The timetabling problem for the University of Bristol

Conor Newton

November 5, 2019

Abstract

This project took place in the summer of 2019 at the University of Bristol. It was supervised by Dr. Ayalvadi Ganesh and was funded by the ESPRC.

1 Introduction

We implement some algorithms and compare the results on the university of bristol data set. The source code (C++) can be found here https://github.com/conornewton/timetabling-project.

Our approach to the timetabling problem proceeds in two stages. In the first stage we use a randomized backtracking algorithm to build an initial solution satisfying the hard constraints. In the second stage we use a meta-heuristic to search the solution space and minimize soft constraint violations. This two-stage optimization approach is fairly standard for solving timetable problems.

1.1 Course structure at the University of Bristol

Each student at the University of Bristol is enrolled onto a set of **Modules**. For example, a first year Mathematics student will be enrolled on *Real Analysis 1A*, *Real Analysis 1B*, *Linear Algebra & Geometry 1* etc. Each of these modules will be made of up **Activities**. An activity is any single module component, such as a *lecture*, *tutorial*, *lab* etc. Each student and teacher has a set of activities that they are required to attend and we aim to assign each activity a room and timeslot. We ignore the concept of a module and just deal with the activities throughout.

Some teachers are not available to teach all of the time, and some teachers require a full day off for research, these are called pathway-one teachers.

For each activity will we consider a set of preferred rooms. Each of these rooms should be large enough and satisfy equipment requirements. This allows us to choose any room from this set.

The academic year at the University of Bristol is split in into 4 parts, TB1A, TB1B, TB2C, TB2D. We treat each part as its own timetabling problem.

1.2 Statistics

To give an understanding of the scale of this problem, we give some information about the data set from TB1A.

There are:

- 4164 activities
- 18154 students
- 2669 teachers
- 692 classrooms

This considers all of the undergraduate and taught postgraduate courses.

2 Definitions

2.1 Problem Formulation

We define some sets and notation that will be used throughout

Symbols	Descriptions
$t = 45$	The number of timeslots available in a week
A	The set of activities $A = \{a_1, \ldots, a_a\}$
A_e	The set of activities $A_e = A \cup \{e\}$ where e is the empty activity
D	The set of departments $D = \{d_1, \ldots, d_d\}$
M	The set of modules $M = \{m_1, \ldots, m_m\}$
R	The set of rooms $R = \{r_1, \ldots, r_r\}$
S	The set of students $S = \{s_1, \ldots, s_s\}$
T	The set of teachers $T = \{t_1, \ldots, t_t\}$
$\text{students}(a)$	The set of students participating in activity $a \in A$
$\text{staff}(a)$	The set of teachers participating in activity $a \in A$
$\text{timeslot}(a)$	The timeslot that activity $a \in A$ is taking place.
$\text{room}(a)$	The room that activity $a \in A$ is taking place
$\text{preferred\_rooms}(a)$	The set of preferred rooms for the activity $a \in A$.
$\text{hours}(a)$	The number of consecutive hours an activity takes up.
$\text{capacity}(r)$	The capacity of the room $r \in R$
$\text{dept}(x)$	The department the room $x \in R$, module $x \in M$ or activity $x \in A$ belongs to
$\text{unavailable}(t)$	The set of hours the teacher $t \in T$ is unavailable to teach.

We define some matrices that help with constraint checking:

Matrices	Descriptions
$T \in M_{r \times t}(A_e)$	The room/timeslot timetable matrix
$ST \in M_{s \times t}(A_e)$	The student/timeslot timetable matrix
$SC \in M_{a \times a}(\mathbb{Z})$	The soft-clash matrix
$HC \in M_{a \times a}(\{0, 1\})$	The hard-clash matrix

2.2 Constraints

In our problem we have two types of constraints, *Hard* and *Soft*. We say that a solution is *feasible* if it satisfies all of the hard constraints. We aim to find a solution that is feasible but also minimizes the number of soft constraints violated.

2.2.1 Hard Constraints

We define and label our hard constraints as follows:

H_1 - Each activity should be scheduled in a distinct room and timeslot pair.

$$\forall a \in A, \text{room}(a) \neq -1 \wedge \text{timeslot}(a) \neq -1$$
$$\forall a, b \in A, \text{room}(a) \neq \text{room}(b) \vee \text{timeslot}(a) \neq \text{timeslot}(b)$$

H_2 - The timeslots chosen must satisfy the lecturers time constraints.

$$\forall a \in A, \forall s \in \text{staff}(a), \text{timeslot}(a) \notin \text{unavailable}(s)$$

H_3 - Each pathway one lecturer should have a full day off.

$$\forall s \in S, \text{pathway\_one}(s) \implies \exists d \in \{0,1,2,3,4\}, \forall a \in \mathit{activities}(s), \text{day}(a) \neq d$$

H_4 - There are no hard clashes.

$$\forall a, b \in A, HC_{a,b} = 1 \implies \text{timeslot}(a) \neq \text{timeslot}(b)$$

H_5 - Activities should not be scheduled for Wednesday afternoon.

$$\forall a \in A, \text{timeslot}(a) \leq 23 \vee \text{timeslot}(a) \geq 27$$

H_6 - Each activity should take place in a preferred room.

$$\forall a \in A, \text{room}(a) \in \text{preferred\_rooms}(a)$$

2.2.2 Soft Constraints

We now define and label our soft constraints. For each soft constraint S_i there will be an associated score function f_i.

S_1 - Minimize Soft Clashes

$$f_1(a,b) = \chi\{\text{timeslot}(a) = \text{timeslot}(b)\} \cdot SC_{a,b}$$

S_2 - A Student should not have more than three hours of lessons in a row without a break.

$$f_2(s) = \sum_{t=0}^{42} \chi\{ST_{s,t} \neq e \wedge ST_{s,t+1} \neq e \wedge ST_{s,t+2} \neq e\}$$

S_3 - The capacity of a chosen room must be greater than the number of students taking a course.

$$f_3(a) = \begin{cases} \#\,\text{students}(a) - \text{capacity}(\text{room}(a)), & \text{if } \#\,\text{students}(a) > \text{capacity}(\text{room}(a)) \\ 0, & \text{otherwise} \end{cases}$$

3 Stage 1 - Backtracking

We generate initial solutions for our problem using a simple backtracking algorithm. It will build a solution satisfying the hard constraints if one exists.

We begin by defining the 'bad_timeslots' variable. This keeps track of timeslots that should not be considered when choosing a new timeslot for a course. This gets populated when a chosen timeslot does not

The main function of the algorithm goes as follows:

Backtrack Algorithm

```
FUNCTION Backtrack(timetable, activities)
    VAR current_activity := 0
    VAR bad_timeslots[][] := {}

    WHILE (current_activity < activities.size)
        VAR timeslot := next_timeslot()
        VAR room := next_room(timeslot)

        IF timeslot == -1
            IF current_activity == 0
                RETURN FALSE
            END IF
            # BACKTRACK HERE!!!
            bad_timeslots[current_activity].clear()
            current_activity := current_activity - 1

            timetable[activities[current_activity].time, ]

        ELSE
            # HERE we move onto the next course
            timetable[ts][room] := current_course
            activities[current_activity].room := room
            activities[current_activity].ts := ts

            current_activity := current_activity + 1
        END IF
    END WHILE

    RETURN TRUE
END FUNCTION
```

4 Stage 2 - Optimization

Generally the optimization stage consists of three main components:

- Objective function - This gives a numerical score to a particular timetable by considering the number of soft constraint violations. The lower the score suggests a solution is better.
- Neighbourhood operators - These allow us to modify our solutions whilst maintaining feasibility.
- Discrete Optimization Algorithm - This will make use of objective function and neighbourhood operators to search for an improved solution given an initial solution.

We will now define our specific implementations of our three components.

4.1 Objective function

The purpose of an objective function is to judge the quality of a feasible solution. We do this by consider considering the number of constraint violations for a particular solution. Furthermore, some constraints may be more important than other's. So, we give each soft constraint an integer weight w_1, w_2, w_3. Our objective function is simply defined as follows:

$$\text{objective}(X) = w_1 \cdot f_1 + w_2 \cdot f_2 + w_3 \cdot f_3$$

Our experimental weighting values were:

4.1.1 Blame function

To effectively choose the swaps we should perform on our current solution, we make use of a blame function. This will the advise us on the activities that are involved in violate constraints the most. We can define the blame operator for each activity $a \in A$ as follows:

$$\text{blame}(a) = w_1 \cdot \sum_{b \in A} f_1(a, b) + w_2 \cdot \sum_{s \in \text{students}(a)} f_2(s) + w_3 \cdot f_3(a)$$

It is also clear that

$$\sum_{a \in A} \text{blame}(a) = \text{objective}(X)$$

We can then choose activities according to their score and apply the swaps appropriately.

4.2 Neighbourhood operators

We will make use of two neighbourhood operators, simple swaps and kempe swaps. They are most common and often the most effective way of improving timetabling solutions.

4.2.1 Simple Swaps

To perform a simple swap we choose an activity $a \in A$ and a random timeslot-room pair $t \in \{0, \dots 44\}, r \in R$. If $T_{r,t} \neq e$ we swap the timeslots and rooms of a and $T_{r,t}$. Otherwise, we choose t, r as a's new time and location. We only perform this swap if our new solution is feasible.

4.2.2 Kempe Swaps

To perform a kempe swap....

4.3 Simulated Annealing

The simulated annealing algorithm produces a new solution by taking a guided random walk around the solution space.

We describe our simulated annealing algorithm with pseudocode below:

Simulated Annealing Algorithm

```
FUNCTION SimulatedAnnealing(timetable)
    CONST initial_temp := 100
    CONST final_temp   := 1
    CONST cooling_rate := 0.999

    VAR current_temp := initial_temp
    VAR current_score := Objective(timetable)

    WHILE (current_temp > final_temp)
        VAR swap := GetRandomSimpleSwap()
        VAR new_timetable := ApplySimpleSwap(timetable, swap)
        VAR new_score := Objective(new_timetable)

        IF (Acceptance(current_score, new_score, current_temp))
            VAR current_score := new_score
            VAR timetable := new_timetable
        END IF

        current_temp := current_temp * cooling_rate

    END WHILE

    RETURN timetable
END FUNCTION
```

The acceptance function chooses whether we should move to a neighbouring solution. If the randomly chosen neighbouring solution is an improvement (according to the objective function) we will move to it, otherwise we will move to it with a probability of $\exp(\frac{current\_score - new\_score}{temp})$. As the algorithm iterates, the probability of moving to a worse solution decreases, this is known as "cooling".

This particular acceptance function was formulated by Kirkpatrick et al. A pseudocode implementation is given below.

Acceptance function

```
FUNCTION Acceptance(old_score, new_score, temp)
    IF (old score < new_score) RETURN TRUE
    ELSE
        VAR u := Unif(0, 1)
        VAR r := Exp((current_score - new_score)/temp)
        RETURN u < r
    END IF
END FUNCTION
```

4.4 Tabu Search

5 Analysis & Results

References

[1] Hwang KS., Lee K.M., Jeon J. (2004) A Practical Timetabling Algorithm for College Lecture-Timetable Scheduling. In: Negoita M.G., Howlett R.J., Jain L.C. (eds) Knowledge-Based Intelligent Information and Engineering Systems. KES 2004. Lecture Notes in Computer Science, vol 3215. Springer, Berlin, Heidelberg

[2] R. Lewis, J.Thompson (2015) Analysing the effects of solution space connectivity with an effective metaheuristic for the course timetabling problem. European Journal of Operational Research 240, 637-648

[3] Zhipeng Lü, Jin-Kao Hao (2010) Adaptive Tabu Search for course timetabling. European Journal of Operational Research 200, 235-244.