Toronto problem

Anthony Tacquet

Literature study

Similar problems

Nurse Scheduling Problem (NSP)

Het *Nurse Scheduling Problem* is een roosterprobleem gericht op het maken van een schema voor verplegers:

1. Het aantal pauzes
2. De toewijzing van verplegers voor elke dienst
3. De tijd tussen twee opeenvolgende werkdagen

Project Scheduling Problem (PSP)

In het *Project Scheduling Problem* is het doel een optimale oplossing te vinden voor het efficiënt beheren van een project qua tijd en kosten.

Ways of solving

Integer Linear Programming (ILP)

Integer Linear Programming is een wiskundig optimalisatie/haalbaarheid algoritme waarbij sommige of alle variabelen beperkt zijn tot gehele getallen. Zowel de doelfunctie als de beperkingen zijn lineair.
Source: https://en.wikipedia.org/wiki/Integer_programming

Genetic Algorithms

In de informatica en operationeel onderzoek zijn genetische algoritmen metaheuristieken geïnspireerd door natuurlijke selectie. Ze behoren tot de bredere klasse van evolutionaire algoritmen en worden vaak gebruikt om hoogwaardige oplossingen te genereren voor optimalisatie- en zoekproblemen door te vertrouwen op bio-geïnspireerde operatoren zoals mutatie, kruising en selectie.
Source: https://en.wikipedia.org/wiki/Genetic_algorithm

Simulated Annealing

Simulated Annealing is een probabilistische techniek om de globale optimale benadering van een functie te benaderen. Het dient als een metaheuristiek om de globale optimalisatie te benaderen in een groot zoekgebied voor een optimalisatieprobleem. In gevallen met talrijke lokale optima kan *Simulated Annealing* het globale optimum identificeren.
Source: https://en.wikipedia.org/wiki/Simulated_annealing

Tabu Search

Tabu Search verbetert de prestaties van lokaal zoeken door de basisregel te versoepelen. Het staat toe dat verslechterende zetten bij elke stap worden geaccepteerd wanneer er geen verbeterende zet beschikbaar is (bijvoorbeeld wanneer de zoektocht vastloopt in een strikt lokaal minimum). Daarnaast worden verboden (vandaar de term "tabu") geïntroduceerd om te voorkomen dat de zoektocht terugkeert naar eerder bezochte oplossingen.
Source: https://en.wikipedia.org/wiki/Tabu_search

Constraint Programming

Constraint Programming is een paradigma voor het oplossen van combinatoriële problemen. Gebruikers geven op declaratieve wijze beperkingen aan op haalbare oplossingen voor een reeks beslissingsvariabelen. Dit maakt doorgaans gebruik van standaardmethoden zoals chronologisch *backtracking* en *constraint propagation*, maar kan aangepaste code gebruiken, zoals een probleemspecifieke vertakking heuristiek.
Source: https://en.wikipedia.org/wiki/Constraint_programming

Critical Path Method (CPM)

Critical Path Method is een essentieel projectmanagement instrument dat de meest uitgebreide reeks taken vertegenwoordigt die moeten worden voltooid om de voltooiing van het hele project te garanderen. Het kritieke pad kan worden bepaald door de langste reeks afhankelijke activiteiten van begin tot eind te meten.
Source: https://www.wrike.com/blog/critical-path-is-easy-as-123/

Program Evaluation and Review Technique (PERT)

Program Evaluation and Review Technique is een methode voor het analyseren van taken die betrokken zijn bij het voltooien van een project, met name de tijd die nodig is om elke taak te voltooien. Het identificeert de minimale tijd die nodig is om het gehele project te voltooien en biedt ruimte voor onzekerheid door planning mogelijk te maken zonder exacte details en duur van alle activiteiten te kennen.
Source: https://en.wikipedia.org/wiki/Program_evaluation_and_review_technique

Inspiration

Ik heb niet veel inspiratie opgedaan van gelijkaardige problemen, ik heb mijn research pas na mijn oplossing gedaan. Nu besef ik dat dit niet de allerbeste keuze was.

Framework

Als framework is er gekozen voor het Odisee Framework. Door enkele kleine aanpassingen te maken aan de *solution* interface en de gekregen algoritmes kon dit framework goed gebruikt worden om een oplossing te berekenen voor het *Toronto Problem*.

Solution

De *MySolution* klas bevat drie lijsten, een lijst met tijdsloten, een lijst met examens en een lijst met studenten. Tijdens het starten van het programma is het de bedoeling te starten met een *feasible* oplossing, waardoor elk examen in een tijdslot wordt geplaatst. Door dit willekeurig te doen zal dit nu eenmaal niet lukken. Om dit vlotter te laten verlopen wordt er gewerkt met een score systeem. Elk examen krijgt een score afhankelijk van in hoeveel tijdsloten dit examen kan zonder een beperking te overtreden. Nadat de scores uitgedeeld zijn zullen de eerste x-aantal examens met de laagste score in een tijdslot worden geplaatst. De score zal dan opnieuw berekend worden op de overblijvende examens.

Moves

Er is gebruikgemaakt van twee move-klassen, namelijk *MyMoveOneRandomExamMove* en *MySwapTwoRandomExamsMove*. Deze twee klassen worden willekeurig gekozen bij de werking van het algoritme.

MyMoveOneRandomExamMove: Deze move zal een examen uit een tijdslot kiezen en dat examen naar een willekeurig ander tijdslot verplaatsen. De verplaatsing zal uiteraard pas gebeuren, nadat er gecontroleerd wordt of er geen harde beperkingen overtreden worden. Dan zal deze move geëvalueerd worden.

MyMoveTwoRandomExamsMove: Deze move zal twee willekeurige examens uit verschillende tijdsloten kiezen. Als de examens in het tijdslot van de andere worden geplaatst zonder harde beperkingen te overtreden, zal deze move worden geëvalueerd.

Costfunction

In deze oplossing worden er twee kostfuncties gebruikt. De eerste kostfunctie wordt berekend aan de hand van de *absolute evaluation*, daarna zal de *delta evaluation* worden gebruikt.

Absolute evaluation: Deze functie zal alle studenten in elk tijdslot doorlopen. Bij elke student zal er dan berekend worden hoe ver het volgende tijdslot is wanneer die student een examen heeft en zo zal de kost berekend worden.

Delta evaluation: Bij *delta evaluation* zullen we ongeveer op dezelfde manier werken als in de *absolute evaluation*, maar er zal niet voor elke student de kost berekend worden. De kost zal nu enkel berekend worden op de studenten die in de examens zitten die verwisseld worden. Dit wordt berekend voor en na de *move* om zo de deltawaarde te berekenen, als die positief is zal de move slecht zijn, zo niet dan zal de move goed zijn. De deltawaarde wordt berekend door de kost voor de *move* af te trekken van de kost na de *move*.

De nieuwe kost wordt als volgt berekend:

$(originalCost + costAfter - costBefore) / students.size()$

Results

With 1000 iterations

Algorithm / Dataset	Steepest Descent	Late Acceptance	Simulated Annealing
St. Andrews 83	164.3	169.3	170.4
Ed HEC 92	13.8	15.3	14.7
Carleton 91	8.2	10.1	9.0

With 2000 iterations

Algorithm / Dataset	Steepest Descent	Late Acceptance	Simulated Annealing
St. Andrews 83	167.9	167.2	168.8
Ed HEC 92	12.9	15.3	15.8
Carleton 91	7.6	8.8	8.8

Hoe groter de dataset wordt, hoe nuttiger het is meer iteraties te gebruiken. CAR91 is een vrij grote dataset waardoor er na 1000 iteraties nog optimalisatie mogelijk was. In tegenstelling tot STA83 en HEC92 die eerder klein waren, was er niet veel optimalisatie meer mogelijk.

Conclusion

Mijn oplossing is zeker niet optimaal; het blijft nog te traag bij het berekenen van de *absolute evaluation* en het vinden van geschikte locaties voor examens in de *move* klassen. Ik heb overmatig gebruikgemaakt van for-loops, terwijl het beter zou zijn geweest om onderlinge verbindingen tussen studenten, examens en tijdsloten te creëren om zo de kosten te berekenen. Dit zou aanzienlijk efficiënter zijn geweest.