# Project Artificiële Intelligentie

Het ontwikkelen van een algoritme dat automatisch lesrooster kan opstellen

Lucas Belpaire en Victor Matthijs

december 2019

# Inhoudsopgave

# 1 Introductie

## 1.1 Beschrijving van het probleem

Het opstellen van lesroosters, of meer algemeen, uurroosters blijft een uitdagend probleem met heel wat toepassingen in verschillende sectoren. Ondanks de vele toepassingen is het opstellen van een dergelijk uurrooster geen triviaal probleem, het behoort namelijk tot de NP-complete problemen[1]. Hieruit volgt dat enkel heuristieke methoden zouden mogelijk zijn om het probleem op te lossen[2].

Het opstellen van uurroosters kan gedefinieerd worden als het toekennen van een verzameling van activiteiten aan groepen, zoals personen, locaties en bepaalde tijdsduren[3][4][5]. Bij het toekennen moet er worden voldaan aan harde beperkingen, die afhangen van het specifieke probleem. Het doel is om een geldige toekenning van alle activiteiten te zoeken, met een zo klein mogelijk aantal overtredingen op de zachte beperkingen.

In dit verslag wordt concreet het opstellen van een lesrooster voor de Universiteit Gent besproken, maar de technieken die gebruikt worden zijn algemeen toepasbaar. De harde beperkingen voor dit probleem zijn als volgt:

- Geen enkele student kan meer dan één les tegelijk volgen.
- Geen enkele lesgever kan meer dan één les tegelijk geven.
- Elk vak moet voldoende lesuren krijgen om aan het opgegeven aantal contacturen te komen.
- Lokaalreservaties kunnen niet overlappen.
- De lokaalcapaciteit mag niet overschreden worden.

De zachte beperkingen:

- Het verschil tussen het aantal zitplaatsen en het aantal studenten per reservatie dient geminimaliseerd te worden. De lokalen moeten dus zo goed mogelijk passen bij de grootte van de groep.
- De laatste twee lesuren van de dag dienen vermeden te worden, alsook lesuren in de inhaalweek.
- Vermijd vier of meer opeenvolgende lesuren zonder pauze.
- Vermijd dat studenten op sommige dagen maar één lesuur hebben.

### 1.2 Beschrijving van de invoerdata

De invoerdata bevat alle nodige informatie om het probleem op te lossen, zoals de lesuren, professoren, curricula en lokalen. Deze data is opgeslagen in een JSON-bestand. De velden, met bijhorende informatie, van dit JSON-bestand bevinden zich in appendix A.

### 1.3 Literatuurstudie en opzoekingswerk

De methode die gebruikt zal worden om tot een oplossing te komen bestaat uit 2 verschillende fases[6]. In de eerste fase wordt er geprobeerd om een geldig uurrooster te bekomen. Dit betekent dat alle lessen geplaatst zijn en dat er geen enkele harde beperking overtreden wordt. Deze fase bestaat uit twee stappen, in de eerste stap wordt er door middel van een heuristische constructie een initieel uurrooster bekomen. Na deze constructie wordt er gebruik gemaakt van tabu search[7] om dit initieel lesrooster, indien nodig, te vervolledigen.

In de tweede fase wordt het lesrooster geoptimaliseerd, het aantal strafpunten veroorzaakt door de overtredingen van zachte beperkingen wordt dus zo klein mogelijk gemaakt. In deze fase zal het lesrooster steeds geldig blijven, er mogen dus nooit harde beperkingen overtreden worden. Deze fase zal gebruik maken van 'simulated annealing' om een optimaal lessenrooster te bekomen.

## 2 Methodiek

Hoewel dit niet expliciet vermeld is in de opgave, is consistentie een belangrijk eigenschap van een goed lesrooster. Het is niet handig als een student voor iedere lesweek een totaal verschillend lesrooster moet volgen. Vandaar dat er zal geprobeerd worden om een zo consistent lesrooster op te bouwen.

Eerst worden alle lessen verdeeld in 4 verschillende types, afhankelijk van hun hoeveelheid lesuren. Daarna zal geprobeerd worden om een lesrooster van 1 week op te stellen, met alle lessen die tot het type met de meeste lesuren behoren. In dit geval bevat type 1 alle lessen die minstens 1 lesuur per week tellen. Dit lesrooster, dat dus 1 week voorstelt, wordt als basis gebruikt om de overige weken op te stellen. Het zal 12 maal gekopieerd worden en zo de 12 lesweken voorstellen. Hierna trachten we de andere types lessen ook toe te voegen aan deze 12 lesroosters. Omdat er 12 keer vertrokken wordt vanuit hetzelfde basis lesrooster, wordt een vrij consistent lesrooster bekomen. Zeker omdat de overige types van lessen in verhouding vrij weinig lessen bevatten.

Het concreet opstellen van een lesrooster bestaat uit 2 fases. In de eerste fase wordt er geprobeerd een geldig lesrooster te bekomen. Dit zal eerst gebeuren door middel van een heuristische constructie. Als bij deze constructie niet alle lessen kunnen worden geplaatst, zal er ook nog tabu search worden uitgevoerd om de resterende lessen te plaatsen.

In de tweede fase wordt door middel van 'simulated annealing' geprobeerd om het aantal overtredingen op zachte beperkingen zo laag mogelijk te krijgen. Belangrijk hierbij is dat geen enkele tussenoplossing een ongeldige, dus overtredingen op harde beperkingen, lesroosters mag opleveren.

## 2.1 Data representatie

Bij de start van het programma wordt eerst alle data ingeladen en omgezet naar object instanties van klassen. De volgende klassen worden hiervoor gebruikt: *Course*, *Lecturer*, *Curriculum*, *Site*, *ClassRoom*, *CourseEvent* en *TimeTable*. Objecten bevatten vaak verwijzingen naar andere objecten. Dit gebeurt door een code (in de vorm van een string) bij te houden die overeenkomt met de code van een ander object. Dankzij mappings die zijn opgeslagen kan zo het gewenste object gevonden worden. Voor dit verslag is het programma geïmplementeerd met Python en worden de mappings voorgesteld door middle van dictionaries. Geen enkel object is dus rechtstreeks verbonden met een ander object.

Een meer gedetailleerde uitleg over de gebruikte klassen bevindt zich in appendix B.

## 2.2 Werking programma

### 2.2.1 Invoer verwerken

De invoerdata zal omgezet worden naar object instanties van de klassen omschreven in appendix B. Tijdens het verwerken worden ook alle mappings opgeslagen zodanig dat het programma later gemakkelijk aan objecten kan geraken.

Zoals eerder vermeld worden de lessen opgedeeld in verschillende types op basis van hoeveelheid lesuren. Daarna wordt er voor ieder lesuur in een bepaald type, een *CourseEvent* object aangemaakt en in de overeenkomende lijst bewaard. We werken met de volgende 4 types:

- Type 1: 1 of meer lesuren per week
- Type 2: 0.5 of meer lesuren per week

- Type 3: 0.25 of meer lesuren per week
- Type 4: overige lessen

Het is mogelijk dat er voor bepaalde lessen *CourseEvent* objecten in verschillende types lijsten worden opgeslagen. Stel dat een les in totaal 18 lesuren telt. Dat betekent dat er over een periode van 12 weken, er 1.5 lesuren per week zal moeten gegeven worden. Omdat er echter geen halve lesuren kunnen worden ingepland, zal dit als volgt worden opgelost: er zal 1 *CourseEvent* object aangemaakt worden en in de lijst met type 1 bewaard worden. Er blijven nu nog 0.5 lesuren over. Er zal dus ook nog 1 *CourseEvent* object worden aangemaakt en in de type 2 lijst bewaard worden.

### 2.2.2 Heuristische constructie van het lesrooster

De heuristische constructie van het lesrooster wordt uitgelegd aan de hand van de pseudocode hieronder:

```
ongeplaatsteEvents = []
gesorteerdeEvents = sorteerCourseEvents(events)
for event in gesorteerdeEVents:
    bp = [] # beschikbare plaatsen
    for (fi_nummer, lesuur) in timetable.empty_positions:
        past = eventPastInLesuur() and lokaalIsGrootGenoeg()
        if past:
            bp.append(fi_nummer, lesuur)
    if lengte(beschikbarePlaatsen) == 0:
        ongeplaatsteEvents.append(event)
    gesorteerdePosities = sorteerPosities(bp)
    bestePositie = gesorteerdePosities.pop()
    timetable.voegEventToe(positie, event)
return ongeplaatsteEvents, timetable
```

In deze eerste fase wordt een lesrooster zo goed mogelijk gevuld. Er wordt een heuristische aanpak gebruikt. Eerst worden de *CourseEvent* objecten gesorteerd op basis van prioriteit. Dit betekend dat moeilijk te plaatsen lessen eerst geplaatst zullen worden. De functie *sorteerCourseEvents* ziet er als volgt uit:

```
def sorteerCourseEventsOpBasisVanPrioriteit(events):
    for event in events:
        rang1 = aantal lesuren / beschikbare lesuren
        rang2 = aantal conflicten
    # van groot naar klein
    sorteer events op basis van rang1, rang2
    return events
```

Bij het sorteren wordt er eerst gekeken naar rang1. Die is gelijk aan het aantal in te plannen uren gedeeld door het aantal nog beschikbare lesuren. Een beschikbaar lesuur is een lesuur waarop zowel alle curricula die het vak volgen, als minstens 1 prof die het vak geeft beschikbaar zijn. Indien 2 events dezelfde rang1 zouden hebben wordt er gesorteerd op basis van rang2. Hierbij wordt het aantal potentiële conflicten berekend. Een conflict komt voor als het vak een lesgever deelt met een ander vak of als het vak een curriculum deelt met een ander vak. Hoe meer conflicten, hoe hoger de prioriteit van een vak.

Vervolgens worden de events in volgorde overlopen. Per event wordt dan een lijst beschikbare posities gezocht, die daarna zullen worden gesorteerd. Het event zal geplaatst worden op de best mogelijke positie. Indien er geen beschikbare posities zijn, zal het event worden toegevoegd aan de lijst ongeplaatsteEvents. De events op deze lijst zullen worden ingepland in de Tabu Search fase. De functie *sorteerPosities* ziet er als volgt uit:

```
def sorteerPosities(posities):
    for (fi_nummer, lesuur) in posities:
        rang1 = berekenNotHomePenalty(fi_nummer)
        rang2 = berekenCapaciteitVerschil(fi_nummer)
    # van klein naar groot
    sorteer posities op basis van rang1, rang2
    return posities
```

Als eerste wordt er gesorteerd op de 'not home penalty', posities die niet op de homesite of op een bepaalde afstand liggen krijgen een grote rang1. Vervolgens wordt er gekeken naar het verschil tussen het aantal studenten en de capaciteit van het lokaal, hoe kleiner dit verschil hoe beter. Hoe kleiner rang1 en rang2 van een positie, hoe beter die positie geschikt is.

Na het overlopen van alle events zullen de ongeplaatste Events en de timetable teruggegeven worden.

### 2.2.3 Tabu search

Indien er in de vorige fase geen compleet en geldig lesrooster bekomen is, zal er door middel van tabu search geprobeerd worden om toch één te bekomen. Het principe achter tabu search is vrij eenvoudig, er zijn verschillende methodes die een bepaalde zet doen die een effect zal hebben op de timetable en de ongeplaatsteEvents. De zet wordt telkens opgeslagen in de tabu lijst. Zetten in die lijst mogen niet meer uitgevoerd worden. De lijst heeft wel een maximale lengte, als ze vol zou komen te zitten zal de oudste zet verwijderd worden. Bijgevolg is het dan mogelijk om deze weer uit te voeren. Nadat een zet wordt uitgevoerd wordt er nagekeken of de bekomen timetable zich dichter bevindt bij een compleet geldige timetable. Dit kan gebeuren door na te gaan hoeveel elementen

ongeplaatsteEvents nog bevat. Indien het aantal elementen gedaald is, wordt er verder gewerkt met de bekomen timetable. Indien het aantal elementen zou gestegen zijn of gelijk gebleven verwerpen we de bekomen timetable en werken we verder met de originele timetable. De pseudocode voor tabu search ziet er als volgt uit:

```
def tabu_search():
    # tabu lijst voor de swap methode
    tabu_swap = []
    # tabu lijst voor de split methode
    tabu_split = []
    #tabu lijst voor de advanced swap methode
    tapu_advanced_swap = []
    while len(events) != 0 & de tijdslimiet niet overschreden:
        if tabu_swap is vol:
            tabu_swap.pop()
        if tabu_split is vol:
            tabu_split.pop()
        if tabu_advanced_swap is vol:
            tabu_advanced_swap.pop()
    x = random(100) # getal tussen 0 en 100
    if x < 33:
        swap(tabu_swap)
        continue
    if x < 66:
        split(tabu_list)
        continue
    if x >= 66:
        advanced_swap(tabu_advanced_swap)
        continue
    return events, timetable
```

De *swap* methode zal 2 willekeurige posities (=tuple: (fi_nummer, lesuur) in het lesrooster verwisselen. De *split* methode zal het event met het grootste aantal studenten, die zich niet in de tabu lijst bevindt, nemen en opsplitsen. *advanced_swap* zal proberen om een event uit het lesrooster te wisselen met een ongeplaatst event. Na het uitvoeren van één van deze functies zal er telkens weer geprobeerd worden om alle andere ongeplaatste events te plaatsen.

### 2.2.4 Simulated Annealing

De laatste fase is het uitvoeren van 'simulated annealing', deze fase heeft als doel om de totale som van alle strafpunten, veroorzaakt door het overtreden van zachte beperkingen, zo laag mogelijk te krijgen. 'Simulated annealing' is

een algoritme voor het oplossen van een optimalisatie probleem, in dit geval dus het optimaliseren van de zachte beperkingen. Het aanvaarden van een tijdelijk slechtere oplossing is een belangrijk kenmerk van de heuristiek. Dit zorgt ervoor dat het algoritme minder snel zal vast zitten in een dal en er dus een globaal optimum kan gevonden worden. Deze simulated annealing methode zal net zoals bij de tabu search twee willekeurige lesuren kiezen en de events op deze plaatsen wisselen, hierna gaat men na of dit een betere oplossing is dan daarvoor. Om specifiek te weten wat beter is, maakt men gebruik van een methode die de totale som van alle strafpunen van de huidige timetable zal berekenen. De pseudo-code voor simulated annealing ziet er als volgt uit:

```
def simulated_annealing(t_max, t_min, steps):
    step = 0
    while total_penalty > 0 & tijdslimiet niet overschreden:
        if 10 times no improvement:
            step = 0

        t_value = t_max*math.exp(t_factor*step/steps)

        if t_value > t_min:
            step += 1

        swap to positions
        if succes:
            no_improvement += 1
        else:
            no_improvement = 0

    return total_penalty
```

In bovenstaande pseudocode maakt men gebruik van de waardes 't_max', 't_min' en 'steps'. 'T_max' staat voor 'maximale temperatuur', 't_min' voor 'minimale temperatuur' deze namen zijn zo gekozen omdat de methode 'simulated annealing' afkomstig is uit metaalbewerking. Hier worden ze gebruikt om tussen de twee waardes te schommelen door gebruik te maken van de stap grootte, hiervoor dient het argument 'step'. Door deze 3 argumenten te gebruiken kan men opzoek gaan naar het globale minimum en dus de beste score.

### 2.2.5 Algemeen verloop

Bij de start van het programma wordt maar 1 timetable en enkel de events van type 1 beschouwd. Om deze timetable te vullen met deze events wordt eerst de construct fase uitgevoerd, gevolgd door tabu search. Daarna zal ook nog de improvement fase worden uitgevoerd om overtredingen op zachte beperkingen te minimaliseren.

Nadat deze eerste timetable is opgesteld, zal hij gekopieerd worden zodat we twee identieke timetables bekomen. Deze twee timetables vullen we dan met events van type 2, compleet analoog aan het vullen van de eerste timetable. Het enige verschil is dat de 'improvement fase' niet meer zal worden toegepast.

Dit proces blijft herhaald worden, zo zullen we de twee bekomen timetables verdubbeld worden, en ogevuldt worden met events van type 3. De bekomen timetables worden ten slotte nog verdrievoudigd en gevuld met events van type 4.

Al de niet geplaatste events worden telkens bijgehouden. Een lege timetable die week 13 voorstelt wordt ten slotte gevuld met deze events. Belangrijk is dat hier wel weer de 'improvement fase' wordt uitgevoerd, aangezien deze week niet consistent moet zijn met de 12 andere weken.

# 3 Resultaten

Het programma zoals beschreven in de sectie methodiek levert een vrij goed bezettingspercentage van de lessen die meegegeven zijn in de invoerdata. Ongeveer 95% van de lessen kunnen op geldige plaatsen geplaatst worden. Het valt op dat het uitvoeren van 'tabu search' weinig effect heeft op de hoeveelheid lessen die geplaatst worden. Als 'tabu search' niet wordt uitgevoerd wordt er ook een bezettings percentage van 95% gehaald. Als de tijdslimiet zeer hoog wordt gezet (25 minuten), zal het eindresultaat ten hoogste een halve procent afwijken.

De volgorde van de rangen in de heuristische constructie fase heeft ook zo goed als geen effect op het bezettingspercentage. Wel zal het lesrooster veel slechter scoren omdat de 'not home penalty' zeer zwaar doorweegt. De volgorde van rangen die beschreven is in de methodiek is bijgevolg de beste keuze.

Aangezien er 13 verschillende lesroosters zijn, is er de mogelijkheid om op ieder lesrooster 'simulated annealing' uit te voeren. Toch wordt er gekozen om dit enkel op het originele lesrooster en het lesrooster dat week 13 voorstelt toe te passen. Er is voor deze aanpak gekozen omdat uit experimenten bleek dat er geen verschil is tussen beide aanpakken qua verbetering in strafpunten. Aangezien er geprobeerd wordt om een zo consistent lesrooster op te bouwen, is het logisch dat 'simulated annealing' niet op ieder lesrooster wordt toegepast.

Uit experimenten bleek ook dat het uitvoeren van 'simulated annealing' nauwelijks invloed heeft op het reduceren van strafpunten. Dit komt voort uit het feit dat de heuristische constructie zeer goed rekening houdt met de 'not home penalty' die zwaar doorgeeft. Er wordt eveneens ook al goed rekening gehouden met de groottes van de lokalen en de afstanden bij het uitvoeren van de heuristische constructie.

# 4 Conclusie

Het is duidelijk dat de algoritmes die beschreven zijn in dit verslag voordelen en nadelen hebben om een goed lesrooster te bekomen. Het grote voordeel van de beschreven aanpak is dat er een consistent lesrooster bekomen wordt. Dit is een belangrijk eigenschap van lesroosters die in de praktijk gebruikt zouden worden. Het is ook duidelijk dat de beschreven heuristische constructie zeer krachtig is. Ook is de implementatie hiervan relatief simpel. Het nadeel van de gebruikte algoritmes is dat ze niet sterk genoeg blijken te zijn. Er zijn complexere algoritmes die er in slagen om een grotere bezettingsgraad te halen. Het toepassen van 'tabu search' en 'simulated annealing' lijkt ook weinig effect te hebben op het finaal bekomen lesrooster.

Hieruit kan geconcludeerd worden dat het gebruik van de heuristische constructie een zeer goede basis kan zijn. Het is relatief gemakkelijk te implementeren en kan zorgen voor een zeer goed lesrooster. Aangezien 'tabu search' en 'simulated annealing' weinig impact hebben, kan het interessant zijn om alternatieve algirmtes toe te passen om de oplossing te verbeteren.

# 5 Appendix

## A Velden invoerdata

| Algemene velden | |
|---|---|
| **Veld** | **Beschrijving** |
| academiejaar | Het academiejaar waar de data over gaat. Dit is gewoon een jaartal (bv. 2018). |
| semester | Het semester van het academiejaar. Dit is een getal, ofwel 1 ofwel 2. |
| kilometerpenalty | Een strafscore per kilometer die de studenten aan verplaatsing moeten afleggen. |
| lateurenkost | Een strafscore die wordt toegekend telkens als de laatste twee lesuren gebruikt worden of er lessen in de inhaalweek plaatsvinden. |
| nothomepenalty | Een strafscore die wordt toegekend voor elke lesactiviteit die niet op de thuisbasis van de studenten georganiseerd wordt. |
| minimaalStudentenaantal | Het minimaal aantal studenten waar je rekening mee moet houden voor lokaalcapaciteit. |
| vakken | Een lijst van alle vakken die ingepland moeten worden |
| sites | Een lijst van beschikbare sites |

| Sites | |
|---|---|
| **Veld** | **Beschrijving** |
| code | De unieke code van deze site binnen het systeem van de UGent. |
| naam | De naam van de site. |
| xcoord | Breedtegraad van de site. |
| ycoord | Lengtegraad van de site. |
| lokalen | Lijst van lokalen die bij deze site horen. |

| Lokalen | |
|---|---|
| **Veld** | **Beschrijving** |
| finummer | De unieke code van dit lokaal binnen het systeem van de UGent. |
| naam | De naam van dit lokaal binnen de site. |
| capaciteit | De maximale capaciteit van dit lokaal, d.i. het grootst aantal studenten dat hier comfortabel kan plaatsnemen. |

| Vakken | |
|---|---|
| **Veld** | **Beschrijving** |
| code | De unieke code van dit vak binnen het systeem van de UGent. |
| cursusnaam | De officiële naam van dit vak. |
| studenten | Het verwacht aantal studenten dat dit vak zal volgen. Merk op dat dit veld soms nul is (zeker voor keuzevakken); in dat geval hanteer je het minimum aantal studenten. |
| contacturen | Het totaal aantal uren dat dit vak beslaat binnen het semester. Je moet voldoende lesmomenten inplannen om per vak aan dit aantal uren te geraken. |
| lesgevers | Een lijst van lesgevers die dit vak begeleiden. Elk element in deze lijst heeft de gebruikelijke eigenschappen: UGentid, voornaam en naam. Hou er rekening mee dat lesgevers maar op één plaats tegelijk kunnen zijn en dat er steeds minstens één lesgever nodig is om een les te geven. |
| programmas | Een lijst van programma's waar dit vak bijhoort. Een programma wordt bepaald door een code (code), een jaar binnen modeltraject 1 (mt1) en de code van de site die de thuisbasis van dit programma bepaalt (homesite). Je moet erop letten dat twee verschillende vakken die bij hetzelfde programma horen (identieke code en MT1) nooit overlappen; studenten kunnen ook maar op één plaats tegelijk zijn. Let op: sommige vakken hangen niet vast aan een programma (bijvoorbeeld universiteitsbrede keuzevakken); hier valt de beperking van niet-overlap dan ook weg. We houden bovendien geen rekening met modeltraject 2, GIT of met studenten die vakken uit ongerelateerde opleidingen opnemen als keuzevakken. |

## B Gebruikte klassen

De klasse *Course* stelt een vak voor. Het bevat volgende velden: *code*, *name*, *student_amount*, *contact_hours*, *course_hours*, *lecturers*, *curricula* en *course_events*. De waardes van deze velden worden uit de invoerdata gehaald. De velden *lecturers* en *curricula* bevatten elk een lijst met codes die verwijzen naar *lecturers* objecten en *curriculum* objecten, en dus niet de objecten zelf. Dit geldt ook voor alle andere klassen en wordt vanaf nu niet meer herhaald in dit verslag. Het veld *course_hours* wordt bekomen door het veld *contact_hours* te vermenigvuldigen met 0,8.

*Lecturer* stelt een lesgever voor. De klasse bevat volgende velden: *ugent_id*, *first_name*, *last_name* en *occupied_time_slots*. De eerste drie velden worden rechtstreeks overgenomen van de invoerdata. Het veld *occupied_time_slots* is een (initieel lege) verzameling, die alle bezette lesuren van de lesgever bijhoudt. Zo is het mogelijk om efficiënt op te zoeken of een lesgever op een bepaald lesuur beschikbaar is.

Ieder curriculum wordt voorgesteld door de klasse *Curriculum*. Deze bevat de velden: *code*, *mt1*, *home_site*, *occupied_time_slots*. De eerste drie velden komen rechtstreeks uit de invoerdata. Het veld *occupied_time_slots* is compleet analoog aan het veld met dezelfde naam in de klasse *Lecturer*.

De klasse *Site* stelt een site voor en bevat volgende velden: *code*, *name*, *x_coord*, *y_coord* en *class_rooms*. Ook deze velden worden overgenomen uit de invoerdata.

*ClassRoom* stelt een lokaal voor en bevat volgende velden: *fi_number*, *name*, *capacity*, *site_id*. Deze velden worden overgenomen uit de invoerdata.

Een van de belangrijkere klassen is *CourseEvent*. Deze stelt een concreet lesuur van een bepaalde les voor. De klasse bevat volgende velden: *code*, *lecturers*, *student_amount*, *lecturers*, *curricula*. Deze velden zijn compleet analoog aan de velden van het overeenkomende *Course* object. Het veld *student_amount* kan echter nog aangepast worden, indien het *CourseEvent* object in twee zou gesplitst worden omdat de groep studenten te groot is om geplaatst te worden. *student_amount* komt hier overeen met het aantal student dat ingepland wordt voor deze specifieke les, en dus niet met het aantal studenten dat dit vak volgen. Verder bevat deze klasse ook nog het veld *assigned_lecturer* dat de ugent_id bevat van de lesgever die wordt ingepland om dit lesuur te geven. Ook al kan een vak door meerdere lesgevers gegeven worden, wordt er enkel gekeken of er 1 lesgever beschikbaar is, deze zal dan bijgehouden worden in dit veld.

*TimeTable* wordt gebruikt om het lesrooster van 1 week voor te stellen. Het veld *timetable* bevat een dictionary die het effectieve lesrooster voorstelt. De sleutels van de dictionary zijn tupels die er als volgt uit zien: (fi_nummer, time_slot). Ie-

dere mogelijke combinatie van lokaal met lesuur wordt als sleutel gebruikt in de dictionary. Indien een les ingepland wordt in de *timetable*, wordt een *CourseEvent* object in het dictionary gestoken met de overeenkomende sleutel. Verder bevat deze klasse ook nog de velden *occupied_positions* en *empty_positions*, die respectievelijk alle plekken in de dictionary bijhouden die al bezet zijn, en die nog vrij zijn. Ten slotte bevat het ook nog een veld *offset* die een integer waarde bevat die wordt gebruikt om bij te houden welke week dit lesrooster voorstelt. Verder bevat de klasse 3 methodes: één om lessen te plaatsen in het lesrooster, één om lessen te verwijderen uit het lesrooster en één om de offset aan te passen.

# Referenties


[1] Cooper, T. B. & Kingston, J. H., 1996. *The complexity of timetable construction problems*. s.l.:Springer.

[2] Pearl, J., 1984. *Heuristics: Intelligent Search Strategies for Computer Problem Solving.Reading*. MA: Addison-Wesley.

[3] Wren A., 1996. *Scheduling, timetabling and rostering a special relationship?* In: Burke E, Ross P (eds) Practice and theory of automated timetabling, vol 1153., Lecture notes in computer science Springer

[4] Schaerf A., 1999 *A survey of automated timetabling*. Artif Intel Rev 13(2):87–127

[5] Carter M., 2013 *Timetabling. In: Gass S, Fu M (eds) Encyclopedia of operations research and management science*. Springer, US, pp 1552–1556

[6] Stephan E. Becker, 2013 *Metaheuristic algorithms for the automated timetabling of university courses*

[7] Glover F., Laguna M., 1998 *Tabu Search*. In: Du DZ., Pardalos P.M. (eds) Handbook of Combinatorial Optimization. Springer, Boston, MA