Personeelsplanning en Kolomgeneratie

BWI Werkstuk

Annemieke van Dongen

Vrije Universiteit Amsterdam
Faculteit der Exacte Wetenschappen
De Boelelaan 1081a
1081 HV Amsterdam

Amsterdam, 1 december 2005

Begeleider: Sandjai Bhulai

Inhoudsopgave

Voorwoord

Een onderdeel van de opleiding Bedrijfswiskunde en Informatica aan de Vrije Universiteit is het schrijven van het BWI-werkstuk. Het doel van het werkstuk is dat de student voor een deskundige manager op een heldere wijze een probleem beschrijft.

Dit is mijn BWI werkstuk, dat gaat over personeelsplanning en kolomgeneratie. Het eerste hoofdstuk gaat over personeelsplanning in het algemeen, wat de verschillende toepassingsgebieden van personeelsplanning zijn, en de meest gebruikelijke oplossingstechnieken. Het tweede hoofdstuk gaat over kolomgeneratie, de geschiedenis, het algoritme en de nadelen. In het derde hoofdstuk geef ik een voorbeeld van hoe kolomgeneratie kan worden gebruikt om een personeelsplanningsprobleem op te lossen. Ik genereer een rooster voor een call center. Tot slot geef ik in het laatste hoofdstuk mijn conclusies.

Graag wil ik Sandjai Bhulai bedanken voor zijn begeleiding. Zijn enthousiasme heeft het schrijven van dit werkstuk voor mij een stuk leuker gemaakt.

Annemieke van Dongen
November 2005

Samenvatting

Personeelsplanning is een heel breed gebied, er zijn veel ingewikkelde problemen, met verschillende oplossingstechnieken. Kolomgeneratie is daar een van. Kolomgeneratie kan gebruikt worden om zeer grote LP-problemen op te lossen.

Een kolom staat voor een mogelijke shift of een werkrooster. Het kolomgeneratie algoritme is gebaseerd op het simplex algoritme, alleen worden niet alle kolommen in het geheugen bewaard, maar worden alleen de kolommen gegenereerd die nuttig zijn. Kolomgeneratie is dus vooral handig als er heel veel mogelijkheden zijn voor roosters, en het vinden van de beste roosters de bedoeling is.

Omdat roosterproblemen van nature geheeltallig zijn (het gaat over mensen), zijn dit geen LP, maar IP problemen. Er zijn verschillende manieren om kolomgeneratie toch op IP problemen toe te passen, er wordt dan een combinatie gemaakt met een branch-and-bound algoritme.

Er wordt een voorbeeld behandeld van hoe een personeelsrooster voor een call center gemaakt kan worden met behulp van kolomgeneratie. Er wordt een heuristiek gebruikt om kolomgeneratie op dit geheeltallige probleem toe te passen. Het resultaat ziet er zo goed uit dat het waarschijnlijk ook optimaal is.

De conclusie is dat kolomgeneratie een doeltreffende methode is om bepaalde problemen op te lossen, maar zeker geen wondermiddel; kolomgeneratie kan niet zomaar op elk probleem toegepast worden, de methode van kolommen genereren is probleemspecifiek en kan erg ingewikkeld zijn.

Hoofdstuk 1

Personeelsplanning

In dit hoofdstuk wordt kort een beschrijving gegeven van wat personeelsplanning inhoudt en welke stappen gevolgd moeten worden. Vervolgens wordt een indeling gemaakt van verschillende toepassingsgebieden van personeelsplanning en de meest gebruikelijke oplossingsmethoden. Voor dit hoofdstuk is gebruik gemaakt van Ernst [5].

1.1 Inleiding

Bij personeelsplanning moet er gezorgd worden dat de juiste mensen op de juiste tijd op de juiste plaats zijn om te werken. Dit is vaak een ingewikkeld proces. Er moeten genoeg werknemers aanwezig zijn om al het werk te doen en het service level te halen. Daarom moet allereerst bepaald worden hoeveel mensen met welke capaciteiten er nodig zijn op welke tijden, terwijl dit niet altijd van te voren vastligt. Vervolgens moeten er 'shifts' bedacht worden, zodat er genoeg mensen aan het werk zijn op elk tijdstip, maar ook zodat de werknemers redelijke werktijden hebben en niet te kort of te lang achter elkaar moeten werken. Dan moeten er nog werknemers aan de shifts gekoppeld worden, waarbij rekening gehouden moet worden met de wensen van individuele werknemers. Ondertussen moet rekening gehouden worden met allerlei zaken die per bedrijf of bedrijfstak verschillen.

Zoals gezegd, is het vaak erg ingewikkeld om een goed rooster te construeren

en het is nog ingewikkelder om een optimaal rooster te construeren, dat de kosten minimaliseert, rekening houdt met voorkeuren van werknemers, shifts eerlijk verdeelt over werknemers en bovendien rekening houdt met alle andere regels van de bedrijfstak.

Daarom maken de meeste personeelsplanners de roosters niet met de hand, maar maken zij gebruik van decision support systems. Maar, omdat er grote verschillen zijn tussen roostersproblemen voor verschillende bedrijfstakken, is er niet een systeem dat voor iedereen een ideaal rooster kan genereren.

1.2 Het roosteringsproces

In deze paragraaf wordt het roosterproces in het algemeen beschreven. Aan bod komt welke problemen er allemaal zijn en wat de verschillen zijn tussen de verschillende roosterproblemen.

1.2.1 Componenten

In deze paragraaf worden de verschillende componenten van personeelsplanning toegelicht. Deze componenten kunnen vaak niet stap voor stap worden uitgevoerd en zullen elkaar gedeeltelijk overlappen. Ook zullen sommige componenten bij bepaalde bedrijfstakken geen rol spelen. Dit is dus geen standaard stappenplan dat zo uitgevoerd kan worden, maar dient om te illustreren welke problemen er allemaal opgelost moeten worden, alvorens een rooster gemaakt kan worden.

Vraag bepalen
Bij dit eerste component moet bepaald worden hoeveel werknemers nodig zijn op verschillende tijdstippen. Hierbij zijn globaal twee situaties te onderscheiden.

Op taken gebaseerde vraag. In deze situatie liggen de taken die uitgevoerd moeten worden van te voren vast. Bij elke taak moet worden vastgesteld hoe lang er aan gewerkt moet worden, wanneer deze taak moet gebeuren, welke capaciteiten er voor nodig zijn en soms ook op welke plaats dat moet gebeuren. Uit alle taken moet bepaald worden hoeveel mensen, met welke

capaciteiten, wanneer, nodig zijn. Bijvoorbeeld roosters voor docenten; het aantal lessen dat gegeven moet worden ligt vast. Voor elke les moet dan worden bepaald welke docenten die les zouden kunnen geven en hoe lang die les duurt. Vervolgens moet een rooster gemaakt worden waarbij elke les gegeven wordt door iemand die daar bevoegd voor is.

Flexibele vraag. Bij flexibele vraag liggen de taken die uitgevoerd moeten worden niet van tevoren vast, maar worden op willekeurige tijdstippen service aanvragen gedaan. Dit komt typisch voor bij call centers, waarbij nooit vooraf bekend is hoeveel mensen gaan bellen. Hier moet dus vooraf geschat worden hoeveel service aanvragen er zullen zijn. Deze schatting moet vervolgens worden omgezet naar shifts, die alle service aanvragen binnen een redelijke tijd afhandelen, zonder dat er al te veel wachttijd ontstaat. In het geval van een call center kan de schatting van de vraag en dus ook het gewenste aantal werknemers per uur verschillen. Het is echter niet gewenst om werknemers voor slechts een uur werk op te roepen, dus is het soms gunstig om op sommige rustige tijdstippen tijdelijk het service level net niet te halen en dit op andere momenten goed te maken met extra werknemers om een extra hoog service level te halen, zodat gemiddeld over de dag het service level wel gehaald wordt.

Shifts bepalen
Hier moeten 'shifts' bedacht worden, zodat er genoeg mensen aan het werk zijn op elk tijdstip, maar ook zodat de werknemers redelijke werktijden hebben. Bij flexibele vraag moet ook rekening gehouden worden met lunchpauzes. Bij op taken gebaseerde vraag is het bepalen van shifts het vormen van goede groepen taken, die door personen uitgevoerd kunnen worden, zodat de vraag gedekt wordt.

Werkroosters construeren
Hier worden de werkroosters gemaakt tot aan de planningshorizon. Hoe dit gebeurt hangt af van de bouwstenen die er zijn. Dit kunnen shifts of taken zijn.

Wanneer de bouwstenen shifts zijn, kan elke shift een werkdag voor een persoon worden, er moet echter wel rekening gehouden worden met extra voorwaarden, bijvoorbeeld een nachtshift mag niet direct gevolgd worden door een dagshift.

Inroosteren van vrije dagen

Dit component bevat de bepaling hoe er moet worden om gegaan met vrije dagen van werknemers. Dit probleem komt vaker voor bij een flexibele vraag dan wanneer de vraag op taken is gebaseerd.

Taken toewijzen
Dit is vaak nodig als shifts zijn bepaald, maar taken nog niet zijn toegewezen aan individuele mensen. Taken worden gegroepeerd en aan shifts toebedeeld op grond van starttijd en duur. Hoe dit gebeurt hangt af van of taken vast staan of verplaatsbaar zijn, of er pauzes zijn in shifts, of overwerken is toegestaan en of specifieke capaciteiten nodig zijn. Taken toewijzen kan ook al gebeuren tijdens het construeren van de werkroosters.

Personeel toewijzen
Uiteindelijk moeten aan de gevormde shifts, of rijen aan elkaar gekoppelde shifts, ook nog personen gekoppeld worden die de juiste capaciteiten hebben. Ook dit gebeurt vaak al tegelijk met het construeren van de werkroosters.

1.2.2 Verschillen tussen roosterproblemen

Zoals gezegd vormen bovengenoemde componenten geen standaard stappenplan. Keuzes die bij een eerdere stap gemaakt worden kunnen heel nadelig zijn voor een latere stap. Voor een rooster dat het beste is, zou men dus het beste alle componenten tegelijk oplossen. Dit is echter vaak veel te ingewikkeld, dus is het toch handiger om het roosterproces te verdelen in deelproblemen. In veel gevallen is het bepalen van de vraag een probleem dat makkelijk van het gehele proces is los te koppelen, zonder dat daarbij het aantal mogelijke oplossingen beperkt wordt. De mate waarin het mogelijk is de verschillende componenten van het roosterproces los te koppelen verschilt tussen roosterproblemen en geeft dus ook totaal verschillende oplossingstechnieken.

Een ander groot verschil tussen roosterproblemen dat ook een groot verschil in oplossingstechnieken geeft, is welke componenten allemaal een rol spelen. Bijvoorbeeld, bij een luchtvaartmaatschappij ligt het aantal piloten per vlucht vast en kan het aantal benodigde piloten direct uit het vluchtrooster verkregen worden. Elke vlucht is een shift, dus de shifts liggen van tevoren vast; het component ‘shifts bepalen’ speelt dus geen rol.

Een ander groot verschil is hoe de vraag bepaald wordt. Het maken van een werkrooster voor een call center met een flexibele vraag, vraagt heel andere technieken dan het maken van een werkrooster voor een bedrijf met een op taken gebaseerde vraag.

1.3 Verschillende toepassingsgebieden

In deze paragraaf staat beschreven welke verschillende toepassingsgebieden van personeelsplanning er zijn en hoe ze veelal aangepakt worden. Het karakteristieke kenmerk van het toepassingsgebied staat cursief gedrukt.

1.3.1 Vervoersbedrijven

De belangrijkste toepassing binnen de vervoers-sector is de luchtvaart. Karakteristieke kenmerken van de vervoers-sector zijn:

1. *Niet alleen tijd, maar ook plaats speelt een rol. Taken worden niet alleen bepaald door een begin- en eindtijd, maar ook door een begin- en eindplaats.* Er moet dus bijvoorbeeld gezorgd worden dat de verschillende vluchten van een piloot op elkaar aansluiten.

2. Alle taken worden verkregen uit een rooster, bijvoorbeeld een vluchtrooster of spoorboekje. Een taak is dan een vlucht of een treinrit tussen 2 stations. Er is dus sprake van *op taken gebaseerde vraag.*

De meest populaire oplossingsmethode is het probleem oplossen in 3 stappen:

1. Het genereren van alle, of een groot aantal mogelijke trips. Bijvoorbeeld een rij vluchten, die mogelijk achter elkaar door dezelfde piloot gevlogen kunnen worden.

2. Het optimaliseren van deze trips. De trips met de minste kosten worden gekozen, zodat alle taken verdeeld zijn.

3. Het verdelen van trips over personen.

1.3.2 Call centers

Het karakteristieke kenmerk van call centers is dat *de hoeveelheid werk niet van te voren bekend is*. Deze moet geschat worden. Verder is er vaak *een service level dat gehaald dient te worden*, bijvoorbeeld 80% van de bellers moet binnen 20 seconden een medewerker aan de lijn hebben. Het kan zijn dat dit service level gemiddeld over de dag gehaald moet worden, of bijvoorbeeld in elk uur. Het aantal bellers kan per uur flink verschillen, daarom moeten shift-lengtes en starttijden verschillen, zodat er goede, kosteneffectieve oplossingen verkregen kunnen worden, die het service level halen. Soms is het daarbij noodzakelijk om op sommige tijden minder mensen aan het werk te hebben, zodat het service level tijdelijk niet gehaald wordt, en dit op andere tijdstippen goed te maken met extra mensen, zodat het service level gemiddeld goed genoeg is.

De meest populaire methode die gebruikt wordt om te bepalen hoeveel mensen er nodig zijn, is met behulp van het Erlang-C wachtrij model. Wachtrijmodellen geven meestal analytisch goede oplossingen, maar vaak moeten wel wat zaken vereenvoudigd worden. Met simulatie kunnen wel veel practische zaken meegenomen worden, maar simulatie kost vaak veel rekentijd. Daarom worden soms analytische methoden en simulatie gecombineerd om te bepalen hoeveel mensen nodig zijn.

Als het aantal benodigde mensen bekend is, moeten er nog shifts en roosters bepaald worden. Er zijn ook call centers waar mensen met verschillende vaardigheden, verschillende telefoontjes behandelen. Dit brengt extra voorwaarden voor het rooster met zich mee, naast de gebruikelijke voorwaarden, zoals openingstijden en maximale shift-lengte. Onder deze voorwaarden kan het aantal mogelijke shifts nog enorm zijn. Het probleem is dan de juiste werkroosters te construeren, met de juiste shifts, zonder te veel kosten. Dit wordt meestal opgelost met technieken uit de Mathematische programmering of met heuristieken.

1.3.3 Gezondheidszorg

Bij de gezondheidszorg gaat het bijvoorbeeld om het inroosteren van verpleegkundigen. Werkroosters moeten ervoor zorgen dat er genoeg werknemers met de juiste opleiding aan het werk zijn om de patienten die op de

afdeling liggen te verzorgen, zodanig dat de nacht en weekend *shifts eerlijk verdeeld worden*. De personeelsplanning in de gezondheidszorg heeft vaak te maken met *veel voorwaarden*. Er moeten bijvoorbeeld altijd plaats, apparatuur en medewerkers beschikbaar zijn voor spoedgevallen; medisch personeel mag niet een dag- en nachtshift achter elkaar draaien; het moet voor het personeel mogelijk zijn om dagen vrij te nemen.

1.3.4 Nood- en beveiligingssystemen

Bij het plannen van het personeel van de politie, ambulance en brandweer is het heel belangrijk dat er aan een *bepaald service level* voldaan wordt. Dit kan bijvoorbeeld een bepaalde maximale responstijd zijn. Ook typisch voor deze nooddiensten is dat de *vraag voor diensten per dag, week of seizoen sterk kan verschillen*. In uitgaansgebieden zal bijvoorbeeld vaker een beroep worden gedaan op de politie op zaterdagavond dan op maandagochtend en in toeristische plaatsen zijn er meer aanvragen tijdens vakanties.

1.3.5 Overheid

De overheid is een zeer grote werkgever en dus een breed toepassingsgebied van de personeelsplanning. Een goede personeelsplanning hier zou de overheidsservices een stuk kunnen verbeteren. Binnen de overheid zijn er weer vele verschillende toepassingsgebieden, zoals de belastingdienst, het leger of een bibliotheek.

1.3.6 Financiële services

De personeelsplanning bij banken lijkt op die van andere service bedrijven. De grootste moeilijkheid is de *variabele vraag over de dag*. Dit kan worden opgelost met parttime medewerkers, om flexibiliteit te creëren, en door over te werken.

1.3.7 Gastvrijheid en toerisme

In hotels zijn personeelskosten een groot deel van de totale kosten. Daarom kan een kleine verbetering in de personeelsplanning al een groot kostenvoordeel met zich meebrengen. In hotels werken *veel mensen met verschillende vaardigheden*, zoals bijvoorbeeld catering, huishouden, receptie of boekhouden. Sommige werknemers hebben meerdere van deze vaardigheden, andere niet. Er moet 24 uur per dag gewerkt worden en de hoeveelheid werk ligt nooit precies van te voren vast. Dit alles maakt personeelsplanning een belangrijk deel van het management in hotels.

1.3.8 Detailhandel

Het maken van een rooster voor een detailhandel kan op dezelfde manier gebeuren als voor een call center. De klanten kunnen gezien worden als bellers en de kassamedewerkers als telefonisten.

1.3.9 Productie

Bij productiebedrijven moet bepaald worden hoeveel personeel per dag nodig is om alle taken te vervullen, zodat de over-all kosten minimaal zijn.

1.4 Oplossingstechnieken

In deze paragraaf worden de meest belangrijke oplossingstechnieken voor personeelsproblemen behandeld.

1.4.1 Constraint programming

Constraint programming is een krachtige methode voor het vinden van een toegelaten oplossing voor een roosterprobleem. Deze techniek is dus voornamelijk bruikbaar als er veel voorwaarden zijn aan het probleem en het

vinden van een bruikbare oplossing al genoeg is, ook al is deze niet optimaal. Deze methode geeft minder goede oplossingen als het vinden van een optimale of bijna optimale oplossing de bedoeling is.

1.4.2 Mathematisch programmeren

Bij technieken uit de Mathematische programmering wordt naar een oplossing met de laagste kosten gezocht, onder een aantal voorwaarden. **Kolomgeneratie** valt onder deze categorie technieken. Deze technieken zijn meestal in staat om een optimale oplossing te vinden, maar het is vaak moeilijk om alle problemen van een roosterprobleem te modelleren, zodat het probleem vaak vereenvoudigd moet worden.

1.4.3 (Meta)heuristieken

Heuristieken zijn strategieën om een goede oplossing te vinden voor een probleem. Ze vinden niet gegarandeerd een optimale oplossing, maar zijn vaak wel robuust: ze kunnen vaak een redelijk goede oplossing vinden voor een groot aantal problemen binnen een redelijke tijd. Ook zijn ze vaak makkelijk te implementeren. Er is sprake van een metaheuristiek als er meerdere heuristieken gecombineerd zijn.

Hoofdstuk 2

Kolomgeneratie

Kolomgeneratie wordt veel toegepast bij het oplossen van roosterproblemen. Het is een techniek die grote lineaire programmeringsproblemen kan oplossen. Lineaire programmering (LP) is een van de meest in de praktijk toegepaste wiskundige methoden. Bij LP wordt de optimale waarde bepaald van een lineaire criteriumfunctie van verschillende beslissingsvariabelen. Deze variabelen moeten voldoen aan een aantal lineaire bijvoorwaarden. Kolomgeneratie kan grote LP-problemen oplossen zonder het hele probleem in een keer te beschouwen. Het LP-probleem in de plaats heeft relatief weinig variabelen, waardoor het kleiner is en makkelijker op te lossen. Het werkt bijzonder goed met set-covering en set-partitioning problemen, omdat deze gestructureerde kolommen hebben.

In dit hoofdstuk wordt eerst kort de geschiedenis van kolomgeneratie behandeld, vervolgens wordt de theorie uitgelegd en het algoritme gegeven. Als laatste wordt de toepassing van kolomgeneratie op personeelsplanning toegelicht.

2.1 Geschiedenis

De geschiedenis van de lineaire programmering begint in de jaren 40, wanneer er op grote schaal wetenschappelijke methoden gebruikt worden om allerlei logistieke problemen op het militaire vlak te analyseren. Na de Tweede

Wereldoorlog realiseerde men zich dat de planningsmethoden, die voor militaire doeleinden ontwikkeld waren, ook bruikbaar waren voor allerlei planningsproblemen in het bedrijfsleven. Het onderzoek op dit gebied bloeide op en leidde in 1947 tot het uitvinden van de simplex methode door George B. Dantzig. De simplex methode wordt tot op de dag van vandaag gebruikt om LP-problemen op te lossen.

De geschiedenis van de kolomgeneratie begint bij Ford en Fulkerson, die in 1958 een algoritme publiceerden voor het 'meerdere goederen stroomprobleem'. Bij dat probleem gaat het er om zo veel mogelijk goederen door een netwerk van bron naar put te brengen. Een toepassing hiervan is bijvoorbeeld een groot communicatienetwerk waarbij verschillende berichten tegelijkertijd door het netwerk moeten kunnen, vanuit verschillende bronnen en naar verschillende eindpunten. In het algoritme van Ford en Fulkerson werden de variabelen slechts impliciet behandeld en zo ontstond het idee van kolomgeneratie. Dantzig en Wolfe (1960) gingen verder met dit fundamentele idee; ze ontwikkelden een strategie om een lineair programma kolomsgewijs uit te breiden. Kolomgeneratie werd als eerst echt gebruikt door Gilmore en Gomory (1961,1963) als deel van een efficiënt heuristisch algoritme om het 'cutting stock' probleem op te lossen. Dit is een probleem waarbij, bijvoorbeeld, rollen papier zo efficient mogelijk in stukken gesneden moeten worden. Kolomgeneratie is tegenwoordig een prominente methode om het hoofd te bieden aan een reusachtig aantal variabelen. Het invoegen van de technieken van de kolomgeneratie in een, op lineaire programmering gebaseerd, branch and bound raamwerk, werd door Desrosiers, Soumis en Desrochers (1984) geïntroduceerd. Ze gebruikten dit voor het oplossen van een vehicle routing probleem, onder tijdsbeperkingen. Dit was de belangrijkste stap in het ontwerp van nauwkeurige algoritmen voor een grote klasse van geheeltallige programma's.

2.2 Theorie

Kolomgeneratie is gebaseerd op de simplex methode, dus hier volgt eerst een korte uitleg van de simplex methode, daarna volgt de uitleg van het kolomgeneratie algoritme.

2.2.1 (Herziene) simplex-methode

Beschouw het volgende standaard lineair programmeringsprobleem

$$\begin{array}{ll} \max & cx \\ \text{onder} & Ax = b \\ & x \geq 0, \end{array}$$

met A een m x n matrix, b een m-dimensionale vector, c een n-dimensionale vector en x de n-dimensionale vector met beslissingsvariabelen. Dit model kan als volgt worden opgelost met de simplex methode [4].

1. Vind een toegestane basisoplossing van het probleem, waarin m beslissingsvariabelen groter dan 0 zijn (basisvariabelen) en $n - m$ beslissingsvariabelen gelijk aan 0 zijn. Laat B de m x m matrix zijn met daarin de kolommen van A die horen bij de m basisvariabelen. Dan is $x_B = B^{-1}b$ de vector met alle basisvariabelen en c_B de vector met alle items van c die horen bij de basisvariabelen. De duale variabelen zijn dan $y = c_B B^{-1}$.

2. Bereken de gereduceerde kosten van alle niet-basisvariabelen j met de formule $yA_j - c_j$.

3. De oplossing is optimaal als de gereduceerde kosten van alle niet-basisvariabelen niet-negatief zijn. STOP het algoritme als dit het geval is.

4. Maak uit matrix B een B_{nieuw} die precies een kolom verschilt van B. Breng een kolom A_s in B_{nieuw}, waarbij A_s een kolom uit A is, behorende bij een niet-basisvariabele met negatieve gereduceerde kosten. De kolomvector die uit B moet worden verwijderd is de vector A_l, waarbij de index l gelijk is aan de index waarvoor het minimum wordt bereikt,
$l = \text{argmin}_{1 \leq i \leq m} \left\{ \frac{(B^{-1}b)_i}{(B^{-1}A_s)_i} | (B^{-1}A_s)_i > 0 \right\}$.
Ga weer naar stap 2 met de nieuwe basismatrix.

2.2.2 Kolomgeneratie

Beschouw wederom het standaard lineair programmeringsprobleem

$$\begin{array}{ll} \max & cx \\ \text{onder} & Ax = b \\ & x \geq 0, \end{array}$$

en veronderstel dat n zeer groot is en m klein. Er zijn dus veel beslissingsvariabelen en weinig restricties. Dus de matrix A heeft zeer veel kolommen. De essentie van kolomgeneratie is hetzelfde als die van de simplex methode. De oplossing wordt steeds iteratief verbeterd door te kijken naar de gereduceerde kosten van niet-basisvariabelen en door kolommen met negatieve gereduceerde kosten toe te voegen aan de basis. Dit gaat door totdat er geen niet-basisvariabelen meer zijn met negatieve gereduceerde kosten en dan is de oplossing optimaal.

Als het aantal kolommen klein is, dan kunnen alle kolommen in het geheugen worden bewaard en kan bij elke iteratie van elke kolom de gereduceerde kosten worden berekend. Echter, als er erg veel kolommen zijn, is het beter om deze dynamisch te genereren tijdens het uitvoeren van het algoritme.

Dit geeft het volgende algoritme voor kolomgeneratie [3]:

1. Kies een kolommenverzameling N_1; $k = 1$.

2. Vorm een matrix D uit de kolommen $(A_i : i \in N_k)$. Eis dat rang(D)=m. Laat verder de vector $c_{N_k} = (c_i : i \in N_k)$ en idem voor x. Dan wordt het beperkte LP-probleem:

   $$\begin{array}{ll} \max & c_{N_k} x_{N_k} \\ \text{onder} & D x_{N_k} = b \\ & x_{N_k} \geq 0. \end{array}$$

   Los deze op met de (herziene) simplex methode.

3. Als er een kolom is met negatieve gereduceerde kosten, ga naar stap 4; anders STOP.

4. Vervang N_k door een N_{k+1}; $k = k + 1$; ga naar stap 2.

Het cruciale deel hier is natuurlijk het virtueel doorzoeken van de niet basiskolommen naar kolommen met negatieve gereduceerde kosten. Dit lijkt erg lastig, maar is toch mogelijk doordat de kolommen van A een bepaalde betekenis hebben, ze staan voor bijvoorbeeld routes, mogelijke werkroosters, of verzamelingen. Dan zijn A en de kosten gedefinieerd op deze structuren en zijn we voorzien van waardevolle informatie over hoe mogelijke kolommen er uit zien.

Beschouw bijvoorbeeld het 'cutting stock' probleem, een klassiek voorbeeld van kolomgeneratie van Gilmore en Gomory: Er zijn papieren rollen van lengte L, en deze kunnen gesneden worden tot meerdere kortere rollen. Er is een vraag d_i naar rollen van lengte i. Het doel is het snijden van de rollen in kortere rollen waar vraag naar is, zodat er zo min mogelijk rollen versneden moeten worden om aan de vraag te voldoen. Dan kunnen de rollen van lengte L in verschillende patronen j gesneden worden. Bijvoorbeeld een rol van $L = 6$ meter kan gebruikt worden om een stuk van 2 meter en een stuk van 4 meter uit te halen, of twee stukken van 3 meter, of een stuk van 5 meter. Dit kan erg veel mogelijkheden opleveren. Dan kan een matrix A gemaakt worden met a_{ij} het aantal papieren rollen van lengte i dat ontstaat als een rol in patroon j gesneden wordt. A is dus een matrix met zeer veel kolommen.

Om ervoor te zorgen dat zo min mogelijk rollen versneden worden kan het volgende model gebruikt worden:

x_j = het aantal keer dat snijpatroon j gebruikt wordt.

$$\begin{array}{ll} \min & \sum_j x_j \\ \text{onder} & Ax \geq d \\ & x_j \in \{0, 1, 2, \ldots\}. \end{array}$$

Hier kunnen we dus kolomgeneratie op toepassen. Als we een toegestane oplossing hebben met bijbehorende duale variabelen y, dan moeten we van de virtuele niet-basiskolommen de gereduceerde kosten berekenen. Deze zijn $yA_j - 1$. Omdat dit een minimaliseringsprobleem is, moeten kolommen met positieve gereduceerde kosten aan de basis toegevoegd worden om de oplossing te verbeteren, in plaats van kolommen met negatieve gereduceerde

kosten. We zijn dus alleen geinteresseerd in de kolom met de grootste gereduceerde kosten, want als deze negatief is, dan hebben we de optimale oplossing gevonden, is deze positief, dan moeten we deze kolom toevoegen aan de basis. De kolom j met de grootste gereduceerde kosten kan in dit geval bepaald worden met een 'knapsack probleem':

z_i = het aantal stukken van lengte i dat ontstaat als volgens dit patroon gesneden wordt.

$$\begin{array}{ll} \max & yz - 1 \\ \text{onder} & \Sigma_i i z_i \leq L \\ & z_i \in \{0, 1, 2, \ldots\}. \end{array}$$

De oplossing z van dit probleem vormt een nieuwe kolom A_j in het hoofdprobleem als $yz - 1$ positief is.

Bij dit voorbeeld hebben de kolommen dus de betekenis van een snijpatroon. Deze interpretatie wordt gebruikt bij het 'knapsack' probleem bij de restrictie $\Sigma_i i z_i \leq L$. Dus doordat de structuur van de kolommen bekend is, kunnen kolommen met positieve gereduceerde kosten gegenereerd worden en worden kolommen die ongunstige resultaten geven buiten beschouwing gelaten.

Eitzen [2] is een ander voorbeeld van hoe kolommen dynamisch gegenereerd worden. Daar gaat het om een roosterprobleem met meerdere vaardigheidslevels. De kolommen van de restrictiematrix A staan daar voor mogelijke roosters voor twee weken, opeenvolgende middag-, avond- of nachtshifts op bepaalde vaardigheidslevels en vrije dagen. Deze kolommen moeten ook aan bepaalde voorwaarden voldoen: er mag geen middag- of avondshift op een nachtshift volgen, een fulltime medewerker moet altijd 5 dagen achtereenvolgens werken, als een werknemer op zaterdag werkt moet hij ook op zondag werken, enzovoorts. Om dynamisch kolommen te genereren gebruikt Eitzen een netwerk waarin een toegestaan pad door het netwerk staat voor een toegestaan werkrooster, dus een kolom. Elk punt in het netwerk correspondeert met een dag, shift en vaardigheidslevel combinatie, en elke lijn tussen twee dag, shift en vaardigheidslevel combinaties betekent dat deze shifts door een persoon achtereenvolgens mogen worden gewerkt. Eitzen gebruikt een langste-pad-algoritme in dit netwerk om het werkrooster met de hoogste positieve gereduceerde kosten te bepalen.

2.3 Relatie met personeelsplanning

Kolomgeneratie is een krachtige methode om grote LP-problemen met zeer veel variabelen optimaal op te lossen. Echter, bij personeelsplanning gaat het om mensen en dus om geheeltallige LP-problemen, IP-problemen genoemd. Toch wordt kolomgeneratie veel toegepast bij roosterproblemen. De meeste bestaande toepassingen combineren kolomgeneratie om de LP-relaxatie (het IP probleem zonder de voorwaarde dat de beslissingsvariabelen geheeltallig moeten zijn) op te lossen met een branch-and-bound algoritme om goede geheeltallige oplossingen te vinden.

Er zijn er drie klassen algoritmes om kolomgeneratie en branch-and-bound te combineren; Vance [7]:

In de eerste klasse algoritmes wordt de kolomgeneratie 'off-line' uitgevoerd. Dat wil zeggen dat van te voren een verzameling kolommen wordt gegenereerd en vervolgens wordt met deze deelverzameling kolommen de optimale oplossing van het gereduceerde IP-probleem gezocht.

De tweede klasse van benaderingen gebruikt dynamische kolomgeneratie om de optimale oplossing van de LP-relaxatie van het probleem te vinden. Vervolgens wordt branch-and-bound toegepast om de optimale geheeltallige oplossing te vinden op de deelverzameling kolommen die gegenereerd zijn bij het oplossen van de LP-relaxatie.

Het probleem met deze eerste twee benaderingen is dat, ook al bestaat er een goede geheeltallige oplossing voor het probleem en ook al worden er goede oplossingen gevonden voor de LP-relaxatie, er is geen garantie dat er een goede of zelfs maar een geldige geheeltallige oplossing bestaat binnen de deelverzameling kolommen die goede LP-oplossingen geven.

In de derde klasse algoritmes wordt kolomgeneratie toegepast in elke tak van de branch-and-bound boom. Deze algoritmes worden **branch and price** algoritmes genoemd [1]. En in tegenstelling tot de vorige twee klassen algoritmes kan een branch and price algoritme een (bewezen) optimale oplossing vinden. Dus dit soort algoritmes hebben een aantal voordelen op de eerste twee, maar doordat er meerdere keren kolomgeneratie wordt toegepast zijn deze algoritmes ook complexer dan algoritmes van de eerste twee benaderingen.

2.4 Nadelen

Het voordeel van kolomgeneratie is dat er zelfs voor zeer grote LP-problemen een bewezen optimale oplossing gevonden kan worden. Er kleven echter ook wat nadelen aan deze methode:

De complexiteit van het hele probleem wordt verschoven naar de stap van het kolommen genereren. Dit kan zo ingewikkeld worden dat hiervoor alsnog een heuristiek nodig is. Dan wordt er niet meer een gegarandeerd optimale oplossing gevonden en had wellicht net zo goed direct een heuristiek toegepast kunnen worden.

Voor geheeltallige problemen zijn extra trucs nodig om een optimale oplossing te verkrijgen, deze zijn tijdrovend.

De manier van kolommen genereren is probleemspecifiek, er moet per probleem bedacht worden welke methode hiervoor het beste gebruikt kan worden.

Het probleem moet te schrijven zijn als LP of IP. Daardoor kunnen vaak niet de daadwerkelijke problemen opgelost worden, maar slechts vereenvoudigingen ervan.

Hoofdstuk 3

Een voorbeeld zelf oplossen

In dit hoofdstuk ga ik een praktijkvoorbeeld behandelen van kolomgeneratie toegepast op personeelsplanning. Het gaat in dit geval om het genereren van een rooster voor een call center. Eerst zal het probleem beschreven worden, daarna het model, vervolgens de implementatie en tot slot zullen de resultaten besproken worden.

3.1 Het probleem

In dit voorbeeld wordt een personeelsrooster gemaakt voor een call center. Dit call center is open van 8:00 uur tot 20:00 uur en voor elk kwartier is een schatting gedaan van het aantal binnenkomende telefoontjes per minuut. Het beantwoorden van een telefoontje kost gemiddeld 1/5 minuut. Er is een 80-20 service level vereist, dat wil zeggen dat 80% van de bellers binnen 20 seconden iemand aan de lijn moet krijgen. Het aantal benodigde mensen kan per kwartier bepaald worden met het Erlang-C wachtrij model. Het geschatte aantal benodigde werknemers verschilt nogal per kwartier, daarom moeten shift lengtes, start en pauze tijden verschillen, zodat er goede oplossingen verkregen kunnen worden, die het service level halen.

Er moet rekening gehouden worden met regels voor werknemers, daarom heb ik de CAO voor call center medewerkers bekeken en daar de volgende regels uit gehaald:

1. De arbeidstijd bedraagt ten hoogste 9 uren per dag.
2. Bij een arbeidstijd per dienst van meer dan 5,5 uur bedraagt de pauzetijd 0,5 uur, eventueel op te splitsen in 2 x 0,25 uur;
3. Bij een arbeidstijd per dienst van meer dan 8 uur bedraagt de pauzetijd 0,75 uur, waarvan 0,5 uur aaneengesloten.

   Om realistische resultaten te krijgen heb ik er de volgende regels bij bedacht:

4. De arbeidstijd bedraagt ten minste 4 uren per dag.
5. Bij een arbeidstijd per dienst van minder dan 5,5 uur bedraagt de pauzetijd 0,25 uur.
6. Een werknemer mag maximaal een blok van 3,25 uur achter elkaar werken zonder pauze.

Het probleem is een rooster te genereren waarbij de totale arbeidstijd minimaal is en in elk kwartier voldaan wordt aan het service level.

3.2 Model

In deze paragraaf wordt een model gegeven, waarmee een rooster met minimale arbeidstijd gemaakt kan worden, dat aan het gestelde service level voldoet.

Eerst wat definities alvorens het model opgesteld kan worden:

De verzameling van mogelijke shifts:
$s \in \{1, 2, \ldots, S\}$, S =het aantal toegestane shifts.
S is zeer groot, omdat er heel veel shifts mogelijk zijn die voldoen aan de regels van de vorige paragraaf.
De verzameling van tijdsintervallen:
$t \in \{1, 2, \ldots, T\}$, T =het aantal tijdsintervallen= 48, (12 uur van 4 kwartier).

x_s = het aantal mensen die shift s werken,
c_s = kosten van shift s = het aantal gewerkte kwartieren in shift s,
$a_{ts} = 1$ als shift s werkt op tijdsinterval t, 0 anders,

d_t = het aantal benodigde agenten op tijdsinterval t, bepaald met het Erlang C wachtrijmodel.

Dan is nu het model:

$$\begin{array}{ll} \min & \sum_{s=1}^{S} c_s x_s \\ \text{onder} & \sum_{s=1}^{S} a_{ts} x_s \geq d_t, \qquad t = 1, \ldots, 48 \\ & x_s \in Z^+. \end{array}$$

De matrix A, waarin alle mogelijke shifts staan gespecificeerd, is echter nog niet bekend. Het is ook niet handig om deze helemaal te maken, want S, het aantal mogelijke shifts, is erg groot. Het kost al erg veel rekentijd om alle mogelijke shifts te genereren en bovendien wordt dan het IP model dan zo groot dat het vinden van een optimale oplossing te tijdrovend of onmogelijk wordt. Daarom is het handig om hier een kolomgeneratie algoritme op toe te passen, zodat alleen de interessante kolommen gegenereerd en bekeken worden.

3.2.1 Kolomgeneratie algoritme

Zoals in paragraaf 2.3 is vermeld is het branch-and-price algoritme het algoritme dat van dit probleem een bewezen optimale oplossing kan vinden. Dit is een algoritme dat meerdere malen kolomgeneratie toepast, en kost veel rekentijd. Daarom kies ik ervoor om een algoritme uit de tweede klasse (zie par 2.3) te gebruiken voor dit probleem, waarbij kolommen worden gegenereerd op basis van het gerelaxeerde LP-probleem. Daardoor geeft dit algoritme niet noodzakelijkerwijs de optimale oplossing.

Er wordt begonnen met een kleine matrix D, met weinig kolommen. Deze kolommen moeten voldoen aan de 6 regels van paragraaf 3.1. Ook moeten deze kolommen zo zijn gedefinieerd, dat er een oplossing van het probleem mogelijk is. Er moet elk tijdsinterval iemand kunnen werken, dus voor elk tijdsinterval moet minstens 1 van de kolommen een 1 hebben. Matrix D is dus gedeelte van de onbepaalde matrix A.

Dan wordt het IP, met weinig kolommen, opgelost als LP, dus de geheeltalligheids-voorwaarde wordt gerelaxeerd, dit noem ik het master model. De duale variabelen van dit gerelaxeerde LP probleem worden vervolgens gebruikt om

een nieuwe shift te genereren met zo groot mogelijke gereduceerde kosten, dit noem ik het sub model. Deze shift wordt dan toegevoegd aan de matrix D van het oorspronkelijke probleem. Hiervan wordt weer de LP-relaxatie opgelost, waardoor nieuwe duale variabelen verkregen worden. Dan wordt weer een nieuwe shift geconstrueerd met zo groot mogelijke gereduceerde kosten en dit gaat door totdat er geen shift meer gevonden kan worden met positieve gereduceerde kosten. Tot slot wordt het IP opgelost, met alle gegenereerde kolommen.

Dus:

1. Kies een paar shifts die voldoen aan de regels en die samen een rooster kunnen vormen. Dus in elk tijdsinterval moet er minstens 1 shift zijn die een dienst inplant op dit tijdsinterval. Deze shifts vormen de kolommenverzameling N_k; k=1.

2. Vorm een matrix D van de kolommen uit N_k. De vector c_{N_k} is de vector met de kosten van de shifts uit N_k. De vector x_{N_k} is de vector met de beslissingsvariabelen die horen bij de shifts uit N_k. Los het gerelaxeerde probleem:

   $$\begin{array}{ll} \min & c_{N_k} x_{N_k} \\ \text{onder} & D x_{N_k} \geq d \\ & x_{N_k} \geq 0, \end{array}$$

   op met de (herziene) simplex methode, (master model).

3. Genereer een nieuwe shift met zo groot mogelijke gereduceerde kosten, (sub model); zijn de maximale gereduceerde kosten positief, ga naar stap 4, anders ga naar stap 5.

4. Voeg de kolom die bij de nieuwe shift hoort toe aan kolommenverzameling N_k. Dit wordt N_{k+1}; $k = k + 1$. Ga naar stap 2.

5. Los het IP probleem:

   $$\begin{array}{ll} \min & c_{N_k} x_{N_k} \\ \text{onder} & D x_{N_k} \geq d \\ & x_{N_k} \geq 0, \end{array}$$

   op met de gegenereerde matrix D.

3.2.2 Het genereren van een kolom

Het vinden van de shift met de grootste gereduceerde kosten kan met het volgende model.

De formule voor de gereduceerde kosten van een kolom j is $\pi A_j - c_j$, waarin π de vector is met duale variabelen, A_j de j-de kolom en c_j de kosten van kolom j. Hier is een kolom een shift, dus wordt gezocht naar de shift waarvoor deze waarde het grootst is.

Laat s een shift zijn, s is dus een vector (kolom):
$s_t = 1$ als in deze shift gewerkt wordt op tijdsinterval t, anders 0,
$c =$ 'kosten' van deze shift = het aantal gewerkte tijdseenheden = $\sum_{t=1}^{T} s_t$.
Gereduceerde kosten van deze shift = $\pi s - c = \sum_{t=1}^{T} \pi_t s_t - \sum_{t=1}^{T} s_t = \sum_{t=1}^{T} (\pi_t s_t - s_t)$.

Dus de criteriumfunctie wordt:
max $\sum_{t=1}^{T} (\pi_t s_t - s_t)$.

Deze shift s moet natuurlijk wel aan de voorwaarden uit paragraaf 3.1 voldoen.

Restricties 1 en 4 zijn eenvoudig te modelleren:
$16 \leq \sum_{t=1}^{48} s_t \leq 36$ minimaal 4 en maximaal 9 uren werk in een shift.

Voor de volgende pauze-restricties zijn een paar hulpvariabelen nodig:
$pauze_t = 1$ als je pauze hebt op tijdstip t, 0 anders.
$y_t = 1$ als je aanwezig bent op tijdstip t (werk of pauze), 0 anders.
Hieruit volgt:
$s_t + pauze_t = y_t$, voor $t = 1, \ldots, 48$, (werk + pauze = aanwezig).
Een werknemer is een aaneengesloten periode aanwezig op het call center, dus de variabele y_t mag maar twee keer veranderen van een 0 naar een 1 of van een 1 naar een 0, namelijk op het moment dat de werkdag begint en op het moment dat de werkdag eindigt. Daarom geldt dat $|y_t - y_{t-1}|$ twee keer 1 mag zijn en verder 0 moet zijn. Dus $\sum_{t=2}^{48} |y_t - y_{t-1}| \leq 2$. Er verandert altijd een 0 naar een 1, behalve als er helemaal aan het begin van de dag met werken begonnen wordt, als $s_1 = 1$. En er verandert altijd een 1 naar een 0, behalve als er helemaal tot aan het eind van de dag gewerkt wordt, als $s_{48} = 1$.
Daarom geldt:
$\sum_{t=2}^{48} |y_t - y_{t-1}| + s_1 + s_{48} = 2$.

Om te zorgen dat de werkdag niet begonnen wordt, of beëindigd wordt met een pauze, is de volgende restrictie nodig:
$pauze_1 = pauze_{48} = 0$.

Het aantal pauzes, restrictie 2, 3 en 5:
Iedereen heeft 1 of 2 pauzes per dag, dus er zijn minimaal 2 en maximaal 4 overgangen van niet pauze naar pauze of andersom:
$2 \leq \sum_{t=2}^{48} |pauze_t - pauze_{t-1}| \leq 4$.

De pauzetijd, restrictie 2, 3 en 5:

aantal kwartieren werktijd	aantal kwartieren pauze
23-26	3
22-31	2
16-21	1

Dit wil zeggen:
aantal kwartier pauze wordt bepaald door: floor(aantal kwartier werk/10.55).
$\sum_{t=1}^{48} pauze_t < \frac{\sum_{t=1}^{48} s_t}{10.55}$,
$\sum_{t=1}^{48} pauze_t > \frac{\sum_{t=1}^{48} s_t}{10.55} - 1$.

Restrictie 6:
$\sum_{j=t}^{t+13} s_j < 14$, voor $t = 1, \ldots, 48 - 13$, je werkt altijd max 3,25 uur achter elkaar.

Geheeltalligheid:
$s_t, y_t, pauze_t \in \{0, 1\}$

Omdat absolute waarden geen lineaire functies moeten een paar restricties nog aangepast worden. De eerste restrictie waarbij zich dit probleem voordoet is $\sum_{t=2}^{48} |y_t - y_{t-1}| + s_1 + s_{48} = 2$. Om hiervan lineaire restricties te maken zijn twee extra hulpvariabelen nodig: d^+ en d^-, die positief moeten zijn. Deze zijn zo gedefinieerd dat $d_t^+ - d_t^- = y_t - y_{t-1}$, dus het deel waar de absolute waarde van genomen moet worden, wordt geschreven als het verschil van twee positieve getallen. Dit is altijd mogelijk. De d_t^+ en d_t^- zijn nu echter nog niet uniek bepaald. Ze zijn wel uniek bepaald als afgedwongen wordt dat voor elke t of d_t^+ of d_t^- nul moet zijn. En dan is de absolute waarde van $y_t - y_{t-1}$ gelijk aan $d_t^+ + d_t^-$. Dus $\sum_{t=2}^{48} |y_t - y_{t-1}| + s_1 + s_{48} = 2$ kan vervangen worden door:
$\sum_{t=2}^{48} (d_t^+ + d_t^-) + s_1 + s_{48} = 2$
$d_t^+ - d_t^- = y_t - y_{t-1}$, voor $t = 2, \ldots, 48$

$d_t^+ + d_t^- < 2$, voor $t = 2, \ldots, 48$
$d_t^+, d_t^- \in \{0, 1\}$.

Op dezelfde wijze kan de restrictie $2 \leq \sum_{t=2}^{48} |pauze_t - pauze_{t-1}| \leq 4$ vervangen worden door:
$2 \leq \sum_{t=2}^{48}(e_t^+ + e_t^-) \leq 4$
$e_t^+ - e_t^- = pauze_t - pauze_{t-1}$, voor $t = 2, \ldots, 48$
$e_t^+ + e_t^- < 2$, voor $t = 2, \ldots, 48$
$e_t^+, e_t^- \in \{0, 1\}$.

Voor het overzicht geef ik nogmaals alle restricties netjes bij elkaar:
$16 \leq \sum_{t=1}^{48} s_t \leq 36$
$s_t + pauze_t = y_t$, voor $t = 1, \ldots, 48$
$pauze_1 = pauze_{48} = 0$
$\sum_{t=1}^{48} pauze_t < \frac{\sum_{t=1}^{48} s_t}{10.55}$
$\sum_{t=1}^{48} pauze_t > \frac{\sum_{t=1}^{48} s_t}{10.55} - 1$
$\sum_{j=t}^{t+13} s_j < 14$, voor t=1,...,35
$\sum_{t=2}^{48}(d_t^+ + d_t^-) + s_1 + s_{48} = 2$
$d_t^+ - d_t^- = y_t - y_{t-1}$, voor $t = 2, \ldots, 48$
$d_t^+ + d_t^- < 2$, voor $t = 2, \ldots, 48$
$2 \leq \sum_{t=2}^{48}(e_t^+ + e_t^-) \leq 4$
$e_t^+ - e_t^- = pauze_t - pauze_{t-1}$, voor $t = 2, \ldots, 48$
$e_t^+ + e_t^- < 2$, voor $t = 2, \ldots, 48$
$s_t, y_t, pauze_t, d_t^+, d_t^-, e_t^+, e_t^- \in \{0, 1\}$.

3.3 Programma

Dit model heb ik geïmplementeerd in GAMS. GAMS is een systeem dat gebruikt kan worden om LP-problemen en andere optimalisatie problemen op te lossen. GAMS bestaat uit een compiler en verschillende solvers. Om dit te implementeren heb ik gebruik gemaakt van Kalvelagen [6].

Eerst wordt in GAMS de data ingevoerd, voor elk tijdsinterval het benodigd aantal medewerkers. Vervolgens worden de twee modellen gespecificeerd. Het hoofdmodel (master) en het submodel om de kolommen met de maximale gereduceerde kosten te vinden (sub). Daarbij worden de variabelen, doelfuncties en de restricties ingevoerd. Bij het master model is sprake

van een groeiende matrix D (dip) en kostenvector c (cp). Daarom wordt er bij het master model gebruik gemaakt van een dynamische verzameling *pp*. Deze dynamische verzameling groeit zo lang er nieuwe kolommen met positieve gereduceerde kosten gevonden worden. Dan wordt het programma geïnitialiseerd, de groeiende matrix D moet in het begin gevuld worden met enkele kolommen, zodat er een toegelaten startoplossing gevonden kan worden. Dan kan de kolomgeneratie starten. Dit gebeurt met een loop.

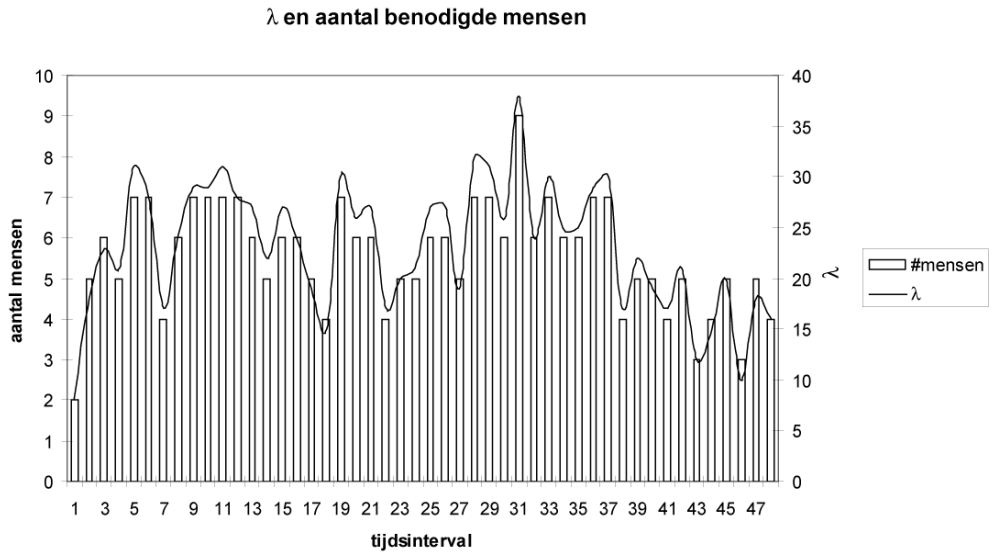
loop{
  los master op met *rmip* minimaliseer z
  los sub op met *mip* maximaliseer z
  if(gereduceerde kosten van gevonden kolom > 0.0001){
    voeg nieuwe kolom toe aan matrix D (dip)
    voeg kosten van nieuwe kolom toe aan c (cp)
    maak dynamische verzameling *pp* groter
  }else STOP
}

Na deze loop zijn alle kolommen gegenereerd en moet alleen nog het master probleem voor de laatste keer opgelost worden.

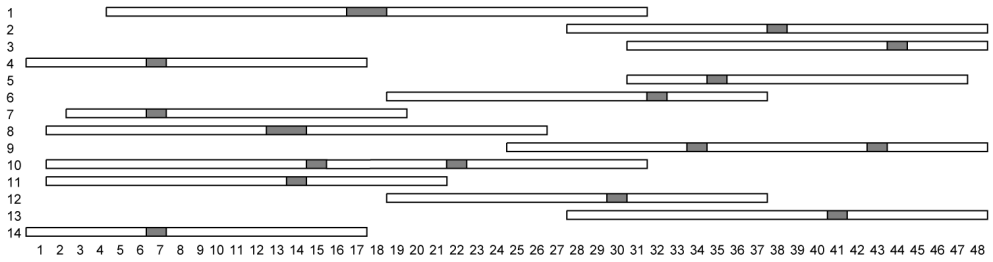
Er zijn verschillende solvers die gebruikt kunnen worden. Voor het oplossen van het master probleem in de loop gebruik ik *rmip*, dit is de solver die gebruikt kan worden om een gerelaxeerd geheeltallig probleem op te lossen. En dat is hier ook het geval. Voor het sub probleem en het uiteindelijke master probleem gebruik ik *mip*, dat is de solver die gebruikt kan worden om (gemixte) geheeltallige problemen op te lossen. De hele code van mijn programma is te vinden in bijlage A.

3.4 Resultaten

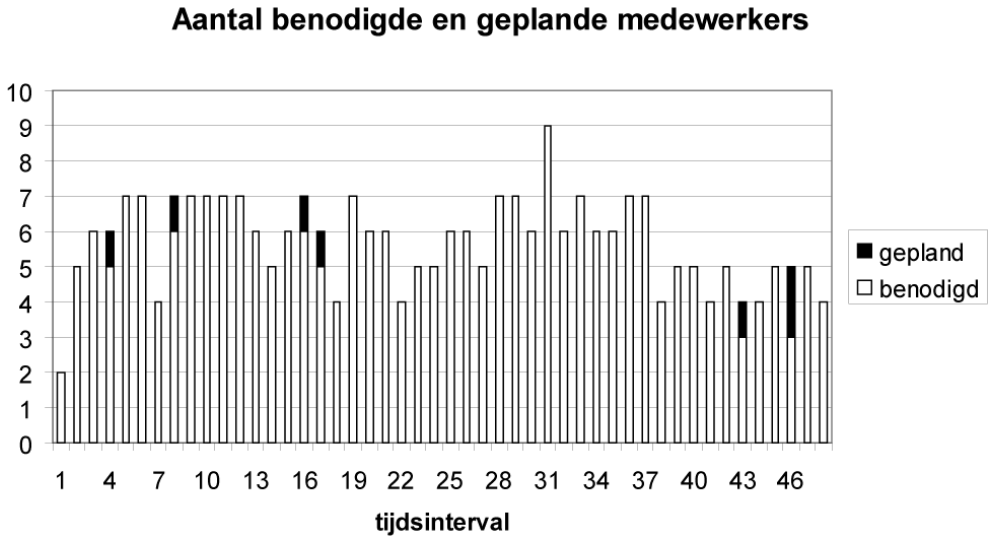
De eerste stap van het oplossen het het probleem is het omzetten van de geschatte λ's, de gegeven bedieningstijd en het gewenste service level, naar het aantal benodigde medewerkers. Dit heb ik gedaan met het Erlang-C wachtrij model.



Op grond van deze aantallen benodigde mensen is het GAMS programma uitgevoerd. Na 3 minuten zijn er 100 kolommen gegenereerd en is het volgende rooster gemaakt:



De witte balken staan voor de shifts en de grijze blokjes daarin zijn de pauzes. Er zijn dus 14 mensen ingepland op deze dag en medewerkers 4 en 14 hebben hetzelfde rooster. Als we deze shifts optellen, kunnen we deze vergelijken met de benodigde aantallen mensen:



Hier zijn de witte staven de benodigde aantallen medewerkers per kwartier en de zwarte staafjes daar bovenop de mensen die te veel zijn ingepland. Er zijn dus 6 kwartieren dat er te veel medewerkers aan het werk zijn, en op kwartier 46 zijn er zelfs 2 mensen te veel. Er wordt dus in totaal 7 kwartier te veel gewerkt. Er valt te zien dat de momenten dat er te veel mensen aan het werk zijn vaak vallen tussen 2 momenten waarop er meer mensen nodig zijn. Ook is te zien dat de pauzes altijd zo gekozen zijn dat ze vallen in een rustige periode. Bijvoorbeeld in tijdsinterval 7 zijn er ineens 3 mensen minder nodig dan in het tijdsinterval daarvoor en daarna, en er hebben dan ook 3 mensen pauze op dat moment. Het rooster ziet er dus goed uit.

Dit algoritme hoeft geen optimale oplossing te geven, omdat de kolommen worden gegenereerd op basis van de LP-relaxatie van het probleem en niet op basis van het probleem zelf. Toch ziet deze oplossing er goed uit en zou wellicht optimaal kunnen zijn. Om dit te testen heb ik het master probleem nog eens opgelost, maar dan met nog veel meer kolommen. Hier kwam ook een oplossing uit met 7 kwartier werk te veel. Dus waarschijnlijk is deze oplossing wel optimaal.

Hoofdstuk 4

Conclusies

Kolomgeneratie is een doeltreffende methode om bepaalde problemen op te lossen; het werkt vooral goed bij problemen met zeer veel mogelijkheden. Er kunnen met kolomgeneratie zeer grote problemen opgelost worden doordat de variabelen impliciet beschouwd worden. Er kleven echter ook wat nadelen aan deze methode. Het probleem moet te schrijven zijn als LP of IP probleem, dus vaak kunnen alleen vereenvoudigde problemen opgelost worden. Verder is de manier van kolommen genereren probleemspecifiek, er moet per probleem bedacht worden welke methode hiervoor het beste gebruikt kan worden, dus kolomgeneratie kan niet zonder meer op alle problemen toegepast worden. Soms kan het genereren van kolommen zo ingewikkeld worden dat hiervoor weer een heuristiek nodig is, en daardoor alsnog niet een optimale oplossing gevonden kan worden.

Kolomgeneratie is ook toepasbaar op personeelsplanningproblemen, hiermee worden vaak goede resultaten bereikt. Een probleem daarbij is dat personeelsplanningsproblemen altijd geheeltallig zijn. De kolomgeneratie moet dan gecombineerd worden met een branch-and-bound algoritme, dit kan tijdrovend zijn of ten koste gaan van de garantie dat de optimale oplossing gevonden wordt.

Bibliografie

[1] Cynthia Barnhard. Branch-and-price: Column generation for solving huge integer programs. 1996.

[2] Guy E. Eitzen. Integer programming methods for solving multi-skilled workforce optimisation problems. 2002.

[3] H.C. Tijms en A.A.N. Ridder. *Mathematische Programmering*. 2003.

[4] H.C. Tijms en E.M.F. Kalvelagen. *Modelbouw in de operations research*. Academic service.

[5] A.T. Ernst. Staff scheduling and rostering: A review of applications, methods and models. 2003.

[6] Erwin Kalvelagen. Column generation with GAMS. 2003.

[7] Pamela H. Vance. A heuristic branch-and-price approach for the airline crew pairing problem. 1997.

Bijlage A

GAMS code

```
$ontext
Scriptie kolomgeneratie
Annemieke van Dongen
$offtext

set i 'kwartieren' /kwartier1*kwartier48/;
set j 'hulpkw' /1*48/;

*-------------------------------------------------------
* Data
*-------------------------------------------------------

table demanddata(i,*)
         demand
kwartier1 2
kwartier2 5
kwartier3 6
kwartier4 5
kwartier5 7
kwartier6 7
kwartier7 4
kwartier8 6
kwartier9 7
```

```
kwartier10 7
kwartier11 7
kwartier12 7
kwartier13 6
kwartier14 5
kwartier15 6
kwartier16 6
kwartier17 5
kwartier18 4
kwartier19 7
kwartier20 6
kwartier21 6
kwartier22 4
kwartier23 5
kwartier24 5
kwartier25 6
kwartier26 6
kwartier27 5
kwartier28 7
kwartier29 7
kwartier30 6
kwartier31 9
kwartier32 6
kwartier33 7
kwartier34 6
kwartier35 6
kwartier36 7
kwartier37 7
kwartier38 4
kwartier39 5
kwartier40 5
kwartier41 4
kwartier42 5
kwartier43 3
kwartier44 4
kwartier45 5
kwartier46 3
kwartier47 5
```

```
kwartier48 4
;

parameter d(i);
d(i) = demanddata(i,'demand');

*-------------------------------------------------------
* Gilmore-Gomory column generation algorithm
*-------------------------------------------------------

set p 'possible shifts' /p1*p1000/;
set iter 'maximum iterations' /iter1*iter1000/;

*-------------------------------------------------------
* Master model
*-------------------------------------------------------

parameter dip(i,p) 'matrix growing in dimension p';
parameter cp(p) 'cost vector growing in dimension p';
integer variable xp(p) 'shifts used';
variable z 'objective variable';
*
* default integer upperbound of 100 is too tight
*
xp.up(p) = sum(i, d(i));

set pp(p) 'dynamic subset';

equations
   mnumshifts 'number of shifts used'
   mdemand(i) 'meet demand'
;

mnumshifts..  z =e= sum(pp, xp(pp)*cp(pp));
mdemand(i)..  sum(pp, dip(i,pp)*xp(pp)) =g= d(i);

model master /mnumshifts,mdemand/;
```

```
* reduce amount of information written to the listing file
master.solprint = 2;
master.limrow = 0;
master.limcol = 0;
master.ITERLIM=1000000;

*-----------------------------------------------------
* Sub model
*-----------------------------------------------------

binary variables
   s(i) 'new shift'
   y(i) 'aanwezig'
   pauze(i) 'pauze'
   dplus(i) 'hulp1'
   dmin(i) 'hulp2'
   eplus(i) 'hulp3'
   emin(i) 'hulp4'
;

equations
   subobj
   sub1
   sub2
   sub3
   sub4
   sub5
   sub6
   sub7
   sub8
   sub9
   sub10
   sub11
   sub12
   sub13
   sub14
   sub15
;
```

```
subobj..  z =e= sum(i, mdemand.m(i)*s(i)-s(i));
sub1(i)..  s(i)+pauze(i) =e= y(i);
sub2..  sum(i, s(i)) =g= 16;
sub3..  sum(i, s(i)) =l= 36;
sub4..  sum(i$ (ord(i) GE 2),dplus(i)+dmin(i))+s('kwartier1')+
   s('kwartier48') =e= 2;
sub5(i)$(ord(i) GE 2)..  dplus(i) - dmin(i) =e= y(i) - y(i-1);
sub6(i)$(ord(i) GE 2)..  dplus(i) + dmin(i) =l= 1;
sub7..  sum(i$ (ord(i) GE 2),eplus(i)+emin(i)) =g= 2;
sub8..  sum(i$ (ord(i) GE 2),eplus(i)+emin(i)) =l= 4;
sub9(i)$(ord(i) GE 2)..  eplus(i) - emin(i) =e= pauze(i) - pauze(i-1);
sub10(i)$(ord(i) GE 2)..  eplus(i) + emin(i) =l= 1;
sub11..  sum(i,pauze(i)) =l= (sum(i,s(i)))/10.55;
sub12..  sum(i,pauze(i)) =g= (sum(i,s(i)))/10.55-1;
sub13(j)$(ord(j) LE 35)..
   sum(i$(ORD(i) GE ORD(j) and ORD(i) LE ORD(j)+13),s(i)) =l= 13;
sub14..  pauze('kwartier1') =e= 0;
sub15..  pauze('kwartier48') =e= 0;

model sub /subobj,sub1,sub2,sub3,sub4,sub5,sub6,sub7,sub8,sub9,
   sub10,sub11,sub12,sub13,sub14,sub15/;

sub.solprint = 2;
sub.optcr = 0;
sub.limrow = 0;
sub.limcol = 0;
sub.ITERLIM=1000000;

*-----------------------------------------------------
* initialization
* get initial set pp and initial matrix dip
*-----------------------------------------------------
set pi(p);
pi('p1') = yes;

dip('kwartier1',pi) = 1;
dip('kwartier2',pi) = 1;
```

```
dip('kwartier3',pi) = 1;
dip('kwartier4',pi) = 1;

dip('kwartier5',pi) = 1;
dip('kwartier6',pi) = 1;
dip('kwartier7',pi) = 1;
dip('kwartier8',pi) = 1;

dip('kwartier9',pi) = 1;
dip('kwartier10',pi) = 1;
dip('kwartier11',pi) = 1;
dip('kwartier12',pi) = 1;

dip('kwartier15',pi) = 1;
dip('kwartier16',pi) = 1;

dip('kwartier17',pi) = 1;
dip('kwartier18',pi) = 1;
dip('kwartier19',pi) = 1;
dip('kwartier20',pi) = 1;

dip('kwartier21',pi) = 1;
dip('kwartier22',pi) = 1;
dip('kwartier23',pi) = 1;
dip('kwartier24',pi) = 1;

dip('kwartier25',pi) = 1;
dip('kwartier26',pi) = 1;

dip('kwartier28',pi) = 1;

dip('kwartier29',pi) = 1;
dip('kwartier30',pi) = 1;
dip('kwartier31',pi) = 1;
dip('kwartier32',pi) = 1;

dip('kwartier33',pi) = 1;
dip('kwartier34',pi) = 1;
```

```
dip('kwartier35',pi) = 1;
dip('kwartier36',pi) = 1;

dip('kwartier37',pi) = 1;
dip('kwartier38',pi) = 1;
dip('kwartier39',pi) = 1;

cp(pi) = 36;
pp(pi) = yes;
pi(p) = pi(p-1);

dip('kwartier5',pi) = 1;
dip('kwartier6',pi) = 1;
dip('kwartier7',pi) = 1;
dip('kwartier8',pi) = 1;

dip('kwartier9',pi) = 1;
dip('kwartier10',pi) = 1;
dip('kwartier11',pi) = 1;
dip('kwartier12',pi) = 1;

dip('kwartier13',pi) = 1;
dip('kwartier14',pi) = 1;
dip('kwartier15',pi) = 1;
dip('kwartier16',pi) = 1;


dip('kwartier19',pi) = 1;
dip('kwartier20',pi) = 1;

dip('kwartier21',pi) = 1;
dip('kwartier22',pi) = 1;
dip('kwartier23',pi) = 1;
dip('kwartier24',pi) = 1;

dip('kwartier25',pi) = 1;
dip('kwartier26',pi) = 1;
dip('kwartier27',pi) = 1;
```

```
dip('kwartier28',pi) = 1;

dip('kwartier29',pi) = 1;
dip('kwartier30',pi) = 1;
dip('kwartier31',pi) = 1;

cp(pi) = 25;
pp(pi) = yes;
pi(p) = pi(p-1);

dip('kwartier28',pi) = 1;

dip('kwartier29',pi) = 1;
dip('kwartier30',pi) = 1;
dip('kwartier31',pi) = 1;
dip('kwartier32',pi) = 1;

dip('kwartier33',pi) = 1;
dip('kwartier34',pi) = 1;
dip('kwartier35',pi) = 1;
dip('kwartier36',pi) = 1;

dip('kwartier37',pi) = 1;

dip('kwartier39',pi) = 1;
dip('kwartier40',pi) = 1;

dip('kwartier41',pi) = 1;
dip('kwartier42',pi) = 1;
dip('kwartier43',pi) = 1;
dip('kwartier44',pi) = 1;

dip('kwartier45',pi) = 1;
dip('kwartier46',pi) = 1;
dip('kwartier47',pi) = 1;
dip('kwartier48',pi) = 1;

cp(pi) = 20;
```

```
pp(pi) = yes;
pi(p) = pi(p-1);

scalar done /0/;

loop(iter$(not done),

*
* solve master problem
*
   solve master using rmip minimizing z;

*
* solve sub problem
*
   solve sub using mip maximizing z;

*
* new shift found?
*
   if(z.l > 0.001,
      dip(i,pi) = s.l(i);
      cp(pi) = sum(i,s.l(i));
      pp(pi) = yes;
      pi(p) = pi(p-1);
   else
      done = 1;
   );
);

abort$(not done) "Too many iterations.";

*
* solve final mip
*

master.optcr=0;
solve master using mip minimizing z;
```

```
file results /def.txt/;
results.pw=350;
results.pc=5;
put results;

loop(i,
   loop(pp,
      put dip(i,pp):1:0
   )
   put /
);
loop(pp,
   put xp.l(pp):1:0
);
putclose;
```