Roosteroptimalisaties

Chun-Fei Lung*, Tim Schwarte, Martijn Terpstra, Jille Timmermans

August 27, 2009

*Chun-Fei Lung heeft helaas door een studieswitch ons projectgroep verlaten.

Contents

1 Inleiding

Voor de cursus Research en Development 1 hebben we een onderzoek gedaan naar een onderwerp binnen de kunstmatige intelligentie. Al snel viel onze intresse op de genetische algoritmes. Hiermee hadden we het middel al gevonden, nu nog een doel. We hebben vele mogelijkheden overwogen maar uiteindelijk viel onze keuze op schoolroosters. Het maken van roosters is iets wat tegenwoordig vaak nog niet goed wordt gedaan. Vaak worden deze bijvoorbeeld nog met de hand gemaakt, wat uiteraard erg arbeidsintensief en tijdrovend is. Gelukkig zijn er ook programma's op de markt die dit werk flink kunnen verlichten. Toch merken we vaak dat bij beide manieren we geen roosters krijgen waar we tevreden mee kunnen zijn: roosters met (een groot aantal) tussenuren bijvoorbeeld.

2 Probleemstelling

Uit ons pilotonderzoek konden we al constateren dat willekeurige, automatisch gegenereerde roosters zijn te verbeteren met genetische algoritmen: telkens kleine willekeurige mutaties toe te passen en de beste roosters te selecteren, zodat á la *Survival of the Fittest* de betere roosters overblijven. Commerciële (en enkele vrije) software doet dat vaak nog beter. Omdat de bron van vrije software veelal vrij te bekijken is, willen wij ons nu ook daar op richten.

Het gaat hier in feite om een optimalisatieprobleem; dat wil zeggen dat door verbeterde roostergeneratie-algoritmes niet alleen roosters kunnen worden gemaakt die zoveel mogelijk aan onze eisen voldoen, maar dat bijvoorbeeld ook reisschema's van vervoersbedrijven verbeterd kunnen worden door iets andere constraints te nemen.

3 Onderzoeksvraag

Onze onderzoeksvraag luidt: Kunnen we met genetic algorithms betere roosters maken dan een bestaand programma zoals FET

4 Theoretisch kader

Enkele belangrijke begrippen uit onze onderzoeksvraag zijn 'mutaties', 'fitness', 'random', 'genetic algorithms', 'hill climbing' en 'local beam search'.

Mutaties

Een kleine wijziging aan - in dit geval - het rooster, waardoor we een net iets ander rooster krijgen, maar welke nog wel een goed rooster is. Bijvoorbeeld: we verwisselen het eerste en het tweede uur op maandag met elkaar.

Fitness

Een score die een rooster krijgt, die een indicatie is van hoe goed dit rooster is.

Random

Hier wordt er willekeurig een rooster gegenereerd dat waarschijnlijk niet zo goed is. Dit wordt gebruikt om de eerste roosters te maken.

Genetische algoritmes

Genetische algoritmes kunnen worden gebruikt om dingen te optimaliseren. Bij genetische algoritmes hebben we allereerst een een selectie (als eerste dus de random roosters). Hierop laten we twee typen veranderingen los om zo een grotere groep roosters te krijgen:

- Mutaties (gebaseerd op een van de rooster uit de selectie)
- Crossover (gebaseerd op twee van de roosters uit de selectie)

Bij crossover proberen we een gedeelte van het eerste rooster een gedeelte van het tweede rooster te laten gebruiken. Vervolgens kiezen we de roosters met de hoogste fitness en maken hiervan de nieuwe selectie. Dit proces wordt steeds herhaald, tot een bepaalde eis is gehaald (generatie,fitness,tijd)

Hill-Climbing

Hill climbing is ook een optimalisatiemethode. In ons geval beginnen we met een willekeurig rooster. We maken meerdere opvolgers met mutaties. Vervolgens kiezen we degene uit met de hoogste fitness: dit wordt nu het nieuwe rooster. We herhalen nu het kiezen van nieuwe roosters, totdat er geen betere meer worden gemaakt. Over het algemeen wordt er bij het genereren van roosters gebruik gemaakt van genetische algoritmes.

Local beam search

Local beam search is een heuristisch zoekalgoritme: het maakt gebruik van meerdere hill climbs. Hierdoor wordt er een zoekboom opgebouwd. We hadden het plan een aantal 'trucjes' van de "Genetic Algorithms" door ons programma te laten gebruiken. We begonnen natuurlijk met de meest voor de hand liggende en simpelste: de mutatie. Het idee van de mutatie in ons programma is het simpelweg omwisselen van 2 uren binnen het rooster van 1 klas. Dit heeft als enorme voordeel dat niet hoeft gecheckt te worden of de klas nog wel alle lessen volgt; en eigenlijk alleen maar even gecheckt hoeft te worden of die docent niet al iets doet op dat uur. Na dit model bedacht te hebben (alleen binnen 1 klas) zaten we wel opeens met het probleem dat een cross-over niet te doen is. Als je een rooster cross-overt met een andere klas binnen dezelfde school haal je vreselijk veel problemen op de hals: * Van beide klassen kloppen de lessen die ze volgen niet meer * Van beide klassen moet je van de helft van de week alle uren controleren of de docent niet dubbel ingeroosterd staat. Door de cross-over te doen met een onafhankelijk rooster; zoals bijvoorbeeld die van een andere school kun je het tweede probleem halveren: Je hoeft alleen nog maar te kijken of de docent van de ene klas klopt.

Dit klinkt waarschijnlijk nog heel simpel om te verbeteren, tot je beseft dat naast het detecteren ook het oplossen van het probleem erg wenselijk zou zijn. Je kunt nu zeggen "oplossen ? nee: we weigeren deze cross-over gewoon, en proberen het nog wel eens"; maar wij denken dat je dan niet erg ver gaat komen met je rooster: tegen die tijd had je ook alle roosters brute-force door kunnen rekenen. En we proberen nou juist efficienter te zijn dan alles doorrekenen door Genetic Algorithms te gebruiken. Het is veel idealer om in plaats van de cross-over opnieuw te proberen te kijken of je het probleem op kan lossen. Bij een mutatie was dat redelijk simpel; je kijkt

welke docent het conflict geeft, en kijkt wanneer in die week die docent wel vrij is. Bij een cross-over is dat veel moeilijker; omdat je moet controleren of alle lessen kloppen en of alle docenten kloppen, per veranderde les.

Vandaar dat ons programma met als enige actie de mutatie is blijven steken; omdat de complexiteit daarvan heel veel simpeler is. En we verwachten niet veel betere resultaten te halen met cross-over dan met mutaties.

5 Methode

We schrijven een nieuw programma dat zo goed mogelijke roosters kan genereren in Java. Het voordeel van het gebruik van Java, is er vrij eenvoudig objectgeörienteerd mee gewerkt kan worden. Iedereen kan bijvoorbeeld afzonderlijk aan eigen onderdelen werken. Een ander voordeel van Java is dat we vrij simpel modules kunnen toevoegen en verwijderen.

In deze fase van ons onderzoeksproject willen we gebruikmaken van enkele algoritmen om automatisch (willekeurig) gegenereerde roosters te optimaliseren. Om een goede, vaste, manier te hebben om de kwaliteit van een al dan niet geoptimaliseerd rooster te bepalen, leggen we van te voren enkele 'eisen' vast, waar we willen dat roosters aan voldoen:

- Roosters moeten zo min mogelijk tussenuren hebben
- Het liefst houden we het laatste uur vrij

Het resultaat en de methode waarmee dat resultaat verkregen is, willen wij vergelijken met de optimaliseermethoden en de daarbijhorende resultaten van vrije software. Voorbeelden hiervan zijn programma's zoals Tablix en FET. Na de resultaten van verschillende programma's een redelijk aantal maal vergeleken te hebben, zouden we een redelijk betrouwbare conclusie moeten kunnen trekken over welk algoritme het best presteert.

We willen dit onderzoeken door zelf een programma te schrijven dat roosters genereert, en ze vervolgens verbetert met behulp van enkele algoritmen; we doen dit met local beam search, waarbij we enkele hill climbs tegelijk uitvoeren. Om de effectiviteit van onze gebruikte algoritmen uit te testen, vergelijken we het resultaat (de door ons programma gegenereerde roosters) met een *open source*-programma dat exact dezelfde constraints stelt en naar dezelfde zaken kijkt voor het bepalen van de fitness. Aangezien we het programma Tablix, waar we aanvankelijk mee wilden werken, niet zover kregen dat het roosters genereerde, zijn we verder gaan zoeken, en uitgekomen op het programma FET. (http://www.lalescu.ro/liviu/fet/)

Om de prestatie van ons pakket te bepalen hebben we deze vergeleken met bestaande roostersoftware. We hebben gekozen voor FET omdat het gratis te verkrijgen is, ook is het eenvoudig om met behulp van een XML-bestand een zelfde rooster in te voeren. Om een goede vergelijking te maken moet je

zowel je eigen software als de software waarmee je vergelijkt dezelfde roosterproblemen laten oplossen. Het betreft hier een rooster met 6 klassen die elk 3 uur per week les hebben van 10 docenten. Per klas zijn dat dus 30 uur. Hier zitten geen beperkingen aan (zoals lokalen die nodig zijn voor een bepaalde les, blokuren, pauzes, enz.).

Om de gegenereerde roosters van beide programma's te vergelijken hebben we een fitnessfunctie ontworpen. De fitness-bepaling van een rooster gaat voor een groot deel aan de hand van het aantal tussenuren dat een rooster bevat. Hoe minder hoe beter. Verder tellen de vrije uren op maandag- en dinsdagochtend ook mee; een beetje uitslapen is immers nooit verkeerd na een zwaar weekend. De beoordeling hierop is wel heel licht in verhouding tot zwaarder wegende factoren, zoals tussenuren. De beoordelingen gaat als volgt: Een rooster begint met 0 punten.

- Per tussenuur gaat hier 10 punten vanaf
- Voor eerste of eerste 2 uren vrij krijg je 5 punten per uur. Maandag en dinsdag tellen dubbel.
- Begint je dag pas het 4e uur en gaat het door tot het 10e uur dan krijg je 20 minpunten.

6 FET probleem

Om een goede vergelijking te kunnen makten tussen ons programma en FET is het noodzakelijk dat beide programma's onder de zelfde condities werken. Dit gaf nogal wat problemen. Het maken van een goede invoer voor FET is moeilijker dan het uiteindelijke rooster. Allereerst ging het testen alleen op een Windows machine. Hoewel de source beschikbaar was hebben we het niet aan de praat gekregen op een niet Windows besturingssysteem. Uiteindelijk hebben we maar Windows binaries genomen. FET heeft een zeer groot assortiment aan constraints. Het nare van FET is dat ze allemaal ingesteld moeten worden. Ook al zijn ze niet van toepassing op je rooster moet je ze wel instellen. Het afstemmen van alle constraints op ons programma heeft veel tijd gekost. Beslissingen die je onbewust neemt in je programmacode moet je weer als constraints terug laten komen in FET. Ook zijn er een aantal mogelijkheden die we met opzet uit ons programma hebben gelaten maar wel in FET terug komen. Er zijn zeer veel mogelijkheden waarop een rooster gemaakt kan worden. De meeste instellingen blijken achteraf pas van toepassing op grote roosters met veel klassen en lessen. Uiteindelijk hadden ze een minimale invloed op ons product. Het vergelijken van de eindproducten moet handmatig en kost veel tijd. Elke keer als we een rooster uit FET kregen moesten we de fitnessfunctie handmatig uitvoeren. Doordat het om veel roosters ging was dit erg arbeidsintensief. De fitness functie van ons eigen programma werkte alleen op de array's in ons programma en hebben we niet geschikt gemaakt om een XML bestand van FET te analyseren.

Doordat we een groot aantal klassen hebben en er legio mogelijkheden zijn bij FET is het niet eenvoudig om een werkende invoer XML te maken. Na 2 keer alles aangepast te hebben hebben we maar een programmatje gemaakt dat de XML maakt. Dit maakt het aanpassen van een grote groep constraints eenvoudig omdat het allemaal lijsten zijn met eenvoudig te bepalen waardes.

7 Resultaten

Beide programma's hebben een rooster geproduceerd en hebben we geanalyseerd.

7.1 Resultaat eigen software

We hebben ons programma een willekeurig rooster laten genereren, die wel correct is. Vanaf dat punt zijn wij begonnen met willekeurig muteren van de roosters. We doen steeds 20 stappen, en doen per stap 3 mutaties. Als er na een stap een beter rooster is dan aan het begin nemen we dat rooster als nieuwe basis en gaan we vanuit daar de volgende 20 stappen doen. Als het binnen 20 stappen niet lukt om een verbetering te vinden gaan we er vanuit dat we een maximum hebben gevonden. Na wat onderzoek bleek dat ons programma geen (of in elk geval zeer weinig) last had van lokale maxima, dus hadden we een globaal maximum gevonden. (er waren soms op andere plekken wel maxima die net iets hoger waren, maar dit leek ons goed genoeg.) Volgens onze fitness score kreeg het eindrooster een score van ongeveer 65 punten.

7.2 Resultaat FET

RenD test

	Klas 1				
	Monday	**Tuesday**	**Wednesday**	**Thursday**	**Friday**
9.00 - 10.00	Vak 5 Docent 5	---	---	Vak 9 Docent 9	Vak 5 Docent 5
10.00 - 11.00	Vak 4 Docent 4	Vak 4 Docent 4	---	Vak 7 Docent 7	Vak 3 Docent 3
11.00 - 12.00	---	---	Vak 8 Docent 8	---	---
12.00 - 13.00	Vak 9 Docent 9	Vak 1 Docent 1	---	Vak 10 Docent 10	---
13.00 - 14.00	Vak 2 Docent 2	Vak 5 Docent 5	---	---	Vak 4 Docent 4
14.00 - 15.00	Vak 8 Docent 8	---	Vak 10 Docent 10	---	---
15.00 - 16.00	---	Vak 1 Docent 1	Vak 9 Docent 9	Vak 6 Docent 6	Vak 7 Docent 7
16.00 - 17.00	Vak 1 Docent 1	Vak 2 Docent 2	Vak 10 Docent 10	Vak 3 Docent 3	---
17.00 - 18.00	---	Vak 6 Docent 6	Vak 8 Docent 8	Vak 3 Docent 3	Vak 7 Docent 7
18.00 - 19.00	Vak 2 Docent 2	---	Vak 6 Docent 6	---	---
	Timetable generated with FET 5.9.4 on 21/06/2009 22:58				

Elke cel in de bovenstaande tabel representeert een lesuur. Cellen waar enkel '—' in staat, zijn tussenuren. Zoals te zien in bovenstaand FET rooster komen er zeer veel tussenuren in voor. Dit veroorzaakt eeen zeer slechte score bij de fitness.

8 Conclusie

8.1 Waarom het verschil

Het rooster dat gemaakt is door FET haalt een zeer lage score. Waarschijnlijk komt dit doordat deze software alleen maar een passend rooster genereert en daarna stopt. Er vindt totaal geen roosteroptimalisatie plaats. Daarentegen stopt ons programma niet op dat moment, maar gaat verder zoeken naar betere roosters. Hierdoor zijn er roosters mogelijk die veel prettiger zijn. We gaan er dan wel vanuit dat veel tussenuren niet als positief ervaren worden.

8.2 Waardoor komt het verschil?

FET is vooral goed in het verwerken van constraints zoals het koppelen van lokalen aan lessen die alleen plaats kunnen vinden in dat lokaal. FET kan een grote hoeveelheid aan variabelen aan; zo kun je ook nog docenten aan een lokaal koppelen. Ook het leeg houden van lokalen gedurende een bepaalde periode is mogelijk. Het programma is vooral geoptimaliseerd om een foutloos rooster te genereren. Optimaliseren is totaal niet aan de orde.

Ons programma maakt een rooster zonder de mogelijkheid tot constraints en gaat daarna nog een keer kijken of het niet beter kan. Het gebruikte *hill climbing*-algoritme zal dan ook niet misstaan in FET. Hiermee kunnen significant betere roosters gemaakt worden dan nu het geval is.

8.3 Wat zou beter kunnen?

Hoewel we een beter rooster hebben, is het waarschijnlijk dat de criteria voor een 'goed' rooster bij de programma's verschillend zijn. Wij hebben ons programma geoptimaliseerd voor compacte en prettig te volgen roosters. FET daarentegen is gemaakt om complexe roosterproblemen op te lossen, waar het ook goed in slaagt - alleen lijkt het een essentieel deel, tussenuren reduceren, over te slaan. Dit is relatief eenvoudig te realiseren met *hill climbing*. Beide programma's hebben een zeer sterke punten. FET is geschikt om complexe roosters te realiseren. Ons programma om roosters compacter te maken. We zijn er van uit gegaan dat alle software dit zou doen. Maar hoe meer we er mee werken, hoe minder goed FET dat kan. In een ideale wereld zou je beide programma's moeten combineren. FET lost de complexe problemen op, ons programma maakt het rooster compact. Je zou dus de uitvoer van FET aan

ons programma moeten geven. Alleen was het te complex om dit probleem op te lossen.

Mogelijk hebben we een probleem verzonnen in roosters maken dat nog niet was uitgewerkt in software. De fitnessfunctie was ook volledig gemaakt om dit probleem te beoordelen. Het al dan niet realiseren van een rooster hebben we niet beoordeeld. Hierdoor is het vergelijken van roosters niet geheel eerlijk verlopen.

Het onderzoek heeft hierdoor een ander resultaat gekregen dan we hadden verwacht. We hebben ons niet zo zeer gericht op het maken van roosters maar vooral op optimalisatie.

9 Discussie

Het was erg jammer dat we onze tool niet op tijd af konden krijgen, was dit gelukt hadden we een betrouwbaardere vergelijking kunnen maken omdat we dan zowel hill climbing en genetic algorithms met exact hetzelfde rooster hadden kunnen laten beginnen. Ook zou de fitness identiek zijn, zouden we ons programma vergelijken met een ander programma dan zou onze fitness een benadering zijn van de fitness van het andere programma of andersom. We hebben ontdekt dat het toepassen van crossover lastig is met rooster omdat we ook rekening moeten houden met het feit dat er geen overlappingen van roosters mogen zijn. Dit zou een reden kunnnen zijn om hill climbing te gebruiken in plaats van genetic algorithms.
Wat we mogelijk ook hadden kunnen doen was, in plaats van zelf een programma maken, er een of twee opzoeken en deze met elkaar vergelijken, waarbij elk programma een ander algorithme gebruikt, maar wel dezelfde fitness.
Wij hadden ons programma vergeleken met FET. ons programma bleek het beter te doen. Ons programma richte zich vooral op het verbeteren van een rooster. FET was vooral bedoelt om een kloppend rooster te maken, en verbeterde daarna het rooster maar weinig. Ons programma zou eigelijk vergeleken moeten worden met een programma dat zich ook voornamelijk richt op het optimaliseren van een rooster. Helaas hebben niet een programma gevonden dat we aan de praat kregen en dat zich voornamelijk richte op optimalisatie. We hadden mischien een programma gevonden als we al in fase 1 waren begonnen met het zoeken naar een dergelijk programma of ons niet hadden beperkt tot open-source programma's. Op dat moment dachten we dat we onze tool op tijd af zouden hebben en zagen we het vergelijken met een bestaand programma als een extra, waar we onze resultaten nog een keer konden controleren.

10 Literatuur

1. Stuart Russell Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall/Pearson Education, USA, 2003

2. Deitel Deitel, *Java How to Program, 7th Edition*, Prentice Hall, USA, 2007