DEGREE PROJECT IN COMPUTER ENGINEERING,
FIRST CYCLE, 15 CREDITS
STOCKHOLM, SWEDEN 2018

Investigating the Reliability of Known University Course Timetabling Problem Solving Algorithms with Updated Constraints

ROBERT BERGGREN

TIMMY NIELSEN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Investigating the Reliability of Known University Course Timetabling Problem Solving Algorithms with Updated Constraints

ROBERT BERGGREN

TIMMY NIELSEN

Master in Computer Science
Date: December 13, 2018
Supervisor: Alexander Kozlov
Examiner: Örjan Ekeberg
Swedish title: Forskning kring pålitligheten hos kända universitetsutbildningsplaneringsproblemlösningsalgoritmer med uppdaterade begränsningar
School of Electrical Engineering and Computer Science
KTH - Royal Institute of Technology
Stockholm, Sweden

Abstract

Scheduling lectures, exams, seminars etc. for a university turns out to be a harder task than what it seems to be at first glance. This problem is known as the University Course Timetabling Problem (UCTP). The UCTP has been hosted for a number of competitions throughout the years by an organization called Practice and Theory of Automated Timetabling (PATAT). Because of these competitions, the problem has been given a standard description and set of constraints as well as standard problem instances for easier comparison of research and work on the subject. However, setting a standard like this have a major drawback; no variety is introduced since new research for finding the greatest method to solve the UCTP is forced to focus on a specific set of constraints, and algorithms developed will only be optimized with these constraints in consideration.

In this research we compared five well known UCTP algorithms with the standard set of constraints to a different set of constraints. The comparisons showed a difference in the rank of performance between the algorithms when some constraints were changed to fit a certain need. The differences were not great but big enough to state that previous research declaring what algorithms are best for the UCTP problem cannot be relied upon unless you use close to identical sets of constraints. If the goal is to find the best algorithm for a new set of constraints then one should not rely on a single previously defined great algorithm but instead take two or three of the top performing ones for the greatest chance of finding the most optimized solution possible.

Sammanfattning

Schemaläggning av föreläsningar, tentamen, seminarier etc. för ett universitet visar sig vara en svårare uppgift än vad det verkar vid första anblicken. Detta problem är känt som University Course Timetabling Problem (UCTP). UCTP har varit centralt i ett antal tävlingar genom åren av organisationen Practice and Theory of Automated Timetabling (PATAT). På grund av dessa tävlingar har problemet fått en standardbeskrivning och en uppsättning specifika begränsningar samt standard problemdata för enklare jämförelse av forskning och arbete i ämnet. Att sätta denna typ av standard har dock en stor nackdel; ingen variation tillförs då ny forskning för att hitta den bästa optimeringsmetoden inom UCTP tvingas att fokusera på en specifik uppsättning begränsningar och algoritmer som utvecklas kommer då endast att optimeras med dessa begränsningar i beaktande.

I den här rapporten jämförde vi fem välkända UCTP algoritmer med standarduppsättningen av begränsningar mot en annan uppsättning begränsningar. Jämförelserna visade en skillnad i prestationsordningen mellan algoritmerna när vissa begränsningar ändrats för att passa ett visst behov. Skillnaderna var inte enorma men tillräckligt stora för att påvisa att tidigare forskning som förklarar vilka algoritmer som är bäst för UCTP-problemet ej är pålitlig om du inte använder nära till identiska uppsättningar av begränsningar. Om målet är att hitta den bästa algoritmen för en ny uppsättning begränsningar, bör man inte lita på en tidigare definierad effektiv algoritm utan istället använda sig utav två eller tre av de starkaste algoritmerna för den största chansen att hitta den mest optimerade lösningen.

Contents

Chapter 1

Introduction

Schools, universities and bigger organizations have always faced the problem of scheduling events (lectures, exams, seminars etc.) into a limited number of time slots, following specified constraints. The problem concerns fitting all scheduled events within an acceptable time-frame whilst avoiding collision between shared courses among students/employees. This problem belongs to the class "NP-complete problems", meaning there is to this day no known algorithm that can compute an optimal solution in polynomial time, essentially concluding that the only way to optimize this problem is through advanced approximation and heuristics.

Todays standard definition of the University Course Timetabling Problem was created by Ben Paechter and accepted by PATAT (see section 2.3) in 2002. It consists of a set of events to be scheduled in 45 time slots (5 days of 9 hours each), a set of rooms in which events can take place, a set of students who attend the events, and a set of features satisfied by rooms and required by events. Each student attends a number of events and each room has a maximum capacity. A solution is achieved when all events have been assigned a time slot and room so that all features required by an event are fulfilled [13].

A solution can be improved by following a set of constraints. The constrains are split into 2 sections, **hard** and **soft** constraints. The **hard** constraints have to be satisfied at all costs, any violation of a hard constraint will render that solution invalid. The **soft** constraints on the other hand may be violated, although each violation of such a constraint will cause a penalty point (penalty cost = 1 for each violation per student). All penalty points are accumulated into a final re-

sult which provides a direct summary of how well the algorithm performed. The objective is to satisfy all hard constraints and to minimize the number of students involved in the violation of soft constraints.

The standard definition and set of constraints of the UCTP are strictly shared among the worlds more popular reports regarding the UCTP [12, 13, 14, 16] which makes comparison between different algorithms possible. This paper aims to compare five well known algorithms for the UCTP with the standard set of constraints to the same algorithms but for a different set of constraints based on what we as students, as well as our fellow students, find of greater importance (see section 1.1 for adjustments). Breaking the pattern of using all the same constraints will potentially give a new perspective on the algorithms and show not only how they perform, but also how they adapt.

1.1 Societal aspects

In a demanding education climate, finding the most optimal algorithm(s) for a preferred set of constraints would provide a better standard of efficient planning, resulting in increased overall society development. A well-customized timetable making the best use of space also provides more free time for people, making their personal lives more manageable in accordance to the schedule they follow on daily basis.

1.2 Problem statement

This report will be focused on comparing five popular UCTP algorithms and conclude if the rank of performance between these algorithms persists with changing constraints. The new set of constraints was created by us conducting a qualitative research through asking fellow students[1] if they liked the new constraints more than the original ones or if they would like to modify them. They all tended to agree that the new set of constraints was better and no further modifications were needed.

When researching the best adjustment of constraints it was found that the hard constraints will remain the same, while the soft constraints will be improved to satisfy the actual opinion among students

[1]DISCLAIMER: There were in total 20 students who agreed to participate in this research anonymously

for an optimal schedule arithmetic. All of this results in the following research question:

Will the rank of performance between five popular UCTP scheduling algorithms persist when changing the soft constraints for the UCTP?

If the rank changes significantly, any results or conclusions from earlier comparisons will be weakened since the previously found "best performing algorithm(s)" might not be the best one(s) to use in practice unless the set of constraints used are close to identical with the standard set. Below follows a comparison between the original constraints and the adjusted constraints.

1.2.1 Original hard constraints, un-adjusted

- No student shall be assigned more than one event at the same time.
- The assigned room is big enough for all the attending students and satisfies all the features required by the event.
- No more than one lecture is allowed at a time slot in each room.

1.2.2 Original soft constraints

- A student should not have a course scheduled in the last time slot of the day.
- A student should not have more than two consecutive lectures without a break.
- A student should not have a single course on a day.

1.2.3 Adjusted soft constraints

- A student should not have a course scheduled in the first or last time slot of the day.
- A student should not have breaks between lectures.
- A student should not have one single lecture on a day.

The constraints are customized for schools consisting of programs containing specific courses linked to that program. This follows from the hard constraint which specifies that no student can attend more than one event at a time. If a student chooses his or her own courses freely this constraint would more than likely be violated, since a student might choose two courses with time slots that coincide. In conclusion, the results of this paper will not take into account if a school offers the opportunity to choose courses freely. Doing this might lead to unattendable classes and is done at a student's own risk.

Feasible solutions are always considered to be superior to unfeasible solutions, independently of the numbers of soft constraint violations. In fact, in any comparison, all unfeasible solutions are to be considered equally worthless. The objective is to minimize the number of soft constraint violations in a feasible solution.

1.3 Problem instances

As mentioned earlier; the timetabling problem has a uniform description in order for different algorithms to be comparable. The problem instances used for different studies have also stayed uniform [12, 13, 14, 16] so that all comparisons are valid. The instances were generated using a problem instance generator created by Ben Paechter [13]. All problem instances produced have a perfect solution, *i.e.* a solution with no constraint violations, hard or soft. There are three classes of problem instances (small, medium and large) which are defined with the sets of parameters shown in Table 1.1.

Class	small	medium	large
Num_events	100	400	400
Num_rooms	5	10	10
Num_features	5	5	10
Approx_features_per_room	3	3	5
Percent_feature_use	70	80	90
Num_students	80	200	400
Max_events_per_student	20	20	20
Max_students_per_event	20	50	100

Table 1.1: Parameters used to produce the different instance classes

For clarification, the parameter *Percent_feature_use* is explained as "[...]the approximate percentage of features used by the events [...]" [13]. Along with these parameters is an integer acting as a seed in the random processes. Using the same integer produces the same problem instance.

1.4 Approach

The algorithms to be compared will be edited to match the new set of constraints. The algorithms will then be run, both with the old and new set of constraints. The results will then be compared to find out if the rank of performance differ between these sets.

The five algorithms used are:

- Ant Colony Optimization (ACO)
- Genetic Algorithm (GA)
- Iterated Local Search (ILS)
- Simulated Annealing (SA)
- Tabu Search (TS)

The reason behind choosing these particular algorithms is that they all read and store data for the problem in the same way and their existing code has almost identical implementation for computing constraint violations. The main difference between the algorithms is the logic and mechanisms applied when computing each step in its heuristic search. This means that any changes added to the way of computing constraint violations will affect all algorithms equally and therefore minimize the risk of corrupt results.

Chapter 2

Background

2.1 Glossary

Abbreviation	Translation	Meaning
UCTP	University Course Timetabling Problem	The problem of scheduling a set of events to a set of rooms in a timetable
Bijective		A one-to-one mapping of a problem to another representation
PATAT	The Practice and Theory of Automated Timetabling	The main conference for research on automated timetabling
Neighbour		A solution achieved by making a single alteration to current solution
Neighbourhood		All neighbours available from current solution
ACO	Ant Colony Optimization	One of the algorithms used in the UCTP research of this report
GA	Genetic Algorithm	One of the algorithms used in the UCTP research of this report

Table 2.1: Words and their meaning

Abbreviation	**Translation**	**Meaning**
ILS	Iterated Local Search	One of the algorithms used in the UCTP research of this report
SA	Simulated Annealing	One of the algorithms used in the UCTP research of this report
TS	Tabu Search	One of the algorithms used in the UCTP research of this report
Swarm Intelligence Algorithm		A term used for optimization algorithms mimicking natural biological systems
Hcv	Hard constraint violations	The amount of hard constraint violations
Scv	Soft constraint violations	The amount of soft constraint violations
Heuristic		An algorithmic approach to solving problems through educated guessing or approximation
Metaheuristic		A wrapper-heuristic using one or more of lower-level heuristics to maximize efficiency of guessing or approximating solutions

Table 2.2: Extension of glossary

2.2 The metaheuristic approach

The fact that there is no method capable of finding an absolute optimal solution to the UCTP is unanimously agreed upon among developers and researchers. This has been a known fact since the late middle of the 20's century and is still to be disproven. This is, in other words, called an "NP-complete problem" which is the exact reason for this being a widely researched subject where progress can, and is, still being made [6, 10].

To solve NP-complete problems, heuristics are used. During the last 2 decades scientific communities have directed a lot of attention towards metaheuristic algorithms, and good progress has been made. A series of metaheuristic methods have been implemented and continuously updated with the goal of improving algorithm performance in regards to the timetabling problem. What has been pinpointed as the most reliable algorithms are those based on the so-called tabu-approach, local search, simulated annealing and a couple more discussed further in the algorithm section [3, 4, 17].

2.3 PATAT

The EURO Working Group on Automated Timetabling (EWG-PATAT) holds PATAT, or "The International Series of Conferences on the Practice and Theory of Automated Timetabling" as their main conference.

PATAT is a forum held biennially, attended by an international community of scientists, researchers and practitioners of the UCTP. The purpose of the forum is to compare, discuss and compete in order to invest in the timetabling community and continuously improve the development rate of modern UCTP heuristics [11].

2.4 Algorithms

We will in the following section briefly go through the five algorithms used in the research. For the interested reader a more detailed explanation of the inner workings of each algorithm as well as pseudo code can be found in Rossi-Doria et al. [12].

2.4.1 Local Search

The local search might be considered the most basic of the heuristic algorithms of this paper as well as in the scientific society.

The main functionality of local search revolves around its so-called *neighbourhood comparison*, where it compares a suggested solution to available *neighbour* solutions. Two solutions are considered neighbours when they are differentiated by one single operation. An operation in this case would be to move one event to another time slot or switching time slots between two events. The fact that the operation makes the minimum number of changes possible to reach a different solution (one move or switch) is what gives rise to the label "neighbour" which essentially means a close-by, or local, solution. In the pseudo-code below, $N(S)$ represents the neighbourhood of solution S.

Algorithm 1 Local Search

1: $S \leftarrow$ *an arbitrary solution*
2: **while** $\exists S' \in N(S)$ *such that cost(S') <* $cost(S)$ **do**
3: $\quad S \leftarrow S'$
4: **return** S

The purpose of local search is to apply the neighbour comparison to a solution until a better local solution is found. This is then repeated for the improved solution until there are no neighbour solutions better than the current solution. This is what is called "local search" [1].

The problem concerning local search is that it can get stuck in a local minimum. If the current solution has no possible single-move operations to improve the solution the algorithm will terminate as it cannot make improvements. This gives rise to a dilemma where there might be a better solution several moves ahead, but these are not explored since no move is done from current solution.

Because of this, regular local search is not often used by its own in optimization problems. It is, though, useful within other algorithms. In fact, most of the other algorithms in schedule optimization use local search as a low-level comparison algorithm when potential solutions are found [15].

2.4.2 Ant Colony Optimization Algorithm (ACO)

The ant colony algorithm is a member of the swarm intelligence algorithms and was originally used as a method for finding an optimal path in a graph. In problems not directly linked to a graph, the algorithm is often transformed into a bijective graph representation. ACO has been successfully applied to several combinatorial problems and has shown good results for the UCTP [16].

The algorithm aims to mimic the social behaviour of ants trying to find the correct path to a source of food. Every ant releases a pheromone which is detectable by other ants trying to navigate, contributing to the probability of an ant choosing the same path as the majority. The networking system revolves around the fact that an ant returns to its starting position after finding a destination, making the shorter paths stronger in pheromone concentration since the shorter traveling time results in less pheromone evaporation during that period.

The most challenging part of simulating an ant colonization process arises when deciding the evaporation rate of the pheromone. A poor choice of evaporation time-limit increases the risk of premature and unoptimized solutions.

The probability of an ant choosing the path from node i to node j is typically determined by an expression as shown below:

$$p_{ij} = \frac{\phi_{ij}^{\alpha} d_{ij}^{\beta}}{\sum_{i,j=1}^{n} \phi_{ij}^{\alpha} d_{ij}^{\beta}},$$

where α and β are a pre-calculated pair of influence parameters ($\alpha \approx \beta \approx 2$), ϕ_{ij} is the pheromone concentration on the route between i and j, and d_{ij} is the desirability of the same route [15, 9].

The internal structure and process of calculation makes the ACO particularly suitable for discrete optimization problems, making it possible to be successfully applied to the timetabling problem. As a matter of fact, the algorithm used for this instance of the UCTP is the first ever ACO-approach to a timetabling problem and has been shown to perform adequately in comparison to the other popular optimization algorithms [12].

2.4.3 Genetic Algorithm (GA)

The genetic algorithm, known also as the evolutionary algorithm, is a computerized representation of evolution through natural selection.

This is one of the most consistently top-performing methods among the different approaches to the timetabling problem [10].

There are three major stages of the simulated evolution:

1. Selection
2. Reproduction
3. Replacement

The selection is based on primitive natural selection; survival of the fittest. Fitter individuals have a higher chance of reproducing new individuals into the algorithm.

The reproduction process concerns combining two parents fit for reproduction, producing a new individual sharing properties from its parents. Mutation is also a part of this stage, meaning to make small inconsistent alterations on a new individual.

In the third and final process, a number of individuals from the original population are replaced by the new ones. In the *generational GA* an entire generation is replaced every iteration, although testing has proven that a *steady-state* approach gives better results, meaning that a subset of individuals are chosen as parents in the reproductive stage as well as replacements in the replacement stage. Therefore, the implementation serving our purpose uses steady-state genetics [12].

2.4.4 Iterated Local Search (ILS)

To increase the reliability of local search, an **iterated** local search algorithm may be applied. This algorithm, as the name implies, is an extension of the original local search where an iterator over local solutions is implemented. The main functionality of this algorithm revolves around building a set of solutions gathered through running the embedded local search algorithm on different neighbourhoods and comparing the solutions in the hopes of finding something close to a global optimal solution.

The change, or *perturbation*, of neighbourhood when a local minimum is found will differ between different interpretations of the algorithm. There is no particular agreement of which perturbation technique is best when switching neighbourhood, which makes this a question of the developers personal choice of design as well as what specific problem you are aiming to resolve. To perturbate through pure

randomization is inconsistent and does not guarantee above-average result quality. Because of this, different methods to increase the reliability of the perturbation are made [7, 12].

The ILS perturbation technique in the algorithm used consists of 3 different moves repeatedly used in a somewhat random order:

- Choose a different time slot for a randomly chosen event
- Swap the time slot of two randomly chosen events
- Choose randomly between the two previous types of moves and a 3-cycle of the time slots for three distinct events

Some amount of randomness in the perturbation is unavoidable when operating on an NP-complete problem since there is no way of knowing where a better solution can be found. The "degree" of randomness may vary between implementations, since there are several methods increasing the odds of finding a good solution space. Specific methods preferred in situations outside of the timetabling problem scope will not be discussed since it is not the main focus of the project, although pseudo-code of the ILS based on an arbitrary perturbation is provided below.

Algorithm 2 Iterated Local Search

1: $S_0 \leftarrow$ *Initial Solution*
2: $S^* \leftarrow LocalSearch(S_0)$
3: **repeat**
4: $\quad S' \leftarrow Perturbation(S^*, history)$
5: $\quad S'^* \leftarrow LocalSearch(S')$
6: $\quad S^* \leftarrow CompareByCriterion(S^*, S'^*)$
7: **until** termination condition is met

2.4.5 Simulated Annealing (SA)

Simulated annealing is based on the process of annealing in physics, which is performed to find the lowest energy ground state of solid materials by heating up the system and letting it cool while monitoring the properties of the material [2].

SA has proven to be preferable in many combinatorial optimization problems for approximating a global optima, although converting this algorithm from its field of use in physics is a complicated task and the

quality depends highly on the design approach and expertise of the developers [5].

The SA algorithm referred to in this paper can be seen as another approach to iterating over a simple local search algorithm. A move to an improving solution is always accepted whilst moving to a worse one can be accepted based on a probability distribution over an evaluation function. The probability distribution used is shown below, making a move less likely to be accepted the worse it is. It is called the Metropolis distribution and it revolves around creating a virtual temperature T for simulating an annealing approach to the problem.

$$p_{accept}(T, s, s') = \begin{cases} 1 & \text{if } f(s') < f(s) \\ e^{-\frac{(f(s') - f(s))}{T}} & \text{otherwise,} \end{cases}$$

where s represent the current solution, s' the neighbouring solution and f(s) the evaluation function on s.

Two different approaches to neighbourhood exploration was implemented for the timetabling problem. The first approach uses local search to improve solutions, in the manner that all possible moves from a solution to the next are considered until a feasible solution is found. The difference from regular local search in this case is that the algorithm does not end after all moves in a neighbourhood have been considered, but instead it continues re-evaluating the solution space in a different order until a stopping criterion is met (in this case the preset time limit). The second approach abandons the local search to use a completely random move selection strategy, making the principles of the algorithm as well as the pseudo-code more complicated but granting better performance. Testing of the two approaches consequently resulted in a decision of using the random move selection implementation in the final version of the simulated annealing algorithm referred to in the results of this report [12].

2.4.6 Tabu Search (TS)

Tabu search is defined by its high abstraction-level design where it is highly dependent on using other heuristics to provide solutions. Although it is a more complicated algorithm than the ILS, it is similar in the sense that it acts as a wrapper around lower level heuristics, or

metaheuristics (usually only one). These heuristics provide new solutions for the TS to filter through in the hopes of identifying the best possible move to make in each step of the process.

TS typically use local search to provide new solutions, although TS may affect the local search in such a manner that it accepts transitioning into a worse solution if no improved solutions are found or if the neighbouring solution is *tabu*-marked. This is to increase efficiency and avoid the known flaw of local search where the algorithm reaches a local minimum.

TS puts previously visited or invalid solutions into a "tabu-list" which is used to avoid visiting solutions already computed whereas you might get circular or time-consuming dependencies. Previously visited solutions are only held as tabu for a certain amount of time, since it may be viable to re-visit a solution if no remarkable improvement is made within a specified time limit in the hopes of transpiring through a different path of solutions which might prove more optimal than what was found in the previous attempt [8].

Some might think that TS refers to another genre of heuristics, known as the tabu hyperheuristic. Although that algorithm uses the same structural principles as TS, it is more advanced in the manner that it indirectly finds solutions through iterating over several different lower-level heuristics, choosing a specific heuristic each iteration based on how well they perform according to a set of implemented rules [3]. No hyperheuristic algorithms were used in the purpose of this project and will thus not be dove into further [12].

Chapter 3

Method

The source code to all the algorithms as well as the problem instances were downloaded from IRIDIA[1]. All algorithms provided "Solution" files (*.h *.cpp) which contained the code for computing both soft and hard constraint violations - consequently, these were the main files edited. The functions used for computing the hard constraint violations remained un-edited since the set of hard constraints were not adjusted between the standard and new set. There were multiple functions for computing the soft constraint violations but they were close to identical and could therefore be summarized easily. The following pseudo-codes show both the original code for computing the standard soft constraint violations (algorithm 3) and the new implementation for computing the improved set of soft constraint violations. (algorithm 4)

[1] http://iridia.ulb.ac.be/supp/IridiaSupp2002-001/index.html

Algorithm 3 Original code for computing soft constraint violations

```
1: scv ← 0                                   ▷ set soft constraint violations to zero to start with
   ▷ classes should not be in the last slot of the day
2: for each e ∈ Events do
3:     if e.assignedtimeslot() mod 9 == 8 then
4:         scv ← scv + e.numberOfStudents()
5: ——————————————————————————————
   ▷ students should not have more than two classes in a row
6: for each s ∈ Students do
7:     for each t ∈ timeslots do
8:         if s attends event at time slot t and
            the 2 previous time slots the same day then
9:             scv ← scv + 1
10: ——————————————————————————————
   ▷ students should not have a single class on a day
11: for each s ∈ Students do
12:     for each d ∈ [1...5] do                ▷ d represent current day
13:         classesDay ← classes for student s in day d
14:         if classesDay == 1 then
15:             scv ← scv + 1
    return scv
```

Algorithm 4 Adjusted code for computing new soft constraint violations

```
1: scv ← 0                                   ▷ set soft constraint violations to zeo to start with
   ▷ classes should not be in the first or the last time slot of the day
2: for each e ∈ Events do
3:     if e.assignedtimeslot() mod 9 == 8 or
         e.assignedtimeslot() mod 9 == 0 then
4:         scv ← scv + e.numberOfStudents()
5: ——————————————————————————————
   ▷ students should not have breaks between lectures
6: for each s ∈ Students do
7:     emptyHours ← 0
8:     lectureEarlierToday ← false
9:     currentlyBreak ← false
10:    for each t ∈ timeslots do
11:        if new day then
12:            emptyHours ← 0
13:            lectureEarlierToday ← false
14:            currentlyBreak ← false
15:        if s attends an event at time slot t then
16:            if currentlyBreak == true then
17:                scv ← scv + emptyHours
18:                emptyHours ← 0
19:                currentlyBreak ← false
20:            lectureEarlierToday ← true
21:        else                              ▷ Doesn't attend an event at time slot t
22:            if lectureEarlierToday == true then
23:                currentlyBreak ← true
24:                emptyHours ← emptyHours + 1
25: ——————————————————————————————
   ▷ students should not have a single class on a day
   ▷ kept from original computeSCV()
26: for each s ∈ Students do
27:    for each d ∈ [1...5] do                ▷ d represent current day
28:        classesDay ← classes for student s in day d
29:        if classesDay == 1 then
30:            scv ← scv + 1
   return scv
```

When running the test data, each algorithm was tested on three small, three medium and two large problem instances to create variety in the results. Multiple independent trials were run to get an average result for each instance making the data more reliable, avoiding the risk of one poorly performing test run rendering the result inconsistent through unlucky variance. Each class of the problem instances have a given number of trials to run and each independent trial has a runtime limit shown in Table 3.1. Testing was performed on a PC with an Intel Core i7 7700K 4.7 GHz and was run on a single core.

In order to know how well an algorithm performs, a fitness rating have been created to compare solutions. The fitness rating is calculated in the following way:

$$f(s) := hcv(s) * C + scv(s),$$

where *s* is a given solution, *hcv(s)* is the number of hard constraint violations, *C* is a constant larger than the maximum possible number of soft constraint violations and *scv(s)* is the number of soft constraint violations. The fitness rating represents the quality of a solution, where a low fitness value means few constraint violations and is therefore a good solution whilst a high fitness value represents a bad solution.

Class	independant trials	runtime limit per trial
small	30	80
medium	20	500
large	4	3600

Table 3.1: Number of independent trials and runtime limit per trial and class

Chapter 4

Results

The results obtained in testing the new constraints against the original ones is shown below. For each individual figure displayed, a single problem instance was used, with the exception of the large problem instances. For each small and medium problem instance, all algorithms managed to return solutions with 100 % sucess rate (meaning no hcv) which is why only the results for small and medium problem instances include box plots displaying the spread of the final scv for multiple runs over the same problem instances and algorithms. In every comparison, an edited version of the graph is provided where the TS is cropped out. This is because of the poor performance of the TS making the remaining results undisguishable.

In the box plots, a box shows the range between the 25 % and the 75 % quantile of the data, the median is indicated by a bar, the whiskers extend to the most extreme data point that is no more than 1.5 times the interquantile range from the box and the outliers are indicated as crosses.

The large problem instances have a tendency to fail at producing feasible solutions. Figure 4.7 shows the percentage of invalid solutions for each algorithm over the collective large problem instances. Because of this, all comparisons between large problem instances have been left out.

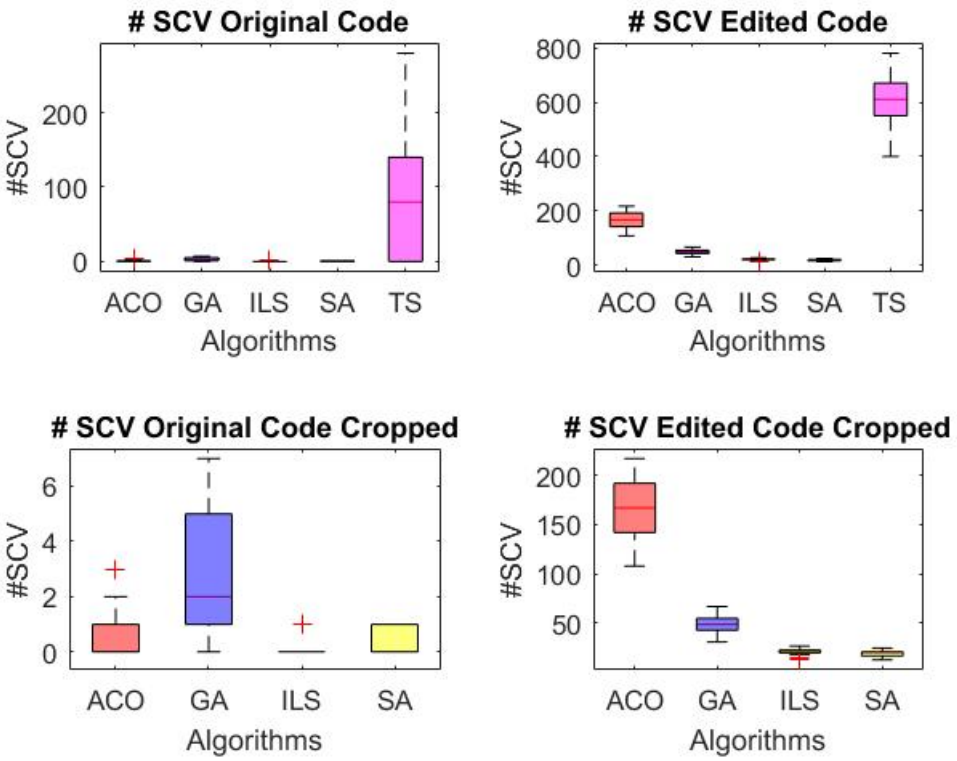

Figure 4.1: Instance: Small 1

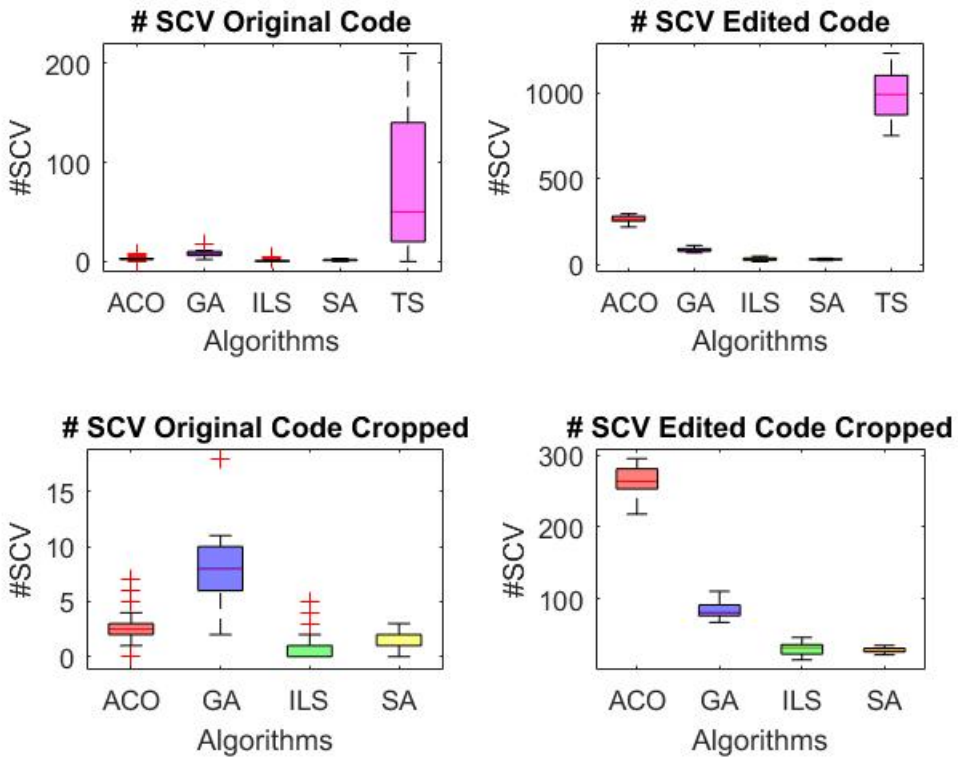

Figure 4.2: Instance: Small 2

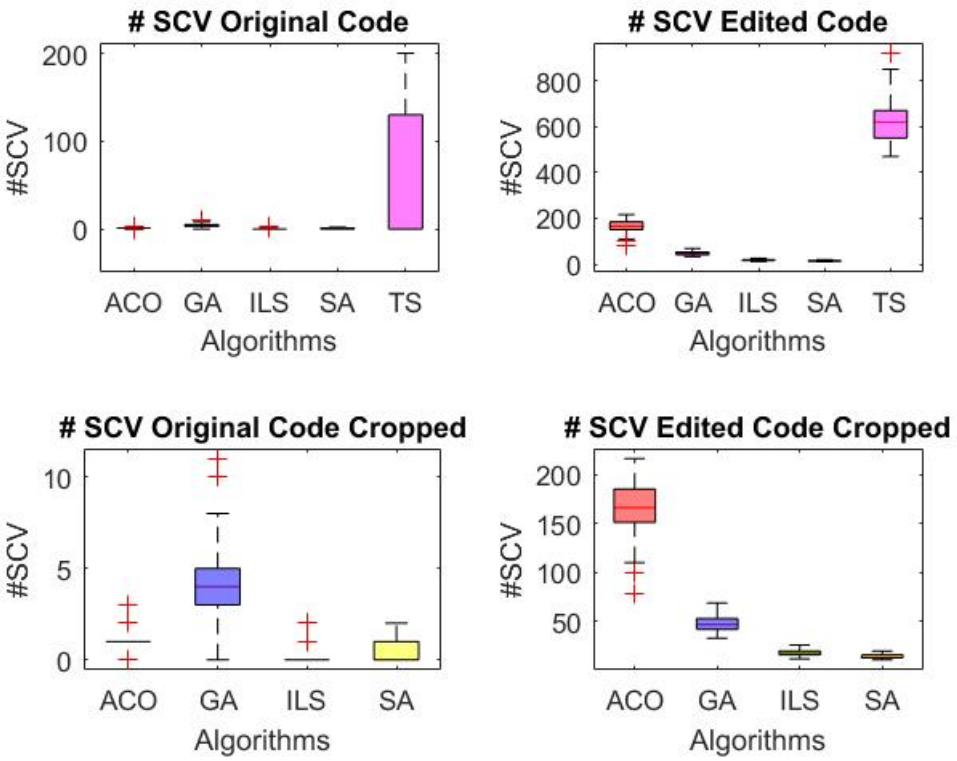

Figure 4.3: Instance: Small 3

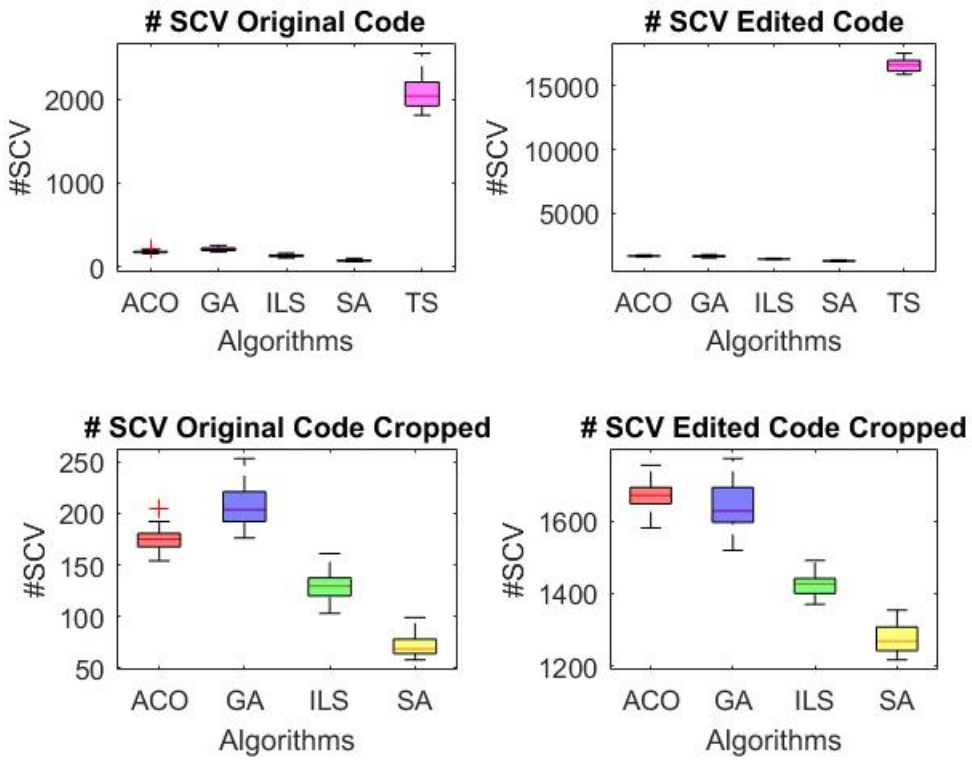

Figure 4.4: Instance: Medium 1

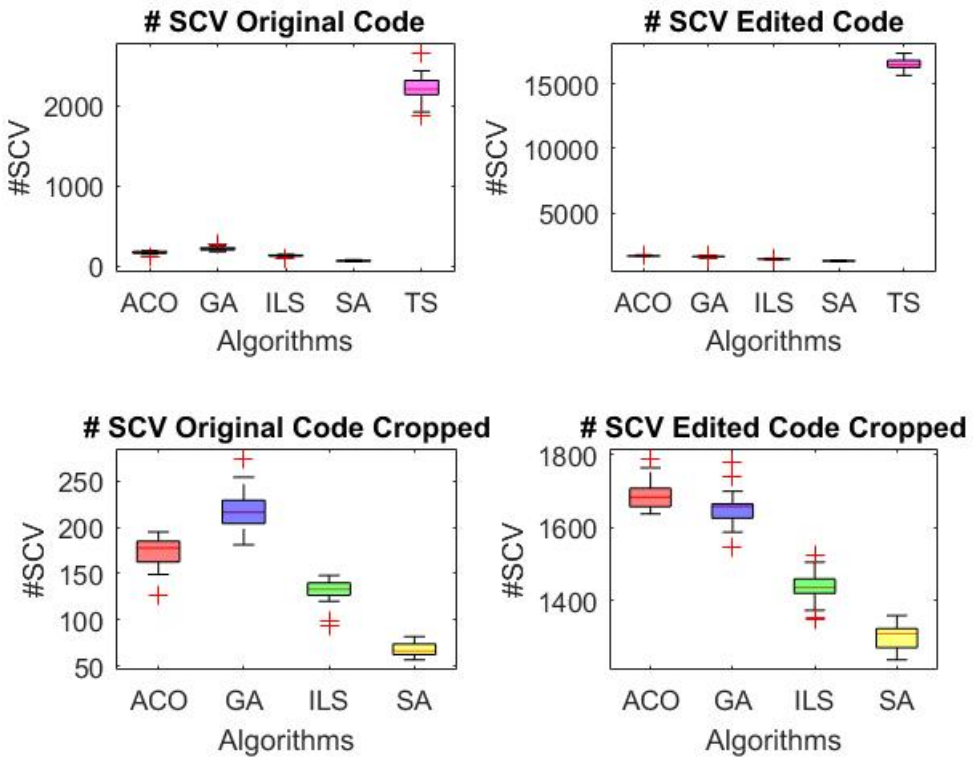

Figure 4.5: Instance: Medium 2

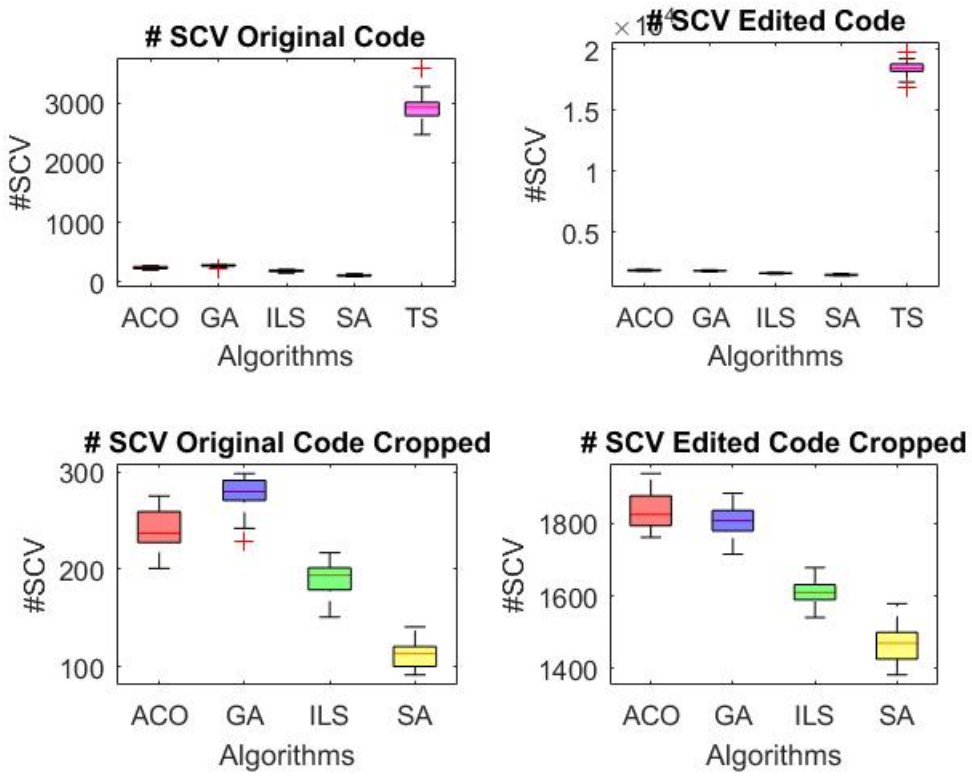

Figure 4.6: Instance: Medium 3

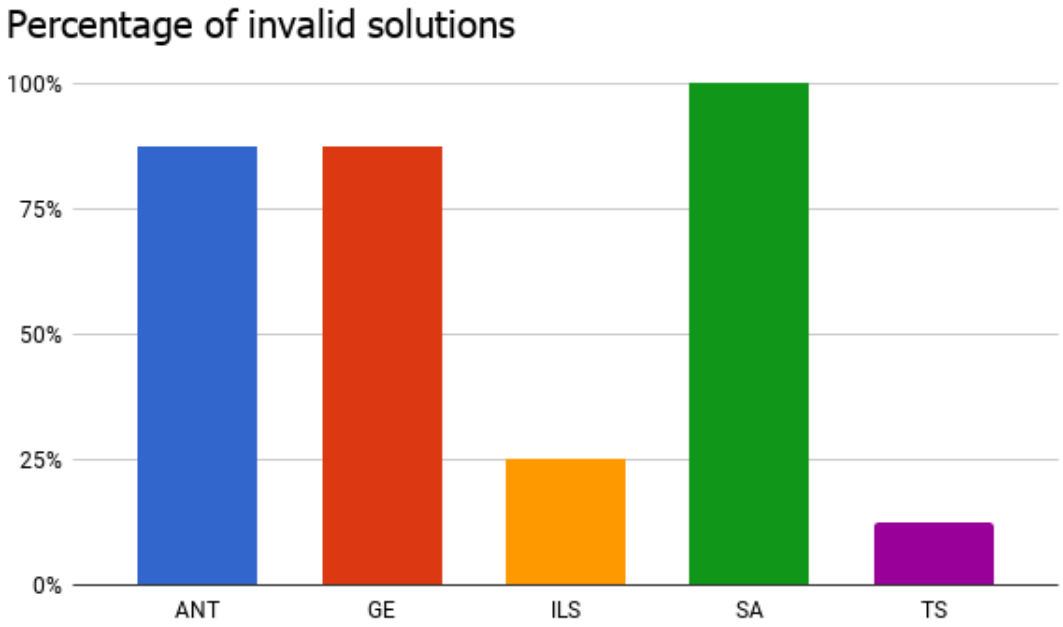

Figure 4.7: Percentage of invalid solutions over all large problem instances

Chapter 5

Discussion

Three main distinctions are made from analyzing the results.

- There is an uprise in average constraint violations across all algorithms when the new constraints are applied.
- The TS is underperforming independently of what set of soft constraints are applied. Worth noting is that it performs best in regards to the hard constraints.
- While the ILS and SA algorithms keeps their rank between all comparisons, ACO and GA tend to perform oppositely, swapping their rank between the old and new set of soft constraints.

5.1 Rising constraint violations

The most obvious reason for the overall increase of constraint violations would be the alterations made on the first soft constraint. The fact that the constraint was extended to also include lectures in the first time slot of the day doubles the risk of a violation on each solution explored. However this is not a satisfying explanation for the great rise in total violations since the total number of violations clearly increased more than by a factor of two.

The less obvious reason for this phenomenon is probably a bi-effect of the change applied at the second soft constraint: *"A student should not have more than two consecutive lectures without a break"* which was effectively replaced with: *"A student should not have breaks between lectures"*.

When a violation is made for each hour of a free period, putting less lectures in a day provides greater risk of violations (unless you only put a single lecture in a day although this is unlikely to happen considering the size of the data sets). To put this more concretely, consider a day consisting of only two lectures. Most of the possible perturbations of two lectures in a day would provide free period violations between the lectures, maximizing when the lectures are put on the first and last time slot of the day where you would get seven violations for the hours between them (not considering the violation to the first soft constraint). On the other hand, each new lecture added to this day would decrease the number of violations.

Now, if you would add two lectures to an empty day based on the original second soft constraint, there is no possibility of this constraint being violated (you would need at least three lectures to get more than two consecutive lectures on a day) and if you add another lecture, there is still a smaller chance of the lecture being put in a violating time slot than in a non-violating one. However, when you start adding even more lectures there will be more violations to this constraint.

This pattern shows that the original constraint is more effective when aiming for a low amount of lectures per day whilst the new constraint revels in having as many lectures as possible in a day. What this conclusion tells us is that the algorithms used are probably more prone to exploring solution spaces with as few lectures as possible in each day. This makes sense as it would be a safer approach when optimizing the timetable according to the original constraints. This would mean that the algorithms need substantially more time to explore until they start considering solutions where the days are filled with more lectures than needed. Because of the time limit set on the tests, this would explain the rise of total violations in regards to the new constraints applied.

Although this might seem detrimental to the study, it does not affect the conclusion of this report since the aim is to compare the rank of the algorithms between the set of solutions - not to compare the total number of violations. Still, it is worth mentioning that if you were to actually implement these algorithms according to these specific new constraints in practice, you might want to alter the heuristics of the algorithms for a more sustainable result.

5.2 The poor performance of TS

Here we discuss hypotheses that could explain the lack of TS performance on the small and medium problem instances.

One hypotheses is that it is simply an overall weak performing algorithm for the UCTP. This conclusion seems quite unrealistic, considering the fact that it was the top performing algorithm for finding a feasible solution to the large problem instances.

The second hypotheses concerns the possibility that the developers of this particular TS code implemented it poorly. This also seems less likely since they were accepted as contestants for the International Timetabling Competition in PATAT 2002, and they are also well-educated with a respectable history [12].

The third hypotheses is that TS excels when running for a long period of time, having stronger potential when crunching through few, but complex, hard constraints violations while being weaker when having to iterate over a big amount of easier soft constraint violations.

5.3 ACO and GA swap

The reason behind the ACO and GA swapping places when constraint alterations are made is hard to define. The complexity of both these algorithms make it very difficult to pinpoint exactly what makes them behave this way as an effect to the altered constraints. The closest thing to a conclusion that can be made is that it has to do with some specific parts of the heuristics of the algorithms, making them more prone to optimize the schedule in a certain manner that fits some criteria better than others. In this case, the constraint alteration makes the GA find optimal paths toward better solutions easier, while the ACO algorithm needs to make more iterating moves before finding solutions fitting the constraints applied.

5.4 ILS and SA fixed rank

The ILS and SA rank stayed the same through all the tests. This goes to show that these two algorithms are reliable and consistent even when substantial changes are made to the constraints. Worth mentioning

however is that ILS outperformed SA regarding the large problem instances, where SA never managed to get a feasible solution. ILS might therefore be preferred for future work and implementation when new constraints are applied and large problem instances are likely to be used.

Chapter 6

Conclusion

The result of this study shows that previous research stating what algorithms are best for the UCTP problem cannot be fully relied upon unless you want to use close to identical sets of constraints. There is a difference in the rank of performance between the algorithms when constraints are changed to fit a certain need, and this should be underlined when interpreting results from other sources and research projects. The difference is not great enough for the worst performing algorithms to become the best performing ones, but it is still quite substantial. This concludes that, for further research and implementations of altering constraints for the UCTP, one should not rely on a single great algorithm for achieving the best possible solution but instead test a few of the best performing ones to reach a satisfactory result for that specific case.

Conclusively: the algorithms do *not* maintain their rank when new constraints are applied.

Bibliography

[1] Vijay Arya et al. "Local search heuristics for k-median and facility location problems". *SIAM Journal on computing* 33.3 (2004), pp. 544–562.

[2] Stephen P Brooks and Byron JT Morgan. "Optimization using simulated annealing". *The Statistician* (1995), pp. 241–257.

[3] Edmund K Burke, Graham Kendall, and Eric Soubeiga. "A tabu-search hyperheuristic for timetabling and rostering". *Journal of heuristics* 9.6 (2003), pp. 451–470.

[4] Edmund K Burke et al. "A graph-based hyper-heuristic for educational timetabling problems". *European Journal of Operational Research* 176.1 (2007), pp. 177–192.

[5] Juan De Vicente, Juan Lanchares, and Román Hermida. "Placement by thermodynamic simulated annealing". *Physics Letters A* 317.5-6 (2003), pp. 415–423.

[6] Shimon Even, Alon Itai, and Adi Shamir. "On the complexity of time table and multi-commodity flow problems". *Foundations of Computer Science, 1975., 16th Annual Symposium on*. IEEE. 1975, pp. 184–193.

[7] Michel Gendreau and Jean-Yves Potvin. *Handbook of metaheuristics*. Vol. 2. Springer, 2010.

[8] Fred Glover. "Tabu search: A tutorial". *Interfaces* 20.4 (1990), pp. 74–94.

[9] Sadaf Naseem Jat and Shengxiang Yang. "A Guided Search Genetic Algorithm for the University Course Timetabling Problem". *Proceedings of the 4th Multidisciplinary International Scheduling Conference: Theory and Applications (MISTA 2009), 10-12 Aug 2009, Dublin, Ireland*. Ed. by J. Blazewicz et al. 2009, pp. 180–191.

[10] Jeffrey H Kingston. "Educational timetabling". *Automated Scheduling and Planning*. Ed. by A. Sima Uyar, Ender Ozcan, and Neil Urquhart. Springer, 2013, pp. 91–108.

[11] *Practice and Theory of Automated Timetabling (PATAT)*. `http://patatconference.org/`. Accessed: 2018-03-25.

[12] Olivia Rossi-Doria et al. "A comparison of the performance of different metaheuristics on the timetabling problem". *International Conference on the Practice and Theory of Automated Timetabling*. Springer. 2002, pp. 329–351.

[13] Olivia Rossi-Doria et al. "A local search for the timetabling problem". *Proceedings of the 4th International Conference on the Practice and Theory of Automated Timetabling, PATAT*. 2002, pp. 124–127.

[14] Nasser R Sabar et al. "A honey-bee mating optimization algorithm for educational timetabling problems". *European Journal of Operational Research* 216.3 (2012), pp. 533–543.

[15] Krzysztof Socha, Joshua Knowles, and Michael Sampels. "A max-min ant system for the university course timetabling problem". *International Workshop on Ant Algorithms*. Springer. 2002, pp. 1–13.

[16] Krzysztof Socha, Michael Sampels, and Max Manfrin. "Ant algorithms for the university course timetabling problem with regard to the state-of-the-art". *Workshops on Applications of Evolutionary Computation*. Springer. 2003, pp. 334–345.

[17] Xin-She Yang. "Metaheuristic Optimization: Nature-Inspired Algorithms and Applications". Springer Berlin Heidelberg, 2013.

Appendix A

Github

For full code and results see our github:
https://github.com/robban96b/RBerggrenTNielsenKex18