Rijksuniversiteit Groningen Faculteit der Wiskunde en Natuurwetenschappen

Vakgroep
Informatica

Rooster Planning

Berend Reitsma

Begeleider: J. Jongejan

RuG

Contents

1 Probleemstelling en context

Een ontwikkeling waar het bedrijfsleven nogal wat belangstelling voor toont betreft het automatiseren van de roosterplanning voor hun personeel. De verwachting is dat men daardoor op kosten kan besparen en dat men in staat is om kwalitatief betere roosters te maken.

Op dit moment zijn er op dit gebied weinig of geen standaard-oplossingen. Afhankelijk van de situatie wordt er van verschillende methoden gebruik gemaakt. Voorbeelden hiervan zijn Monte Carlo, Simulated Annealing, Genetische Algoritmen, (Mixed Integer) Lineair Programmeren, Constraint Logic Programming en dergelijke.

Er is mij gevraagd om vooral te kijken naar de mogelijkheden die binnen het pakket ZKR[1] te gebruiken zijn. Dit is een pakket voor het plannen van diensten van ziekenhuispersoneel. Hierbij wordt het personeel per dag in een bepaalde dienst ingeroosterd. Eén van de kenmerken van het systeem is dat het interactief werkt. Dit sluit daarom die technieken uit die relatief langzaam zijn.

Op dit moment wordt met dit pakket alleen een planning gemaakt naar tijd. Een volgende fase is het plannen van werkplekken. Met de gebruikte methode binnen ZKR is dit niet zondermeer mogelijk.

Na enige studie blijkt dat binnen ZKR een model gebruikt wordt wat vrij veel lijkt op een model voor Lineair Programmeren (LP). Voor mij is dat één van de redenen om naar het gebruik van LP methoden te gaan kijken. Een andere reden is gelegen in de eenvoud van een dergelijk model. Het is een uitdaging om te kijken wat we met behulp van zo'n model nog kunnen beschrijven. Na enkele testen blijkt dat we met behulp van LP ook snel een oplossing kunnen krijgen. Dit is ook een vereiste in verband met het interactieve karakter van het systeem.

Het inroosteren van personeel kan op meerdere manieren gedaan worden. De eerste is door flexibel plannen. Daarvoor wordt het personeel alleen dan ingeroosterd als er bepaalde taken uitgevoerd moeten worden. Het zal duidelijk zijn dat hier een groot aantal vrijheidsgraden zijn. Dat heeft zijn weerslag op de grootte van het model en de rekentijd die nodig is om het model door te rekenen.

Een andere manier is het het inperken van het aantal vrije vrijheidsgraden. Eén vrij gebruikelijke methode daarvoor, is het vaststellen van vaste werktijden. In een ziekenhuis betekent dit dat het personeel in diensten ingeroosterd wordt. Voor de planner betekent dit dat hij alleen nog aan moet geven hoeveel mensen en met welke kwalificatie hij nodig heeft om alle taken uit te voeren.

Naast de behoefte aan gekwalificeerd personeel komen er ook andere zaken om de hoek kijken zoals werkbelasting, afspraken, CAO-eisen en dergelijke. Deze gegevens zullen ook in het model verwerkt moeten worden. Behalve aan het inroosteren van personeel in een bepaalde dienst is er ook behoefte aan het plannen van werkplekken. Dit geeft het roosterprobleem extra complexiteit. Daarnaast willen we ook graag gebruik maken van de kwalificaties die het personeel bezit. Als men personeel heeft dat op meerdere plaatsen ingezet kan worden, geeft dat een extra mogelijkheid om een beter rooster te maken.

[1] ZKR (ziekenhuis roostering) is een produkt van IKS

In de huidige literatuur heb ik weinig gegevens kunnen vinden over roosterplanning. Enkele door mij aangeschreven personen hadden zich vooral gericht op een specifieke oplossing van hun roosterprobleem. Een klein bericht op Internet heeft mij uiteindelijk een handvat gegeven waardoor ik in staat was om een meer algemeen model op te zetten voor de roosterplanning.

Nadat duidelijk geworden was hoe de structuur van het probleem er uit zag, heb ik nog een aantal voorbeelden kunnen vinden die eenzelfde structuur bezaten. Ik heb echter niets kunnen vinden over het veralgemeniseren van het probleem.

2 Probleemanalyse

Er zijn verschillende soorten roosterproblemen. In dit verslag ga ik een specifiek soort roosters bespreken. Dit zijn roosters waarbij personeel gedurende een dag of een dagdeel aan een bepaalde taak werkt. Deze taak ligt in principe vast. Het rooster moet een oplossing geven, zodanig dat er voldoende personeel aanwezig is tijdens die dag of dat dagdeel, om alle taken uit te voeren.

Dit verslag gaat dus niet in op het samenvoegen van werkzaamheden tot een taak. Deze situatie doet zich voor op een afdeling van een ziekenhuis. Het personeel op zo'n afdeling werkt in diensten. Er is per dienst een specifikatie gemaakt van de benodigde hoeveelheid personeel om alle werkzaamheden uit te voeren. Daar komt nog bij dat bepaalde taken alleen maar door gekwalificeerd personeel uitgevoerd mogen worden.

Een oplossing van dit roosterprobleem moet er voor zorgen dat (indien mogelijk) tijdens elke dienst er minimaal een bepaalde bezetting met een gegeven kwalificatie aanwezig is. Daarnaast moet er getracht worden om ook naar een aantal randvoorwaarden te kijken. Dus als er een rooster mogelijk is dat beter aan de randvoorwaarden voldoet, dan zou het prettig zijn als dat rooster ook gekozen wordt.

Er zijn een aantal methodes om het roosterprobleem aan te pakken. Eén daarvan is door middel van heuristieken te proberen een acceptabel rooster samen te stellen. Daarna kan nog geprobeerd worden een iets beter rooster te maken door in het rooster te gaan schuiven.

Een variant hierop is te maken door niet één enkel rooster te maken, maar een aantal roosters. Door deze te vergelijken kunnen nieuwe roosters gemaakt worden die misschien een beter resultaat opleveren.

Een andere mogelijkheid krijgen we door alle mogelijke roosters te genereren. Daarna hoeven we alleen nog maar de beste eruit te selecteren. Het voordeel van de laatste methode is, dat we precies weten dat we de beste oplossing te pakken hebben. Het grote nadeel is de benodigde rekentijd en opslagcapaciteit. Een rooster met een beperkt aantal in te roosteren dagen, taken en personen is misschien nog wel door te rekenen, maar het aantal roosters neemt wel exponentieel toe met het aantal dagen, taken en personen.

We hoeven deze methode echter niet direct weg te gooien. Als we er in slagen om het grote probleem in een aantal deelproblemen op te splitsen, dan is de exponentiële groei ook kleiner. Als alles een beetje meezit dan kunnen we ook roosterproblemen met een reële omvang toch nog doorrekenen.

Vanuit de menselijke planner gezien is het roosterprobleem als volgt te zien:

- Er is een verzameling taken die uitgevoerd moeten worden op een bepaald tijdstip. Dit tijdstip is in principe van te voren al vastgelegd. Dit heeft te maken met het inperken van het aantal vrijheidsgraden van het probleem.

- Er is een verzameling personen waaruit gekozen moet worden om tot de uitvoering van de gestelde taken te komen.

- Er zijn een groot aantal regels die hun invloed hebben op de uiteindelijke vorm en gewenstheid van een rooster. Voorbeelden van dergelijke regels zijn CAO-eisen, onderlinge werkverdeling en persoonlijke afspraken.

- Dan zijn er nog de arbeidsgegevens van het personeelsbestand die meegenomen moeten worden in het planningsproces. Dit heeft bijvoorbeeld te maken met de kwalificaties van een persoon, de werkbelasting ten opzichte van andere personen en dergelijke.

Een planner zal onder deze omstandigheden een keuze moeten maken voor bepaalde roosters. Dat hierbij niet alle mogelijke roosters bekeken worden zal niemand verbazen. Meestal wordt een rooster gekozen op grond van ervaringsfeiten. Omdat dit soort informatie meestal nooit expliciet gegeven wordt, is het voor een informatiesysteem vrijwel onmogelijk om ook met dit soort informatie te werken.

Hoe we het echter ook wenden of keren, we zullen hier op de een of andere manier mee moeten leren leven. Als we in een dergelijk pakket een manier hebben om op eenvoudige wijze regels toe te voegen, zijn we in staat om na de ingebruikname de nog niet expliciet gemaakte regels later toe te voegen.

Bij een systeem voor roosterplanning kan men onderscheid maken tussen interactieve en niet-interactieve systemen. Bij een interactief systeem zijn vooral de responstijden belangrijk. Het systeem moet dan binnen een vooraf ingestelde tijd met een oplossing komen. Dat deze oplossing dan sub-optimaal is, is minder van belang. De gebruiker van een dergelijk systeem kan altijd nog besluiten dat hij nog wel iets langer op een betere oplossing wil wachten.

Een dergelijke eis legt wel beperkingen op aan de oplossingsmethoden. Een techniek die de toegestane responstijd met factoren overschrijdt, kan dan niet toegepast worden. Het liefst gebruikt men een methode die op vrij korte termijn al een redelijke oplossing geeft, waarna deze verder geoptimaliseerd kan worden.

Bij een niet-interactief systeem is de kwaliteit van de oplossing vaak veel belangrijker. Men is wel bereid om enkele uren of dagen te wachten voordat het systeem met een oplossing komt. Het zal duidelijk zijn dat bij dit soort systemen meer oplossingsmethoden aanvaardbaar zijn dan bij een interactief systeem. Ook krijgen we met grotere systemen en met meer vrijheidsgraden te maken. Men is bij een dergelijk systeem veel meer geïntereseerd in de kwaliteit van een oplossing, dan in de tijdsduur.

Als we bijvoorbeeld naar een afdeling in een ziekenhuis kijken, zien we vaak de volgende eigenschappen:

- Er is een roostereenheid gedurende welke een aantal taken c.q. diensten gewerkt moet worden. Meestal is dit een dag of een dagdeel. Daarom spreken we vaak van een dag of dagdeel als we een roostereenheid bedoelen.

- Per dag of dagdeel (= roostereenheid) moeten een aantal diensten en taken uitgevoerd worden. Het personeel werkt in diensten aan een bepaalde (globale) taak. Deze taak of dienst beschrijft dan weer een samenstel van verschillende werkzaamheden. Het samenstellen van een dienst is meestal een eenmalige taak van de planner, en blijft daarna gedurende lange tijd vrijwel hetzelfde. Het samenstellen van een dienst gaan we hier niet behandelen.

- Per dag wordt er door een persoon maar in één dienst gewerkt. Per dienst wordt er één taak uitgevoerd (= samenstel van werkzaamheden). Het is in principe mogelijk om tijdens die dienst meerdere taken uit te voeren. We bekijken welke konsekwenties dit heeft, maar gaan in eerste instantie uit van één taak per dienst.

- Het personeel bezit een aantal kwalificaties die het geschikt maakt voor de uitvoering van bepaalde taken. Al het personeel is niet gekwalificeerd voor alle taken, zodat we aan moeten kunnen geven welke kwalificaties een persoon bezit, en welke kwalificaties nodig zijn voor de uitvoering van een taak.

Om de complexiteit van het roosterprobleem terug te dringen zijn we genoodzaakt om een opsplitsing in deelproblemen te maken. Als we dit niet willen, dan kunnen we ons slechts beperken tot vrij kleine problemen. Omdat het de bedoeling is dat een dergelijk systeem in de praktijk moet werken, ontkomen we niet aan een vereenvoudiging.

We gaan daarvoor het rooster opsplitsen in roosters per persoon. Het uiteindelijke rooster is dus te zien als een samenstelling van een aantal persoonlijke roosters. De keuze van een rooster per persoon kan dan niet meer afhangen van omstandigheden die niet op dit niveau bekend zijn. over het algemeen valt met deze beperking wel te leven. Als dit soort keuzes echt van belang zijn, dan moet een gedeelte van het rooster handmatig ingevuld worden, of er moet andere software komen.

Per persoon gaan we dus een keuze maken voor een rooster. Deze keuze kunnen we af laten hangen van de wens op niveau van het totaal rooster. Er wordt geprobeerd een zodanig rooster te kiezen dat er aan een aantal minimum eisen voldaan wordt in het totaal rooster. In mindere mate wordt gepoogd om een optimum voor het totaal rooster te verkrijgen. Dit komt omdat we daartoe niet altijd in staat zijn.

Bestaande oplossing binnen ZKR

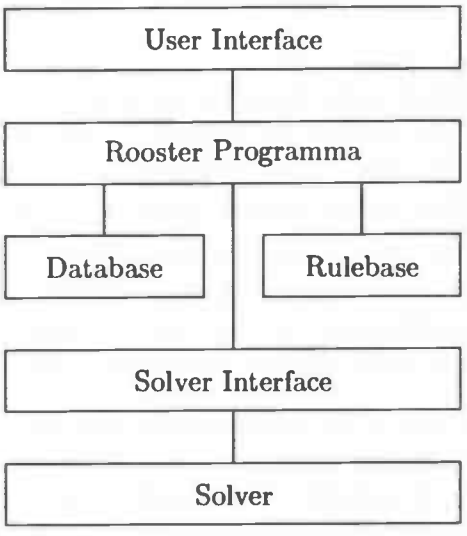
Ik ben uitgegaan van de bestaande scheiding binnen het programma ZKR. Daarbij wordt op een gegeven moment per persoon een aantal roosters gegenereerd met daarbij een waarderingswaarde per rooster. Deze gegevens worden aan een routine aangeboden die probeert een oplossing te vinden. Als laatste wordt een aantal gevonden oplossingen naar de gebruiker teruggekoppeld.

Ik heb geprobeerd of er ook een betere en meer generieke methode was om een oplossing te zoeken. Daarbij heb ik mij uiteindelijk gericht op Integer Lineair Programmeren. Mijn idee is om de mogelijkheid open te laten voor het gebruik van de reeds bestaande code voor het genereren van roosters en de terugkoppeling naar de gebruiker. Alleen het kiezen van een oplossing uit de totale verzameling van oplossingen gaat dan veranderen.

Voor een optimaal gebruik van de mogelijkheden van Lineair Programmeren zal de manier aangepast moeten worden waarop de roosters gegenereerd worden. Op dit

moment worden eerst alle roosters gegenereerd, en worden daarna de niet toegestane roosters geschrapt. Het is beter (in termen van geheugengebruik en processortijd) om alleen maar de toegestane roosters te genereren. Als we bijvoorbeeld ook werkplek planning willen gaan doen, dan krijgen we een vrij grote oplossingsruimte met maar een beperkte hoeveelheid toegestane oplossingen. Dit leidt er toe dat een meer generieke methode nodig is om dit soort problemen aan te kunnen.

Het schematische model van het rooster programma ziet er als volgt uit:



Een user-interface stelt de gebruiker in staat om op een simpele manier de parameters voor het systeem in te stellen. Ook kan de database met personen, taken en dergelijke up-to-date gehouden worden.

Het rooster programma zelf moet er voor zorgen dat de onderlinge consistentie gehandhaaft blijft. In de database staan onder andere de gegevens over personen, de uit te voeren taken en roosters. Tevens is er een rulebase aanwezig waar regels in staan waaraan een rooster moet voldoen. Het programma gebruikt deze regels om correctheid en gewenstheid van een rooster te bepalen.

De solver is een onderdeel waarmee geprobeerd wordt om een rooster samen te stellen met behulp van software. Op deze manier wordt gepoogd om het rooster-proces te automatiseren. De interface voor de solver zorgt voor een vertaling van de gegevens tussen het roosterprogramma en de solver.

Tijdens mijn afstuderen heb ik me met name gericht op de solver en op de interface hiervoor. De huidige solver is gebaseerd op een heuristiek. Deze werkt op zich wel, maar is niet flexibel genoeg om ook nog werkplek planning te gaan doen. Na enig zoeken ben ik tot de conclusie gekomen dat een solver gebaseerd op Integer Lineair Programmeren de huidige solver kan vervangen. Daarna ben ik gaan zoeken naar mogelijkheden om zoveel mogelijk uit de LP solver te persen. Dit heeft vooral betrekking op de formulering van het oorspronkelijke roosterprobleem, zodanig dat een LP solver er weg mee weet.

De reden dat ik mij tot LP beperkt heb, ligt in het feit dat het onderliggende model vrij simpel is. In principe bestaat het model uit een verzameling lineaire ongelijkheden

die een oplossingsruimte beschrijven. Met behulp van LP wordt een optimum oplossing in deze ruimte gezocht. De eenvoud zit in het lineair zijn van alle betrekkingen. We zijn op een vrij eenvoudige manier in staat om het roosterprobleem om te schrijven naar een LP probleem. Voordat ik andere technieken van stal wilde halen, heb ik eerst geprobeerd om zoveel gebruik te maken van de mogelijkheden van een LP model.

Definities

In de loop van het verslag maken we gebruik van enkele termen die we hier nader definiëren.

- Een *rooster* is een beschrijving van geplande taken of diensten op een reeks dagen, behorende bij een persoon. In feite is een rooster een opsomming van dagen of dagdelen. Elke dag binnen zo'n rooster bestaat weer uit een opsomming van taken of diensten die uitgevoerd (kunnen) worden als het rooster gevolgd wordt.

  Elk rooster behoort bij één specifiek persoon. Dit betekent dat we een unieke identificatie van een persoon hebben als we een rooster selecteren. We hoeven dan alleen rekening te houden met een (grote) verzameling roosters, en niet ook nog met een verzameling personen waaruit gekozen moet worden.

- Een *roosterhorizon* geeft aan hoeveel dagen we in één keer inroosteren. Doordat het inroosteren van een dag invloed heeft op alle andere dagen, zullen we in principe alle dagen in een keer vast willen leggen. Omdat de grootte van het model een exponentieel gedrag vertoont in het aantal te plannen dagen, zullen we het aantal dagen moeten beperken. Dit noemt men dus een roosterhorizon.

- Een *(lineair) mathematisch model* is het onderliggende model voor Lineair Programmeren (LP). Dit model bestaat uit een stelsel lineaire ongelijkheden in combinatie met een lineaire optimalisatie-funktie. In het geval van Mixed Integer LP kan ook nog aangegeven worden of een variabele geheeltallig is of niet.

  Vaak wordt het model als een matrix gepresenteerd. Daarbij stellen de rijen de ongelijkheden voor, staan de variabelen boven de kolommen, en staan de factoren in de matrix.

- *Lineair Programmeren* (LP) is een manier om een oplossing te zoeken voor een (lineair) mathematisch model. Met behulp van lineaire algebra wordt het stelsel lineaire ongelijkheden doorgerekend. Hierbij wordt bijvoorbeeld gebruik gemaakt van het schoonvegen van een matrix.

- *Mixed Integer LP* (MILP) geeft ons de mogelijkheid om ook nog aan te geven of een bepaalde variabele geheeltallig moet zijn of niet. In veel gevallen is er de eis van geheeltalligheid. LP houdt hier in principe geen rekening mee.

In het vervolg van dit verslag gaan we kijken naar de mogelijkheden van Lineair Programmeren. De nadruk ligt vooral op de ontwikkeling van een mathematisch model dat met behulp van een dergelijke methode opgelost kan worden.

3 Definitie van een rooster

In dit verslag gebruiken we de term 'rooster'. Dit is echter niet een eenduidig begrip. In de meeste gevallen verstaan we onder een rooster een matrix waarin personeel uitgezet wordt tegen te werken dagen. Elk element uit de matrix bevat dan een taakomschrijving voor een persoon op een gegeven dag.

Voor een personeelslid is een rooster iets wat hem persoonlijk aangaat, en is dus alleen die enkele rij (of kolom) uit de matrix waarin zijn werkzaamheden staan. In dit verslag wordt de term rooster in beide gevallen gebruikt. Meestal betreft het echter een rooster voor een enkel persoon, en wordt in het andere geval over een samenstel van (persoonlijke) roosters gesproken.

Ook gaan we het begrip rooster verder uitbreiden. Een rooster schrijft in de praktijk voor elke dag één taak voor. We kunnen dit uitbreiden door per dag een verzameling van mogelijke taken op te nemen. Het klassieke rooster krijgen we door op elke dag een verzameling ter grootte één te nemen. Meerdere taken in de verzameling geven een keuze aan tussen de betreffende taken.

Door deze uitbreiding zijn we in staat om ook de nog niet ingeroosterde toestand te beschrijven. Deze kan namelijk worden voorgesteld door een rooster met op elke dag een verzameling van alle mogelijke taken. Met behulp van dit rooster kunnen andere roosters gemaakt worden onder voorwaarde dat elke verzameling taken op een gegeven dag een subset is van de verzameling in het oorspronkelijke rooster. Op deze manier zijn we ook in staat om (bijvoorbeeld handmatig) eerst een nadere specificering op een bepaalde dag aan te geven.

Een lege verzameling op een dag geeft aan dat de betreffende dag in het huidige rooster niet ingeroosterd is. Dit kan gebruikt worden als het rooster-proces gefaseerd doorlopen wordt. Een niet ingeroosterde dag kan dan in een volgende fase alsnog ingeroosterd worden. In alle andere gevallen kan de lege verzameling niet voorkomen.

Let wel dat een niet ingeroosterde dag niet hetzelfde is als een vrije dag. In het geval van een vrije dag wordt een taak 'vrij' gedefinieerd, en bestaat de verzameling op de betreffende dag uit alleen de taak 'vrij'.

In de laatste fase bestaat een rooster dus uit een reeks dagen met op elke dag precies één taak in de verzameling. De verzameling van taken op een dag is een subset van de verzameling in het rooster in de voorgaande fase.

Vanuit een rooster kan een volgend rooster gegenereerd worden door één of meerdere verzamelingen te verkleinen. Hierdoor onstaan er verschillende roosters, waaruit een keuze gemaakt kan worden. Het verkleinen van een verzameling kan op een aantal manieren gebeuren. De meest belangrijke bestaat uit het toepassen van externe regels die een beperking opleggen ten aanzien van de geldigheid van een rooster. Door deze regels zullen sommige taken uit een verzameling nooit in een geldig rooster voorkomen. Deze elementen kunnen we dus zonder problemen uit die verzameling verwijderen.

Stel er is een regel die zegt dat na een nachtdienst geen dagdienst mag volgen. Als er dan een bepaalde dag alleen nachtdiensten mogelijk zijn, dan kunnen we gevoegelijk alle dagdiensten uit de volgende dag verwijderen. Als er echter behalve die nachtdienst ook nog een dagdienst als mogelijkheid aanwezig is, mogen we de dagdiensten op de volgende dag niet verwijderen.

Een tweede mogelijkheid tot verkleinen krijgen we door het opsplitsen van een ver-

zameling in meerdere delen en de daardoor verkregen roosters met voorgaande manier afzonderlijk verder te verkleinen. Door het uitsplitsen van de nachtdiensten in het voorgaande voorbeeld, kunnen we in het ene rooster de daarop volgende dagdiensten dus verwijderen, en blijven ze in het tweede rooster aanwezig.

Verder kunnen we verzamelingen verkleinen door zelf een beperking op te leggen aan de mogelijke roosters. Als een planner alleen maar nachtdiensten wenst te roosteren, kunnen we het rooster verkleinen door alle diensten die geen nachtdienst zijn, uit het rooster te verwijderen. Daarna kunnen we de voorgaande technieken gebruiken voor verdere bewerking van de roosters.

We kunnen stoppen met het verkleinen van de verzamelingen als elke mogelijke opeenvolging van taken op de gegeven dagen is toegestaan, waarbij de taken gekozen worden uit de betreffende verzameling van die dag.

In het meest simpele geval is dat zo als elke verzameling precies een element bevat, en die enkele opeenvolging een geldig rooster oplevert. Als er een lege verzameling op een gegeven dag is, dan is het rooster geldig als er uiteindelijk minstens één taak ingepland kan worden op die betreffende dag, zodanig dat er een geldig rooster uit volgt. Dit betekent dat in het geval van een lege verzameling er naar de verzameling in het 'super'rooster gekeken moet worden. Als er in deze verzameling een taak voorkomt die binnen het huidige rooster past zonder dat de regels overtreden worden, dan kan het rooster als geldig beschouwd worden.

4 Generatie van roosters

Bij het genereren van de mogelijke roosters gaan we dus telkens één of meerdere verzamelingen opsplitsen. Hierdoor krijgen we roosters die meer specifiek zijn. In eerste instantie kunnen we een simpele techniek gebruiken voor het opsplitsen. Daarbij worden alle taken behalve één uit alle verzamelingen verwijderd. Dit levert een rooster op met op elke dag een verzameling met precies één of nul elementen. Vanuit dit rooster worden dan alle mogelijke combinaties gegenereerd. Uiteindelijk levert dit proces een verzameling roosters op waarbij op een aantal dagen één taak ingeroosterd staat. Over de andere dagen doet het rooster geen uitspraak (een lege verzameling op de betreffende dag). Daarna worden, door het toepassen van de opgelegde externe regels, alle roosters die niet aan de regels voldoen uit de verzameling van roosters verwijderd.

We hebben nu roosters gegenereerd die aan alle regels voldoen. Bij het uiteindelijke roosterproces kan zondermeer uit deze verzameling van roosters een keuze gemaakt worden.

Deze techniek kan gebruikt worden om een roosterplanning in fasen uit te voeren. Daarbij worden per fase één of meerdere taken gelijktijdig ingeroosterd. Door het aantal taken per fase te beperken, wordt ook de hoeveelheid roosters waaruit gekozen moet worden kleiner. Dit betekent ook een kleinere zoektijd om tot een oplossing te komen.

Duidelijk nadeel van deze techniek is de beperkte keuzevrijheid die we krijgen. Doordat we onze zoekruimte beperken, kunnen we hoogstwaarschijnlijk niet de meest optimale oplossing krijgen. In de praktijk wordt echter vaak op deze manier geroosterd, en dat levert nog steeds redelijk goede roosters op.

Zo worden op een afdeling van een ziekenhuis vaak eerst alle nachtdiensten ingeroosterd, daarna alle avonddiensten, en als laatste de dagdiensten. Bij het inroosteren van werkplekken kunnen we bijvoorbeeld als eerste de moeilijke werkplekken inroosteren, en als laatste die plekken die bijna altijd wel lukken.

Een vergroting van de zoekruimte kunnen we bewerkstelligen door het tijdelijk samenvoegen van taken tot één 'super'taak. Dit kunnen we doen als er geen enkele beperking is ten aanzien van verwisseling van de taken in een rooster. We kunnen taak A en taak B samenvoegen als alle regels die voor taak A gelden ook gelden voor taak B en omgekeerd. We maken nu een taak AB en werken daarmee verder. Na de keuze van een rooster kunnen we de taken A en B weer uit het rooster uitsplitsen naargelang de behoefte.

We kunnen ons dit voorstellen door een dienst op locatie A en een dienst op locatie B. In de regels is er geen beperking opgelegd tussen de beide locaties. Beperkingen die gelden voor locatie A gelden ook voor locatie B en omgekeerd. Door nu te abstraheren van de afzonderlijke locaties, en beide locaties in een 'super'locatie AB op te nemen, kunnen we uiteindelijke keuze tussen beide locaties uitstellen. Als er een persoon op locatie A nodig is dan is het rooster een mogelijkheid, zowel als op locatie B. Door het stellen van een ondergrens aan het totaal aantal personen op beide locaties, zorgen we ervoor dat er voldoende mensen aanwezig zijn om beide locaties te bemannen. Dus als op locatie A één persoon nodig is en op locatie B ook, dan zijn op 'super'locatie AB minstens twee personen nodig.

Na het roosterproces worden de 'super'taken weer uitgeplitst naar de afzonderlijke komponenten. Het is nu zeker dat aan de vraag voldaan kan worden, omdat we er voor gezorgd hebben dat er voldoende mensen aanwezig zijn.

Deze techniek komt vooral tot zijn recht als we grotere groepen samen kunnen stellen. Vooral op het gebied van werkplek planning geeft dit mogelijkheden. Als er bijvoorbeeld binnen één afdeling meerdere werkplekken zijn, is het waarschijnlijk zo dat er wel beperkingen zijn ten aanzien van het gekwalificeerd zijn van personeel voor een bepaalde werkplek, maar niet tussen de werkplekken onderling. In dat geval kunnen de werkplekken gecombineerd worden tot een 'super'werkplek. Door het toevoegen van ondergrenzen aan alle mogelijke combinaties binnen een 'super'werkplek, wordt uitgesloten dat er te weinig personeel beschikbaar is om alle werkplekken te bezetten.

Door het samenvoegen wordt het aantal roosters drastisch ingeperkt. Nadeel is wel dat er een extra aantal regels bijkomt om een minimale bezetting te garanderen. Groot voordeel van deze techniek is de grotere flexibiliteit tegen een relatief kleine hoeveelheid extra regels.

Het resultaat van deze actie is een verzameling personen die op de 'super'werkplek ingeroosterd worden. Na het inroosteren kunnen we op een simpele manier deze personen een definitieve werkplek toewijzen, waarbij we er zeker van zijn dat we alle werkplekken kunnen bezetten.

Verderop in dit verslag zullen we daar nog op terug komen.

5 Omschrijving eisen planning

Tijdens het maken van een roosterplanning krijgen we te maken met extra eisen aangaande de wenselijkheid van bepaalde roosters. Dit heeft meestal te maken met zaken zoals het personeelsbeleid, afspraken met bepaalde personen, CAO-eisen en dergelijke.

Omdat dergelijke zaken van invloed zijn op het wel of niet toestaan van een specifiek rooster, zullen we tijdens het genereren van de roosters hier al rekening mee moeten houden. In de navolgende lijst noemen we een aantal zaken die van toepassing kunnen zijn op een roosterplanning.

- Afspraken over bepaalde diensten worden vaak handmatig vastgelegd. Het roosterpakket zal dus in staat moeten zijn om informatie over al ingeroosterde dagen mee te nemen tijdens het genereren van de roosters.

- Vrije dagen, cursusdagen e.d. worden meestal van te voren ingeroosterd. Dit betekent dat bij de generatie van roosters hier expliciet rekening mee gehouden moet worden. Door het toevoegen van een kwalificatie 'vrij' en de betreffende dagen met 'vrij' in te roosteren, kan dit opgelost worden.

- Er is meestal een afspraak over het aantal dagen dat achtereen gewerkt wordt. Dit betreft een minimaal, een maximaal en een gewenst aantal dagen en/of diensten. De eisen wat betreft het minimaal en maximaal aantal dagen, zijn vaak harde eisen. Dit betekent dat er geen roosters gegenereerd mogen worden die hier niet aan voldoen. Een voorkeur voor een bepaald aantal dagen kan uitgedrukt worden door extra kosten mee te geven aan de roosters die afwijken van het gewenste aantal dagen.

- Bepaalde volgordes van diensten zijn niet toegestaan. Te denken valt bijvoorbeeld aan een 'dagdienst' na een 'nachtdienst'. Bij het genereren van de roosters moeten dergelijke combinaties uitgesloten worden.

- Bepaalde volgordes van diensten verdienen de voorkeur. Als bijvoorbeeld op de een dag een 'nachtdienst' gedraaid wordt, dan verdient het de voorkeur om een hele serie 'nachtdiensten' te draaien. Aan dergelijke roosters kan een voorkeur gegeven worden boven roosters die hier niet aan voldoen.

  Hier valt ook de negatieve voorkeur onder. Dus liever geen 'avonddienst' voor een vrije dag.

- Een speciale plaats neemt het weekeinde in. In veel gevallen is er een maximum gesteld aan het aantal weekeinden dat men ingeroosterd mag worden. Meestal is er een voorkeur voor het om-en-om vrij zijn tijdens een weekeinde. Dit vertaald zich dus weer in een voorkeur voor roosters die aan deze regel voldoen.

- Vaak zijn er een aantal standaard roosters waar men zoveel mogelijk mee wil werken. Te denken valt aan series van bepaalde diensten achterelkaar aan. Dit heeft als voordeel dat alle roosters wat meer uniform zijn. Min of meer uniforme roosters kunnen er voor zorgen dat de arbeidsomstandigheden wat prettiger worden. Roosters die hier niet aan voldoen, kunnen we een hogere penalty geven, om er voor te zorgen dat ze minder gauw gekozen worden.

- Als iemand de voorkeur uitspreekt voor het inroosteren van bepaalde diensten op bepaalde dagen, kan ook hier weer door middel van de kostenfunctie die voorkeur uitgedrukt worden in de gegenereerde roosters.

6 Beschrijving van het mathematisch model

Bij het gebruik van LP of MILP zullen we de informatie die door de menselijke planner wordt aangevoerd, om moeten werken naar een struktuur die geschikt is voor LP. Hiervoor hebben we een mathematisch model tot onze beschikking. De informatie moet omgezet worden naar een stelsel van lineaire vergelijkingen (of ongelijkheden). Met behulp van technieken uit de lineaire algebra wordt dan een oplossing gezocht voor dit stelsel.

Voor het beschrijven van een roosterprobleem maken we gebruik van een (lineair) mathematisch model. In dit model hebben we variabelen en lineaire relaties tussen deze variabelen. Deze relaties worden meestal constraints genoemd. In de meeste gevallen is de oplossing niet eenduidig bepaald. Er is meestal sprake van een verzameling oplossingen en niet van precies één punt in de oplossingsruimte. Daarom is er ook nog een te optimaliseren (lineaire) functie aanwezig. Met behulp van deze optimalisatie-functie kunnen we in de oplossingsruimte zoeken zodanig dat een optimum bereikt wordt voor deze functie onder handhaving van de constraints.

Een voorwaarde van het model is, dat we een punt nodig hebben van waaruit we kunnen beginnen te zoeken. Dit punt wordt in de oorsprong gelegd. Daarnaast is er de eis dat alle variabelen in het model niet-negatief zijn. Door toepassing van lineaire algebra kunnen we elk model omschrijven zodanig dat aan deze voorwaarden voldaan wordt. Vanuit de oorsprong wordt daarna stapsgewijs naar een oplossing toegelopen.

Van deze restricties merkt de gebruiker vaak weinig. Het verplaatsen van het startpunt naar de oorsprong kan zonder tussenkomst van de gebruiker afgehandeld worden. De enige restrictie waar de gebruiker wel iets van merkt is het niet-negatief zijn van variabelen ($x_i \geq 0$). Door het toepassen van lineaire algebra zijn we in staat om alle andere bereiken te construeren met behulp van deze variabelen. Een niet-positieve variabele ($x_i \leq 0$) krijgen we door substitutie van x_i door $-y_i$. Door het optellen van een constante kan de ondergrens van 0 verschoven worden. En door substitutie van x_i door $-y_i^- + y_i^+$ simuleren we een variabele zonder onder- of bovengrens.

Na het doorrekenen van het mathematisch model moeten we de substities omgekeerd toepassen om de originele variabelen van de juiste waarde te voorzien.

De constraints in het mathematisch basismodel zijn gelijkheden. Door het toevoegen van een vrije variabele die niet in de optimalisatie-functie voorkomt, kunnen we de kleiner-gelijk en groter-gelijk constraints beschrijven. Deze bewerking kan zonder tussenkomst van de gebruiker gebeuren, zodat de gebruiker hier geen rekening mee hoeft te houden.

De volgende twee constraints zijn gelijkwaardig:

$$\sum_{i=1}^{n} C_i x_i \geq B$$

$$\sum_{i=1}^{n} C_i x_i - y = B$$

Doordat y een variabele is met een ondergrens van 0, zal deze er voor zorgen dat de sommatie nooit kleiner kan worden dan B. Door het optellen van y inplaats van ze eraf te trekken krijgen we een kleiner-gelijk constraint. Let wel dat de variabele y verder in geen andere vergelijking gebruikt mag worden, anders ontstaat er een niet gewenste afhankelijkheid tussen de constraints.

De basisvorm van de constraint heeft ook tot gevolg dat er geen groter-dan of kleiner-dan constraints gemaakt kunnen worden. De keus is dus beperkt tot groter-gelijk ($\geq$), kleiner-gelijk ($\leq$) en gelijkheid ($=$).

Doordat de waarde van de variabelen afhangt van de constraints, is hiermee ook verklaart waarom het bereik van een variabele altijd een gesloten einde heeft. Het onmogelijk zijn van een groter-dan of kleiner-dan, maakt ook een open einde onmogelijk. We praten dus niet over een positieve variabele, maar over een niet-negatieve variabele.

Omdat het aantal oplossingen van een dergelijk model vaak zeer groot is, is er een optimalisatie-functie in het model aanwezig. Bij het oplossen van het model wordt dan geprobeerd om deze functie te optimaliseren onder handhaving van de constraints.

Er zijn twee typen optimalisatie-functie mogelijk: minimalisatie en maximalisatie. Beide zijn gelijkwaardig. Het onderscheid is vooral traditioneel bepaald. Afhankelijk van de gebruikte context wordt soms over minimaliseren danwel over maximaliseren gesproken. In het geval van de roosterplanning gaan we van kosten uit. Elk rooster geeft ons een kostenfactor. In dat geval gebruiken we minimalisatie om de kosten te minimaliseren.

Een ander punt is de representatie van het model. Het model wordt vaak als matrix gepresenteerd. In de rijen staan dan de constraints beschreven. De kolommen staan voor de variabelen in het model. De elementen in de matrix zijn de coëfficiënten van de variabelen binnen de constraints.

6.1 Lineair Programmeren

Lineair Programmeren (LP) is één van de manieren om een oplossing te vinden voor een lineair mathematisch model. De variabelen zijn in principe niet geheeltallig binnen LP. Als we geheeltallige oplossingen willen dan spreken we van Integer LP. Bij zowel geheeltallige als niet-geheeltallige variabelen in een oplossing spreken we van Mixed Integer LP (MILP). In het geval dat de variabelen alleen de integer waarden 0 of 1 aan kunnen nemen, spreken we van 0-1 LP of binair LP.

De restrictie van geheeltalligheid zorgt er voor dat de benodigde tijd voor het vinden van een oplossing met factoren omhoog gaat. In het algemeen wordt MILP uitgevoerd door een boom op te bouwen van gewone LP-oplossingen. Daarbij wordt telkens één van de variabelen vastgezet op een integer waarde. Door het toepassen van enkele

heuristieken probeert men de boom zo klein mogelijk te houden, en daarmee de looptijd ook zoveel mogelijk te beperken.

Met behulp van LP of MILP zijn we in staat om een oplossing te zoeken voor het lineaire mathematische model. De variabelen in dit model zijn in eerste instantie ongebonden. Als we alle variabelen op de een of andere manier een waarde toewijzen, dan zijn we in staat om de constraints uit te rekenen. Het resultaat van een constraint is een boolse waarde. Alle constraints moeten de waarde *TRUE* opleveren. Als niet aan alle constraints voldaan wordt, dan is de huidige keuze van de variabelen niet de juiste. Dit betekent dat we op zoek moeten naar een andere keuze.

Dit toewijzen van waarden aan variabelen gebeurt met behulp van een LP algoritme. Wij hebben er alleen voor te zorgen dat we een lineair mathematisch model construeren, dat ons probleem beschrijft. Als we zo'n model gemaakt hebben, pakken we een standaard algoritme uit de kast en laten we het model voor ons uitrekenen.

Als het model doorgerekend is, hebben alle variabelen een waarde gekregen. Als we voorbij gaan aan het feit dat er geen oplossing mogelijk is, dan weten we zeker dat aan al onze constraints voldaan is. Door een vertaling te maken van regels binnen het rooster naar constraints in het mathematisch model, dwingen we opvolging van de regels af.

Een willekeurige oplossing is meestal niet een goede oplossing. Zo'n oplossing is een onderdeel van de ruimte van alle toegestane oplossingen. Door het gelijktijdig optimaliseren van een optimalisatie-functie, kunnen we het algoritme naar een bepaald soort oplossingen sturen. Als we een juiste keuze maken van de optimalisatie-functie, kunnen we een betere oplossing krijgen dan zonder zo'n functie.

Een dergelijke optimalisatiefunctie kunnen we gebruiken om een voorkeur uit te drukken voor de ene combinatie boven de andere. Binnen het roosterprobleem kan op deze manier onderscheid gemaakt worden tussen personen of werkplekken. Het feit dat één persoon een beter resultaat oplevert dan een ander, geeft als resultaat dat indien mogelijk voor de eerste persoon gekozen zal worden, boven de tweede. In de optimalisatie-functie geven we dit aan door voor de eerste persoon een lagere kosten-factor mee te nemen dan voor de tweede persoon.

6.2 Modellering

Binnen het mathematisch model maken we onderscheid tussen variabelen en constraints. De variabelen gaan we koppelen aan variabelen binnen het roosterprobleem. De constraints gebruiken we om een vertaling te maken van de regels die een uitspraak doen over de variabelen in het roosterprobleem.

Omdat de constraints de beperkende factor zijn, zullen we eerst moeten inventariseren hoe de regels binnen het roosterprobleem er uit zien. De meeste regels doen een uitspraak over de regulering van de werktijden. Ze stellen bijvoorbeeld een minimum of een maximum aantal te werken dagen vast, of een aantal vrije dagen of weekeinden. Ook gaan ze over de onderlinge werkverdeling.

Daarnaast zijn er regels die bepaalde eisen stellen met betrekking tot de te vervullen taken. Te denken valt dan aan kwalificatie van personeel en de hoeveelheid personeel.

Als we alle regels alleen met behulp van constraints willen vertalen, dan krijgen we een zeer groot aantal constraints. Als we bijvoorbeeld het geval nemen dat een persoon

minimaal 2 dagen achtereen moet werken, dan heeft dat tot gevolg dat elk tweetal achtereenvolgende dagen per persoon een constraint genereerd. Omdat er nog een aantal van dit soort regels kunnen optreden, krijgen we ontzettend veel constraints.

Omdat de rekentijd van het systeem mede afhangt van de hoeveelheid constraints die in het mathematisch model aanwezig zijn, zijn we wel genoodzaakt om hier een oplossing voor te vinden.

Deze oplossing vinden we door het expliciet genereren van alle mogelijke roosters per persoon. In feite maken we een uitputtende verzameling van alle roosters die we maar kunnen bedenken, en die aan de gestelde eisen voldoen. We maken nu een apart stuk software die voor ons die verzameling genereert. Uit deze verzameling van roosters hoeven we alleen nog maar te kiezen.

Doordat we per persoon alle mogelijke roosters genereren, kunnen we alle regels die alleen op die persoon van toepassing zijn, weglaten uit het mathematisch model. Deze regels zijn namelijk al verwerkt in de gegenereerde roosters. Alleen de regels die nog betrekking hebben tussen personen onderling, zullen we moeten vertalen.

De keuze van een rooster is nu een variabele geworden in het roosterprobleem. Deze variabele kunnen we dus koppelen aan een variabele in het mathematisch model. Hiervoor koppelen we elk rooster aan een binaire variabele. De keuze van een rooster wordt dan bepaald door de waarde van de desbetreffende variabele.

Door het toevoegen van constraints moeten we er voor zorgen dat alleen valide selecties plaats vinden. Dus de selectie van twee roosters bij één persoon mag niet voorkomen.

We gebruiken de binaire variabelen ook in de constraints die de behoefte aan personeel specificeren. Dit doen we door de variabele mee te tellen in de constraint als dat rooster een bijdrage levert aan de vervulling van de behoefte. Dit doen we door een constraint te kiezen van de volgende vorm:

$$\sum_{i=1}^{n} C_i x_i \geq B$$

met

C_i $= 1$, indien x_i een bijdrage levert aan de vervulling van de behoefte B.
$= 0$, indien x_i niet meegeteld mag worden.
x_i : de binaire variabele die gekoppeld is aan een persoonlijk rooster.
B : een positief getal dat de totale behoefte aan personeel aangeeft.

Doordat elke x_i een binaire variabele is, krijgen we geen gedeeltelijke selectie van roosters. Door deze constraint hebben we onze zoekruimte teruggebracht tot een ruimte waarin tenminste B roosters geselecteerd worden. Tevens weten we zeker dat die roosters aan de aanwezigheid en kwalificatie van het personeel voldoen. Indien dit niet het geval is, dan is C_i gelijk aan 0.

Op deze manier zijn we in staat om een koppeling aan te brengen tussen een gespecificeerde behoefte en roosters om aan deze behoefte te kunnen voldoen.

Omdat we meerdere roosters per persoon hebben, moeten we tussen de roosters een wederzijdse uitsluiting construeren. Dit doen we door de volgende constraint op te stellen:

$$\forall p \in P : \sum_{i \in T_p} x_i = 1$$

met

P	: de verzameling van personen.
T_p	: de verzameling rooster identifiers van persoon p,
x_i	: de binaire variabele i die gekoppeld is aan rooster i en persoon p.

Dit is het basismodel waar we mee gaan werken. Hiermee zijn we in staat om toegestane oplossingen te vinden. Als we daarnaast ook nog een voorkeur aan willen geven voor bepaalde roosters, maken we gebruik van de optimalisatie-functie binnen LP. LP probeert namelijk een functie te optimaliseren onder handhaving van de aangegeven constraints. Door een juiste keuze van de coëfficiënten in de optimalisatie-functie kunnen we onze voorkeur uitdrukken voor het ene rooster boven het andere.

Als bijvoorbeeld een aantal personen een zwaardere werkbelasting hebben dan anderen, dan kunnen we de kosten voor het selekteren van een dergelijk persoon hoger maken. Door de optimalisatie-functie zal het LP algoritme proberen om deze personen een lichter rooster te geven.

Deze fase levert dus een programma op wat roosters kan genereren en daar een kostenfactor aan hangt. Daarna worden deze roosters in het mathematisch model ingepast, tezamen met de constraints die voor wederzijdse uitsluiting zorgen. Een 0-1 LP algoritme rekent het model door en geeft, indien mogelijk, een oplossing. Deze oplossing geeft dan aan welke roosters er geselecteerd zijn uit de verzameling van roosters. Als de oplossingsruimte niet leeg is, zal het algoritme altijd in staat zijn om een oplossing te vinden. Als de constraints het niet mogelijk maken om een legale oplossing te vinden, kunnen er altijd nog enkele trucs uit de LP hoed getrokken worden om hier wat aan te doen. Zo kunnen we een constraint afzwakken door een extra variabele in de constraint mee te nemen. Door aan deze variabele een zeer grote kostenfactor mee te geven, zal deze variabele in de meeste gevallen gelijk zijn aan nul. Alleen als niet op een normale manier aan de constraint voldaan kan worden, zal de variabele van een waarde voorzien worden. Voor verdere bestudering verwijs ik naar standaard literatuur op het gebied van de operations research.

7 Uitwerking

Ik ga nu proberen stap voor stap een mathematisch model op te bouwen. In het oorspronkelijke probleem hebben we met de volgende omstandigheden te maken:

- Er zijn een aantal diensten volgens welke het personeel werkt. Dit zijn een dagdienst, avonddienst en een nachtdienst.
- Een persoon heeft in principe maar één kwalificatie.

- Per dienst is er een behoefte aan een aantal mensen met een gespecificeerde kwalificatie.

De volgorde van oplossen zoals deze door de planner gebeurt is als volgt: Eerst worden alle nachtdiensten ingeroosterd, daarna de avonddiensten en als laatste de dagdiensten. Dit proces wordt dus niet in één keer gedaan maar in gedeelten. Dit heeft als voordeel dat het voor de planner overzichtelijker werkt.

In mijn eerste opzet ben ik van deze situatie uitgegaan wat betreft het laten oplossen door de LP solver.

De strategie voor het inroosteren van een nachtdienst ziet er dan als volgt uit:

- Eerst wordt er gekeken welke kwalificatie er nodig is tijdens een nachtdienst. Alle personen die aan deze kwalificatie voldoen en in een nachtdienst mogen werken worden geselekteerd voor het verdere proces.

- Van elk persoon dat geselekteerd is wordt een verzameling roosters opgesteld volgens welke die persoon mogelijkerwijs ingeroosterd kan worden. Met een rooster wordt hier bedoeld een persoonlijk rooster, dus dat onderdeel uit het totale rooster dat alleen op deze persoon van toepassing is.

- De gegenereerde roosters per persoon worden van een waardering voorzien. Deze waardering wordt uitgedrukt als een kostenfactor voor dat specifieke rooster. Dus als dat rooster gekozen wordt, dan levert dat een kostenfactor op voor de totale beoordeling van het uiteindelijke rooster.

- Er moet voor gezorgd worden dat alleen die roostercombinaties gekozen kunnen worden, die ervoor zorgen dat de hoeveelheid gekwalificeerd personeel voldoende is. Naast deze eis wordt geprobeert om de totale kosten zo laag mogelijk te houden.

In principe willen we alle dagen in één keer inroosteren. Op die manier kunnen we de globale kosten in de hand houden. Als we telkens stap voor stap gaan roosteren dan kunnen we in locale minima terecht komen, en geeft het eindresultaat grotere kosten dan bij het in één keer inroosteren.

In theorie is er geen beperking op het aantal dagen of dagdelen dat in één keer ingeroosterd wordt. In de praktijk krijgen we te maken met het exponentiële karakter van het systeem. Het aantal roosters dat gegenereerd kan worden is exponentieel in het aantal dagen. Daarom wordt er bijna altijd met een zogenaamde roosterhorizon gewerkt. Dit geeft het aantal dagen aan dat in één keer ingeroosterd mag worden.

Een variatie hierop is om een verschuivende horizon toe te passen. We nemen bijvoorbeeld een horizon van 7 dagen, en nemen alleen de eerste 4 dagen over. Daarna beginnen we vanaf dag 5 weer opnieuw. Op deze manier hopen we dat we voldoende invloed vanuit de toekomst in het huidige rooster mee nemen.

Tijdens testen blijkt een roosterhorizon van 7 tot 12 dagen nog vrij goed te werken. Bij een grotere horizon is het aantal gegenereerde roosters dermate groot dat het uitrekenen van het systeem te veel tijd gaat kosten.

De volgende stap is het omzetten van het roosterprobleem naar een omschrijving die we aan een LP solver aan kunnen bieden.

Elk rooster wordt geïdentificeerd door een binaire variabele. Daarna voegen we per persoon een constraint toe voor de wederzijdse uitsluiting.

Van alle regels hebben we alleen nog te maken met de inter-personele regels. Alle regels die betrekking hebben op één persoon, hebben we verwerkt in de generatie van de roosters. De regels die overblijven hebben betrekking op de hoeveelheid personeel die gevraagd wordt om een taak uit te voeren tijdens een bepaalde dienst.

Deze vraag kunnen we vertalen met behulp van een groter-gelijk constraint. We tellen het aantal roosters waarmee aan onze vraag voldaan kan worden, en eisen dat dit aan een minimum voldoet. Op deze manier verzekeren we ons ervan dat er een combinatie van roosters gekozen wordt, waarmee we aan onze vraag kunnen voldoen.

Dit doen we voor elke dag dat een gegeven kwalificatie verlangd wordt. Als we de matrix van het mathematisch model bekijken, dan zien we daar de gegenereerde roosters in de kolommen terug als een opeenvolging van nullen en enen. Het niet ingeroosterd zijn voor de betreffende kwalificatie is te zien als een '0', en het wel ingeroosterd zijn als een '1'. Als we het rooster uitdrukken als een opeenvolging van enen en nullen, in de volgorde van de dagen, dan zien we exact dat rijtje weer terug in de kolom van de matrix.

Als we meerdere kwalificaties in de matrix hebben staan, dan zien we de roosters met andere kwalificaties in de desbetreffende rijen als een opeenvolging van enen en nullen.

Voorbeeld:

- Persoon A bezit de kwalificatie VPK.
- Rooster 1 bij persoon A geeft aan dat A op maandag ingeroosterd is.
  Rooster 2 geeft aan dat maandag niet ingeroosterd is.
- Op maandag is er behoefte aan 3 personen met kwalificatie VPK.
- Als we bij persoon A aankomen, zien we dat deze de kwalificatie VPK bezit. Dat is inderdaad waar we naar zochten. Vervolgens gaan we alle gegenereerde roosters van persoon A bekijken.
  - Rooster 1 geeft aan dat persoon A voor maandag ingeroosterd is.
    Rooster 2 geeft aan dat A maandag niet ingeroosterd is.
  - Alleen rooster 1 wordt dus meegenomen in de constraint die 3 personen met kwalificatie VPK op maandag moet afdwingen.
  - Rooster 2 werkt niet mee om aan de vraag voor de huidige constraint te voldoen. Alhoewel persoon A de juiste kwalificatie heeft, wordt dit rooster niet meegenomen in het opbouwen van de constraint.
- Op deze manier bekijken we alle andere roosters van persoon A en alle roosters van de overige personen.

Hetzelfde kunnen we doen met de andere dagen die we nog in willen roosteren. En natuurlijk voor alle andere kwalificaties.

Uiteindelijk levert dit een mathematisch model op wat een afspiegeling is van het oorspronkelijke roosterprobleem. Door dit model aan een solver aan te bieden, wordt er een

oplossing gezocht die optimaal is voor het mathematisch model. Omdat we geprobeerd hebben een afspiegeling te maken van ons oorspronkelijke roosterprobleem, hopen we ook een optimaal rooster te krijgen.

In ieder geval weten we zeker dat alle taken ingeroosterd zijn. Ook weten we zeker dat per persoon precies één rooster geactiveerd is. Alleen in het geval dat er geen oplossing gevonden kan worden, is het mis gegaan met de invulling van de taken. In dat geval is het technisch mogelijk om in de solver te achterhalen welke constraint niet vervuld kan worden. Dit is echter niet een voorziening die standaard aanwezig is in alle solvers. In die gevallen zal men zelf een aanpassing in de code moeten maken om de gewenste gegevens te achterhalen.

Omdat we weten welke regels welke constraints genereren, kunnen we dus ook een terugkoppeling maken naar de gebruiker van het roosterprogramma. Een dergelijke fout zal verder door de gebruiker afgehandeld moeten worden, want het is het resultaat van eisen die te restrictief zijn.

Als we dit alles samenvatten, dan nemen we de volgende stappen:

- We kiezen het aantal dagen dat we in willen roosteren.
- We selecteren per dag de gewenste kwalificaties en aantallen.
- We selecteren de personen die in aanmerking komen voor het roosterproces.
- We genereren per persoon een verzameling roosters waaruit gekozen moet worden.
- Daarna genereren we een mathematisch model met daarin alle gegenereerde roosters, en eisen aangaande de vervulling van de taken.
- Op dit model laten we een LP algoritme los. Dat levert een oplossing voor het model.
- Als er een oplossing is voor het mathematisch model, dan kunnen we een oplossing maken voor het roosterprobleem. Dit doen we door die roosters als uiteindelijk rooster te kiezen, waarvan de binaire variable de waarde '1' heeft. Alle andere roosters kunnen we negeren.
- Het uiteindelijke resultaat van deze actie is dat we een aantal personen dusdanig ingeroosterd hebben dat de gespecificeerde taken uitgevoerd kunnen worden.

8 Diensten en werkplekken

Eén manier om met diensten om te gaan is door er expliciet rekening mee te houden. We kunnen diensten echter ook meenemen in de kwalificaties. Een kwalificatie geeft dan niet alleen een taakomschrijving, maar ook de dienst waarin die taak wordt uitgevoerd. In plaats van een kwalificatie 'vpk' krijgen we dan de kwalificaties 'vpk_dag', 'vpk_avond' en 'vpk_nacht'.

Een voordeel is de uniformiteit van de beschrijving. Als er sprake is van diensten hoeft men tijdens het modelleren er geen extra rekening mee te houden. Ook kan op deze manier eenduidig aangegeven worden welke kwalificaties een persoon heeft. Als

een verpleegkundige alleen maar overdag en 's avonds inzetbaar is, dan zijn alleen de kwalificaties 'vpk_dag' en 'vpk_avond' van toepassing, en niet 'vpk_nacht'.

Ook werkplekken kunnen we op een dergelijke manier in een kwalificatie versleutelen. Als er bijvoorbeeld 3 verschillende afdelingen zijn, met elk hun eigen team van verpleegkundigen, kunnen we een verpleegkundige bijvoorbeeld indelen als 'vpk_A', 'vpk_B' of 'vpk_C', al naar gelang in welk team hij meedraait. Een verpleegkundige die bijvoorbeeld zowel in team A als in team B mag werken krijgt dus zowel de kwalificatie 'vpk_A' als 'vpk_B'. Het zal duidelijk zijn dat de combinatie van werkplekken en diensten op deze manier ook mogelijk is.

De enige plaats waar we met deze materie te maken krijgen, is bij het genereren van de roosters. Omdat we voor een generieke structuur kiezen om een rooster te beschrijven, zijn we in staat om dit te verwerken. Alleen bij het opstellen van de regels die de generator gebruikt om de roosters te genereren, moeten we er voor zorgen dat er geen illegale combinaties kunnen onstaan. De generator zelf wordt er in principe niet veel complexer door.

9 Meerdere teamsamenstellingen

Een ander probleem is gelegen in het samenstellen van een team tijdens een dienst. In een aantal gevallen heeft men een aantal teamsamenstellingen waar uit gekozen mag worden. Dit betekent een aanpassing van het mathematisch model.

Een voorbeeld van zo'n samenstelling in een ziekenhuis is: 3 ervaren verpleegkundigen en 2 onervaren verpleegkundigen. Deze personen vormen samen een team dat dan een bepaalde dienst gaat draaien, bijvoorbeeld een nachtdienst. Een andere teamsamenstelling krijgen we door een andere verhouding en hoeveelheid van kwalificaties. Een teamsamenstelling zegt dus iets over de gewenste kwalificaties en niet over specifiek groep personen.

In het simpelste geval heeft men de voorkeur voor meer personeel tijdens een dienst. Dus in plaats van 3, liever 4 ervaren verpleegkundigen. Dit kan opgelost worden door een pseudo-persoon mee te nemen in het model. Deze pseudo-persoon wordt voorgesteld door een vrije binaire variabele in de desbetreffende constraints. Deze variabele krijgt in de optimalisatiefunctie een grote kostenfactor mee. Dit heeft tot gevolg dat het algoritme tot elke prijs gaat proberen om voldoende mensen in te roosteren en zodoende de kosten zo laag mogelijk te houden. Als dat echt niet lukt zit er niets anders op dan gebruik te maken van de extra variabele (lees: pseudo-persoon). Uiteindelijk heeft dit tot gevolg dat er één persoon minder ingeroosterd wordt tijdens de desbetreffende dienst.

Als verschillende teamsamenstellingen niet meer als lineaire combinaties op te schrijven zijn, zullen we onze toevlucht tot andere methoden moeten nemen. De eerste meest simpele is natuurlijk om per samenstelling het model apart door te rekenen. In de meeste gevallen is er een prioriteit aan een samenstelling gekoppeld, en zijn we eigenlijk alleen geïnteresseerd in de samenstelling met de hoogste prioriteit. Als we er in slagen om hieraan te voldoen, hoeven we niet verder te rekenen.

Als de teamsamenstellingen moeilijker zijn, zullen we expliciet met elke samenstelling rekening moeten houden. In dit geval kunnen we gebruik maken van zogenaamde 'either-or' constructies. Per samenstelling krijgen we een extra (binaire) variabele binnen

het model die de selectie van die specifieke samenstelling aangeeft. Per samenstelling gaan we dan constraints genereren, waarbij we er voor zorgen dat deze constraints alleen actief worden wanneer de samenstelling actief is. Dit activeren en deactiveren van een constraint doen we door het selectief toevoegen van een extra variabele met een grote coëfficient aan de constraint. De constraints krijgen dan de volgende gedaante: (onderscheid naar $\geq$ en $\leq$)

$$M\overline{y} + \sum_{i=0}^{n} C_i x_i \;\geq\; B$$

$$-M\overline{y} + \sum_{i=0}^{n} C_i x_i \;\leq\; B$$

met

M	: een grote constante, die groter is dan de B.
$\overline{y}$	: de negatie van de binaire variabele die de selectie van een teamsamenstelling aangeeft.
C_i	$= 1$, indien x_i een bijdrage levert aan de vervulling van de behoefte B.
	$= 0$, indien x_i niet meegeteld mag worden.
x_i	: de binaire variabele die gekoppeld is aan rooster i.

Als we naar de constraints kijken zien we dat er constante M aan de constraint toegevoegd is. In het geval dat y gelijk is aan 1, is $\overline{y}$ gelijk aan 0, en valt de constante M weg uit de constraint. Op dat moment blijft de originele constraint over. In het geval dat $\overline{y}$ gelijk aan 1 is, zorgt de constante M er voor dat in alle gevallen aan de constraint voldaan wordt. De aanwezigheid van M is in alle gevallen voldoende voor het voldoen aan de constraint. Op deze manier is de constraint redundant geworden als y gelijk is aan 0. Op deze manier zorgen we er voor dat constraints alleen maar in één geval actief worden. Door er voor te zorgen dat tussen de samenstellingen een wederzijdse uitsluiting aanwezig is, is maar één samenstelling actief.

Natuurlijk kunnen we ook hier, door middel van een kostenfunctie, sturing geven aan de keuze van een samenstelling. Gelijkwaardige samenstellingen geven we gelijke kosten. Samenstellingen van lagere prioriteit geven we hogere kosten. De hoogte van deze kosten bepaalt het omslagpunt, waarbij er nog gekozen wordt voor die ene samenstelling, tegen de kleinere kosten die de roosters met zich mee brengen.

10 Meerdere kwalificaties

Tot nu toe zijn we er van uitgegaan dat er bij het maken van een rooster ieder persoon maar met één kwalificatie per keer in het proces meedoet. Als een persoon tijdens het genereren met meer dan één kwalificatie meetelt, dan krijgen we een probleem bij het genereren van de mogelijke roosters.

Stel dat iemand op meerdere werkplekken ingezet kan worden, die geen onderlinge relaties hebben. Als deze persoon bijvoorbeeld 4 kwalificaties heeft, dan vermenigvuldigt

het aantal mogelijke roosters met een factor 5 voor elke dag binnen de roosterhorizon. Op elke dag kan hij namelijk één van deze kwalificaties vervullen of niet ingeroosterd worden. Bij een roosterhorizon van 7 dagen geeft dat dus 5^7 (= 78125) mogelijke roosters bij deze persoon. Het zal duidelijk zijn dat dit niet erg wenselijk is. Bovendien is het ook nog zeer goed voor te stellen dat iemand nog meer dan deze 4 kwalificaties bezit.

Voor dit probleem zijn een aantal oplossingen. Alle oplossingen hebben echter het nadeel dat ze niet alle mogelijke roosters meer toestaan en/of dat de kans groot wordt dat een sub-optimale oplossing gemaakt wordt.

De meest simpele oplossing is het verkleinen van de roosterhorizon naar een kleiner aantal dagen. Dit beperkt het aantal te genereren roosters drastisch. Nadeel hiervan is dat er nu nog meer dan voorheen op lokale gronden een keuze voor een rooster gemaakt wordt. Dit is echter zeker een manier die aandacht verdient.

Een tweede oplossing is het uitsluiten van een aantal mogelijke roosters. Dit gebeurt bijvoorbeeld door per rooster maar één kwalificatie in te roosteren. Bij 4 kwalificaties en 7 dagen krijgen we dus $4 \cdot 2^7$ mogelijke roosters in plaats van 5^7. Hiermee sluiten we echter wel een zeer groot aantal roosters uit, en het is nu nog maar de vraag of er wel een oplossing voor het ontstane model te vinden is.

Een variant op de voornoemde oplossing is het samenvoegen van meerdere kwalificaties binnen één 'super'kwalificatie. Bij het genereren van de roosters laten we dan even in het midden in welke specifieke kwalificatie iemand gaat werken, maar geven alleen aan dat hij in deze zogenaamde 'super'kwalificatie gaat werken. Het aantal roosters verminderd dan tot 2^7. Na het doorrekenen van het model moeten we echter nog een laatste slag over de oplossing doen om uit de 'super'kwalificatie de specifieke kwalificatie toe te wijzen. Als we echter zeker weten dat we aan de behoefte kunnen voldoen, dan is dat laatste geen probleem meer. Ook qua rekentijd stelt deze fase niets voor, omdat we deze toewijzing per dag uit kunnen rekenen zonder ons iets van andere dagen aan te trekken.

Deze laatste oplossing wil ik graag beter bekijken, om te zien wat voor mogelijkheden we hier nog hebben. Dit doen we door een voorbeeld uit te werken.

Voorbeeld

Stel dat we te maken hebben met 2 afdelingen in een ziekenhuis. Elke afdeling heeft zijn eigen verpleegkundigen. Afgezien van de diensten hebben de verpleegkundigen dus maar één kwalificatie. Stel dat er nu ook een verpleegkundige is die op beide afdelingen werkt. Deze heeft dus 2 kwalificaties. In een werkweek kan deze verpleegkundige dus afwisselend voor de ene dan wel voor de andere afdeling werken.

Bij het genereren van de roosters worden beide kwalificaties gekoppeld binnen een 'super'kwalificatie. Bij het construeren van de constraints die de behoefte specificeren, nemen we de 'super'kwalificatie in beide specifieke kwalificaties mee. Dus deze persoon wordt op één dag zowel meegetelt bij de eerste als bij de tweede afdeling.

Nu krijgen we echter het probleem dat het mathematisch model niet meer een correcte omschrijving is van het roosterprobleem. Deze verpleegkundige kan door het huidige model gelijktijdig op beide afdelingen ingeroosterd worden. Het zal duidelijk zijn dat dit niet de bedoeling is.

Een oplossing kan gevonden worden door niet alleen een behoefte te specificeren per afdeling, maar ook door specificatie van de behoefte bij vereniging van beide afdelingen. Stel dat afdeling A 10 verpleegkundigen nodig heeft en afdeling B 15. Door het dubbel tellen van die ene verpleegkundige missen we op het totaal 1 verpleegkundige. Als we nu ook nog specificeren dat de afdelingen A en B tezamen 25 verpleegkundigen nodig hebben, dan zorgen we ervoor dat aan de behoefte voldaan kan worden.

De eerste constraints zorgen er voor dat er tenminste 10 personen voor afdeling A en 15 voor afdeling B aanwezig zijn. Door te stellen dat het totaal aantal verpleegkundigen voor de afdelingen A en B 25 moet zijn zorgen we ervoor dat er aan de gevraagde behoefte voldaan kan worden. Na het doorrekenen van het model doen we een afsluitende toewijzing van een specifieke kwalificatie uit de 'super'kwalificatie.

Deze methode is ook uit te breiden naar meerdere kwalificaties. Als we bijvoorbeeld niet 2 maar 3 of meer afdelingen hebben met een aantal personen die multi-inzetbaar zijn, dan krijgen we eenzelfde constructie met een extra constraint die het totaal aan personen over de vereniging van de afdelingen specificeert.

Een volgend probleem dient zich aan bij een overlap tussen de zogenaamde 'super'-kwalificaties. Stel dat we de afdelingen A, B, C hebben. Een aantal personen bezit de kwalificatie voor de afdelingen A en B. Een groep andere personen bezit de kwalificaties voor de afdelingen B en C. We kunnen dan niet zondermeer stellen dat het verenigen van alle afdelingen tot een correct model leidt. Er zijn situaties mogelijk die een niet-correcte oplossing opleveren.

Hier volgt een voorbeeld van een dergelijke configuratie. P_A, P_B en P_C zijn verzamelingen waarin de personen k, l, m, n en o op de volgende manier in voor komen:

$$P_A = \{k, l, m\}$$

$$P_B = \{m, n\}$$

$$P_C = \{n, o\}$$

De volgende constraints voldoen:

$$\sum p \in P_A \geq 1$$

$$\sum p \in P_B \geq 2$$

$$\sum p \in P_C \geq 2$$

Ook de extra constraint voldoet want er zijn 5 personen aanwezig:

$$\sum p \in P_A \cap P_B \cap P_C \geq 5$$

Helaas is dit niet een correcte oplossing, want er is niet genoeg personeel voor afdeling B of C. In A zitten twee personen die nergens anders ingezet kunnen worden dan op A. Dit betekent automatisch dat op één van de andere afdelingen er een tekort optreedt.

Als we beter naar het probleem kijken, zien we dat we een oplossing kunnen krijgen door niet alleen een uitspraak te doen over de vereniging van alle afdelingen, maar door

ook uitspraken te doen over subsets van die vereniging. Dus door ook te kijken naar de behoefte per twee- en per drietal afdelingen.

In het voorgaande voorbeeld betekent dit het toevoegen van de volgende constraints:

$$\sum p \in P_A \cap P_B \geq 3$$

$$\sum p \in P_B \cap P_C \geq 4$$

$$\sum p \in P_A \cap P_C \geq 3$$

We zien nu dat niet aan de behoefte voldaan kan worden omdat er nu niet aan alle constraints voldaan kan worden. Dit is echter een probleem van de planner en niet van ons systeem. Wij zijn nu tenminste in staat om aan te geven of een oplossing wel of niet goed is.

Door van elke combinatie de behoefte aan te geven, zijn we weer in staat om een correct model te construeren. Probleem hierbij is echter dat we op deze manier nogal veel extra constraints nodig hebben. Dit heeft tot gevolg dat de rekentijd nogal toe kan nemen. Het vermoeden is dat een groot aantal van deze constraints overbodig is.

In het voorbeeld kunnen we hoogstwaarschijnlijk de laatste constraint weglaten omdat er geen directe relatie is tussen A en C. Voor mijn gevoel kunnen we dat op de volgende manier aanpakken: Per persoon hebben we een verzameling van kwalificaties. Nu maken we een systeem van deze verzamelingen. Daarna gaan we van elk tweetal verzamelingen uit dit systeem de wederzijdse restverzameling en de vereniging nemen. Dit levert ons een drietal verzamelingen op die we toevoegen aan het bestaande systeem van verzamelingen. Dit doen we net zolang todat er geen nieuwe verzamelingen meer toegevoegd worden. In het uiterste geval hebben we de machtsverzameling opgebouwd. Waarschijnlijker is het echter dat het systeem een deelverzameling van die machtverzameling is.

Op dit moment kan ik dit nog niet met een bewijs onderbouwen. Om veilig te zijn kunnen we altijd de machtsverzameling nemen. Mijn vermoeden wordt echter gestaafd door het feit dat er niets aan het systeem veranderd als we één kwalificatie door twee subkwalificaties vervangen. Als alle personen die eerst kwalificatie A hadden, nu kwalificaties A_1 en A_2 hebben, dan is aan de omstandigheden niets gewijzigd. Het door mij voorgestelde systeem van verzamelingen blijft ook even groot. Alleen de machtsverzameling breidt zich uit.

Dit geeft reden om hier verdere studie naar te doen.

11 Conclusie

Tijdens mijn afstuderen heb ik laten zien op welke wijze de huidige solver van ZKR vervangen kan worden door een solver gebaseerd op LP. Tevens is er een methode ontwikkeld om op een meer universele manier met rooster-problemen om te gaan. Zo is het nu mogelijk om ook werkplekplanning te gaan doen met de ontwikkelde technieken. Door het veralgemeniseren van een kwalificatie kunnen ook een dienst en een werkplek als kwalificatie aangemerkt worden. Voor het systeem is het om het even wat de achtergrond van een kwalificatie is.

Voor het verruimen van een kwalificatie is het ook nodig dat er een ruimere definitie van een rooster komt. Ook dat is gebeurd.

Echte puristen zouden zelfs een dag nog als kwalificatie aan kunnen merken. Voordeel hiervan is dat we dan nog maar met één grootheid te maken hebben in plaats van twee. Deze mogelijkheid laat ik echter open voor de toekomst. Mijn inziens is ze echter wel het bestuderen waard.

De voornaamste plaats waar nu naar gekeken moet worden, betreft het genereren van de roosters. Ik heb gemerkt dat de echte intelligentie in dat gedeelte zal zitten. Voor eenvoudige gevallen zoals een minimum of maximum aantal dagen is het vrij eenvoudig om een generator te bouwen. Moeilijker wordt het al bij de beoordeling van een rooster of het nemen van een beslissing over het wel of niet genereren van een rooster.

Ik heb mij echter beperkt tot het mathematisch model voor solver. Dat is in principe het fundament waar alles mee staat of valt. Als we er in slagen om een zo universeel mogelijk gereedschap te krijgen, kunnen we later veel meer situaties aan.

Wat betreft het vertalen naar een mathematisch model, denk ik dat vooral het tijdelijk samenvoegen van kwalificaties grote mogelijkheden biedt. Hierdoor kunnen we grote delen van de zoekruimte samenbrengen tot één punt. Deze methode geeft wel een aantal constraints extra, maar dat wordt gecompenseerd door de veel grotere vrijheid die we krijgen in de keuze van een rooster.

Verder verwonder ik mij erover dat niemand anders een dergelijk systeem heeft opgezet met behulp van Lineair Programeren. Als ik terug kijk op het resultaat wat bereikt is, dan is het uiteindelijk nog vrij elementair. Ik heb echter geen enkele aanwijzing kunnen vinden van een dergelijke generieke aanpak van het roosterprobleem zoals dat hier gebruikt wordt.

12 Referenties

Wiliams, H.P., *Modelbuilding in mathematical programming*, John Wiley and Sons, New York

Panne, C. van de, *Linear programming and related techniques*, North Holland Publishing Company, Amsterdam

Garfink, R.S., and G.L. Nemhauser, *Integer programming*, John Wiley and Sons, New York

news:sci.op-research

news:comp.constraints

Appendix: Een voorbeeld

Lijst met personen en hun kwalificaties.

persoon	**kwalificaties**
Greven-Weerdmeester, W	vrij, secr_dag
vd Wal, K	vrij, vpke_dag, vpke_avond, vpke_nacht
Wilmink-Pel, T	vrij, vpke_dag, vpke_avond, vpke_nacht
Boonstra-Venema, A	vrij, vpke_dag, vpke_avond, vpke_nacht
de Jong-Swart, EC	vrij, vpke_dag, vpke_avond, vpke_nacht
Konstapel-Haverkamp, DB	vrij, vpke_dag, vpke_avond, vpke_nacht
Krom, T	vrij, vpke_dag, vpke_avond, vpke_nacht
Niemarkt, JCM	vrij, vpke_dag, vpke_avond, vpke_nacht
Rijnbeek-Schaapveld, ELF	vrij, vpke_dag, vpke_avond, vpke_nacht
Schokker-Jongsma, R	vrij, vpke_dag, vpke_avond, vpke_nacht
leerling1	vrij, ll_dag, ll_avond, ll_nacht
leerling2	vrij, ll_dag, ll_avond, ll_nacht

Lijst met uit te voeren taken. De elementen stellen de benodigde hoeveelheid personeel voor.

	ma	**di**	**wo**	**do**	**vr**	**za**	**zo**
secr_dag	1	1	1	1	1		
vpke_dag	3	3	2	2	2	2	2
ll_dag	1	1	1	1	1	1	1
vpke_nacht	2	2	2	1	1	2	1

Voorbeeld van een aantal regels die gelden voor het roosterproces.

Maximaal 10 dagen achter elkaar
Minimaal 2 dagdiensten achter elkaar
Maximaal 5 dagdiensten achter elkaar
Minimaal 3 nachtdiensten achter elkaar
Maximaal 7 nachtdiensten achter elkaar
Na een nachtdienst geen dagdienst
Maximaal 2 weekends achter elkaar
Voorkeur voor een reeks diensten van één type
Voorkeur voor om de week een weekend werken

Een gedeelte van de gegenereerde roosters voor 'Greven-Weerdmeester, W'. Een 1 stelt voor dat de betreffende taak ingeroosterd is op die dag.

kwalificatie	**ma**	**di**	**wo**	**do**	**vr**	**za**	**zo**	**kosten**
secr_dag	1	1	1	1	1			6
	1	1	1	1				5
	1	1	1					4
	1	1						3
		1	1	1	1			5
		1	1	1				4
		1	1					3
			1	1	1			4

De gegenereerde constraints zoals ze dan in de LP tabel te zien zijn. Deze worden gegenereerd a.d.h.v. de voorgaande roosters en de takentabel

	Greven-Weerdmeester, W									
		x_1	x_2	x_3	x_4	x_5	x_6	x_7	x_8	
Minimize:	$\cdots$	6	5	4	3	5	4	3	4	$\cdots$	
secr_dag,ma	$\cdots$	1	1	1	1					$\cdots$	≥ 1
secr_dag,di	$\cdots$	1	1	1	1	1	1	1		$\cdots$	≥ 1
secr_dag,wo	$\cdots$	1	1	1		1	1	1	1	$\cdots$	≥ 1
secr_dag,do	$\cdots$	1	1			1	1		1	$\cdots$	≥ 1
secr_dag,vr	$\cdots$	1				1			1	$\cdots$	≥ 1

```
/*-------------------------------------------------------------------------
 * FILE    : gen3.c
 * AUTHOR : Berend Reitsma
 *
 */


void InitMatrixSettings(void)
/* Berekent welke taken en personen per dag nodig zijn.
 * Bepaalt hoogte en breedte van de matrix.
 * Initialiseert boekhouding per persoon en per taak (cross ref).
 * Per persoon moeten de roosters al gegenereerd zijn.
 */

void CreateMatrix(void)
/* Maakt de matrix. Afmetingen zijn tevoren bepaald m.b.v InitMatrixSettings()
 */

void SetMatrix(uint row, uint col, uint val)
/* Zet een element in de matrix
 */

void SetCost(uint col, uint val)
/* Zet kostenfactor voor een variabele
 */

void SetRH(uint row, int constr, uint val)
/* Zet de right-hand-side van een constraint
 */


void InitMatrix(void)
/* Initialiseert en vult de matrix
 */

void PrintSolution(void)
/* Print een eenvoudige tabel met de namen en de dagen waarop gewerkt
 * moet worden.
 */

void PrintSolutionSch(void)
/* Print de oplossing op een dusdanige manier dat deze weer als input in een
 * verdere fase kan dienen.
 */

void SetSolution(void)
/* Initialiseert de variabelen solTaskGroup en solPattern.
 * Dient voor een snelle lookup van de gekozen oplossing.
 */

void SetSolutionSched(void)
/* Voert de definitieve toewijzing van taken aan personen uit nadat de
 * matrix is doorgerekend.
 * Splitst daarbij ook de 'super'taken.
 */


/*-------------------------------------------------------------------------
 * FILE    : hgen.c
 * AUTHOR : Berend Reitsma
 *
 */

void CompressMix(void)
/* Verwijdert subsets van grotere sets uit PInfo.taskMixList
 */

void MixSingleTask(AList *taskList)
/* Voegt elke task afzonderlijk toe in PInfo.taskMixList
 */

void AddDayToPatterns(bitint *bitmask, PersonRec *person, uint day)
/* Breidt alle bitmasks uit met de huidige dag
 */

int Check(CheckComboRec *checkv, bitint mask)
/* Algemene check functie voor het checken van geldige roosters
 */

int CheckEnd(struct GenRec *rec, bitint mask)
```

```
/* Staart checking van een rooster. Kijkt een aantal dagen vooruit naar mogelijk
 * vast ingeroosterde dagen.
 */

void Gen0(struct GenRec *rec, bitint mask)
/* Probeert een rooster te genereren met alleen een 0 op deze positie.
 */

void Gen0End(struct GenRec *rec, bitint mask)
/* Probeert een rooster te genereren met alleen een 0 op deze positie.
 * Als dit slaagt, wordt het rooster bij de huidige verzameling roosters
 * gevoegd.
 */

void Gen1(struct GenRec *rec, bitint mask)
/* Idem aan Gen0, alleen dan met een 1
 */

void Gen1End(struct GenRec *rec, bitint mask)
/* Idem aan Gen0End, alleen dan met een 1
 */

void Gen01(struct GenRec *rec, bitint mask)
/* Probeert zowel een rooster met een 0 als een 1 te genereren.
 */

void Gen01End(struct GenRec *rec, bitint mask)
/* Probeert zowel een rooster met een 0 als een 1 te genereren.
 * Elk geldig rooster wordt toegevoegd aan de verzameling.
 */

void GenError(struct GenRec *rec, bitint mask)
/* Genereert een fout als deze functie aangeroepen wordt.
 * Wordt gebruikt om interne fouten af te vangen indien de structuren
 * niet goed gebouwd zijn.
 */

void GeneratePatterns(void)
/* Genereert alle roosters voor alle benodigde personen en taken.
 */

void GeneratePatterns_1(PersonRec *person)
/* Genereert alle roosters bij een persoon.
 * Wordt aangeroepen door GeneratePatterns.
 */

void GeneratePatterns_0(PersonRec *person, AList *taskList)
/* Genereert alle roosters bij een persoon voor alle taken uit taskList.
 * Gebruikt de check-functies.
 */

void AddPattern(TaskGroupRec *p, bitint pat, uint cost)
/* Voegt een rooster toe aan de huidige set van roosters.
 */

uint CalcCost(bitint mask)
/* Berekent een (eenvoudige) kostenfactor
 */

void PrintTaskList(FILE *out, AList *taskList)
/* Print alle taken, gescheiden door komma's.
 */

/*-------------------------------------------------------------------------*/

void CleanTodo(void)
/* Hulpfunctie voor het verwijderen van overbodige taken uit PInfo.taskList
 */

void SetupDayInfo(void)
/* Hulpfunctie voor het opzetten van een tabel met behoefte per dag per taak.
 */

int ListExists(AList *list, AList *listList)
/* Test of een lijst al in een lijst van lijsten voorkomt.
 * Alleen voor taaklijsten !!!
 */

void Create3List(AList *intersect, AList *dif1, AList *dif2, AList *l1, AList *l2)
/* Genereert doorsnede en wederzijdse verschillen van twee (taak)lijsten.
```

```
 */

void DayInfoAddTaskList(uint index, AList *list)
/* Voegt een taaklijst toe aan een dag
 */

void ExtendDayInfo(void)
/* Voegt de benodigde extra taaklijsten toe aan de dagen, om er voor te zorgen
 * dat er geen dubbele inroostering kan plaats vinden.
 * Elke doorsnede en vereniging binnen de huidige verzameling wordt toegevoegd.
 * Deze functie maakt gebruik van een nog niet bewezen stelling...
 */

void ExtendDayInfo_1(AList *combList, AList *newList, AList *intersect, int n)
/* Voegt alle doorsnedes aan de verzameling toe.
 */

void ExtendDayInfo_2(AList *combList, AList *newList, AList *unionList, int n)
/* Voegt alle verenigingen aan de verzameling toe.
 */

/*---------------------------------------------------------------------------
 * FILE   : hlpmat.c
 * AUTHOR : Berend Reitsma
 *
 */


/* Matrix manipulatie functies
 */

LPRec *CreateLP(int nrRows, int nrCols)
void DeleteLP(LPRec *lp)
void LPSetMatrix(LPRec *lp, int row, int col, int val)
void LPSetCost(LPRec *lp, uint col, int val)
void LPSetRH(LPRec *lp, uint row, int constr, uint rh)

/*---------------------------------------------------------------------------*/

void PrintTerm(FILE *out, int factor, int varnr)
/* Hulpfunctie voor printen van een LP matrix
 */

void PrintLPProblem(FILE *out, LPRec *lp)
/* Print de LP matrix in een formaat geschikt voor lp_solve
 */

int ReadLPResult(FILE *in, LPRec *lp)
/* Leest de oplossing van de LP matrix in
 */

int SolveWithLib(LPRec *lp)
/* Roept lp_solve aan d.m.v. een gelinkte lib
 */

int SolveSingle(LPRec *lp)
/* Probeer een unieke oplossing te vinden
 */

int Solve(LPRec *lp)
/* Roept lp_solve aan. Afhankelijk van compile-time opties als extern programma
 * of als functie in de lib.
 */


/*---------------------------------------------------------------------------
 * FILE   : hpers.c
 * AUTHOR : Berend Reitsma
 *
 */


/* Een aantal functies voor personen administratie
 */

PersonRec *CreatePerson(char *name)
PersonRec *LookupPerson(AList *personList, char *name)
PersonRec *CreateOrLookupPerson(AList *personList, char *name)

SchedRec *CreateSched(uint day)
```

```
SchedRec *LookupSched(AList *schedList, uint day)
SchedRec *CreateOrLookupSched(AList *schedList, uint day)

void PersonSetTaskList(PersonRec *p, AList *l)
void PersonAddTask(PersonRec *p, TaskRec *t)
void PersonAddTaskGroup(PersonRec *p, TaskGroupRec *r)
void PersonSetSched(PersonRec *p, uint day, AList *taskList)
AList *PersonGetSched(PersonRec *p, uint day)
void SchedAddTask(SchedRec *r, TaskRec *task)


/*-------------------------------------------------------------------------
 * FILE   : hrule.c
 * AUTHOR : Berend Reitsma
 *
 */


/* Een aantal 'rules' met install, handle en check functies */

void HandleMix(void **argv)


/* Maximaal aantal dagen achtereen ingeroosterd */
CheckComboRec InstallCheckMax(int day, AList *taskList, bitint *bitmask, void **argv)
int CheckMax(void *data, bitint mask)
void HandleMax(void **argv)

/* Minimaal aantal dagen achtereen ingeroosterd */
CheckComboRec InstallCheckMin(int day, AList *taskList, bitint *bitmask, void **argv)
int CheckMinHalf(void *data, bitint mask)
int CheckMinEnd(void *data, bitint mask)
void HandleMin(void **argv)

/* Maximaal 2 weekeinden achtereen ingeroosterd */
CheckComboRec InstallCheckMax2WE(int day, AList *taskList, bitint *bitmask, void **argv)
int CheckMax2WE5(void *data, bitint mask)
int CheckMax2WE6(void *data, bitint mask)
void HandleMax2WE(void **argv)


void AddCheckFn(CheckFnRec *r)
/* Voegt een checkfunctie toe aan de bestaande lijst
 */

RuleFnRec *FindRuleFn(char *name)
/* Zoekt een rule a.d.h.v. de naam
 */


/*-------------------------------------------------------------------------
 * FILE   : htask.c
 * AUTHOR : Berend Reitsma
 *
 */


/* Een aantal functies voor administratie van taken
 */

TaskRec *CreateTask(char *name)
TaskRec *LookupTask(AList *taskList, char *name)
TaskRec *CreateOrLookupTask(AList *taskList, char *name)

RequiredRec *CreateRequired(uint day)
RequiredRec *LookupRequired(AList *requiredList, uint day)
RequiredRec *CreateOrLookupRequired(AList *requiredList, uint day)

void TaskAddPerson(TaskRec *task, PersonRec *person)
void TaskSetRequired(TaskRec *task, uint day, uint count)
uint TaskGetRequired(TaskRec *task, uint day)

void TaskListSetId(AList *taskList)
/* Zet task->id gelijk aan index in taskList voor snel indexeren.
 */

void SetTag(AList *l, int tag)
/* Zet task->tag. Voor gebruik in verzameling rekening e.d.
 */
```

```
int DoesIntersect(AList *a, AList *b)
/* Checkt op doorsnede van twee (taak)lijsten
 */

void CreateIntersect(AList *dst, AList *a, AList *b)
/* Geeft de doorsnede van twee lijsten
 */

void CreateUnion(AList *dst, AList *a, AList *b)
/* Geeft de vereniging van twee lijsten
 */

int IsSubset(AList *a, AList *b)
/* Checkt of lijst a een subset is van b
 */

/*--------------------------------------------------------------------------
 * FILE   : read.c
 * AUTHOR : Berend Reitsma
 *
 */


void SkipComment(void)
/* Springt over commentaar heen in de datafile
 */

int GetPattern(void)
/* Leest een pattern in
 */

int GetToken(void)
/* Leest een token in
 */

void ScanInfoFile(char *filename)
/* Leest de info file in. (filenaam info e.d.)
 */

void ScanPersFile(char *filename)
/* Leest de personfile in.
 */

void ScanSchedFile(char *filename)
/* Leest de schedulefile in. (Ingeroosterde dagen)
 */

void ScanTaskFile(char *filename)
/* Leest de taskfile in. (Nog te vervullen taken)
 */

void ScanRuleFile(char *filename)
/* Leest de rulefile in. (Regels voor het genereren van roosters)
 */

void ScanTodoFile(char *filename)
/* Leest todo file in. (Huidige fase van roosteren)
 */

void ScanTaskListExceptions(AList *tList, AList *globalList)
/* Hulpfunctie voor inlezen van een tasklist.
 */

void ScanTaskList(AList *tList, AList *globalList)
/* Leest een tasklist in.
 */

void ScanTaskListExists(AList *tList, AList *globalList)
/* Leest een tasklist in. Alle taken moeten gedefinieerd zijn.
 */

/*--------------------------------------------------------------------------
 * FILE   : sparse2.c
 * AUTHOR : Berend Reitsma
 *
 */


/* Sparse matrix implementatie
```

```
 */

void MoreRoom(SMatrixBucket *b, uint n)
/* Hulpfunctie voor het uitbreiden van een bucket.
 */

ushort SBSearchEl(SMatrixBucket *b, ushort nr)
/* Zoek index van nr in de bucket.
 */

ushort SBInsertEl(SMatrixBucket *b, ushort nr, int value)
/* Voeg nr in de bucket toe met waarde value
 */

void SBDeleteEl(SMatrixBucket *b, ushort nr)
/* Verwijdert nr uit de bucket.
 */

void DelSMatrixEl(SMatrix *m, ushort row, ushort col)
/* Verwijdert row,col uit de matrix.
 */

void SetSMatrixEl(SMatrix *m, ushort row, ushort col, int value)
/* Voeg row,col toe aan de matrix.
 */

int GetSMatrixEl(SMatrix *m, ushort row, ushort col)
/* Vraagt de waarde van row,col op.
 */

SMatrix *CreateSMatrix(ushort nrRows, ushort nrCols)
/* Maakt een sparse matrix.
 */

void DeleteSMatrix(SMatrix *m)
/* Verwijdert de sparse matrix.
 */

void SMatrixBucketCopy(SMatrixBucket *dst, SMatrixBucket *src)
/* Kopieert een bucket.
 */

void SMatrixCopy(SMatrix *dst, SMatrix *src)
/* Kopieert een sparse matrix.
 */

void SMatrixSetID(SMatrix *m)
/* Zet het id-veld van elk element in de matrix voor het snel indexeren.
 * Dit is een zeer onveilige functie. Na een insert moet deze opnieuw
 * uitgevoerd worden om consistentie te bewaren.
 */

uint NextCol(SMatrix *m, uint row, uint col)
/* Geeft de eerstvolgende niet-nul kolom in de rij of het aantal kolommen.
 */

/*-----------------------------------------------------------------------------
 * FILE   : write.c
 * AUTHOR : Berend Reitsma
 *
 */

void WriteTaskSet(FILE *out, AList *taskList)
/* Schrijft een tasklist naar file
 */

void WritePersFile(char *filename)
/* Schrijft een personfile
 */

void WriteTaskFile(char *filename)
/* Schrijft een taskfile
 */

void WriteSchedFile(char *filename)
/* Schrijft een schedulefile
 */
```