rather than population-based algorithms. Therefore, this work will implement a TS algorithm and compare it to a GA implemented by Yamazaki and Pertoft (2014). They used it to solve a simplified UCTP inspired by KTH. TS is picked because it is a popular local-search based algorithm.

1.2.1 Problem statement

This work implements a TS algorithm that solves this instance of the UCTP. The TS is compared to the GA implemented by (Yamazaki and Pertoft, 2014) with regards to scalability.

The current state of the art is hybrid algorithms that combine GAs and TS. Hybrid algorithms have been shown to be better than other tested methods (Jat and Yang, 2011). Therefore, this work does not intend to declare whether a GA or a TS is best universally. The purpose of this work is to explore how the approaches differ for this particular instance, and specifically investigate whether a TS algorithm alone is sufficient, considering the small size of this UCTP instance.

This work does not apply soft constraints to the problem. The reason for implementing soft constraints would be to explore how the different algorithms handle them. Because (Yamazaki and Pertoft, 2014) did not implement soft constraints, eventual soft constraints in this work could only be compared to speculations of how Yamazaki and Pertoft's (2014) algorithm would handle them. The time limit of the work also plays a part in this. However, thoughts on soft constraints are discussed in the discussion section.

Chapter 2

Background

This section describes Yamazaki and Pertoft's (2014) GA, how TS works and pseudocode for a TS.

2.1 Yamazaki and Pertoft's (2014) Genetic algorithm

GAs use techniques and terminology inspired by natural evolution. This work will use the terminology described by Yamazaki and Pertoft (2014). However, the terminology used to describe GAs is not always consistent in research so discretion is advised.

When describing GAs, a canditate solution is called a chromosome and a set of chromosomes is called a population. The properties of chromosomes are called genes. A fitness function uses these genes to determine the quality of a chromosome. New chromosomes are created by combining genetic operators: crossover and mutation. Crossover means combining two parent chromosomes to create a new child chromosome. It is analogous to real world reproduction. Mutation is the process of randomly changing chromosomes to expand the search space. The details of Yamazaki and Pertoft's (2014) GA can be found in the original paper.

2.2 Tabu search

The following is a short overview of tabu search. For a detailed description, see Reeves (1993).

A problem with local search algorithms is that they tend to get stuck in a limited part of the search space, as mentioned earlier. TS adresses this by allowing the algorithm to move to neighbouring solutions that are worse than the current solution if no improving neighbours are avaliable. The algorithm also discourages revisiting explored search spaced by adding every visited solution to a data structure called the tabu list, which is not necessarily a list in practice, for a limited number of iterations. This number is called tabu tenure. The algorithm is forbidden to explore a solution in the tabu list if that solution has a remaining tabu tenure larger than zero. Reeves (1993) includes an exception to this rule in his description. That exception is called the aspiration criterion. The aspiration criterion allows a solution on the tabu list to be revisited if the solution is an improvement over all previous solutions.

2.2.1 Tabu search pseudocode

This section contains pseudocode for the TS. Note that the aspiration criterion on line 11 allows a solution to be visited if it is better than the previously best solution.

Algorithm 1 Tabu search

```
 1: $tabuList \leftarrow []$
 2: $currentSolution \leftarrow initialSolution$
 3: $bestSolution \leftarrow s$
 4: while $fitness(bestSolution) < 0$ do
 5:     $bestCandidate \leftarrow null$
 6:     for $candidate \in currentSolution.getNeighbourhood$ do
 7:         if $(\neg tabuList.contains(candidate))$ then
 8:             if $(fitness(candidate) > fitness(bestCandidate))$ then
 9:                 $bestCandidate \leftarrow candidate$
10:             end if
11:         else if $(fitness(candidate) > fitness(bestSolution))$ then
12:             $bestCandidate \leftarrow candidate$
13:         end if
14:     end for
15:
16:     $currentSolution \leftarrow bestCandidate$
17:     if $fitness(bestCandidate) > fitness(bestSolution)$ then
18:         $bestSolution \leftarrow bestCandidate$
19:     end if
20:     $tabuList.push(bestCandidate)$
21:     if $tabuList.size > tabuTenure$ then
22:         $tabuList.removeFirst()$
23:     end if
24: end while
25: return $bestSolution$
```

Chapter 3

Method

This section describes the input data, how the experiments were performed, details regarding the implementation as well as the hardware used to run the experiments.

3.1 The implementation

The implementation of the TS extends the OpenTS library (Harder, 2001). OpenTS is a Java library that lets the user supply definitions for a solution, a move and a fitness function and provides the underlying TS algorithm.

Necessary parts for comparing the algorithms were taken from the work of Yamazaki and Pertoft (2014) and supplemented to work with the OpenTS library.

3.1.1 Fitness function

The fitness function used was the same as the one used by Yamazaki and Pertoft (2014). The function takes a timetable as input and returns an integer value. A value is subtracted from the timetable's fitness for every breached hard constraint in the timetable. A timetable without breached hard constraints has a fitness of zero. The fitness function and the hard constraints are defined as follows:

$$fitness(timetable) = -(2sgdb(timetable)+ldb(timetable)+4rcb(timetable)+4rtb(timetable))$$

$sgdb$ returns the number of student group double bookings
ldb returns the number of lecturer double bookings
rcb returns the number of room capacity breaches
rtb returns the number of room type breaches

3.1.2 Choosing moves

A move has to be defined when implementing a local search-based algorithm. This work utilizes three different types of moves as defined by Jat and Yang (2008):

- n_1: Swaps an event with a free timeslot.
- n_2: Swaps two events.
- n_3: Swaps three events e_1, e_2 and e_3 by either moving e_1 to e_2's position, e_2 to e_3's position and e_3 to e_1's position or moving e_1 to e_3's position, e_2 to e_1's position and e_3 to e_2's position.

3.1.3 Generating moves

One n_1 move was generated for every combination of a scheduled event and a free timeslot. The size of the set of all n_1 moves, N_1, is equal to the number of events times the number of empty timeslots.

$|N_1| = |E| * |F|$ where E is the set of all events and F is the set of all free timeslots.

One N_2 move was generated for every combination of two scheduled events. No repetition is allowed because an event cannot be swapped with itself. Because swapping event e_1 with e_2 is the same as swapping e_2 with e_1, the order of events does not matter. Therefore the number of n_2 moves generated is equal to the number of events choose two.

$$|N_2| = \binom{|E|}{2}$$

There are $\binom{|E|}{3}$ event triplets like (e_1, e_2, e_3). Two n_3 moves were generated for every such triplet because there are two ways of swapping three events.

$$|N_3| = 2 * \binom{|E|}{3}$$

3.1.4 Combinations of move sets

All combinations of the sets N_1, N_2 and N_3 were tested when generating the timetable solution. Some combinations were unable to solve the timetable instances and were left out of the results. For example, the combination consisting of only two way swaps (N_2) could rarely solve the problem, because many initializations produced boards unsolvable by n_2 moves. Table 3.1 shows which combinations of move sets were used and the size of the combinations.

Combination	Number of Moves	Used
N_1	$\lvert E\rvert * \lvert F\rvert$	Yes
N_2	$\binom{\lvert E\rvert}{2}$	No
N_3	$2 * \binom{\lvert E\rvert}{3}$	No
$N_1 + N_2$	$\lvert E\rvert * \lvert F\rvert + \binom{\lvert E\rvert}{2}$	Yes
$N_1 + N_3$	$\lvert E\rvert * \lvert F\rvert + 2 * \binom{\lvert E\rvert}{3}$	Yes
$N_2 + N_3$	$\binom{\lvert E\rvert}{2} + 2 * \binom{\lvert E\rvert}{3}$	No
$N_1 + N_2 + N_3$	$\lvert E\rvert * \lvert F\rvert + \binom{\lvert E\rvert}{2} + 2 * \binom{\lvert E\rvert}{3}$	Yes

Table 3.1: Specification of move sets

3.1.5 Evaluation

A move's fitness was evaluated by copying the current timetable, performing the move on the copy, and executing the fitness function on the modified copy. That method of evaluation is called absolute evaluation. Another method of evaluation is incremental evaluation. Incremental evaluation does not evaluate the entire solution. Instead it only evaluates the parts of the solution that were modified by the last move. Incremental evaluation was not used because Yamazaki and Pertoft's (2014) fitness function lacks the functionality required.

3.2 How the tests were performed

An accepted solution was defined to be a timetable with a fitness of zero. Therefore, the solver was programmed to run until a fitness of zero was achieved. The number of iterations required to reach a fitness of zero was measured. The time required to solve a problem instance was also measured.

Every experiment had to be repeated because the initial states were randomized. The sample size of 20 was chosen because Yamazaki and Pertoft (2014) used the same sample size.

3.2.1 Hardware

The experiments were performed on a 64-bit Arch Linux laptop with an Intel Core2 Duo T7500 CPU @ 2.2GHz and 2GB RAM.

3.2.2 Comparison

Yamazaki and Pertoft (2014) chose to measure scalability of their algorithm by the number of generations required to obtain a solved timetable. The scalability of the TS was measured by the number of iterations and time required. Therefore the generations and iterations were compared. In order to get a valid comparison of the time required, Yamazaki and Pertoft's (2014) GA was used to solve the input data again on the aforementioned hardware.

3.3 Input data

The input data is the same as the data used by (Yamazaki and Pertoft, 2014). Each room has a total of 20 schedulable timeslots per week: five days per week and four timeslots per day. The input data specifies the number of events and the timeslots, but not where the events are to be placed initially. Therefore, the solver randomly schedules each event to a free timeslot during initialization. Details regarding the input data are shown in Table 3.2.

Table 3.2: Input Data Specifications (Yamazaki and Pertoft, 2014)

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

Chapter 4

Results

This chapter describes initialization of the input data and shows the results of the experiments.

4.1 Initialization

Figure 4.1 shows the variation in initial fitness values. The horizontal solid line is the mean fitness value of the different input sizes. The dotted lines are one standard deviation above and below the mean fitness value.

Table 4.1 contains the actual values for the mean fitness value and standard deviation at initialization.

Table 4.1: Mean values and standard deviations

Input data	Mean value	Standard deviation
XS	149.5	17.38
S	254.1	23.81
M	426.4	36.00
L	617.5	27.26

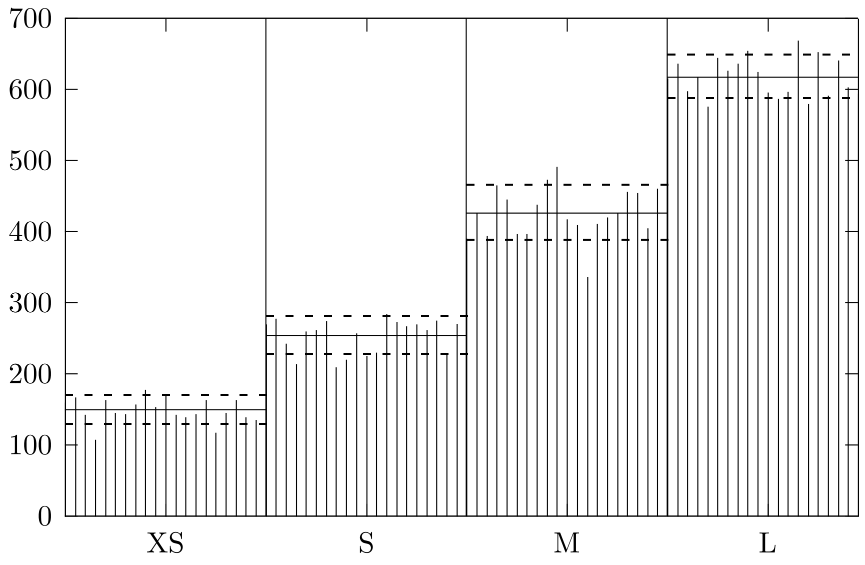

Figure 4.1: Fitness values (negative) after randomized initialization.

4.2 Scalability

4.2.1 TS

Figure 4.2 shows the mean number of iterations required by the TS to find a solution. Figure 4.3 shows the mean amount of time required.

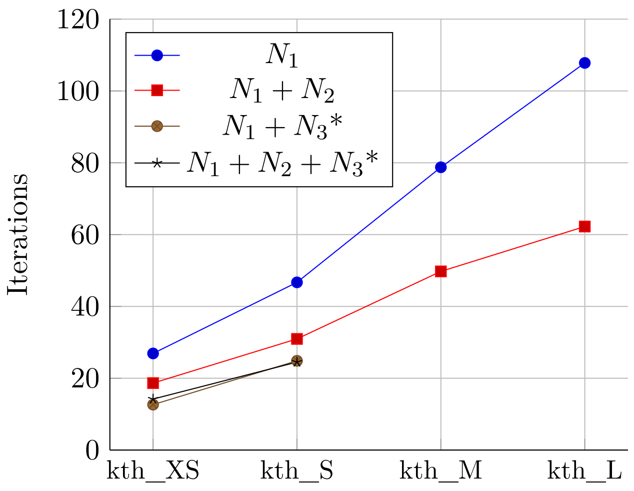

Figure 4.2: Iterations required

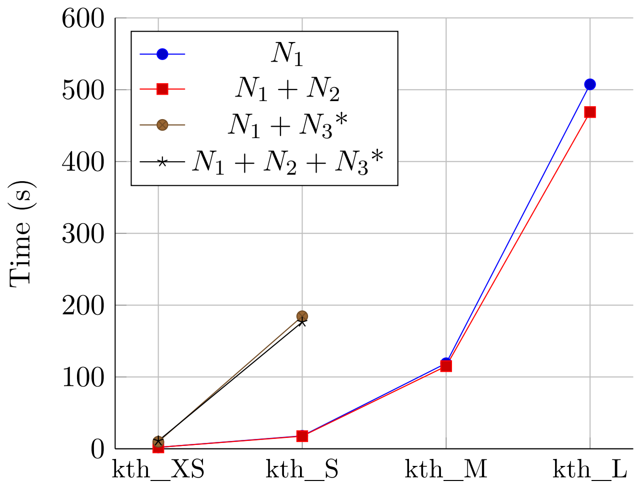

Figure 4.3: Time required

Plots marked with * in the legend were not run to completion because of slow performance. An analyzis of the results follows in the next chapter.

4.2.2 Yamazaki and Pertoft's (2014) GA

Figures 4.4 and 4.5 show the results of rerunning Yamazaki and Pertoft's (2014) GA on the aforementioned hardware.

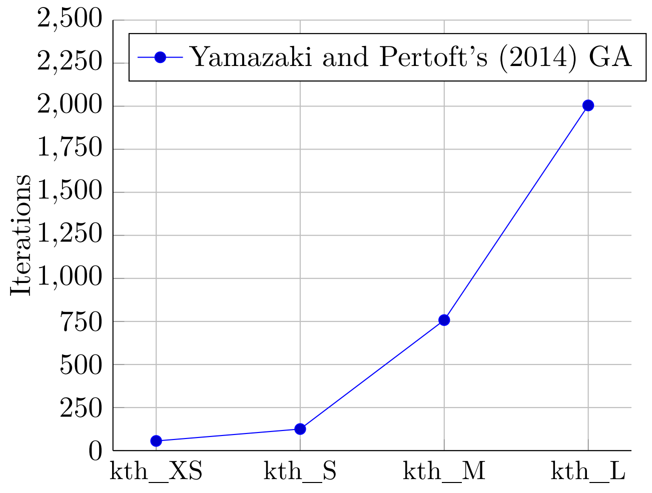

Figure 4.4: Iterations required

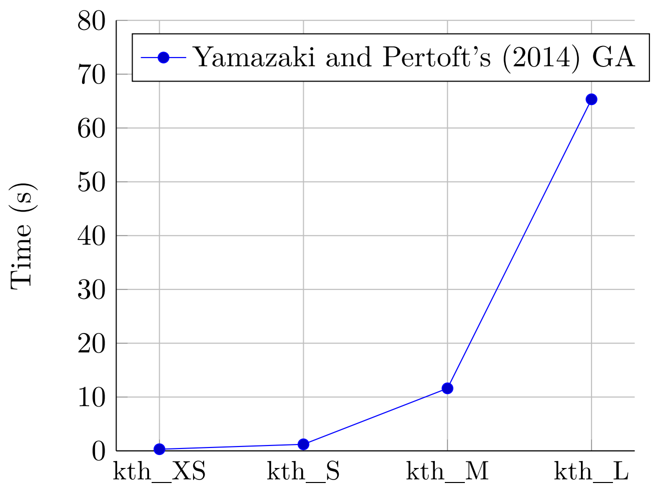

Figure 4.5: Time required

Chapter 5

Discussion

This chapter contains analysis and discussion of the results.

5.1 Initialization

The initial positions of the events are randomized, as mentioned in the method section. Table 4.1 and figure 4.1 show that the standard deviations of the initial fitness values are small. This supports the conclusion that trying to optimize the initial state of a solution will not make a noteable difference in the end result.

5.2 The results of the TS

The scalability of the TS can be measured by looking at how the number of iterations grow and time increases when the number of events grow. Figures 4.2 and 4.3 show that the number of iterations required to solve the problem scales linearly and that the time required grows polynomially or exponentially.

5.2.1 Number of moves

It is given that the number of generated moves M depends on the number of events ($|E|$). In the case of N_2 and N_3, $|E|$ is the only variable needed to determine M as shown in the method section. N_1 additionally requires the number of free timeslots $|F|$ to determine M. $|F|$ is assumed to be approximately equal to $|E|$ because the event density is near 0.5 in each input file. Using these facts and the formulae from the method section we can determine how the number of moves scales when $|E|$ increases.

$N_1 \in \mathcal{O}(|E|^2)$ because

$$|N_1| = |E| * |F| \approx |E| * |E| = |E|^2$$

.

$N_2 \in \mathcal{O}(|E|^2)$ because

$$|N_2| = \binom{|E|}{2} = \frac{|E|!}{2!(|E|-2)!} = \frac{|E|(|E|-1)}{2} = \frac{|E|^2 - |E|}{2}$$

$N_3 \in \mathcal{O}(|E|^3)$ because

$$|N_3| = 2*\binom{|E|}{3} = 2*\frac{|E|!}{3!(|E|-3)!} = 2*\frac{|E|(|E|-1)(|E|-2)}{6} = 2*\frac{|E|^3 - 3|E|^2 + 2|E|}{6}$$

Note that although both N_2 and N_3 asymptotically grow polynomially with the number of events, N_3 grows considerably faster for practical purposes. For example, consider that kth_L has 159 events. Then N_2 contains $\binom{159}{2} = 12561$ moves, but N_3 contains $2 * \binom{159}{3} = 1314718$. This results in a considerable difference of magnitude, which explains why no combinations containing N_3 could finish the experiments in reasonable time.

5.2.2 Evaluation

The fitness function also scales with input data. Each time the fitness function is called, four types of hard constraints are investigated:

- Student group double bookings
- Lecturer double bookings
- Room capacity breaches
- Room type breaches

All four types of hard constraints require iterating through every room timetable, every day and every timeslot. The number of days and number of timeslots remain constant, but the number of rooms increases with larger test data.

5.3 Comparison

This section contains details about the comparison between the algorithms.

Figures 4.4, 4.5 and 4.3 show that the growth patterns of the algorithms are similiar. However, it is clear that the GA is faster.

5.3.1 Evaluation

A GA works by selecting two timetables (parents) from a group of timetables (the population), creating an offspring by combining the two timetables, mutating the offspring timetable and then evaluating the fitness of that timetable. It does this for all groups of timetables. (Yamazaki and Pertoft, 2014) use a population size of 100 timetables, which means that they evaluate a timetable 100 times per generation. In the case of the TS which uses the same evaluation of a timetable, the evaluation happens once per possible move. Using the move sets N_1 and N_2 on kth_XS, this means 3239 times per iteration. This is one explanation for the difference in execution time as well as general differences between the two algorithms.

5.3.2 Characteristics

Yamazaki and Pertoft (2014) noted that their GA converged towards an acceptable solution considerably faster during the first generations. Due to this the authors concluded that their GA could be used to solve a problem for a set number of generations. The algorithm then stops and feeds the result to another algorithm. Doing so would utilize the fast initial stage of the GA while avoiding the slower stages.

The TS solves the problem linearly regarding iterations, but every iteration takes a long time. Reducing the number of iterations would therefore make the TS faster. The natural conclusion is what Yamazaki and Pertoft (2014) suggested in the previous paragraph, namely that combining the TS with the GA would be beneficial.

5.4 Improvements

This section contains information pertaining the different potential improvements of the TS.

5.4.1 Only generate moves for events violating constraints

An area of optimization to be explored is to reduce the amount of moves generated. One way of accomplishing that would be to focus on the events that are breaching constraints. Because the TS already automatically chooses the move that best improves the timetable, no change would occur regarding the number of iterations required. However, the main benefit would be that the TS would not have to evaluate every possible move. This might be the most important improvement because the vast number of moves generated is closely related to the slow execution times.

5.4.2 The fitness function and incremental evaluation

As discussed in the method section, the TS utilizes absolute evaluation, meaning that moves cannot be evaluated without evaluating the entire timetable. One improvement would be implementing a method of incremental evaluation. Incremental evaluation would allow the evaluation of moves without evaluating the entire timetable. Because the TA evaluates a large set of moves each iteration, such an improvement could reduce the time required by a considerable amount.

5.4.3 Extending the problem with soft constraints

Real life scheduling problems are often modelled by using both hard and soft constraints. Yamazaki and Pertoft (2014) were planning to implement soft constraints by letting fulfilled soft constraints add a positive integer to the fitness value of a timetable. That approach requires that a solution without violated hard constraints is found before soft constraints are taken into consideration. Otherwise a solution with violated hard constraints might be accepted as correct because of a large number of fulfilled soft constraints pushing the fitness value above zero. Regardless, soft constraints would be a necessary addition to a real world scheduler.

5.4.4 Parallelisation

The OpenTS library natively supports a multithreaded TS (Harder, 2001). The multithreaded solver evenly splits the neighbourhood into partitions and distributes the partitions to different threads. Such a solver has obvious benefits for computers with multiple processor cores. However, parallelisation requires that the entire application is well threaded, which the TS currently is not.

Chapter 6

Conclusion

This work implements a TS algorithm and compares it to a GA. Note that time has limited the extent of the research and the implementation. The TS proved to be too slow to be a recommended substitution for Yamazaki and Pertoft's (2014) GA. The TS is slow because a large number of moves are evaluated each iteration. To speed it up, the TS could be combined with the GA to create a UCTP solver that is efficient in all stages as seen in earlier research (Jat and Yang, 2011). The GA would initially search a wider search space and find a near optimal solution. Then the TS would finish the search by finding a locally optimal solution. This work hypothesizes that the most beneficial improvement to the TS would be to reduce the amount of moves generated per iteration. The suggested approach is to only evaluate the moves of events that breach hard constraints, which would reduce the size of the neighbourhood and thus consume less computational time.

Bibliography

Mohammed Azmi Al-Betar and Ahamad Tajudin Khader. A harmony search algorithm for university course timetabling. *Annals of Operations Research*, 194(1): 3–31, 2012.

Y Awad, A Dawood, and A Badr. An evolutionary immune approach for university course timetabling. *IJCSNS*, 11(2):127, 2011.

Leonora Bianchi, Marco Dorigo, Luca Maria Gambardella, and Walter J Gutjahr. A survey on metaheuristics for stochastic combinatorial optimization. *Natural Computing: an international journal*, 8(2):239–287, 2009.

EK Burke, DG Elliman, and RF Weare. A genetic algorithm based university timetabling system. In *Proceedings of the 2nd East-West International Conference on Computer Technologies in Education*, volume 1, pages 35–40, 1994.

Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. Genetic algorithms and highly constrained problems: The time-table case. In *Parallel problem solving from nature*, pages 55–59. Springer, 1991.

Marco Dorigo and Thomas Stützle. *Ant Colony Optimization*. Bradford Company, Scituate, MA, USA, 2004. ISBN 0262042193.

Wilhelm Erben. A grouping genetic algorithm for graph colouring and exam timetabling. In *Practice and Theory of Automated Timetabling III*, pages 132–156. Springer, 2001.

R Harder. Opents–java tabu search framework, 2001.

Sadaf Naseem Jat and Shengxiang Yang. A memetic algorithm for the university course timetabling problem. In *Tools with Artificial Intelligence, 2008. ICTAI'08. 20th IEEE International Conference on*, volume 1, pages 427–433. IEEE, 2008.

Sadaf Naseem Jat and Shengxiang Yang. A hybrid genetic algorithm and tabu search approach for post enrolment course timetabling. *Journal of Scheduling*, 14 (6):617–637, 2011.

Rhydian Lewis. A survey of metaheuristic-based techniques for university timetabling problems. *OR spectrum*, 30(1):167–190, 2008.

Colin R Reeves. *Modern heuristic techniques for combinatorial problems.* John Wiley & Sons, Inc., 1993.

Peter Ross, Emma Hart, and Dave Corne. Some observations about ga-based exam timetabling. In *Practice and Theory of Automated Timetabling II*, pages 115–129. Springer, 1998.

Andrea Schaerf. A survey of automated timetabling. *Artificial intelligence review*, 13(2):87–127, 1999.

Hiroyuki Vincent Yamazaki and John Pertoft. *Scalability of a Genetic Algorithm that solves a University Course Scheduling Problem Inspired by KTH.* 2014.