University Timetabling

Curriculum-based Course Timetabling using ALNS and TABU

Andreas Madsen – s123598

April 25. 2016

Contents

1 Introduction **2**

2 Problem Description **3**
2.1 Constraints . . . 3
2.2 Objective . . . 4
2.3 Datasets . . . 5
2.4 Choosing the metaheuristic models . . . 6

3 Metaheuristic description **8**
3.1 Mutatation operations . . . 8
3.2 Greedy initialization . . . 9
3.3 Tabu Search general description . . . 10
3.4 Tabu specialization . . . 11
3.5 ALNS general description . . . 12
3.6 ALNS specialization . . . 14

4 Parameter tuning **16**
4.1 Tabu . . . 16
4.2 ALNS . . . 17

5 Test results **19**

6 Conclusion **20**

A Notaton **21**

B Move and swap operation **22**

C Code ReadMe **24**

1 Introduction

The curriculum-based university timetabling problem is about assigning courses, which are associated to a set of curriculums, to a schedule. Each slot in this schedule consists of a room and a time, where the time is discretized into a given number of days and periods per day. Each course are to be scheduled a given number of times.

The curriculum part of the problem, is about making sure a student who attend a given curriculum can take all courses, meaning there are no overlapping courses within the curriculum. There is also an optimization component, which prefers schedule where there are no holes for a given curriculum during a day (called curriculum compactness).

For solving this problem a meta heuristic approach is taken. Tabu Search and ALNS will both be used to solve the problem independently and then be compared.

2 Problem Description

The problem is very precisely defined in the project description [1]. The problem will be restated here, but also discussed in terms of how the the delta calculations can be done.

Delta calculations is the change in the objective value when performing a simple operation. In this case only adding or removing a single course from a slot (room and time) is considered. In section 3.1 it will be discussed how other operations can be implemented using these primitives.

The used notation is the same as in the project description [1] and may also be found in appendix A.

2.1 Constraints

It will always be valid to remove a course, thus the following constraints only need to be checked when adding a course to the schedule solution.

2.1.1 Time availability

$$\sum_{r \in R} x_{c,t,r} \leq F_{c,t} \quad \forall c \in C, t \in T \tag{2.1}$$

A course may only be assigned once in the same time slot. The dataset also contains a set of unavailable time slots for each course. For unavailable time slots $F_{c,t} = 0$ otherwise $F_{c,t} = 1$.

Delta: This constraint is really two separate constraints. First maintain an indicator that is true if the given course is assigned to the given time. Secondly one can use a lookup table to check if the time slot is defined unavailable.

2.1.2 Room availability

$$\sum_{c \in C} x_{c,t,r} \leq 1 \quad \forall t \in T, r \in R \tag{2.2}$$

A slot consisting of room and time may only be used once.

Delta: Maintain an indicator that is true if the slot (time and room) is used.

2.1.3 Max lectures

$$\sum_{t \in T, r \in R} x_{c,t,r} \leq L_c \quad \forall c \in C \tag{2.3}$$

Prevent assigning more slots to a course than what is needed.

Delta: Maintain a counter of how many times a given course is assigned.

2.1.4 Conflicting courses

$$\sum_{r \in R} x_{c_1,t,r} + \sum_{r \in R} x_{c_2,t,r} \leq 1 \quad \forall c_1, c_2 \in C, t \in T, \chi(c_1, c_2) = 1 \tag{2.4}$$

Conflicting courses may not be assigned the same time slot. Courses are conflicting if taught by the same lecture or part of the same curriculum.

Delta: Pre-calculate a list of conflicting courses for each course and maintain a list of courses assigned to each time slot. Then check that there are no conflicting courses assigned to the given time slot.

2.2 Objective

There are 5 objectives which are combined to a single objective score:

$$\text{Objective} = 10U + 5W + 2A + 1P + 1V \tag{2.5}$$

The delta calculations for the add and remove operations are mostly the same, just with the sign flipped. For P and W there are minor differences. Furthermore it can be assumed that the constrains are met, as those have already been checked.

2.2.1 Room capacity (V)

$$V = \sum_{t \in T, r \in R} V_{t,r}(x), \quad V_{t,r} = \max\left\{0, \sum_{c \in C} S_c \cdot x_{c,t,r} - C_r\right\} \tag{2.6}$$

A course has an associated number of students S_c and a room has an associated capacity C_r. V is the penalty for exceeding this capacity.

Delta: One can simply lookup S_c and C_r for a given $(course, time, room)$ combination and calculate V.

2.2.2 Unscheduled (U)

$$U = \sum_{c \in C} U_c(x), \quad U_c(x) = \max\left\{0, L_c - \sum_{t \in T, r \in R} x_{c,t,r}\right\} \tag{2.7}$$

A course has to be scheduled a given number of times L_c. Anything less than this is penalized.

Delta: Since the new solution is valid, one can simply subtract or add one to the U sum.

2.2.3 Room stability (P)

$$P = \sum_{c \in C} P_c(x), \quad P_c(x) = \max \left\{ 0, \left\| \left\{ r \in R \middle| \sum_{t \in T} x_{c,t,r} \geq 1 \right\} \right\| - 1 \right\} \tag{2.8}$$

A course should be assigned to as few different rooms as possible.

Delta: On addition check if r is a new room for the course. On remove check if room r is used only once in the current solution.

2.2.4 Minimum working days (W)

$$W = \sum_{c \in C} W_c(x), \quad W_c(x) = \max \left\{ 0, M_c - \left\| \left\{ d \in D \middle| \sum_{t \in T(d), r \in R} x_{c,t,r} \geq 1 \right\} \right\| \right\}$$

A course should be assigned time slots over M_c different days.

Delta: Count the number of days the course is spread over. On addition check if this count is less than M_c and check that the course isn't already scheduled this day. On remove check that the count isn't greater than M_c and that the course is scheduled only once this day.

2.2.5 Curriculum compactness (A)

$$A = \sum_{q \in Q, t \in T} A_{q,t}(x), \quad A_{q,t}(x) = \begin{cases} 1 & \text{if } \sum_{c \in C(q), r \in R} x_{c,t,r} = 1 \wedge \sum_{\substack{c \in C(q), r \in R, \\ t' \in T, \Upsilon(t,t')=1}} x_{c,t',r} = 0 \\ 0 & \text{otherwise} \end{cases}$$

Penalizes gaps/holes over a day in a curriculum.

Delta: This is the most complicated objective in terms of delta calculation, because one also need to adjust the penalty for adjacent courses in the same curriculum. Thus if the course before was penalized then remove that penalty, similarly for the course after. If there are no courses before and after add a penalty.

2.3 Datasets

There are 12 datasets, each is given as 7 files:

- `basic.utt` - Contains basic meta information about the problem. Number of courses, rooms, days, periods pr day, etc.
- `courses.utt` - Contains information about each course.
- `curricula.utt` - Contains the number of courses associated to each curriculum. This file was not used, as the information could be inferred from `relation.utt`.

- `lecturers.utt` - Contains a list of lecturers. This file was not used, as the information could be inferred from `courses.utt`.
- `relation.utt` - Relational table that binds curriculum and course.
- `rooms.utt` - Contains information about each room.
- `unavailability.utt` - Contains the list of unavailable time slots for each course.

Using `basic.utt` the 13 datasets can be summarized:

dataset	Courses	Rooms	Days	Periods per day	Curricula	Constraints	Lecturers
1	30	6	5	6	14	53	24
2	82	16	5	5	70	513	71
3	72	16	5	5	68	382	61
4	79	18	5	5	57	396	70
5	54	9	6	6	139	771	47
6	108	18	5	5	70	632	87
7	131	20	5	5	77	667	99
8	86	18	5	5	61	478	76
9	76	18	5	5	75	405	68
10	115	18	5	5	67	694	88
11	30	5	5	9	13	94	24
12	88	11	6	6	150	1368	74
13	82	19	5	5	66	468	77

Table 1: Information from `basic.utt` for each dataset.

2.4 Choosing the metaheuristic models

As seen the problem is highly constrained. ALNS is usually good for highly constrained, thus ALNS is an obvious choice.

The objective function is not multi objective, but do consist of many separate sub-objective functions. This suggest Evolutionary Algorithms may be a good choice, as two good solution may minimize two different sub-objectives, thus a crossover between the two solutions could minimize both and become a very good solution. However because the problem is so tightly constrained, it will be difficult to come up with a good crossover algorithm.

GRAPS is likely a poor choice, as the many constraints makes it difficult to come up with many good greedy algorithms.

The neighborhood of a given solution is at least, all possible $(course, time, room)$ additions and removal operations. However because of the big penalty (+10) of removing a $(course, time, room)$ combination, this is unlikely to ever be an immediate good choice. Thus it make sense to expand the neighborhood to include all possible moves to an available slot and all possible swaps between two $(course, time, room)$ combinations. This expanded neighborhood is huge, thus Tabu Search may also be a good choice.

2.4.1 Conclusion

ALNS is chosen because of the many constraints. Tabu Search is chosen because of the large neighborhood. Evolutionary Algorithms is an interesting choice and may be worth investigating, however that is out of the scope in this project.

3 Metaheuristic description

3.1 Mutatation operations

The solution schedule is defined as list of $(c \in C, t \in T, r \in R)$. Which means course c is assigned to a slot given by the time t and room r.

In this project 4 move operations are used. The operations are:

- `Add`(c, t, r) - adds course c to the slot (t, r).
- `Remove`(c, t, r) - remove course c from the schedule slot (t, r).
- `Move`(c, t_0, r_0, t_1, r_1) - moves course c from (t_0, r_0) to (t_1, r_1).
- `Swap`$(c_0, t_0, r_0, c_1, t_1, r_1)$ - course c_0 and course c_1 swap slots.

To avoid copying the solution object, it should be possible to calculate the Δ cost without actually changing the solution object. Those functions are prefixed with `Simulate`, while the functions that actually changes the solution object are prefixed with `Mutate`.

Algorithm 1 Add a course c to slot (t, r)

```
1 function SimulateAdd(c, t, r)
2     if not ValidAdd(c, t, r) then          ▷ Discussed in section 2.1
3         return None
4     return CostAdd(c, t, r)                ▷ Discussed in section 2.2

5 function MutateAdd(c, t, r)
6     Δ ← SimulateAdd(c, t, r)
7     if Δ ≠ None then
8         Objective ← Objective + Δ
9         update data structures             ▷ Discussed in section 2.1 and 2.2
```

Algorithm 2 Remove a course c from slot (t, r)

```
1 function SimulateRemove(c, t, r)
2     if not ValidRemove(c, t, r) then       ▷ Discussed in section 2.1
3         return None
4     return CostRemove(c, t, r)             ▷ Discussed in section 2.2

5 function MutateRemove(c, t, r)
6     Δ ← SimulateRemove(c, t, r)
7     if Δ ≠ None then
8         Objective ← Objective + Δ
9         update data structures             ▷ Discussed in section 2.1 and 2.2
```

Note that one typically simulates the operation to check that $\Delta < 0$ and then mutate the solution object. Thus it makes sense to add Δ as an optional argu-

ment to the mutate functions, to avoid unnecessary calculations. This was done in the implementation but is for simplicity excluded here.

The `Move` and `Swap` operations can then be implemented, using using the `Add` and `Remove` primitives. This is not the most efficient implementation as it requires mutation of the current solution, which are then reverted in the simulation case. However given the time constraints of the project, more efficient implementations wasn't made. See appendix B for how `Move` and `Swap` was implemented using the primitives.

3.2 Greedy initialization

For this particular problem a schedule with no courses scheduled is a valid solution. Thus a special initialization procedure is not necessary. However initializing using empty solution will cause ALNS and in particular Tabu Search to end up spending many iterations doing simple course additions. This is wasteful as the fixed optimization time could be spend much more productive.

Thus a simple randomized greedy initialization procedure is used to quickly go from the empty solution to a much more optimal solution.

Algorithm 3 Performs a greedy optimization of a solution

```
1  function GreedyInitialization(solution)
2      ▷ Create list of missing courses in random order
3      courses ← list()                          ▷ List with fast push operation
4      for all (c, n) in MissingCourses(solution) do
5          Push(c on courses), n times
6      courses ← Shuffle(courses)
7
8      ▷ Create list of available slots
9      slots ← AvaliableSlots(solution)
10     slots ← Shuffle(slots)
11     slots ← Deque(slots)                              ▷ Double-ended queue
12
13     for all c in courses do
14         for i from 1 to Length(slots) do
15             (t, r) ← PopRight(slots)
16
17             ▷ add (c, t, r) to solution if it improves the objective
18             Δ ← SimulateAdd(c, t, r)
19             if Δ < 0 then
20                 MutateAdd(c, t, r)                                ▷ Use slot
21                 break
22             else
23                 PushLeft((t, r) on slots)          ▷ The slot is still avaliable
```

The algorithm tries to add all missing courses to the schedule, but it will only add the course if it improves the solution in the current situation.

3.3 Tabu Search general description

Tabu search performs local search using a neighborhood definition, it then picks the best new solution. This process is repeated in each iteration.

Doing just the local search will cause the search algorithm to each a minimum, unfortunately this minimum is very likely to only be a local minima. To escape a local minima Tabu Search performs non-optimal moves once a minima is reached, this process is called diversification. This can however cause a cycling behavior, where the opposite or same moves are just applied again, thus reaching the same local minima.

Tabu search attempts to solve the cycling problem by maintaining a tabu list. The tabu list contains all temporarily illegal solutions. Once an optimal change is applied the new solution is added to the tabu list. When performing a local search, the tabu list is checked before the change is attempted, this prevents one from reaching the same solution twice.

Storing all previous solutions in a tabu list is not very efficient, both in terms of space and the computational resources required for checking if two solutions are equal. Instead the neighborhood is defined by a set of moves. Once a move is applied to the solution the opposite move is added to the tabu list. By using moves one only needs to store the individual moves and check if two moves are equal, this is much more efficient.

Finally only typically also applies intensification to the Tabu Search algorithm. Intensification prevent the algorithm from diversifying too much. Intensification can be anything from take core good components from the globally best solution to just restoring the globally best solution. Usually intensification also involves resetting the tabu list.

Algorithm 4 Generalization of the Tabu search algorithm

```
1  function TabuSearch(solution_init)
2      s_global ← solution_init                          ▷ Globally best solution
3      s_local ← solution_init                           ▷ Current solution
4      tabu ← LimitedSet()                               ▷ May have infinite space
5
6      repeat
7          Δ, move ← LocalSearch(s_local, tabu)          ▷ Find best move ∉ tabu
8          if Δ < 0 then
9              s_local ← Apply(move on s_local)
10             Add(Opposite(move) on tabu)
11         else
12             if s_global hasn't been updated for awhile then
13                 s_local ← Intensify(s_global)         ▷ Intensification is optional
14             s_local ← Divserify(s_local)              ▷ Diversification is optional
15         if Cost(s_local) < Cost(s_global) then
16             s_global ← s_local
17     until no more time
18     return s_global
```

The Tabu Search algorithm have the following parameters:

name	type	description
diversification	boolean	Should diversification be used.
intensification	integer	How many iterations without a globally better solution is allowed, before the intensification procedure is used. One may choose ∞ to disable intensification.
tabu limit	integer	As new moves are added to the tabu list, old moves may be removed from the list. This parameter controls how many moves the tabu list should remember.

Table 2: Parameters for generalized Tabu search

3.4 Tabu specialization

The local search scans the neighborhood consisting of:

- All possible additions of a course to an available slot.
- All possible removals of a $(course, time, room)$ combination from the schedule.
- All possible moves of an existing $(course, time, room)$ combination to an available slot.
- All possible swaps between two $(course, time, room)$ combinations.

The swap operation is in particularly expensive, on a typical run without using the swap operation, the tabu search will spend 0.6 to 0.8 seconds per iteration. Adding the swap operation adds an additional 2.5 to 3.3 seconds per iterations. Adding swap moves to the neighborhood thus introduces a tradeoff between having a large neighborhood and performing many iterations. To optimize this tradeoff a parameter was added.

Diversification is done by removing random $(course, time, room)$ combinations from the schedule. How may there should be removed is controlled by a parameter.

Intensification is done by simply restoring the globally best solution and resetting the tabu lists.

The Tabu Search have just a single tabu list. However because the neighborhood is expressed using different move operations, individual tabu lists for each operation where used. This was done for implementation simplicity, joining the tabu lists is completely possible and given more time one should do this.

name	type	description
diversification	integer	How many $(course, time, room)$ combinations should be removed. May be zero to disable diversification.
intensification	integer	Same as in the generalized tabu search.
tabu limit	integer	This parameter controls the tabu limit of all the tabu lists.
allow swap	{always, dynamic, never}	If *always* the swap neighborhood is always checked. If *never* the swap neighborhood is never checked. Additionally if *dynamic*, the swap neighborhood is only checked if none of the other operations could reduce the objective.

Table 3: Parameters for specialized Tabu search

3.5 ALNS general description

LNS is an search algorithm that uses a destroy method and a repair method. The destroy method will remove a part of the current solution. The destroy method in it self is unlikely to improve the objective value, thus it is followed by a repair method that will search for a better solution. The destroy and repair methods are typically stochastic.

The idea is that by removing parts of the solution without expecting a better solution, the search neighborhood becomes very large without being computationally expensive. This will of cause require that the repair method to do more good than the destroy method does harm, at least on average. However the performance of the destroy and search method will most likely depend on the specific dataset. ALNS generalizes LNS such that more than one destroy and one repair algorithm can be used. ALNS will then dynamically choice the best destroy and repair algorithms for the specific dataset.

ALNS choices the repair and destroy method by updating the probability of selecting the different repair and destroy methods. In each iteration this probability is used to randomly select the method.

The probability update equations requires the following model parameters:

name	type	description
λ	ratio $\in [0, 1]$	remember parameter used in the moving average update of the probabilities.
w_{global}	positive integer	reward for a globally better solution
$w_{current}$	positive integer	reword for a locally better solution
w_{accept}	positive integer	reward for accepting the new solution
w_{reject}	positive integer	*reward* for rejecting the new solution

Table 4: Parameters for generalized ALNS search

First the reward is calculated as $\Psi = \max\{w_{global}, w_{current}, w_{accept}, w_{reject}\}$

where the w values are zero if the corresponding reward condition weren't met. If the selected destroy method has index d and the repair method index r, the preference values are then updated as:

$$p_d^- = \lambda p_d^- + (1 - \lambda)\Psi \tag{3.1}$$
$$p_r^+ = \lambda p_r^+ + (1 - \lambda)\Psi \tag{3.2}$$

Here p^- are the preference values for destroy and p^+ are for repair.

To convert the preference values to actual probabilities, simply scale by the sum:

$$\phi_d^- = \frac{p_d^-}{\sum_i p_i^-} \tag{3.3}$$
$$\phi_r^+ = \frac{p_r^+}{\sum_i p_i^+} \tag{3.4}$$

To sample from these distributions, can use compare the commutative sum with a uniformly random number between 0 and 1.

Algorithm 5 Samples using the preference values p

```
function SampleFunction(p)
    φ ← Scale(p)
    c ← CumSum(φ)
    rand ← UniformRandom(0, 1)
    return Bisect(c, rand)          ▷ Finds first index where rand < c_i
```

The entire algorithm can now be stated as:

Algorithm 6 Generalization of the ALNS search algorithm

```
function AlnsSearch(solution_init)
    s_global ← solution_init                    ▷ Globally best solution
    s_local ← solution_init                     ▷ Current solution
    p⁺, p⁻ ← vector of 1s

    repeat
        d ← SampleFunction(p⁻)
        r ← SampleFunction(p⁺)
        s_local ← Repair(Destroy(s_local, d), r)

        Ψ ← max{w_global, w_current, w_accept, w_reject}
        p_d⁻ ← λ p_d⁻ + (1 − λ)Ψ
        p_d⁺ ← λ p_d⁺ + (1 − λ)Ψ

        if Cost(s_local) < Cost(s_global) then
            s_global ← s_local
    until no more time
    return s_global
```

3.6 ALNS specialization

The problem specific ALNS implementation is almost identical to the generalized ALNS. The destroy methods remove (course, time, room) combinations according to specific rules, and the repair methods adds missing courses to the schedule. Because removing a course is always valid and adding courses can be validated continuously, the w_{accept} and w_{reject} parameters serves no purpose, thus the gain simply becomes:

$$\Psi = \max\{w_{global}, w_{current}\} \tag{3.5}$$

The parameters are:

name	type	description
λ	ratio $\in [0, 1]$	remember parameter used in the moving average update of the probabilities.
w_{global}	positive integer	reward for a globally better solution
$w_{current}$	positive integer	reword for a locally better solution
$remove$	positive integer	number of courses removed in each destroy function

Table 5: Parameters for generalized ALNS search

3.6.1 Destroy functions

There are 4 destroy functions, they all remove a given number ($remove$) of courses using some strategy.

Algorithm 7 remove random (course, time, room) combinations from the solution

```
1 function DestroyFullyRandom(solution)
2     for (c, t, r) in UniformSample({(c, t, r)}, remove) do
3         MutateRemove(c, t, r)
```

Algorithm 8 remove random (course, time, room) combinations from a curriculum

```
1 function DestroyCurriculum(solution)
2     q ← UniformSample(Q, 1)
3     for (c, t, r) in UniformSample({(c, t, r) | c ∈ C(q)}, remove) do
4         MutateRemove(c, t, r)
```

Algorithm 9 remove random (course, time, room) combinations from a day

```
1 function DestroyDay(solution)
2     d ← UniformSample(D, 1)
3     for (c, t, r) in UniformSample({(c, t, r) | t ∈ d}, remove) do
4         MutateRemove(c, t, r)
```

Algorithm 10 remove random (course, time, room) combinations where the course is fixed

```
function DestroyCourse(solution)
    c_d ← UniformSample(C, 1)
    for (c, t, r) in UniformSample({(c, t, r) | c = c_d}, remove) do
        MutateRemove(c, t, r)
```

3.6.2 Repair functions

The repair methods attempt to insert all missing courses.

Algorithm 11 picks the first slot for a course that has $\Delta < 0$

```
function VeryGreedyRepair(solution)
    for all (c, missing) in MissingCourses(solution) do
        for all (t, r) in AvaliableSlots(solution) do
            Δ ← SimulateAdd(c, t, r)
            if Δ < 0 then
                MutateAdd(c, t, r)
                missing ← missing − 1
                if missing = 0 then break
```

Algorithm 12 evaluate Δ independently and takes the best for each course

```
function BestPlacementRepair(solution)
    for all (c, missing) in MissingCourses(solution) do
        ▷ Find the best missing slots assuming independent Δ
        slots ← AvaliableSlots(solution)
        best ← MinSort(slots by SimulateAdd(c, t, r), missing)

        for all (t, r) in best do
            Δ ← SimulateAdd(c, t, r)              ▷ revalidate improvement
            if Δ < 0 then
                MutateAdd(c, t, r)
```

An issue with the chosen repair methods is that `BestPlacementRepair` will almost always perform better than `VeryGreedyRepair`. However it also require much more computation time. ALNS does not penalize computation time, thus it will likely often choose `BestPlacementRepair` even if `VeryGreedyRepair` was better because it allowed more iterations.

Also note that the repair methods aren't random. This was done for implementation simplicity. It also improves speed as there no need to generate the full list of missing courses and available slots. Because the destroy methods are random, it is unlikely that the lack of randomness is a big issue. However given more time this would be worth exploring.

4 Parameter tuning

In order to find the best set of parameters for the ALNS and Tabu Search, different parameter combinations was tried (see section 4.1 and 4.2). Each parameter combination was tried 3 times using different random initializations.

Because the problems aren't equally difficult and because the objective value isn't normalized, the objective value for each dataset can't be directly compared. To accommodate this the best objective value for each dataset is used to normalize the objective, this score is called the *gap*:

$$\tilde{z}_i = \frac{z_i - z^*}{z^*} \tag{4.1}$$

Here z_i is the objective value and z^* is the best objective value for the dataset.

Because one wishes to avoid overfitting of the parameters, a subset of the entire dataset is chosen for parameter optimization, this is called the training dataset. As there do not appear to be any pattern in the dataset id, all odd dataset are chosen for parameter optimization.

dataset id	1	3	5	7	9	11	13
Tabu	35	706	896	1390	765	36	794
ALNS	24	211	761	211	200	5	166
both	24	211	761	211	200	5	166

Table 6: Best objective value for each training dataset

4.1 Tabu

A grid search over all 4 parameters (see table 9) is performed. This is 108 (μ, σ) pairs, which is too much data to visualize. Thus tables with two parameters and the remaining fixed to the best parameters are shown instead.

		`intensification`		
		2	10	∞
`diversification`	0	(4.90, 0.52)	(5.36, 0.37)	(5.53, 0.71)
	1	(5.07, 0.97)	(5.34, 0.17)	(5.29, 0.63)
	5	(5.19, 0.41)	(4.14, 0.12)	(5.06, 0.77)

Table 7: Shows (μ, σ) with `allow_swap=dynamic` and `tabu_limit=40` fixed

The choice of diversification and intensification appears to be somewhat important. There don't appear to be any linear trend, it is the combination $diversification = 5$ and $intensification = 10$ that yields good result. This makes sense since intensification reverts the diversification, thus they need to fit together. $diversification$ is on the edge of the grid search, given more time one should investigate this parameter direction further.

		tabu_limit			
		10	20	40	∞
allow_swap	never	(5.53, 0.70)	(5.91, 0.89)	(6.52, 0.60)	(5.93, 0.43)
	always	(9.12, 0.18)	(8.77, 0.26)	(9.16, 0.87)	(8.82, 0.44)
	dynamic	(5.52, 0.69)	(5.10, 0.59)	(4.14, 0.12)	(5.45, 0.27)

Table 8: Shows (μ, σ) with `diversification=5` and `intensification=10` fixed

Using the dynamic swap neighborhood generally outperforms the other options, though for some tabu limits only slightly. However for the correct parameters, the results shows that having a dynamic neighborhood can greatly outperform a fixed neighborhood. This idea is something that could be applied to other problems as well.

The best parameters are chosen solely based on the μ values. This is because the σ values don't vary too much, but the chosen parameters also turns out to have the best σ within the shown subset.

parameter	search space	value
allow swap	$\{\text{never}, \text{always}, \text{dynamic}\}$	dynamic
tabu limit	$\{10, 20, 40, \infty\}$	40
intensification	$\{2, 10, \infty\}$	10
diversification	$\{0, 1, 5\}$	5

Table 9: Best Tabu search parameters with $\mu = 4.139$ and $\sigma = 0.122$

4.2 ALNS

		remove		
		1	3	5
update_lambda	0.9	(0.62, 0.04)	(0.58, 0.02)	(0.99, 0.04)
	0.95	(0.46, 0.13)	(1.09, 0.05)	(1.22, 0.05)
	0.99	(0.35, 0.06)	(1.86, 0.04)	(1.72, 0.15)

Table 10: Shows (μ, σ) with `w_global=10` and `w_current=10` fixed

$remove = 1$ is not unreasonable as it allows for fine tuning by only inviting minor changes. However a swap operation would requires at least $remove = 2$, given more project time this choice should be explored. $\lambda = 0.99$ is also reasonable since the fast destroy and repair function allows for many iterations and preferring a function can have long term effects.

		w_current			
		1	3	5	10
w_global	5	(0.45, 0.15)	(0.55, 0.14)	(0.67, 0.04)	(0.39, 0.09)
	10	(0.42, 0.12)	(0.58, 0.01)	(0.55, 0.06)	(0.35, 0.06)
	20	(0.52, 0.18)	(0.40, 0.11)	(0.43, 0.11)	(0.58, 0.10)

Table 11: Shows (μ, σ) with `update_lambda=0.99` and `remove=1` fixed

It is a bit strange that w_{global} and $w_{current}$ should have the same value. However it is likely that a globally better solution is not reached because of one good choice of a repair and destroy method, but instead a long line of good choices. The at the time chosen repair and destroy functions are thus not a direct contributing factor to the globally better solution.

parameter	search space	value
λ	$\{0.9, 0.95, 0.99\}$	0.99
w_{global}	$\{5, 10, 20\}$	10
$w_{current}$	$\{1, 3, 4, 10\}$	10
remove	$\{1, 3, 5\}$	1

Table 12: Best ALNS parameters with $\mu = 0.3502$ and $\sigma = 0.0594$

5 Test results

The best parameters form the parameter tuning in section 4, are used to test both the Tabu and ALNS model. All odd numbered datasets was used for parameter tuning, thus those datasets shouldn't be used for testing. But for completeness the search algorithms was reapplied on the training set. This time the parameters are fixed thus the number of runs can be increased to 5 runs per dataset.

		Tabu	ALNS
train	1	(3.00, 0.84)	(0.22, 0.18)
	3	(3.07, 0.08)	(0.23, 0.15)
	5	(0.14, 0.04)	(0.07, 0.05)
	7	(6.33, 0.22)	(0.20, 0.11)
	9	(3.19, 0.24)	(0.13, 0.09)
	11	(12.80, 2.45)	(0.80, 0.81)
	13	(3.90, 0.17)	(0.11, 0.11)
test	2	(3.17, 0.27)	(0.09, 0.11)
	4	(4.88, 0.35)	(0.09, 0.08)
	6	(5.06, 0.35)	(0.05, 0.05)
	8	(5.19, 0.43)	(0.12, 0.07)
	10	(6.22, 0.39)	(0.14, 0.07)
	12	(0.92, 0.10)	(0.06, 0.06)
all train		(4.63, 0.46)	(0.25, 0.14)
all test		(4.24, 0.15)	(0.09, 0.04)

Table 13: Test and train results over 5 runs using best parameters

The Tabu mean is a bit too high given the standard deviance calculated in section 4. This appears to be caused by dataset number 11, where the Tabu search performs extremely poorly. But overall the values are close to what one would expect, from the parameter tuning in section 4. The test objective values are actually surprisingly good, as they are a little smaller than those produced by the train datasets.

6 Conclusion

ALNS consistently outperforms Tabu Search. As both algorithms reaches a point where it becomes hard to find a much better solution, this must be because ALNS covers a much broader solution space.

Diversification in Tabu Search is meant to move the search to another part of the solution space. But clearly this is not accomplished sufficiently. This is likely because Tabu Search spends a lot of time just in validating solutions, this is particularly the case with the swap operation. ALNS don't do this, as it just looks at the best placement for missing courses. Improving the performance of the move and swap operations could make a huge difference for Tabu Search. The fact that the dynamic neighborhood consistently outperforms any other neighborhood credits this hypothesis too.

Even if the move and swap operations where made faster, it is possible that the complexity of the problem makes it impossible for those operations to become sufficiently fast. Using a dynamic neighborhood thus still makes sense. In particularly this is something that could be applied to many other problems. It has definitely been shown to make a big improvement.

In parameter tuning for ALNS it was found that using the same $w_{current}$ and w_{global} gave the best results. This is something that is not intuitive at first, but as discussed indicates that a globally better solution is found by a long line of good repair and destroy choices. If one did not have the computational resources to optimize these parameters, setting them to the same value may be a good choice.

To improve the ALNS method further, one could analyze the repair and destroy method selection probabilities to learn about what works and more importantly doesn't work for some datasets. Using this knowledge more fine tuned algorithms could be constructed. However one also have to be careful not to do such must fine tuning that it becomes overfitting.

At last one could explore combining the methods. A simple way of doing this, could be to initialize the Tabu Search using the ALNS solution. This would ensure that the ALNS solution is in a local minima and there isn't some nearby better solution, that can be found by simple diversification.

A Notaton

Notation and text descriptions are from the project description [1].

Sets

C The set of courses

L The set of lecturers

R The set of rooms

Q The set of curricula

T The set of time slots. i.e. all pairs of days and periods

D The set of days

$T(d)$ The set of time slots that belongs to day $d \in D$

$C(q)$ The set of courses that belongs to curriculum $q \in Q$

Parameters

L_c The number of lectures there should be for course $c \in C$

C_r The capacity of room $r \in R$

S_c The number of students attending course c

M_c The minimum number of days that course c should be spread across

$$F_{c,t} = \begin{cases} 1 & \text{if course } c \in C \text{ is available in time slot } t \in T \\ 0 & \text{otherwise} \end{cases}$$

$$\chi(c_1, c_2) = \begin{cases} 1 & \text{if course } c_1 \in C \text{ is different from course } c_2 \in C \ (c_1 \neq c_2) \text{ and conflicting, ie. are taught by the same lecturer or are part of the same curriculum.} \\ 0 & \text{otherwise} \end{cases}$$

$$\Upsilon(t_1, t_2) = \begin{cases} 1 & \text{if time slot } t_1 \text{ and } t_2 \text{ belongs to the same day and are adjacent to each other} \\ 0 & \text{otherwise} \end{cases}$$

Decision Variables

$$x_{c,t,r} = \begin{cases} 1 & \text{if class } c \in C \text{ is allocated to room } r \in R \text{ in time slot } t \in T \\ 0 & \text{otherwise} \end{cases}$$

B Move and swap operation

Algorithm 13 Move a course c from slot (t_0, r_0) to (t_1, t_2)

```
1  function SimulateMove(c, t_0, r_0, t_1, r_1)
2      Δ_remove ← SimulateRemove(c, t_0, r_0)
3      if Δ_remove = None then
4          return None
5      MutateRemove(c, t_0, r_0)
6
7      Δ_add ← SimulateAdd(c, t_1, r_1)
8      if Δ_add = None then
9          MutateAdd(c, t_0, r_0)                ▷ Revert remove operation
10         return None
11
12     MutateAdd(c, t_0, r_0)                    ▷ Revert remove operation
13
14     return Δ_remove + Δ_add

15 function MutateMove(c, t_0, r_0, t_1, r_1)
16     Δ ← SimulateMove(c, t_0, r_0, t_1, r_1)
17     if Δ ≠ None then
18         MutateRemove(c, t_0, r_0)
19         MutateAdd(c, t_1, r_1)
```

Algorithm 14 Course c_0 and course c_1 swaps slots.

```
1  function SimulateSwap(c_0, t_0, r_0, c_1, t_1, r_1)
2      Δ_{remove,0} ← SimulateRemove(c_0, t_0, r_0)
3      if Δ_{remove,0} = None then
4          return None
5      MutateRemove(c_0, t_0, r_0)
6
7      Δ_{remove,1} ← SimulateRemove(c_1, t_1, r_1)
8      if Δ_{remove,1} = None then
9          MutateAdd(c_0, t_0, r_0)                    ▷ Revert remove operation
10         return None
11     MutateRemove(c_1, t_1, r_1)
12
13     Δ_{add,0} ← SimulateAdd(c_0, t_1, r_1)
14     if Δ_{add,0} = None then
15         MutateAdd(c_1, t_1, r_1)                    ▷ Revert operations
16         MutateAdd(c_0, t_0, r_0)
17         return None
18     MutateAdd(c_0, t_1, r_1)
19
20     Δ_{add,1} ← SimulateAdd(c_1, t_0, r_0)
21     if Δ_{add,1} = None then
22         MutateRemove(c_0, t_1, r_1)                 ▷ Revert operations
23         MutateAdd(c_1, t_1, r_1)
24         MutateAdd(c_0, t_0, r_0)
25         return None
26
27     MutateRemove(c_0, t_1, r_1)                     ▷ Revert operations
28     MutateAdd(c_1, t_1, r_1)
29     MutateAdd(c_0, t_0, r_0)
30
31     return Δ_{remove,0} + Δ_{remove,1} + Δ_{add,0} + Δ_{add,1}

32 function MutateSwap(c_0, t_0, r_0, c_1, t_1, r_1)
33     Δ ← SimulateSwap(c_0, t_0, r_0, c_1, t_1, r_1)
34     if Δ ≠ None then
35         MutateRemove(c_0, t_0, r_0)
36         MutateRemove(c_1, t_1, r_1)
37         MutateAdd(c_0, t_1, r_1)
38         MutateAdd(c_1, t_0, r_0)
```

C Code ReadMe

Requirements

- python 3.5 or 3.4
- numpy (only used in gridsearch for storing results)
- nose (only used in test for running tests)
- tabulate (only used in plot for generating latex tables)

This install script will setup python 3 on the HPC cluster, but it is much more complicated that what is needed for this project: `https://github.com/AndreasMadsen/my-setup/tree/master/dtu-hpc-python3`

Running the solver

Only python 3.4 or python 3.5 is required for running the solver.

```
python3 ./solver.py courses.utt lecturers.utt rooms.utt
    curricula.utt relation.utt unavailability.utt 300
```

Examples

- `scripts/inspect_tabu.py` will run a single optimization on the first dataset using TABU
- `scripts/inspect_alns.py` will run a single optimization on the first dataset using ALNS
- `scripts/grid_search_tabu.py` the grid search script used for the TABU optimization
- `scripts/grid_search_alns.py` the grid search script used for the ALNS optimization
- `scripts/test_tabu.py` the test script used for the TABU optimization
- `scripts/test_alns.py` the test script used for the ALNS optimization

Directories

The code is structured into the following directories:

- `dataset`: contains `Database` constructor, used for initializing a dataset
- `gridsearch`: code for running many optimizations using different parameters
- `judge`: code for validating the solution using `Judge.exe`.
- `plot`: code for generating latex tables and other summaries
- `script`: scripts used for manual inspecting and running on the HPC cluster
- `search`: the ALNS and TABU implementations are in here
- `solution`: contains `Solution` constructor that contains the an solution. Its methods simulate moves or mutate the solution. It also contains the random initialization procedure.
- `test`: test scripts, run them using `make test`

References

[1] M. Lindahl and N.-C. F. Bagger, "University timetabling - curriculum-based course timetabling", *DTU*, 2015.