Job shop scheduling in de werkelijkheid

De efficiëntie en flexibiliteit van een multi-agent systeem

Auteur: A.C. Leenhouwers
Studentnummer: 3019292
Opleiding: Kunstmatige Intelligentie
Universiteit Utrecht

Begeleider: dr. F.P.M. Dignum

datum: 31-03-15
Bachelor eindwerkstuk

Inhoudsopgave

1. Introductie

Binnen het onderzoeksgebied van kunstmatige intelligentie (KI) is er veel aandacht gericht op heuristieken voor optimalisatie problemen. Een bekend voorbeeld hiervan is het *Job-shop scheduling problem* (JSSP). Hierbij is vaak het doel om jobs, die op machines uitgevoerd moeten worden, zo in te roosteren dat de totale productietijd minimaal is. Het aantal mogelijke oplossingen, waaruit het optimale rooster volgt, neemt zeer snel toe. Het simpelweg berekenen van alle oplossingen en het optimale rooster kiezen is daarom niet efficiënt. Vanuit het operationeel onderzoek (OR) perspectief zijn er optimale oplossingen gevonden voor specifieke testsituaties. Binnen dit onderzoek vertaalt men deze theoretische situaties naar wiskundige problemen en maakt men gebruik van abstracte wiskundige technieken om tot een optimale oplossing te komen. Deze benadering verschilt met onderzoek binnen KI, waarbij men vanuit het domein van het probleem kijkt. KI tracht de specifieke theoretische situaties naar de werkelijkheid te brengen, met behulp van bijvoorbeeld technieken geïnspireerd uit de natuur. Bij het oplossen van een probleem gebruiken mensen immers geen ingewikkelde wiskunde technieken, maar bepaalde slimme heuristieken. Vanuit verschillende benaderingen zijn er KI technieken gevonden, die het JSSP op intelligente wijze oplossen. Echter zoeken deze technieken over het algemeen naar oplossingen in een statische omgeving. In de realiteit gaan echter machines kapot en is vaak het aantal uit te voeren jobs niet van tevoren bekend. In het proefschrift “Agent technology in agile multiparallel manufacturing and product support” beschrijft Moergestel (2014) een flexibel multi-agent systeem dat dynamisch om kan gaan met het inroosteren van jobs. In dit systeem is het doel niet om het optimale schema te vinden, maar een kloppend schema te geven zonder de deadlines van de jobs te overschrijden. Moergestel bespreekt de technieken die hier tot toe in staat zijn, maar laat niet zien hoe goed deze technieken presteren. Dit leidt tot de volgende probleemstelling van dit werkstuk:

Hoe groot is het verlies in optimalisatie ten behoeven van flexibiliteit in het job-shop scheduling multi-agent systeem van Moergestel?

Om deze vraag te beantwoorden geef ik antwoord op de onderzoeksvragen:

- Welke KI technieken bestaan er om het optimum te benaderen?
- Wat zijn de voordelen in flexibiliteit in het systeem van Moergestel?
- Welke heuristieken gebruikt het systeem om tot zinvolle oplossingen te komen?
- Hoe efficiënt is het systeem van Moergestel in het oplossen van bekende job-shop testvoorbeelden?

Dit werkstuk is als volgt ingedeeld: Hoofdstuk 3 geeft de definitie van het JSSP en diens variaties. Hoofdstuk 4 geeft antwoord op de eerste onderzoeksvraag. De tweede onderzoeksvraag wordt beantwoord in hoofdstuk 5. De voordelen van de flexibiliteit worden behandeld in hoofdstuk 6. Hoofdstuk 7 beslaat het experimentele onderzoek naar de efficiëntie van het multi-agent systeem.

2. Methode van onderzoek

Een literatuurstudie naar de definitie van het JSSP en diens varianten, staat aan het begin van dit onderzoek. Daarop volgt onderzoek naar gevonden KI technieken en in het bijzonder het flexibele multi-agent systeem van Moergestel. Om inzicht te krijgen in de werking en de efficiëntie van de beschreven multi-agent heuristieken, vindt er een experimenteel onderzoek plaats. Net als onderzoekers binnen de KI

en het operationeel onderzoek, gebruik ik testvoorbeelden uit de OR-library om de efficiëntie van de heuristieken te toetsen. Het systeem implementeer ik in JAVA en pas ik toe op elf instanties uit de OR-library. De gevonden minimale productietijden vergelijk ik met de reeds gevonden optima voor de testvoorbeelden. In het bijzonder neem ik de instantie ft06 met zes jobs, zes machines en een optimum van 55 in beschouwing. Aan de hand van dit hanteerbare probleem kan ik analyseren waarin en waarom de implementatie van Moergestel's systeem verschilt.

3. Job-shop scheduling probleem

In het klassieke job-shop scheduling probleem (JSSP) bestaan er aantal taken/jobs (n) die op een aantal machines (m) uitgevoerd worden. Een taak bestaat uit één of meerdere operaties (s), die een bepaalde tijd in beslag nemen. Wanneer er meerdere operaties zijn, worden deze gerepresenteerd door een n-tuple (v) waarbij de volgorde van belang is. Op een willekeurig tijdstip kan een machine maar één operatie behandelen en kan er per taak maar één operatie uitgevoerd worden. Een operatie moet in één keer uitgevoerd worden door een machine en mag dus niet onderbroken worden om plaats te maken voor een andere operatie. Het doel van het JSSP is om ten eerste een geldig schema te vinden, waarbij alle jobs volgens de bovenstaande voorwaarden zijn ingeroosterd. Daarnaast kan een JSSP verschillende optimalisatie doelen hebben. Over het algemeen zoekt men naar een planning waarbij de totale productietijd van alle taken (makespan) minimaal is. Bij het job shop scheduling probleem met deadlines (JSSD), waardeert men het schema aan de hand van de job die het meest de deadline (Maximum lateness) overschrijdt. Een ander voorbeeld is het minimaliseren van het aantal taken die de deadlines overschrijden. Sommige onderzoeksvragen beschouwen een combinatie van optimalisatie doelen. (Çalis et al, 2013)

Er zijn verschillende variaties op JSSP zoals open job-shop scheduling, waarbij de volgorde van de uit te voeren operaties niet van belang is. In een flow shop setting is het aantal machines gelijk aan het aantal operaties en bestaan de taken uit dezelfde sequentie van operaties. Bij een parallel shop bestaat elke taak uit één operatie die op één van de beschikbare identieke machines uitgevoerd worden (J.K. Lenstra et al, 1977). De variatie die het dichts bij de werkelijkheid ligt, is de JSSD. Taken hebben mogelijk hierbij variërende starttijden en deadlines.

De omgeving waarin de JSSP en diens variaties liggen, is in de traditionele zin statisch. Het aantal taken en machines staat namelijk van tevoren vast. In een dynamische omgeving is dit niet geval en verschijnen er op willekeurige momenten nieuwe jobs. Algoritmen die in een statische omgeving werken, zijn niet direct toepasbaar op dynamische problemen. In veel gevallen beschouwt men algoritmen die offline inroosteren. Het productieproces, zij het theoretisch, vindt dan pas plaats nadat er een oplossing is gevonden. Bij online inroosteren vindt de productie gelijktijdig plaats met het inroosteren. Een JSSP in dynamische omgeving en dat online inroosteren vereist, sluit het meest aan met een werkelijke productielijn.

Sommige job scheduling problemen staan pre-emption toe. Normaal gesproken moet een operatie zo zijn ingeroosterd dat deze in één keer uitgevoerd kan worden. In het geval van pre-emption kan een algoritme een operatie opsplitsen om zo ruimte te maken voor een operatie van een andere job.

In simpele instanties van JSSP, bijvoorbeeld drie taken en één machine, is kunstmatige intelligentie niet perse nodig. Echter is het aantal te beoordelen oplossingen gelijk aan (n!)^m. In het geval dat er 4 jobs zijn en 2 machines zijn er dus al 576 mogelijke oplossingen. De tijd die nodig is om tot de optimale oplossing te komen aan de hand van de input, definieert de complexiteit van job-shop problemen. Over het algemeen

zijn Job shop problemen NP-hard (J.K. Lenstra et al, 1977), wat wil zeggen dat er niet binnen polynominale tijd een oplossing gevonden kan worden. Vanwege deze complexiteit is het niet efficiënt om blind alle mogelijke oplossingen te doorlopen. Daarom is het vinden van slimme heuristieken voor het JSSP een actief onderzoeksgebied binnen de KI.

4. Kunstmatige intelligentie optimalisatie technieken

In de periode van 1960 tot 1980 richtten voornamelijk onderzoekers binnen het operationeel onderzoek zich op het JSSP. Over het algemeen gebruikte men branch and bound algoritmen om optimale schema's te vinden (Nakano et al, 1991). Deze algoritmen doorlopen echter een groot deel van de zoekruimte. Het kost dus veel tijd om tot een optimale oplossing te komen. Met behulp van KI technieken waren onderzoekers in staat om slimme heuristieken te vinden voor het JSSP. Deze heuristieken vinden niet alleen sneller een oplossing, maar ook de kwaliteit van de KI heuristieken overtreft de kwaliteit van branch and bound algoritmen (Çalis et al, 2013). KI zoekt immers vanuit het domein naar intelligente technieken. Niet alle algoritmen zijn in staat het optimum te vinden, maar zijn kwalitatief en qua toepasbaarheid beter dan OR-technieken. Voor een historisch overzicht van de gevonden technieken zie Çalis et al (2013). Er is geen bewijs gevonden dat een algoritme in het algemeen beter is dan andere algoritmen. In sommige situaties, zoals bijvoorbeeld een groot aantal jobs, werkt een bepaald algoritme wel beter. De meeste onderzoeken in de afgelopen vijftien jaar waren gericht op het ontwikkelen van algoritmen. Maar een paar onderzoekers probeerde werkelijke industriële productieproblemen op te lossen (Çalis et al, 2013). De algoritmen in de volgende drie paragrafen beschreven, geven een beeld van hoe KI onderzoekers het JSSP benaderen.

4.1 Genetic algoritms (GA)

Het concept van natuurlijke selectie inspireerde KI onderzoekers om genetische algoritmen te ontwikkelen. In een GA wordt een populatie van mogelijke oplossingen geëvolueerd naar een populatie met betere oplossingen. Initieel bevat de populatie willekeurige individuen met een sequentie van bits of DNA dat een oplossing representeert. Een fitness functie bepaalt aan de hand van dit genotype voor elk individu zijn prestatiewaarde. Vervolgens selecteert het algoritme een groep uit de populatie die de basis vormt voor de opvolgende generatie. In het simpelste geval bestaat de geselecteerde groep uit de individuen die het best presteren, maar er bestaan ook andere strategieën voor het selectieproces. Met behulp van operaties op het genotype van de individuen zoals overerving, recombinatie, mutatie en cross-over creëert het algoritme nieuwe nakomelingen. Voor een bepaald aantal generaties wordt het algoritme opnieuw toegepast. Wanneer de selectiemethoden en operaties op de genotypen goed zijn gekozen, bevat de laatste generatie de beste oplossing van alle generaties Russel et al, 2003). Dit betekent echter niet dat het algoritme de optimale oplossing heeft gevonden.

4.2 Ant colony optimalisation (ACO)

Als collectief vertonen mieren intelligentie (swarm intelligence), terwijl een individu niet intelligent is. De methode waarop mieren een pad vinden naar voedsel, vormt de basis voor ant colony optimalisation. Wanneer een mier voedsel vindt, laat hij op de terugweg een feromonenspoor achter. Andere mieren volgen dit spoor en verhogen de feromonenconcentratie van het pad. Na verloop van tijd verdampen de feromonen en de mate waarin, hangt af van de concentratie. Wanneer er twee paden naar de voedselbron zijn, dan

volgen de meeste mieren het kortste pad. Immers kunnen mieren in een bepaald tijdsbestek vaker het korte pad doorlopen dan het lange pad. Het ACO algoritme past de principes van het plaatsen van een feromonenspoor en het verdampen van dit spoor toe om een kortste pad te vinden in een zoekprobleem. Bij ACO bouwen de mieren in een ronde stap voor stap een deeloplossing op totdat de doelbestemming is bereikt. Per stap heeft de mier een probabilistische keuze uit mogelijke opvolgende stappen. Een hogere feromonenconcentratie geeft een hogere kans dat een stap gekozen wordt. Wanneer alle mieren een oplossing hebben gevonden, is de ronde afgelopen en vindt er een update van de feromonensporen plaats. Afhankelijk van kosten van een gevonden pad, is dit een positieve of negatieve update van het feromonenspoor. Na een bepaald aantal ronden is de feromonenconcentratie van kortste pad zo hoog, dat vrijwel alle mieren dit pad volgen. Dit gevonden kortste pad hoeft niet perse de optimale oplossing te zijn. In 2000 is bewezen dat sommige varianten van ACO convergent zijn en dus in een eindig tijdsbestek het optimum vinden. (Guhtjar, 2000)

4.3 Artificial neural networks (ANN)

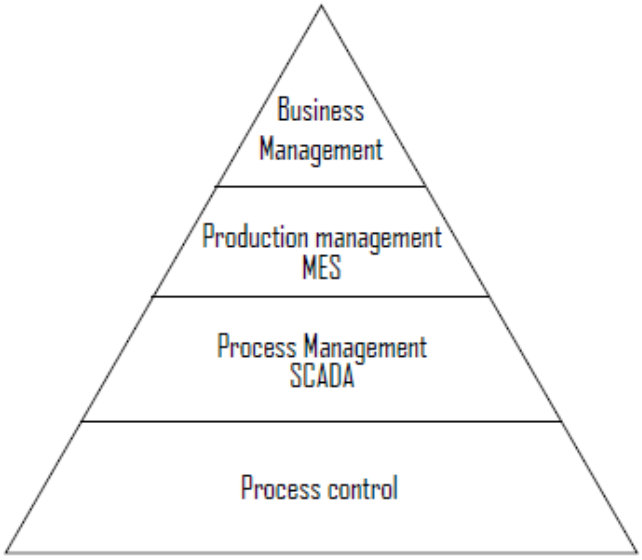
Een kunstmatig neuraal netwerk is gebaseerd op de neuronen en diens verbindingen in hersenen. Een ANN is een systeem dat in staat is om op verschillende manieren te leren. Met supervised learning leert een ANN aan de hand van het herhaaldelijk invoeren van testvoorbeelden en diens gewenste output. Na het leren van deze testvoorbeelden, is een goed werkend netwerk in staat om nieuwe problemen op te lossen. Een ANN bestaat uit een input-, output en verborgen laag. De input laag representeert de benodigde informatie voor een JSSP en de output laag de gevonden oplossing. In de hidden layer vindt de daadwerkelijke computatie plaats. In een laag communiceren neuronen met elkaar via verbindingen met een gewicht. Een neuron heeft een activatiefunctie die alle binnenkomende signalen omzet naar een output waarde. Deze output waarde vormt samen met het gewicht van de verbinding een input waarde voor een opvolgende neuron. Een ANN is in staat om te leren door de gewichten van de verbindingen aan te passen. Bij het leerproces genereert het netwerk een oplossing voor een leervoorbeeld. Wanneer deze afwijkt van de gewenste output van het leervoorbeeld, bepaalt het algoritme de foutmarge van de output. De gewichten in het netwerk worden aan de hand van deze foutmarge en een leerconstante een beetje aangepast. In theorie verkleint de foutmarge van de output dus per leervoorbeeld (Russel et al, 2003).

5. Multi-agent system voor flexibele productie

De reeds beschreven kunstmatige intelligentie technieken sluiten niet goed aan met een werkelijke productielijn. In de praktijk is het aantal te verwerken jobs niet vanaf het begin bekend, gaan machines kapot, verschuiven deadlines, komen er op willekeurige momenten nieuwe orders binnen en is er sprake van online inroosteren. Daarnaast is elke productielijn verschillend. In sommige productielijnen zijn er bijvoorbeeld machines die verschillende taken uit kunnen voeren of betreft de productielijn een combinatie van unieke en identieke machines. Merk op dat er wel varianten van andere KI technieken bestaan die wel met een dynamische omgeving om kunnen gaan. Bijvoorbeeld in de paper van (Bierwidth et al, 1995) gebruikt men genetische algoritmen om dynamisch in te roosteren. Echter is het met behulp van een multi-agent systeem op een intuïtieve en ogenschijnlijk simpele manier mogelijk om flexibel en dynamisch in te roosteren.

5.1 Standaard geautomatiseerd productieproces

Volgens Moergestel zijn er tekortkomingen aan het standaard geautomatiseerde productieproces. De gebruikte software is gebaseerd op het gelaagde automation pyramid model (zie figuur 1). De bovenste laag betreft de business management software. Op dit niveau komen de orders binnen en worden klanten en leveranciers behandeld. In de tweede production management laag maakt men over het algemeen gebruik van Manufacturing Execution System (MES) software. Deze laag is verantwoordelijk voor het besturen en verwerken van orders op een hoger niveau. Zo kent een MES bijvoorbeeld grondstoffen toe aan een productie order. De process management laag is verantwoordelijk voor het vertalen van orders naar productie stappen, het opbouwen van productie informatie en het toezicht houden op de werkelijke productie. De software in deze laag wordt vaak SCADA of Supervisory control and data acquisition genoemd. De onderste process control laag betreft de software die de machines bestuurt.



Figuur 1

De eigenschappen van het standaard proces zijn samen te vatten in vier eigenschappen. Grote batches, ofwel het herhaaldelijk produceren van één product, zijn nodig om de kostenefficiëntie te verwezenlijken. Het kost weinig tijd en geld (overhead) wanneer men van batch switcht van hetzelfde product. Het switchen naar een ander product geeft een hoge overhead. Immers moet bijvoorbeeld vaak de SCADA software opnieuw geconfigureerd worden. Als laatste is er een moeilijke transitie van product ontwikkeling naar de daadwerkelijke productie. De producten in ontwikkeling worden los gemaakt en dus niet door de machines die in de productielijn zitten. Er vindt dus nog een fase van ontwikkeling plaats, waarbij het product klaargemaakt moet worden voor massa productie, genaamd upscaling.

Moergestel ziet zes tekortkomingen in het standaard geautomatiseerd productieproces. Het is alleen geschikt voor massa productie, aangezien het switchen van producten voor een hoge overhead zorgt. Men maakt gebruik van gespecialiseerde machines, die vaak in gebruik moeten zijn om kostenefficiënt te zijn. Pipeline batch productie, ofwel elk product volgt dezelfde sequentie van machines, komt in de problemen wanneer er iets mis is met een machine. Meeste automatiseringsmethoden zijn push-driven, hierbij produceert een bedrijf een aantal producten zonder te wachten op orders van klanten. Dit kan echter resulteren in overproductie, wat een verspilling van geld, materiaal en tijd betekent. De transities van concept naar product naar massa productie kunnen te veel tijd in beslag nemen. De meeste SCADA en MES implementaties zijn niet geschikt voor decentralisatie en hebben dus een single point of failure.

5.2 Flexibel multi-agent systeem van Moergestel

Om de tekortkomingen van het standaard geautomatiseerd productieproces op te vangen introduceert Moergestel een equiplet. Een equiplet is een relatief goedkope machine waarop verschillende opzetstukken geplaatst kunnen worden. Elke equiplet heeft dus dezelfde basis, maar de functionaliteit komt van het geplaatste opzetstuk. De equiplets staan met elkaar en een ondersteunend systeem via een netwerk in verbinding. Op het ondersteunende systeem is de benodigde software voor een frond-end te downloaden. Met de goedkope equiplets is het relatief makkelijk een multi-parallelle productielijn op te stellen. In deze situatie met redundante equiplets ondervindt het systeem weinig last wanneer een machine problemen heeft. Tevens is het mogelijk om een kleine batch van producten simultaan te verwerken naast een andere batch. Bijvoorbeeld bij het ontwikkelen van een nieuw product is het upscaling proces niet van toepassing omdat het test product gelijktijdig met een andere batch geproduceerd wordt door de equiplets. Het switchen naar een batch van een ander product, is ook gemakkelijker door het omwisselen van de front-ends. Als laatste is de schaalbaarheid groter bij een productielijn met equiplets dan bij standaard productielijnen. Men hoeft bij het vergroten van de productieomgeving niet te bepalen in welke mate een machine belast wordt.

Moergestel beschrijft een systeem gebaseerd op autonome agenten die op een natuurlijke wijze aansluiten op het beschreven multi-parallel systeem. Een product agent representeert een job en is onder andere verantwoordelijk voor het plannen van de te kiezen machines en aan de hand van deze planning een schema te maken. Equiplet agenten zijn verantwoordelijk voor het uitvoeren van de operaties en bevatten de informatie over de operaties die de machine uit kan voeren. De product agenten zijn in staat om een taak in te roosteren aan de hand van een prikbord. Voor elk tijdstip staat op het prikbord welke machine door welke product agent is gereserveerd. Wanneer een product agent niet in staat is om zijn taak in te roosteren, onderhandelt hij met de andere agenten om zo tot een kloppend schema te komen.

Een product agent beschikt over de volgende eigenschappen:

- Hij heeft zijn leefomgeving, ofwel de hardware locatie waarop de software draait, op de onboard hardware van een equiplet of op een ondersteunend systeem.
- Alle agenten zijn modulair opgebouwd zodat zij dezelfde basis hebben. Tevens maakt deze modulariteit het mogelijk om de agent uit te breiden met nieuwe modules.
- Product agenten bouwen een data log op van alle specifieke uit gevoerde operaties en eigenschappen van het product.
- Hij moet in staat zijn om van leefomgeving te veranderen. De product agent met diens product informatie kan nodig zijn voor bijvoorbeeld logistieke doeleinden.
- De agent weet de volgorde waarin de operaties uitgevoerd moeten worden.
- Hij moet in staat zijn om te onderhandelen met andere agenten.

Voor een equiplet agent gelden de volgende eigenschappen:

- Hij leeft omwille van de schaalbaarheid op de onboard hardware van zijn equiplet.
- Een equiplet agent weet welke operaties hij kan uitvoeren en kan dit door communiceren.
- Hij kan een product agent vertellen of hij een operatie met specifieke parameters kan uitvoeren.
- Equiplet agenten geven aan de product agenten door hoeveel tijd een operatie in beslag neemt.
- Hij moet in staat zijn tot interactie met een menselijke equiplet operator.
- De software voor een front-end is te downloaden vanaf een ondersteunend systeem.

5.3 Gecentraliseerd versus gedistribueerd MAS

Men spreekt van een gedistribueerd systeem wanneer de computaties niet op één plek plaats vinden. In het systeem van Moergestel geldt dit wanneer de product agenten op de equiplet hardware leven. Tijdens de communicatie met een equiplet, bijvoorbeeld bij het uitvoeren van een operatie, verblijven zij op de equiplet hardware. Het voordeel hiervan is dat er geen single point of failure is. Bij een gecentraliseerd systeem kan de hardware, waarop alle productagenten leven, overbelast raken. Er is deze situatie echter geen sprake van werkelijke communicatie tussen product agenten. Dit kan voordelig zijn wanneer de inter-agent communicatie op verschillende apparaten het netwerk te zwaar belast. Een gedistribueerd systeem heeft een betere schaalbaarheid maar heeft als nadeel dat niet alle informatie op één plek zit. In het systeem van Moergestel staat de informatie over het schema wel centraal, namelijk op het prikbord. Als laatste sluit het concept van een MAS niet aan met de definitie van een gecentraliseerd systeem. Intuïtief gezien bestaat een autonome agent los van andere agenten en behoeft hij een eigen leefruimte.

5.4 Voordelen van het multi-agent systeem

Het MAS is in staat om alle variaties van het JSSP op te vangen. In tegenstelling tot andere KI algoritmen is dit mogelijk zonder aanpassingen in het basissysteem te hoeven maken. Het maakt een product agent niet uit hoe de omgeving eruit ziet. Hij communiceert alleen met andere equiplet- en product agenten of raadpleegt het prikbord. Het kan voor een KI algoritme problemen opleveren wanneer een machine meerdere taken uit kan voeren. Het systeem van Moergestel is echter zo ingericht, dat de equiplet agenten de mogelijk uit te voeren operaties door geven. De product agenten bevatten zelfs heuristieken om de keuze tussen opeenvolgende equiplets te optimaliseren. In zijn paper behandelt Moergestel daarnaast de volgende voordelen van zijn multi-agent systeem. De parallelle eigenschap van het systeem maakt het mogelijk om verschillende batches naast elkaar uit te voeren. Dit is vooral gewenst bij het ontwikkelen van een product, omdat de equiplets ook beschikbaar zijn voor het produceren van het testproduct. Product agenten bouwen een data log op van alle specifieke uit gevoerde operaties en eigenschappen van het product. Zij kunnen na de productie blijven bestaan in verband met de product informatie of het transport naar een andere locatie. Het is ook mogelijk om de agent via internet te raadplegen, mocht dit voor de eindgebruiker gewenst zijn. Het gebruik van relatief simpele onderdelen (hardware en software) in het systeem, maakt vervanging goedkoop en biedt flexibiliteit. Wanneer een equiplet wordt verwijderd of toegevoegd aan het systeem, past de software zich automatisch aan op de nieuwe situatie. Zo kan men ook de betrouwbaarheid van het systeem testen op een kleine schaal. Het MAS geeft een goede aanvulling op het standaard automatisering piramide model, waarbij product agenten de MES laag vervangen en equiplet agenten de SCADA laag. Deze lagen bevatten over het algemeen inflexibele gecentraliseerde software en hebben single point of failure. Het is mogelijk om het prikbord uit te breiden met een overzicht dat per operatie weergeeft op welke tijdstippen zij uitgevoerd worden. Zo heeft een operator toezicht op de productielijn en ziet hij snel eventuele bottlenecks. Hij kan een equiplet toevoegen wanneer een operatie populair wordt. Het systeem is niet alleen makkelijk uitbreidbaar qua software en hardware. Maar door de modulariteit van de product agenten is het mogelijk om nieuwe onderhandelingstechnieken, zonder bestaande code te wijzigen, toe te voegen. Zoals bijvoorbeeld FDD/MWKR prioriteitsregel, die volgens de literatuur het best presteert (Sels et al). Het is zelfs mogelijk om andere KI algoritmen te koppelen aan de agenten.

6. Inrooster heuristieken in het systeem van Moergestel

Een product agent moet eerst een planning vinden voordat hij een job kan inroosteren. Hij hoeft geen planning te maken wanneer een equiplet maar één operatie uit voert. Moergestel behandelt in zijn proefschrift algoritmen die voor een korte planning zorgen. Het is bijvoorbeeld gunstig om een planning te vinden waarbij er zo min mogelijk gewisseld wordt van equiplet. Het plannen van een job is echter niet toepasbaar op de instanties uit de OR-library en is niet van belang voor dit onderzoek. Zie hoofdstuk 3.1 van Moergestel (2014) voor de uitleg van deze algoritmen.

Het inroosteren door agenten heeft de volgende karakteristieken:

- Resources (machines) zijn beschikbaar voor alle agenten.
- Alle operaties nemen een tijdseenheid van gehele getallen in.
- Agenten moeten bereid zijn om samen te werken.
- Een reeds ingeroosterd schema van een agent mag alleen opgegeven worden wanneer er een kloppend alternatief bestaat voor die agent.
- Agenten hebben geen afgeschermde informatie.

Een nieuwe product agent probeert alle operaties in het productie pad in te roosteren aan de hand van de gegeven start tijd en deadlines. In een geldig rooster mogen de deadlines van de producten niet overschreden worden. Aan de hand van het prikbord kan de agent zien op welke tijdstippen een machine beschikbaar is. Dit prikbord geeft daarnaast per machine aan welk tijdstip door een product agent is gereserveerd of ingeroosterd. Er is een protocol nodig voor het te kiezen vrije tijdstip. Zoals bijvoorbeeld First Fit, waarbij de agent steeds kiest voor het eerst beschikbare tijdstip. Bij Just in time roostert de agent als eerste de laatste operatie in op een beschikbaar tijdstip zo dicht mogelijk bij de deadline. Uit de resultaten van Moergestel blijkt dat First fit het meest geschikt is van vier getoetste protocollen. Met First Fit is het aantal jobs dat succesvol ingeroosterd worden namelijk het grootst.

Om het aantal deadline overschrijdingen te verminderen is er een inroostermethode of prioriteitsregel nodig. Moergestel behandelt in zijn proefschrift de volgende methoden:

- Fixed priority: hierbij krijgen producten van tevoren een prioriteitslabel mee. Dit kan in de praktijk nuttig zijn. Echter zijn alle agenten in dit systeem gelijk dus hebben zij dezelfde prioriteit.
- Earliest deadline first (EDF): jobs die een eerdere deadline hebben krijgen voorrang.
- Least slack first (LSF): Slack is gedefinieert als de totale beschikbare tijd tot de deadline minus de som van alle operatie tijden.
- Smallest critical ratio first (CR): critical ratio is gedefinieerd als het totale aantal tijdseenheden beschikbaar gedeeld door het aantal benodigde tijdseenheden.
- Shortest process first (SPF): de job met de kleinste productietijd krijgt voorrang.

Aan de hand van deze methoden vindt er onderhandeling plaats tussen de agenten. Er zijn twee varianten te onderscheiden, weak en strong negotiation. Bij weak negotiation kan een product agent pas onderhandelen wanneer hij bij het inroosteren zijn deadline overschrijdt. De keuze van de deadline speelt een grote rol. Als de deadline ruim is, dan vinden er weinig onderhandelingen plaats. In het slechtste geval zijn er zelfs geen onderhandelingen en bestaat het inroosteren alleen uit First Fit. Bij strong negotiation moet een product agent op het moment dat deze aangemaakt is, eerst onderhandelen voordat hij zijn job in roostert. Een nadeel van deze variant van onderhandelen is dat er meer inter-agent communicatie optreedt. Dit vraagt meer rekenkracht van het systeem en kan het zoeken naar een oplossing lang laten duren. Een systeem kan overbelast raken wanneer er te veel product agenten in het systeem bestaan.

Een product agent probeert alle operaties aan de hand van First Fit in te roosteren. Het prikbord geeft weer welke tijdstippen beschikbaar zijn. Tijdens het inroosteren reserveert de agent de benodigde tijdstippen. Wanneer de agent een geldig schema vindt, claimt hij deze tijdstippen op het prikbord. Indien een rooster de deadline van het product overschrijdt, moet de agent herinroosteren om tot een geldig rooster te komen. Om geldig schema te verkrijgen probeert de product agent operaties op een vroeger tijdstip in te roosteren. Hij checkt welke agenten een lagere prioriteit hebben en vraagt aan hen om de geclaimde tijdstippen op te geven. Indien de agent de agent een eigen schema heeft gevonden, moet hij daarna een geschikt schema vinden voor de agenten die hun tijdstippen hebben vrijgegeven. Op deze manier komt het niet voor dat agenten in oneindigheid proberen te herinroosteren. Merk op dat wanneer product agenten allebei dezelfde prioriteit hebben, dat dan degene die al een geldig schema heeft niet zijn tijdstippen vrij geeft.

7. Experimenteel onderzoek

Bij het opstellen van het experimenteel onderzoek, kwam naar voren dat de OR testvoorbeelden niet direct toepasbaar zijn op het geïmplementeerde systeem. Er moest daarom vastgesteld worden wat de probleembeschrijving van de OR instanties is en waarin de implementatie verschilt.

7.1. Probleem beschrijving

Voor alle de voorbeelden uit de OR-library geldt:

- Een job bestaat uit een gegeven aantal operaties dat gelijk is aan het aantal machines.
- Alle operaties in een job moeten in de gegeven volgorde uitgevoerd worden.
- Een operatie heeft een tijdswaarde van gehele getallen.
- Een operatie mag niet bestaan uit deel operaties, dus geen half producten.
- Alle machines zijn uniek, dus kan een operatie maar door één machine uitgevoerd worden
- Alle machines kunnen maar één operatie uitvoeren.
- De transport van en naar een machine zit inbegrepen in de tijdsduur die voor een operatie staat.
- Pre-emption is niet toegestaan.
- De jobs, operaties en diens productietijden worden gerepresenteerd door een tabel waarin een rij een job voorstelt. Een rij bevat de gegevens van de uit te voeren operaties. Per operatie zijn er getallen, één geeft het nummer van de machine die de operatie uit voert en de ander geeft de tijdsduur.

De gevonden optima voor de gekozen OR instanties zijn overgenomen van tabel 2 uit de paper van Kammer et al (2010).

7.1.1 Onderhandelingen

Uit de resultaten van Moergestel komt naar voren dat EDF en LSF overall de twee beste prioriteitsregels zijn. Bij een inroosterprobleem waarbij het aantal jobs klein is, presteert CR echter ook goed. Omdat de gekozen JSSP instanties maximaal twintig jobs hebben, is het van belang om CR ook in beschouwing te nemen.

De weak en strong variaties van de onderhandelingsmethoden zijn gebaseerd op een dynamische omgeving. Bij de bekende JSSP instanties zijn starttijden niet van toepassing en zijn alle jobs vanaf het

begin af aan bekend. Echter is dit niet het geval wanneer een testvoorbeeld aan de implementatie van het multi-agent systeem aangeboden wordt. Het programma leest het tabel van jobs uit en aan laat de agenten inroosteren aan de hand van de gegeven volgorde. Dit geeft echter problemen wanneer er weak negotiation plaats vindt. Stel dat van de zes jobs de tweede job de hoogste prioriteit heeft. Nadat de eerste job is ingeroosterd, komt de inroostering van de tweede job na de eerste. De tweede job overschrijdt niet zijn deadline en hoeft er dus geen onderhandeling plaats te vinden. Een opvolgende product agent overschrijdt zijn deadline en daarom moet de eerste product agent zijn gereserveerde tijdstippen vrijgeven om zo tot een geldige planning te komen. Wanneer de eerste en tweede job wisselen van plek in het tabel, dan wordt de job met de hoogste prioriteit als eerste ingeroosterd. Echter waren deze tijdstippen in het voorgaande schema nodig voor een andere agent. Omdat de eerste agent de hoogste prioriteit heeft, geeft hij zijn gereserveerde tijdstippen niet vrij en kan er geen geldig schema gevonden worden. De volgorde waarin de agenten inroosteren is dus van belang. In theorie is het mogelijk om alle mogelijke volgorden van de jobs te evalueren. Maar in de praktijk betekent dit het testen van n! mogelijkheden. In het geval dat n = 15 zijn er dus 1.307.674.368.000 mogelijkheden. In de tweede hypothetische situatie zal het systeem met strong negotiaton ook geen geldig schema vinden. Maar in dat geval is de volgorde van de jobs in ieder geval niet van belang. Om deze reden neem ik in dit onderzoek alleen strong negotiation in beschouwing. De verwachting is dat het nadeel van de toeneming van de inter-agent communicatie bij strong negotiation in de test voorbeelden, die een maximaal twintig jobs hebben, voor geen problemen in de berekentijd zal zorgen. Merk op dat in een realistische dynamische setting de volgorde van de jobs bij weak negotiation tot minder problemen leidt, omdat jobs zich over het algemeen op willekeurige tijdstippen aanmelden.

7.1.2 Deadlines

Een belangrijk verschil is dat het systeem werkzaam is met deadlines, terwijl dit niet het geval is bij de bekende OR instanties. Wanneer deze testvoorbeelden niet verrijkt worden met deadlines, zal een product agent alleen First Fit toepassen en vinden er geen onderhandelingen plaats. In dit onderzoek beschouw ik vier mogelijkheden voor het toevoegen van deadlines:

1. Total production time deadline (TPTD): Alle jobs krijgen dezelfde dezelfde deadline dus is EDF niet nuttig. Er vinden geen onderhandelingen plaats en First Fit bepaalt het schema. De deadline is aan de som van alle minimale productietijden van alle jobs. De minimale productietijd is de som van de tijdsduur van alle operaties in een job.
2. Equal slack deadline (ESD): Elke job krijgt een deadline die gelijk is aan de minimale productietijd plus een vaste waarde. Dit betekent dat de slack voor elke job hetzelfde is, daarom LSF niet van toepassing is en alleen First fit geldt.
3. Total work content (TWK) *(Blackstone et al., 1982)*: De deadline die een job krijgt hangt af diens minimale productietijd vermenigvuldigd met een spelingsfactor. Deze factor moet zo gekozen worden dat het systeem in staat is om tot een gelding schema te komen.
4. Fixed value deadline (FVD): Alle jobs krijgen dezelfde vrij gekozen deadline. Het gebruik van deze deadline vindt alleen plaats bij de analyse van testvoorbeeld ft06. In dat geval wordt de deadline gelijk gesteld aan het optimum en het resultaat geanalyseerd. Tevens is EDF niet van toepassing omdat alle jobs dezelfde deadline hebben.

7.1.3 Implementatie

Het systeem van Moergestel zal gesimuleerd worden middels een implementatie in de programmeertaal JAVA. Deze implementatie simuleert niet alle kenmerken van het multi-agent systeem. Een belangrijk verschil dat de dynamische eigenschappen niet zijn geïmplementeerd. In dit onderzoek wordt namelijk de effectiviteit getoetst aan de hand van de statische testvoorbeelden en diens optimale waarden. Het is daarom niet van belang dat het systeem in staat is om nieuwe jobs en dus nieuwe product agenten toe te voegen op willekeurige momenten. Tevens hoeft het prikbord niet dynamisch vergroot te worden. Het maximale aantal tijdstippen is van tevoren bekend, ofwel de som van de minimale productietijden van elke job. Dit komt namelijk overeen met de situatie waarin één machine alle jobs uitvoert. Wanneer een tweede machine van toepassing is, dan is de totale minimale productietijd altijd gelijk of kleiner aan de situatie met één machine. Een ander verschil met het systeem van Moergestel is dat de product agenten in de implementatie niet gedistribueerd zijn. De computatie vind namelijk plaats op één computer (Intel Core i7-2600 CPU @ 3.40Ghz, 8 GB RAM). Dit betekent dat er nooit twee product agenten tegelijk proberen in te roosteren en dat een agent dus ook geen tijdstippen op het prikbord hoeft te reserveren tijdens het inroosteren. Het doel van dit onderzoek is niet om de functionaliteit van een gedistribueerd systeem te analyseren, vandaar dat dit niet van toepassing hoeft te zijn. Een ander verschil is dat er geen implementatie is voor de equiplet agent. In dit onderzoek hoeft een equiplet agent niet een operatie daadwerkelijk uit te voeren. Omdat een machine maar één operatie uit kan uitvoeren, hoeft een equiplet agent ook niet te communiceren welke operaties hij uit kan voeren. Tevens hoeft hij aan een product agent niet door te geven hoe lang de operatie duurt. De datatabel uit de testvoorbeelden geeft namelijk aan welke machine de operatie uitvoert en hoe lang dit duurt. Als laatste mogen product agenten niet de deadline overschrijden. De voorwaarde dat agenten alleen een reeds ingeroosterd schema mogen opgeven wanneer er een geldig alternatief gevonden wordt, is dus niet van toepassing.

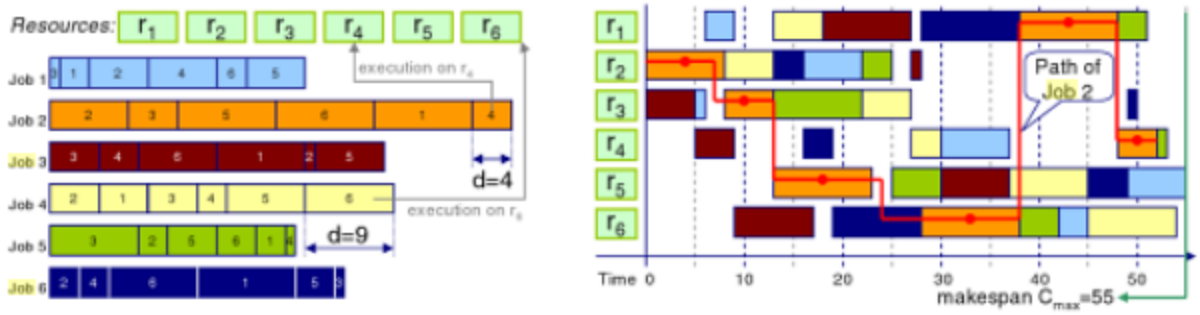
7.2 Resultaten

7.2.1 Analyse van probleem instantie ft06

Het testvoorbeeld ft06 betreft een probleem met zes jobs, zes machines en een optimale minimale makespan van 55. Tabel1 geeft de dataset weer zoals gegeven in de OR-library. Als toevoeging op deze dataset is ook de minimale productietijd per job toegevoegd. Figuur 2 uit het boek van Bergmann et al (2008) geeft de optimale oplossing in de vorm van een Gantt diagram (rechts).

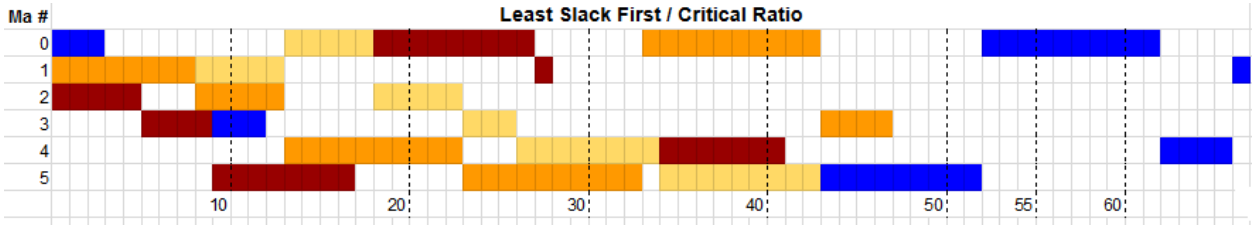
Job #	Operatie 1		Operatie 2		Operatie 3		Operatie 4		Operatie 5		Operatie 6		MPT
1	2	1	0	3	1	6	3	7	5	3	4	6	26
2	1	8	2	5	4	10	5	10	0	10	3	4	47
3	2	5	3	4	5	8	0	9	1	1	4	7	34
4	1	5	0	5	2	5	3	3	4	8	5	9	35
5	2	9	1	3	4	5	5	4	0	3	3	1	25
6	1	3	3	3	5	9	0	10	4	4	2	1	30

Tabel 1



Figuur 2

Om te testen of LSF of CR tot de optimale oplossing komen, is FVD toegepast om de deadline op 55 zetten. Het systeem komt immers tot de optimale oplossing wanneer alle jobs vóór de deadline van 55 ingeroosterd zijn. Merk op dat wanneer alle jobs dezelfde deadline hebben, LSF en CR dezelfde resultaten geven. Bijvoorbeeld bij LSF heeft job vier voorrang op job drie, want slack is 20 en 21 respectievelijk. De vierde job zal ook prioriteit hebben over job 3 bij CR, de critical ratio is namelijk respectievelijk 1.57 en 1.61. Met beide prioriteitsregels blijkt het systeem niet in staat om het optimum van 55 te bereiken. De zesde job is niet in te roosteren zonder de deadline te overschrijden. Wanneer de zesde product agent zijn job wil inroosteren, moeten de eerste en vijfde product agenten de geclaimde tijdstippen vrijgeven. Figuur 3 geeft de situatie weer nadat de zesde product agent zijn job heeft ingeroosterd. Zoals te zien is overschrijdt hij daarbij de deadline van 55.



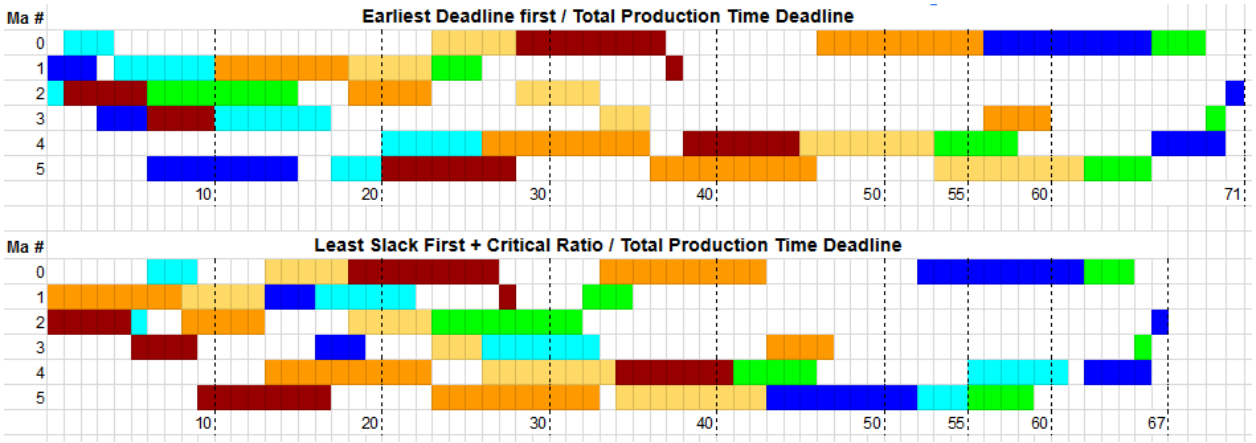
Figuur 3

Het systeem is niet in staat om tot het optimum van 55 te komen omdat de keuze van de vrije tijdstippen afhangt van First Fit. Zoals te zien is in figuur 2 oplossing moet bijvoorbeeld de vierde operatie van de tweede job op machine vijf plaats maken voor de derde operatie van de zesde job. Dit is echter niet mogelijk door alleen het gebruik van First Fit en onderhandelingsmethoden. De tweede job heeft in deze situatie namelijk altijd voorrang. Met het gebruik van deze technieken, kan een job niet initieel een hogere prioriteit hebben en voor latere operaties een lagere prioriteit.

Tabel 2 geeft de gevonden optima met de verschillende deadline methoden en prioriteitsregels (zie ook figuur 4). Opvallend is dat EDF een optimaal schema vindt bij de onderhandelingsmethode TPTD. In dit geval vinden er geen onderhandelingen plaats, alle deadlines zijn immers hetzelfde. Bij toeval geeft de volgorde waarin de product agenten aangemaakt worden, met de toepassing van standaard First Fit een betere oplossing. Wanneer de eerste en de tweede job van positie wisselen, dan heeft de makespan een andere waarde. Om de volgorde uit te sluiten van dit experiment is er voor strong negotiation gekozen. Maar het blijkt nu dus dat de volgorde nog steeds van belang is, zij het in het triviale geval zonder onderhandelingen.

	TPTD	ESD		TWK		Optimum	Afwijking
	Makespan	Makespan	Slack	Makespan	Allowance		
EDF	*71*	79	43	79	2.6	71	1,29090909
LSF	67	*71*	45	94	2.0	67	1,21818181
CR	67	67	41	70	2.5	67	1,21818181

Tabel 2



Figuur 4

7.2.2 Prestaties op overige probleem instanties

Tabel 3 geeft de resultaten weer van de tien probleem instanties uit de OR-library. Hieruit is af te lezen dat het systeem gemiddeld 43% van het optimum afwijkt. Bij de yn1 instantie scheelt het zelfs 62%. Het lijkt alsof de afwijking groter wordt, wanneer het aantal jobs en machines toenemen. Echter is deze hypothese niet getoetst in dit experiment en is er vanuit de resultaten onvoldoende bewijs voor deze claim. Er blijkt geen voorkeur te zijn voor een bepaalde onderhandelingstechniek.

Instantie	n x m	Optimum	EDF			LSF			CR			Best	Afwijking
			TPTD	ESD	TWK	TPTD	ESD	TWK	TPTD	ESD	TWK		
la02	10 x 5	655	*912*	982	982	945	*912*	966	**893**	893	912	893	1,363358
la19	10 x 10	842	*1208*	1165	1165	**1146**	*1208*	1302	1172	1172	1236	1146	1,361045
ft10	10 x 10	930	*1410*	1343	1349	**1271**	*1410*	1530	1475	1475	1283	1271	1,366666
orb1	10 x 10	1059	*1376*	1528	1528	1391	*1376*	1489	**1319**	**1319**	1625	1319	1,245514
la21	15 x 10	1046	*1610*	1768	1763	**1452**	*1610*	1806	1700	1700	1610	1452	1,388145
la27	20 x 10	1235	*2119*	2119	2059	**1844**	*1955*	2161	2119	2119	2059	1844	1,493117
la40	15 x 15	1222	*1867*	1904	1904	**1726**	*1867*	2231	1794	1794	1962	1726	1,412438
abz7	20 x 15	655	*1162*	1045	1045	1035	*1162*	1082	1155	1155	**1010**	1010	1,541984
abz9	20 x 15	656	***1034***	1146	1146	1077	*1034*	1159	1185	1485	1098	1034	1,576219
yn1	20 x 20	846	*1461*	**1375**	**1375**	1383	*1461*	1547	1507	1507	1455	1375	1,625295

Tabel 3

Om geldige schema's te vinden, hebben de onderhandelingsmethoden een vrij grote deadline nodig. In tabel 4 is te zien dat bij ESD de slack waarde vrijwel even hoog is als de gegeven optimale makespan. In sommige gevallen is de slack waarde zelfs hoger dan de makespan. Alle oplossingen werden gevonden in maximaal 5 seconden.

Instantie	n x m	Optimum	TPTD Deadline	EDF		LSF		CR	
				ESD Slack	TWK Ratio	ESD Slack	TWK Ratio	ESD Slack	TWK Ratio
la02	10 x 5	655	2643	597	2.66	527	2.49	568	4.00
la19	10 x 10	842	5346	686	2.43	655	2.24	646	2.43
ft10	10 x 10	930	5109	810	2.26	870	2.34	984	2.85
orb1	10 x 10	1059	5409	911	2.48	823	2.17	888	2.97
la21	15 x 10	1046	7994	1194	3.09	1036	2.53	1218	3.00
la27	20 x 10	1235	10832	1590	3.60	1326	3.16	1590	3.60
la40	15 x 15	1222	11472	1185	2.32	1139	2.34	1121	2.64
abz7	20 x 15	655	7366	678	2.86	823	2.73	802	3.03
abz9	20 x 15	656	7442	749	2.93	600	2.74	783	3.34
yn1	20 x 20	846	11760	777	2.28	832	2.31	952	2.46

Tabel 4

8. Conclusie

Om antwoord te geven op de probleemstelling “Hoe groot is het verlies in optimalisatie ten behoeven van flexibiliteit in het job-shop scheduling multi-agent systeem van Moergestel?“, moeten eerst de onderzoeksvragen in beschouwing worden genomen.

Welke KI technieken bestaan er om het optimum te benaderen?

Binnen het onderzoeksgebied van KI zijn er diverse algoritmen gevonden voor het JSSP. Ondanks dat niet niet alle technieken de optimale oplossing vinden, is de kwaliteit van het algoritme en de zoektijd beter dan de branch-and-bound OR-algoritmen. Uit de literatuur blijkt dat niet één algoritme over het algemeen beter is dan de anderen (Çalis, 2013).

Wat zijn de voordelen in flexibiliteit in het systeem van Moergestel?

De flexibiliteit van het systeem geeft vele voordelen, zo maakt het voor systeem niet uit hoe de omgeving eruit ziet. De software past zich automatisch aan wanneer de situatie of productievoorwaarden veranderden. Dankzij deze schaalbaarheid is het mogelijk om de productielijn te testen op kleine schaal. Alle JSSP variaties zijn op te lossen zonder aanpassingen in het systeem van Moergestel, in tegenstelling tot andere KI methoden. Het is wel van belang dat een job deadlines heeft, anders vinden er geen onderhandelingen plaats. Het systeem kan meerdere batches van verschillende grootte naast elkaar uitvoeren. Dit is over het algemeen niet mogelijk in het standaard automatisering proces. Door deze parallelle eigenschap vervalt het upscaling proces in bij het ontwikkelen van een product. Door de MES en SCADA te vervangen door de product- en equiplet agenten, wordt het standaard proces flexibel en geven beide lagen niet meer een single point of failure. Product agenten bouwen een data log op van alle specifieke uit gevoerde operaties en eigenschappen van het product. Deze informatie kan bruikbaar zijn voor logistieke doeleinden of online toegankelijk zijn voor de eindgebruiker. Tevens kan de eindgebruiker aan de hand van het prikbord

eventuele bottlenecks in de productielijn opsporen en indien nodig een equiplet toevoegen. Naast deze flexibiliteit is het systeem ook flexibel in de uitbreidbaarheid van de product agenten. Door diens modulariteit is het mogelijk om nieuwe onderhandelingstechnieken toe te voegen, zonder bestaande code te wijzigen. Het is zelfs mogelijk om andere KI algoritmen te koppelen aan de agenten, om zo hybride KI heuristieken te verwezenlijken.

Welke heuristieken gebruikt het systeem om tot zinvolle oplossingen te komen?
De product agenten zijn verantwoordelijk voor het vinden van een geldig schema. De equiplet agents geven aan het prikbord door op welke tijden de machines beschikbaar zijn. Een product agent probeert in eerste instantie zijn job in te roosteren aan de hand van First Fit. Voor alle operaties in een job zoekt hij hierbij op het prikbord naar de equiplet die de operatie op zijn vroegst kan uitvoeren. Wanneer een product agent niet in staat is om een geldig schema te vinden, moet hij onderhandelen met de overige agenten. Aan de hand van een prioriteitsregel, zoals bijvoorbeeld Earliest Deadline First, vraagt hij aan alle agenten die een lagere prioriteit hebben om de gereserveerde tijdstippen op te geven. Na het succesvol inroosteren van zijn job, is de product agent ook verantwoordelijk voor het vinden van een schema voor de agenten met lagere prioriteit. Op deze manier is het niet mogelijk dat agenten tot in de oneindigheid proberen te onderhandelen. Wat niet binnen de reikwijdte van dit onderzoek valt is het plannen van een job. Het systeem van Moergestel bevat wel heuristieken voor het kiezen van het kortste pad langs de benodigde equiplets.

Hoe efficiënt is het systeem van Moergestel in het oplossen van bekende job-shop testvoorbeelden?
Bij het testvoorbeeld mt06 zijn de onderhandelingsmethoden gematigd efficiënt. Gemiddeld geven zij een verhoging van 25% van de optimale makespan. In het geval van EDF en TPTD vindt het systeem zijn optimum wanneer er geen onderhandelingen plaats vinden. De gevonden oplossingen voor de overige instanties wijken gemiddeld 42% af van het optimum. In deze gevallen vindt alleen EDF met de toepassing van First fit zonder onderhandelingen zijn optimum. Pas bij relatief grote deadlines zijn de onderhandelingsmethoden in staat een geldig schema te vinden. In de praktijk kan het zijn dat een deadline van deze grootte niet is toegestaan in een productielijn. Met het gebruik van strong negotiation vind het systeem voor alle instanties binnen 5 seconden een oplossing. In de praktijk kan weak negotiation toegepast worden om zo sneller een oplossing te vinden. De volgorde van de jobs is niet meer belangrijk omdat zij in de dynamische productielijnen op willekeurige momenten binnen komen.

Het doel van dit onderzoek is niet om te bepalen of de hoogte van de afwijking, de berekentijd en de grootte van de deadline goed of slecht is. Naar mijn idee is het niet eens mogelijk om dit te bepalen. De waardering hangt af van de eisen van de productielijn en de gebruiker van het systeem. De schaalbaarheid, uitbreidbaarheid, flexibiliteit en betrouwbaarheid van het systeem zijn voor sommige gebruikers het belangrijkst. Dit onderzoek heeft wel aangetoond dat KI technieken en in het bijzonder het mulit-agent systeem van Moergestel, in staat zijn om redelijk goede oplossingen te geven voor het JSSP. In tegenstelling tot de OR-algoritmen, zijn niet alle KI heuristieken in staat om de optimale oplossing vinden. Echter zijn OR-technieken niet direct toepasbaar op een dynamische omgeving en winnen KI methoden flexibiliteit en robuustheid.

9. Discussie

Het door Moergestel beschreven systeem vormt een goede basis voor een praktische toepassing van het job schop scheduling probleem. De uitbreidbaarheid van het systeem maakt het mogelijk om de efficiëntie te verhogen. De behandelde prioriteitsregels zijn naar mijn idee vrij simpel. Het toepassen van bijvoorbeeld de

FDD/MWKR prioriteitsregel (Sels et al, 2012)., zal de afwijking met het optimum verkleinen. Ook bij het kiezen van tijdstippen, in dit geval First fit, is er ruimte voor verbetering. In vervolgend onderzoek kan er gezocht worden naar efficiëntere methoden, zoals bijvoorbeeld een combinatie van First fit en Just in time. Tevens is de manier waarop agenten onderhandelen interessant voor vervolgend onderzoek. Geeft bijvoorbeeld de onderhandelingstechniek Contract nett protocol betere resultaten?
Het grootste nadeel en mogelijk knelpunt is de overhead aan inter-agent communicatie. Wanneer het aantal jobs te hoog is, wordt het systeem te zwaar belast. In dat geval is het mogelijk om over te stappen naar weak negotiation. In een vervolg onderzoek zou ik toetsen wanneer de overhead voor problemen zorgt en wat de efficiëntie is van weak negotiation in een dynamische omgeving.

10. Reflectie

Terugkijkend op dit onderzoek heb ik het literatuur onderzoek zwaar onderschat. Mijn focus lag meer op wat ik wilde implementeren en onderzoeken. Terwijl het in het begin veel belangrijker is om het brede onderzoeksgebied in te perken tot één aspect. Daarnaast zal ik een volgend onderzoek eerder gaan nadenken over een duidelijke structuur. Het is makkelijker om in een logische volgorde een stuk te schrijven. Ik heb sowieso al moeite met het schrijven van een stuk, maar heb alsnog de benodigde tijd verkeerd ingeschat.

De implementatie van het systeem kan naar mijn idee een stuk beter. Ik ben dan ook van plan om op een later moment de code aan te passen zodat de GRAPS principes van object georiënteerd programmeren van toepassing zijn. Tevens kwam ik er pas laat achter dat de implementatie wordt gezien als gereedschap voor het onderzoek. Mits het programma de juiste werking heeft, is de daadwerkelijke implementatie niet belangrijk. Om deze reden heb ik weinig energie meer gestopt in het verbeteren en becommentariëren van de code.

Al met al heb ik veel geleerd van dit bachelor eindwerkstuk. Bijvoorbeeld het inkaderen van de probleemstelling en wanneer een probleem interessant is voor KI, dwong mij om op een andere manier naar de materie te kijken.

11. Samenvatting

Binnen de Kunstmatige Intelligentie (KI) vindt veel onderzoek plaats naar optimalisatie problemen. Dit werkstuk neemt het job shop scheduling problem (JSSP) in beschouwing. Hierbij bestaan er aantal jobs die op een aantal machines uitgevoerd moeten worden. Een job bestaat uit één of meerdere operaties die een bepaalde tijd in beslag nemen. Het doel is om een schema te vinden dat een minimale de totale productietijd of makespan heeft. Binnen het Operationeel Onderzoek (OR) heeft men wiskundige algoritmen gevonden die het optimum vinden. Veel algoritmen, zij het vanuit KI of OR, vinden een oplossing voor de theoretische probleemstelling. In de realiteit gaan machines stuk en verschijnen er op willekeurige tijdstippen nieuwe jobs. In het proefschrift “Agent technology in agile multiparallel manufacturing and product support” beschrijft Moergestel een multi-agent systeem dat flexibel jobs in roostert. In dit werkstuk is onderzocht hoe groot het verlies in optimalisatie is ten behoeven van flexibiliteit het systeem van Moergestel. Als eerst heeft er een literatuurstudie plaats gevonden naar het JSSP, bestaande KI algoritmen en het flexibele multi-agent systeem. In dit systeem bestaan er product agenten die verantwoordelijk zijn voor het inroosteren van een job. Wanneer een agent zijn deadline overschrijdt, zoekt hij naar agenten die een lagere prioriteit hebben.

Deze agenten geven de gereserveerde tijdstippen vrij, zodat de agent wel een geldig schema vindt. Is dit niet het geval, dan is er door het systeem geen oplossing gevonden. Het multi-agent systeem is geimplementeerd in JAVA en is toegepast op bekende JSSP voorbeelden uit de OR-library. De verkregen resultaten zijn vergeleken met de bewezen optima voor die JSSP instanties. Uit de resultaten is gebleken dat het systeem gemiddeld 43% van het optimum af wijkt. Het doel van dit onderzoek was niet om te bepalen of dit goed of slecht is. Dit onderzoek heeft wel aangetoond dat het mulit-agent systeem van Moergestel, in staat zijn om redelijk goede oplossingen te geven voor het JSSP. In tegenstelling tot de OR-algoritmen, is het system niet iin staat om de optimale oplossing vinden. Echter zijn de OR-technieken niet toepasbaar op een dynamische omgeving en winnen KI methoden flexibiliteit en robuustheid.

Appendix A: Literatuur

Bergmann, R., Lindemann, G. & Kirn, S. (2008). *Multiagent Systems Technology: 6th German Conference, MATES 2008, Kaiserslautern, Germany, September 23-26, 2008.* Heidelberg: Springer Verlag Berlin.

Bierwirth, C., Kopfer, H., Mattfeld, D., & Rixen, I., (1995). Genetic Algoritm based scheduling in a dynamic manufacturing environment. *Proceedings of the Second Conference on Evolutionary Computation, 1995 (volume 1).*

Blackstone, J.H., Phillips, D.T., & Hogg, G.L., (1981). A state-of-the-art survey of dispatching rules for manufacturing job shop operations. *International Journal of Production Research 1982 (Volume 20, Issue 1), 27-45*

Çalis B & Bulkan S. (2013). A research survey: review of AI solution strategies of job shop scheduling problem. *Journal of Intelligent Manufacturing, 2013.*

Gutjahr, W.J., (2000). A graph-based Ant System and its convergence, *Future Generation Computer Systems, 2000 (volume 16), 873-888*

Kammer, M., van den Akker, M. & Hoogeveen, H (2010). Identifying and exploiting commonalities for the job-shop scheduling problem. *Computers & Operations Research 38, 2011, 1556-1561*

Lenstra, J.K., Rinnooy Kan, A.H.G., & Brucker, P. Complexity of machine scheduling problems. *Annals of Discrete Mathematics, 1977 (1), 343-362*

Moergestel, L (2014). *Agent technology in agile multiparallel manufacturing and product support* (proefschrift). Utrecht: Universiteit Utrecht

Nakano, R. & Yamada, T. (1991). Conventional genetic algorithms for job shop problems. *In Proceedings of the 4th international conference on genetic algorithms, 1997, 474–479*

Russel, S. & Norvig P., (2003). *Artificial Intelligence, a modern approach*. New Jersey: Pearson Education Inc.

Sels,V., Gheysen, N., & Vanhoucke, M.(2012). A comparison of priority rules for the job shop scheduling problem under different flow time- and tardiness-related objective functions, *International Journal of Production Research, 2012*(Vol. 50, No. 15), 4255–4270

Appendix B: JAVA implementatie

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse MAS: Dit is de main klasse voor dit project. Het maakt een blackboard
aan van x aantal machines en y aantal tijdstippen (in de huidige implementatie
is dit een vaste waarde) en maakt aan de hand van een OR-library tabel voor
elke job een product agent aan. Per product agent worden ook de uit te voeren
operaties aangemaakt  en meegegeven aan de agent. Daarna laat de methode
setAll() een deadline klasse alle deadlines instellen. Vervolgens moeten de
agenten één voor één een rooster vinden voor de job. De agent die inroosterd
krijgt een referentie mee van alle reeds ingeroosterde agenten. Als laatst
wordt de makespan berekend en geprint.
*/
package mas;

import java.util.*;

public class MAS {

        public static void main(String[] args) {
        Blackboard blackboard = new Blackboard(20, 3000);
        setAll(blackboard);
        blackboard.printSchedule("output");

        }

        public static void setAll(Blackboard blackboard){
                ArrayList<ProductAgent> allAgents = new ArrayList<>();
                ArrayList<ProductAgent> scheduledAgents = new ArrayList<>();
                Strategy strat = new CR();
                ReadMagicSquare square = new
ReadMagicSquare("C:/Users/Asrax/Documents/NetBeansProjects/MAS/src/mas/square.txt");
                ArrayList<ArrayList<Integer>> operations = square.getMatrix();
```

```
		for(int i=0;i < operations.size();i++){
			ArrayList<Operation> op = new ArrayList<>();
			for(int j=0;j < operations.get(i).size();j+=2){
			Operation opNew = new Operation(operations.get(i).get(j),
operations.get(i).get(j+1));
		op.add(opNew);
		}
		ProductAgent agentNew = new ProductAgent(i+1, op, blackboard,
scheduledAgents, 0, strat);
		allAgents.add(agentNew);
		}
		Deadline dl = new DeadlineTotalPT();
		dl.calcDeadline(allAgents);
		for(ProductAgent pa : allAgents){
			scheduledAgents.add(pa);
			pa.firstSchedule(pa);
		}

		int makespan = 0;
		for(ProductAgent pa : allAgents){
			int temp = pa.completionTime(pa);
				if(temp > makespan){
					makespan = temp;
				}
			}
		System.out.println("makespan: " + makespan);
	}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse Blackboard: deze klasse represnteert het prikbord. Het is verantwoorde-
lijk voor het initialiseren van het prikbord, het geven van het schema van één
machine, het inroosteren van een schema van een product agent, het opgeven van
reserveerde tijdstippen en het afdrukken van het gevonden schema in een output
file.
*/
package mas;

import java.util.*;
import java.io.*;

public class Blackboard {
	ArrayList<ArrayList<Integer>> schedule = new ArrayList<>();

	public Blackboard(int machines, int timeSteps){
		for(int i=0; i < machines; i++){
			fillMachine(timeSteps);
		}
	}

	private void fillMachine(int timeSteps){
		ArrayList<Integer> row = new ArrayList<>();
		for(int i=0;i < timeSteps;i++){
			row.add(0);
		}
		schedule.add(row);
	}

	public ArrayList<Integer> getMachineRow(int machine){
		return schedule.get(machine);
	}
```

```
	public void reserve(Operation op, int agentNo){
		for(int i=0;i < op.timesteps;i++){
			schedule.get(op.machine).set(i + op.getStartTime(), agentNo);
		}
	}

	public void giveUp(Operation op, int agentNo) {
		for(int i=0;i < op.timesteps;i++){
			schedule.get(op.machine).set(i + op.getStartTime(), 0);
		}
	}

	public void printSchedule(String file){
		String strFilePath =
"C:/Users/Asrax/Documents/NetBeansProjects/MAS/src/mas/" + file +".txt";
		PrintStream ps;
		try {
			ps = new PrintStream(new FileOutputStream(strFilePath));
			for(int j=0; j < schedule.get(0).size();j++){
				ps.print(j+1 + "\t");
			}
			ps.println();
			for(int row=0;row < schedule.size();row++){
				for(int col=0; col < schedule.get(0).size();col++){
					int s = schedule.get(row).get(col);
					ps.print(s + "\t");
				}
				ps.println();
			}
			ps.close();
		}
		catch (FileNotFoundException e) {
			System.out.println(e.getMessage());
		}
	}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse CR: Deze klasse bevat de berekeningen voor het vergelijken van de
critical ratio. De klasse geeft de lijst van agenten terug die een lagere
prioriteit hebben.
*/
package mas;

import java.util.ArrayList;

public class CR implements Strategy {
	@Override
	public ArrayList<ProductAgent> priority(ProductAgent pa, ArrayList<ProductAgent>
all){
		ArrayList<ProductAgent> agents = new ArrayList<>();
		float cr1 = (float) pa.getDeadline() / pa.getProductionTime();
		for(ProductAgent agent : all){
			float cr2 = (float) agent.getDeadline() / agent.getProductionTime();
			if(cr2 > cr1) {
				agents.add(agent);
			}
		}
		return agents;
	}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Interface Deadline: bepaalt de interface voor de deadline klassen. Met het
gebruik van deze interface hoeven de overige klassen niet te weten wat voor
deadline klasse het betreft.
*/
package mas;
import java.util.*;

public interface Deadline {
	public void calcDeadline(ArrayList<ProductAgent> allPA);
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse DeadlineESD: De klasse geeft aan elke job dezelfde hoeveelheid slack
*/
package mas;

import java.util.ArrayList;

public class DeadlineESD implements Deadline{
	int slack;

	public DeadlineESD(int a){
		slack = a;
	}

	@Override
	public void calcDeadline(ArrayList<ProductAgent> allPA) {
		for(ProductAgent pa : allPA){
			int deadline = pa.getProductionTime() + slack;
			pa.setDeadline(deadline);
		}
	}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse DeadlineFVD: De klasse geeft aan elke job dezelfde deadline.
*/
package mas;

import java.util.ArrayList;

public class DeadlineFVD implements Deadline {
	int fixed;

	public DeadlineFVD(int a){
		fixed = a;
	}

	@Override
	public void calcDeadline(ArrayList<ProductAgent> allPA) {
		for(ProductAgent pa : allPA){;
			pa.setDeadline(fixed);
		}
	}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse DeadlineTWK: De klasse berekent de deadline voor elke job aan de hand
de TWK methode en geeft deze deadline aan de agent.
*/
package mas;

import java.util.ArrayList;

public class DeadlineTWK implements Deadline{
	double allowance;

	public DeadlineTWK(double a){
		allowance = a;
	}

	@Override
	public void calcDeadline(ArrayList<ProductAgent> allPA) {
		for(ProductAgent pa : allPA){
			int deadline = (int) ((int) pa.getProductionTime() * allowance);
			pa.setDeadline(deadline);
		}
	}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse DeadlineTWK: De klasse berekent de totale productietijd van alle jobs en
stelt deze deadline in bij elke product agent.
*/
package mas;
import java.util.*;

public class DeadlineTotalPT implements Deadline{
	@Override
	public void calcDeadline(ArrayList<ProductAgent> allPA) {
		int totalPT = 0;
		for(ProductAgent pa : allPA){
			totalPT += pa.getProductionTime();
		}
		for(ProductAgent pa : allPA){
			pa.setDeadline(totalPT);
		}
	}

}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse EDF: Deze klasse bevat de berekeningen voor het vergelijken van de
earliest deadline. De klasse geeft de lijst van agenten terug die een lagere
prioriteit hebben.
*/
package mas;

import java.util.*;

public class EDF implements Strategy {
	@Override
	public ArrayList<ProductAgent> priority(ProductAgent pa, ArrayList<ProductAgent>
all){
		ArrayList<ProductAgent> agents = new ArrayList<>();
		for(ProductAgent agent : all){
			if(agent.getDeadline() > pa.getDeadline()) {
				agents.add(agent);
			}
		}
		return agents;
	}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse LSF: Deze klasse bevat de berekeningen voor het vergelijken van de
least slack. De klasse geeft de lijst van agenten terug die een lagere
prioriteit hebben.
*/
package mas;

import java.util.*;

public class LSF implements Strategy {
	@Override
	public ArrayList<ProductAgent> priority(ProductAgent pa, ArrayList<ProductAgent>
all){
		ArrayList<ProductAgent> agents = new ArrayList<>();
		int slack = pa.getDeadline() - pa.getProductionTime();
		for(ProductAgent agent : all){
			if(agent.getDeadline() - agent.getProductionTime() > slack) {
				agents.add(agent);
			}
		}
		Collections.sort(agents, Slack);
		return agents;
	}

public static Comparator<ProductAgent> Slack = new Comparator<ProductAgent>() {
	public int compare(ProductAgent pa1, ProductAgent pa2) {
		int slack1 = pa1.getDeadline() - pa1.getProductionTime();
		int slack2 = pa2.getDeadline() - pa2.getProductionTime();

		/*For ascending order*/
		return slack1-slack2;
```

```
                }
        };

}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse operation: Deze klasse houdt bij op welke machine een operatie uitgevoerd
moet worden, hoeveel tijdstippen dit in beslag neemt en de starttijd van een
ingeroosterde operatie.
*/
package mas;

public class Operation {
        public final int machine;
        public final int timesteps;
        private int startTime;

        public Operation(int x, int y) {
                machine = x;
                timesteps = y;
         }

         public void setStartTime(int i){
                startTime = i;
         }

         public int getStartTime(){
                return startTime;
         }

}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Klasse ProductAgent: Deze klasse bezit alle benodigde informatie die nodig om
een job in te roosteren. Het doorzoeken van het blackboard verloopt als volgt:
de agent vraagt aan het prikbord per operatie om de desbetreffende machine rij.
De agent doorloopt de rij en telt het aantal opeenvolgende nullen (tijdstip
beschikbaar). Wanneer het aantal opeenvolgende nullen gelijk is aan de
productietijd van een operatie, wordt de huidige index minus het aantal
nullen + 1 als starttijd van een operatie opgeslagen.
(het tellen van de nullen begint al bij index nul.
Dus wanneer de productietijd 1 is en het er dus bij nul begonnen moet worden,
moet er één bij opgeteld worden om zo je juiste starttijd van de operatie
namelijk nul te verkrijgen). Bij het claimen van de tijdstippen geeft de
productagent één voor één zijn operaties door aan het prikbord. Het prikbord
zoekt aan de hand van de operatie informatie in de dubbele array met het machine
nummer (= rij in tabel). De starttijd van de operatie geeft de index (kolom)
aan. Voor het aantal productiestappen schijft het prikbord vanaf het startpunt
het agentnummer schrijft in het tabel. Het vrijgeven van tijdstippen verloopt
precies hetzelfde alleen wordt er steeds een nul ingevuld ipv het agentnummer.
Op deze manier hoeft niet steeds heel de dubbele array doorgelopen te worden,
waarbij gezocht wordt naar het agent nummer.
*/
package mas;

import java.util.*;

public class ProductAgent {
        int agentNo;
        ArrayList<Operation> operations;
        ArrayList<ProductAgent> allPA;
        Blackboard blackboard;
```

```
	Strategy strategy;
	ArrayList schedule;
	private final int start;
	private int deadline;
	private int productionTime;

	ProductAgent(int n, ArrayList<Operation> o, Blackboard b, ArrayList<ProductAgent>
a, int s, Strategy st) {
		agentNo = n;
		operations = o;
		blackboard = b;
		allPA = a;
		start = s;
		strategy = st;
		calculatePT();
	}

	@Override
	public String toString() {
		return "[ No=" + agentNo +  " deadline=" + deadline + ", production time=" +
productionTime + "]";
	}

	public int getAgentNo(){
		return agentNo;
	}

	public ArrayList<Operation> getOperations(){
		return operations;
	}

	public int getStart(){
		return start;
	}

	public void setDeadline(int d){
		deadline = d;
	}

	public int getDeadline(){
		return deadline;
	}
```

```
public int getProductionTime(){
	return productionTime;
}

private void calculatePT(){
	int sum = 0;
	for(Operation op : operations){
		sum = sum + op.timesteps;
	}
	productionTime = sum;
}

/*
Uncomment regels om weak negotiation toe te passen.
*/
public void firstSchedule(ProductAgent pa){
//	int finishedTime = schedule(pa);
//	if(pa.getDeadline() < finishedTime -1) {
	reschedule(pa);
//	}
//	else reserveSlots(pa);
}

/*
method schedule(ProductAgent): Vanaf de starttijd zoekt de agent per
operatie naar een slot van tijdstippen waarin de operatie uit te voeren is.
*/
public int schedule(ProductAgent pa){
	int scheduledTime = pa.getStart();
	for(Operation op : pa.getOperations()){
		findFreeSlot(scheduledTime, op);
		scheduledTime = op.getStartTime() + op.timesteps;
	}
	return scheduledTime;
}

/*
Method reschedule(ProductAgent): Aan de hand van de prioriteits strategie
vindt de product agent alle agenten die een lagere prioriteit hebben. Deze
geven hun gereserveerde tijdstippen en de product agent probeert opnieuw
een geldig rooster te vinden.
*/
public void reschedule(ProductAgent pa1) {
```

```
                ArrayList<ProductAgent> lowerAgents = strategy.priority(pa1, allPA);
                for(ProductAgent pa2 : lowerAgents) {
                        for(Operation op : pa2.getOperations()){
                                blackboard.giveUp(op, pa2.getAgentNo());
                                op.setStartTime(0);
                        }
                }
                int finishedTimePA1 = schedule(pa1);
                if(pa1.getDeadline() <= finishedTimePA1 -1) {
                //finishedTime geeft laatste operatiestap +1
                        System.out.println("Schedule failed because of product agent: " +
pa1.getAgentNo());
                }
                else {
                        reserveSlots(pa1);
                        for(ProductAgent pa2 : lowerAgents) {
                                int finishedTimePA2 = schedule(pa2);
                                if(pa2.getDeadline() <= finishedTimePA2 -1) {
                                        System.out.print("Schedule failed because of product
                agent: " + pa2.getAgentNo());
                                }
                                else reserveSlots(pa2);
                        }
                }
        }

        public void findFreeSlot(int initTime, Operation op){
                ArrayList<Integer> row = blackboard.getMachineRow(op.machine);
                int count = 0;
                for(int i=initTime;i < row.size(); i++) {
                        if(row.get(i) == 0) {
                                count++;
                        } else {
                                count = 0;
                        }
                        if(count == op.timesteps){
                                op.setStartTime(i - count +1);
                                break;
                        }
                }
        }

        public void reserveSlots(ProductAgent pa){
```

```
			for(Operation op : pa.getOperations()){
				blackboard.reserve(op, pa.getAgentNo());
			}
		}

		public int completionTime(ProductAgent pa){
			int completionTime = 0;
			for (Operation op : pa.getOperations()){
				completionTime = op.getStartTime() + op.timesteps;
			}
			return completionTime;
		}
}
```

```
/*
Bachelor eindwerkstuk KI:
Job shop scheduling in de werkelijkheid:
De efficientie en flexibiliteit van een multi-agent systeem
Door A.C. Leenhouwers, 3019292

Interface Deadline: bepaalt de interface voor de deadline stategie klassen. Met
het gebruik van deze interface hoeven de overige klassen niet te weten wat voor
strategie klasse het betreft.
*/
package mas;

import java.util.*;

public interface Strategy {
	public ArrayList<ProductAgent> priority(ProductAgent pa, ArrayList<ProductAgent>
all);
}
```