Lokale zoekmethoden voor een een-machine volgordeprobleem

H. A. J. Crauwels [1]

L. N. Van Wassenhove [2]

Samenvatting

Omdat voor vele problemen de optimale algoritmen te veel rekentijd en computergeheugen vereisen, wordt momenteel binnen de onderzoekswereld veel aandacht gespendeerd aan heuristische zoekprocedures. De meeste procedures zijn spin-offs van de zogenaamde *local neighbourhood search* heuristieken: simulated annealing, threshold accepting, tabu search en genetic algorithms. De oplossing die door deze methodes gevonden wordt, is meestal beter dan deze verkregen door het toepassen van eenvoudige prioriteitsregels. Deze verschillende methodes worden geïllustreerd aan de hand van een klassiek machinevolgordeprobleem: het minimaliseren van de totale levertijdoverschrijding.

1. Inleiding.

De intelligentie van een planningssysteem wordt gevormd door de schedulingsmotor. Zoals de studie van Ernst & Young [5] aangeeft, zijn er verschillende categorieën te onderscheiden: heuristische systemen, op theory-of-constraints gebaseerde systemen, op simulatie gebaseerde systemen, kennisgebaseerde systemen en optimizers. Simulatie en optimizers zijn technieken die reeds lang binnen de wereld van het operationeel onderzoek gebruikt worden. Theory-of-constraints is geïntroduceerd door Goldratt [9], maar elementen uit deze theorie worden nu ook in andere systemen aangewend, bijvoorbeeld bij bottleneck-analyses. De kennisgebaseerde systemen omvatten onder andere expertsystemen en "constraint satisfaction" technieken. In deze tekst wordt een aantal heuristische methodes besproken.

De reden waarom op heuristische methoden beroep moet gedaan worden, is te zoeken in de resultaten van de *complexiteitstheorie*. Deze theorie verstrekt het mathematische raamwerk voor de classificatie van problemen als "gemakkelijk" en "aartsmoeilijk". Voor de aartsmoeilijke types kan men problemen van beperkte grootte optimaal oplossen met behulp van enumeratieve optimalisatieprocedures (zoals branch-and-bound). Voor grotere problemen, degene die men in de praktijk opgelost wil zien, vragen deze optimale procedures teveel rekentijd en/of teveel computergeheugen. Men is dan aangewezen op suboptimale technieken. Algemeen in gebruik zijn constructieheuristieken, welke resulteren in prioriteitsregels. Vanwege de eenvoud kan zo'n regel snel een oplossing vinden, maar voor grote problemen is de kwaliteit van de oplossing dikwijls vrij slecht. Deze oplossingen kunnen echter als start dienen voor zoekheuristieken.

De meest klassieke methode hiervan is descent waarbij men in de omgeving van een oplossing naar een betere oplossing zoekt. Elementen in de omgeving worden geconstrueerd

---

[1] H. Crauwels, De Nayer instituut, J.De Nayerlaan 5, 2860 Sint-Katelijne-Waver, België
Tel. (+32) 15 31 69 44 (correspondentie)

[2] L. Van Wassenhove, INSEAD, Boulevard de Constance, 77305 Fontainebleau, Frankrijk
Tel. (+33) 1 60 72 42 95

door een kleine wijziging aan de oplossing aan te brengen, door bijvoorbeeld een aantal taken van plaats te wisselen. Vanuit deze betere oplossing gaat men iteratief verder zoeken tot geen betere oplossing meer gevonden wordt. Het resultaat is een lokaal optimum dat afhankelijk is van de startoplossing. Om die invloed van de startoplossing weg te werken, herhaalt men de procedure voor een aantal verschillende startoplossingen. Hierdoor verkrijgt men een verzameling lokale optima en men kiest hiervan de beste als oplossing van het probleem. In de jaren tachtig zijn er alternatieven voor het probleem van lokale optima en startoplossing-afhankelijkheid ontstaan: simulated annealing, threshold accepting en tabu search. Wanneer in de omgeving van een oplossing geen betere oplossing kan gevonden worden, wordt eventueel naar een slechtere oplossing gegaan om van daaruit verder te zoeken. De procedure wordt verder gezet tot aan een vooraf ingesteld stopcriterium (bijv. aantal iteraties) voldaan is. Deze methodes worden meta-heuristieken genoemd omdat ze op een heuristische manier het zoekproces (zelf een heuristiek) sturen zodat verschillende regio's van de oplossingsruimte bezocht worden.
In tegenstelling tot deze omgevingszoekmethodes, waarin met een iteratieproces van één oplossing naar een volgende gewerkt wordt, start een genetisch algoritme met een beperkte populatie van oplossingen. Vanuit deze populatie worden iteratief volgende populaties berekend door middel van genetische operatoren waaronder reproductie (het overleven van de betere oplossingen), crossover (het uitwisselen van deelinformatie tussen twee oplossingen) en mutatie (aanbrengen van een willekeurig kleine wijziging aan een oplossing). In dit iteratief proces verwacht men dat de gemiddelde en beste waarde van de oplossingen van populatie tot populatie verbetert.
Deze verschillende methodes worden in deze tekst beschreven en de werking van hun parameters belicht. Ook wordt voor elk van deze procedures een implementatie voor het klassieke probleem van het minimaliseren van de totale levertijdoverschrijding besproken. Afsluitend wordt een vergelijking gegeven van de resultaten verkregen met deze verschillende technieken.

2. Zoeken met behulp van omgevingen.

Een lokale zoekheuristiek is een iteratieve procedure waarbij een verzameling van oplossingen X geëxploreerd wordt naar een globaal optimum. In wat volgt zal steeds een minimalisatie probleem behandeld worden. Er wordt gestart van een willekeurige oplossing $x_i \in X$ en bij elke stap i, wordt een nieuwe oplossing x_{i+1} uit de omgeving $N(x_i)$ van de actuele oplossing x_i onderzocht om eventueel de nieuwe actuele oplossing te worden.
Hierbij moeten twee elementen nader gedefinieerd worden. Een omgeving van een oplossing x_i is een deelverzameling $N(x_i) \subset X$, welke elementen bevat die vanuit de oplossing via een eenvoudige bewerking bereikt kunnen worden. Bij een machinevolgordeprobleem bestaat zo'n bewerking meestal uit het omwisselen van taken. Daarnaast moet ook gespecificeerd worden hoe de nieuwe oplossing gekozen wordt. De gebruikelijkste manier is elementen te kiezen waarvan de doelfunctiewaarde minstens even goed is als de waarde van de actuele oplossing: $F(x_{i+1}) \leq F(x_i)$. Wanneer geen betere oplossing gevonden kan worden in de omgeving, wordt het zoekproces gestopt. Deze strategie wordt *descent* genoemd.

Descent Algoritme

- Initialisatie: kies een willekeurige oplossing $x_1 \in X$

- Stap $i = 1, 2, \ldots$: x_i duidt de actuele oplossing aan

  a) kies een element x_{i+1} uit de omgeving $N(x_i)$

  b) indien $F(x_{i+1}) \leq F(x_i)$ dan wordt x_{i+1} de nieuwe actuele oplossing, in het andere geval blijft x_i de actuele oplossing

  c) indien in de volledige omgeving geen betere oplossing meer kan gevonden worden, wordt de procedure gestopt.

De zoektocht wordt dus bij de best gevonden oplossing gestopt, maar dit is niet noodzakelijk een optimale oplossing omdat ze afhankelijk is van de startoplossing en het gebruikte type van omgeving: men heeft slechts een lokaal optimum gevonden. Door van verschillende willekeurig gegenereerde startoplossingen te vertrekken, kan men verschillende lokale optima verkrijgen, waarvan het beste misschien vrij dicht bij het globaal optimum ligt. In dit geval spreekt men van *multi-start descent.*
Verschillende alternatieven waarbij nu en dan overgegaan wordt naar een oplossing met een minder goede kwaliteit, worden in de volgende hoofdstukken besproken. De onderlingende redenering is telkens dat men uit een lokaal optimum moet *klimmen* om van daaruit naar een beter lokaal optimum te evolueren.

3. Simulated annealing.

Een eerste methode die op dit principe gebaseerd is, is *simulated annealing.* Wanneer voor een slechtere oplossing gekozen wordt, is hier afhankelijk van een dalende functie die de kans aangeeft om een 1 % toename van de doelfunctie te accepteren. Deze aanvaardingskans K_l start bij een waarde K_0 en daalt in L niveaus naar een eindkans K_f. Het verloop van deze aanvaardingskansfunctie kan bijvoorbeeld lineair: $K_{l+1} = K_l - \frac{K_0 - K_f}{L-1}$ of convex: $K_{l+1} = aK_l$ met $a = (K_f/K_0)^{1/(L-1)}$ zijn.
Voor deze kans op een bepaald niveau wordt een Boltzmann-achtige verdeling gebruikt, $\exp(-\delta/t)$ met δ de toename in de doelfunctiewaarde en t een parameter die naar analogie met het fysisch gloeiproces, temperatuur genoemd wordt. Omdat K_l gedefinieerd is als de aanvaardingskans op een 1% toename, kan hieruit de temperatuur t berekend worden: $t = -0.01F^*/\ln K_l$. Op basis van t kan dan de kans bepaald worden een willekeurige toename δ te aanvaarden.
Gedurende elk aanvaardingskansniveau wordt de omgeving van een oplossing onderzocht. Dit kan de volledige omgeving zijn, die sequentieel bekeken wordt of een aantal elementen die willekeurig in de omgeving gekozen worden. Een bijkomend element dat moet bepaald worden is het stopcriterium, bijvoorbeeld door het vastleggen van het aantal aanvaardingskansniveaus.

Simulated Annealing

- Initialisatie: kies een willekeurige oplossing $x_1 \in X$

  $F^* \leftarrow F(x_1)$; $x^* \leftarrow x_1$

  starttemperatuur: $t_0 = -0.01F^*/\ln K_0$ en $l = 0$

- Stap $i = 1, 2, \ldots$: x_i duidt de actuele oplossing aan

  a) kies een element x uit de omgeving $N(x_i)$

b) indien $F(x) \leq F(x_i)$ dan $x_{i+1} \leftarrow x$
indien $F(x) < F^*$ dan $F^* \leftarrow F(x)$; $x^* \leftarrow x$

c) anders, kies r, een random getal in [0,1]
indien $r < \exp(-(F(x) - F(x_i))/t_l)$ dan $x_{i+1} \leftarrow x$

d) indien de omgeving $N(x_i)$ onderzocht is, wordt de temperatuur aangepast:
$l \leftarrow l + 1$; $K_l = K_{l-1} - (K_0 - K_f)/(L-1)$; $t_l = -0.01F^*/\ln K_l$

e) indien K_l gelijk is aan de eindwaarde K_f, wordt de procedure gestopt.

Simulated annealing werd voor de eerste maal als oplossingsmethode voor een combinatorisch probleem voorgesteld door Kirkpatrick *et al.* [13]. Nadien zijn tientalle artikels verschenen welke zowel de theoretische (bijv. Van Laarhoven en Aarts [19]) als praktische aspecten (bijv. Eglese [4]) van deze methode beschrijven. Een overzicht van de toepassingen op problemen uit het operationeel onderzoek wordt gegeven door C. Koulamas *et al.* [14]. Johnson *et al.* [11][12] bespreken een implementatie voor twee problemen uit de grafentheorie waarbij uitgebreid ingegaan wordt op de parameterkeuze.

4. Threshold accepting.

Deze methode werd geïntroduceerd door Dueck en Scheuer [3]. De werking is te vergelijken met simulated annealing op één belangrijk element na. Bij simulated annealing is de aanvaarding van een verslechtering van de oplossing afhankelijk van een kansfunctie. Of naar een slechtere oplossing gegaan wordt, wordt voor een deel door toeval bepaald. Bij threshold accepting wordt steeds naar een slechtere waarde gegaan zolang de verslechtering kleiner blijft dan een bepaalde waarde (de *threshold*). Analoog aan de aanvaardingskansfunctie moet hier een thresholdfunctie gedefinieerd worden in functie van het aantal drempelniveaus L. Dit verloop kan weer verschillende vormen aannemen, bijvoorbeeld lineair: $T_{l+1} = T_l - \frac{T_0 - T_f}{L-1}$, waarbij T_0 en T_f de initiële en einddrempelwaarde zijn. Deze waarden moeten op één of andere manier gekozen worden, bijvoorbeeld als een fractie van de startdoelfunctiewaarde $F(x_1)$.

5. Tabu search.

Bij *tabu search* wordt niet een willekeurig element uit de omgeving $N(x_i)$ onderzocht om dat eventueel als volgende oplossing te nemen, maar wordt, in principe, het beste element x uit de omgeving gekozen. Men beweegt dus van x_i naar x. Wanneer $F(x) \leq F(x_i)$ is dit te vergelijken met *descent*. Wanneer echter geen betere oplossing kan gevonden worden, wordt toch overgegaan naar de minst slechte uit de omgeving.
Om te voorkomen dat men in volgende iteraties terug gaat naar oplossingen die reeds bezocht zijn, wordt een lijst van verboden of zogeheten *taboe bewegingen* bijgehouden. Deze lijst is cyclisch georganiseerd: achteraan wordt de tegengestelde beweging $x \rightarrow x_i$ toegevoegd terwijl vooraan een taboe beweging uit de lijst verdwijnt en dus terug toegelaten wordt. Omdat in vele gevallen niet de volledige specificatie van de tegengestelde beweging op de taboe lijst bijgevoegd wordt, maar slechts een karakteristiek kenmerk ervan, is het mogelijk dat meer elementen in de omgeving verboden worden dan nodig is. Soms zijn er zelfs taboe elementen met een betere doelfunctiewaarde. Om deze elementen niet te missen, kan een aspiratie criterium ingebouwd worden. Zo'n aspiratie criterium is een voorwaarde die toelaat de taboe status van een element te negeren. Een voorwaarde

kan bijvoorbeeld zijn dat het element een betere doelfunctiewaarde heeft dan de waarde die tot dan toe gevonden is.

Tabu Search

- Initialisatie: kies een willekeurige oplossing $x_1 \in X$

  $F^* \leftarrow F(x_1)$; $x^* \leftarrow x_1$

- Stap $i = 1, 2, \ldots$: x_i duidt de actuele oplossing aan

  a) zoek het beste element x uit de omgeving $N(x_i)$ dat niet taboe is, tenzij aan het aspiratie criterium voldaan is

  b) $x_{i+1} \leftarrow x$; indien $F(x) < F^*$ dan $F^* \leftarrow F(x)$; $x^* \leftarrow x$

  c) pas de taboe lijst aan

  d) de procedure wordt gestopt indien aan het stopcriterium voldaan is, bijvoorbeeld een gegeven aantal iteraties uitgevoerd.

Sinds de introductie door Glover [6], is tabu search uitgeprobeerd op tal van combinatorische problemen. Praktische richtlijnen voor het ontwerp van een tabu search algoritme zijn samengebracht in Glover *et al.* [7].

6. Genetische algoritmen.

Een genetisch algoritme is een iteratieproces waarbij uit een populatie oplossingen een nieuwe populatie gegenereerd wordt met behulp van genetische operatoren. Betere oplossingen hebben meer kans om te overleven en geven ook hun intrinsieke eigenschappen door aan elementen in de volgende populatie. Deze nieuwe elementen worden gecreëerd door delen van reeds bestaande elementen met elkaar te combineren (de crossover operator). Men verwacht dat tijdens het iteratieproces steeds betere oplossingen ontstaan en dat convergentie optreedt naar het globaal optimum. Omwille van de analogie met genetische processen, worden de elementen van de populatie, die elk een oplossing voorstellen, soms ook chromosomen genoemd.
Een basisidee bij een genetisch algoritme is "survival of the fittest". De doelfunctiewaarde van een oplossing (z_i) moet dus omgezet worden naar een fitheidswaarde (f_i). Voor een minimalisatieprobleem kan dit bijvoorbeeld als volgt gebeuren: $f_i = z_{\max} - z_i$ waarbij $z_{\max}$ de grootste doelfunctiewaarde in de populatie is. Om voortijdige convergentie tegen te gaan, kunnen deze fitheidswaarden geschaald worden. Bij lineaire schaling worden de f_i's zodanig aangepast dat de gemiddelde geschaalde fitheidswaarde gelijk blijft aan de niet-geschaalde waarde en dat het beste populatie element een aantal duplicaten in de nieuwe populatie krijgt, gelijk aan een parameter (C_{mult}).
Tijdens een iteratiestap worden uit de populatie chromosomen geselecteerd op basis van hun (geschaalde) fitheidswaarde. Een goed chromosoom wordt vaker *gereproduceerd* in de nieuwe populatie dan een slecht. In een populatie met M chromosomen wordt het verwachte aantal chromosomen e_i berekend met $e_i = (f_i / \sum f_i) * M$. Het aantal geselecteerden voor elk chromosoom is gelijk aan het geheel deel van e_i. Dit wordt aangevuld door het fractionele deel van e_i van groot naar klein te sorteren en hiervan de eerste elementen te nemen totdat er M geselecteerd zijn.

Een belangrijk element bij het ontwerp van een genetisch algoritme, is hoe een oplossing voorgesteld wordt. Deze oplossingscodering bepaalt mede de efficiëntie van de crossover operator, welke voor de uitwisseling van deelinformatie tussen twee oplossingen zorgt. De klassieke manier die bij het maximaliseren van functies gebruikt wordt, is een binaire voorstelling. Crossover wordt uitgevoerd door de twee oplossingen op een door het toeval bepaalde plaats door te knippen en de twee staartstukken met elkaar om te wisselen.
Voorbeeld met het snijpunt na het 7^e element in de oplossing:

Oorspronkelijk:		Na uitwisselen:
1011 110 1 1111 0011	→	1011 110 1 1001 1001
0000 101 1 1001 1001		0000 101 1 1111 0011

Bij machinevolgordeproblemen is de natuurlijke voorstelling van een oplossing een permutatie van n getallen. Crossover met uitwisseling van staartstukken leidt hier echter naar irrealistische oplossingen. Daarom wordt dikwijls de *Partially Mapped Crossover* (PMX) operator gebruikt. Hierbij worden de oplossingen twee aan twee beschouwd. Er worden met behulp van random getallen twee snijpunten gegenereerd en de delen tussen deze twee snijpunten worden tussen de twee oplossingen uitgewisseld. Na deze omwisseling is het mogelijk dat bepaalde taken tweemaal in de volgorde voorkomen: eenmaal in het gewisselde interval en eenmaal erbuiten. Deze laatste wordt dan vervangen door de taak die in het oorspronkelijke interval op die plaats stond.
Voorbeeld met snijpunten na het 3^e en het 5^e element in de oplossing:

Oorspronkelijk:		Na uitwisselen:		En aanpassen:
6 8 7 1 4 5 2 3	→	6 8 7 2 3 5 2 3	→	6 8 7 2 3 5 1 4
4 7 1 2 3 6 5 8		4 7 1 1 4 6 5 8		3 7 2 1 4 6 5 8

Of crossover tussen twee elementen plaatsvindt of niet, wordt bepaald door de crossoverkans parameter. Gewoonlijk ligt deze waarde tussen 0.8 en 1.0.
Daarnaast wordt tijdens elke iteratie ook de *mutatie* operator toegepast. Een klein aantal veranderingen (bepaald door de mutatiekans parameter) worden op willekeurige plaatsen in de nieuwe populatie aangebracht. Bij een binaire codering kunnen bijvoorbeeld enkele bits van waarde veranderd worden; bij een permutatie voorstelling kunnen twee taken in een oplossing van plaats omgewisseld worden.
Het iteratieproces stopt wanneer een vooraf vastgelegd aantal generatiestappen uitgevoerd is. Het kan ook nuttig zijn om bijkomende stopcriteria in te bouwen op basis van de graad van convergentie van de verschillende chromosomen in een populatie.

Genetic algorithm

Stap 1. Creëer een initiële populatie en noem deze de actuele populatie. Evalueer de actuele populatie. Het beste element wordt x^* met bijhorende waarde F^*.

Stap 2. Genereer een nieuwe populatie vanuit de actuele populatie door middel van de genetische operatoren reproductie, crossover en mutatie.

Stap 3. Evalueer de nieuwe populatie, het beste element is x_i. Indien $F(x_i) < F^*$ dan $F^* \leftarrow F(x_i)$, $x^* \leftarrow x_i$. Indien aan een convergentie criterium voldaan is, stop. Vervang de actuele door de nieuwe populatie en noem deze de actuele populatie. Indien het maximum aantal generaties nog niet is uitgevoerd, ga naar Stap 2.

De basis voor genetische algoritmen werd gelegd door Holland [10]. Een inleiding tot de theorie van genetische algoritmen en een aantal toepassingen kan gevonden worden in Goldberg [8]. Een tutorial voor de ontwikkeling van een genetisch algoritme wordt gegeven door Davis [2].

7. Voorbeeld: totale gewogen levertijdoverschrijding.

Probleemstelling.

Gegeven zijn n taken die uitgevoerd moeten worden op één machine. Ook zijn voor elke taak de bewerkingstijd p_i, het gewicht $w_i > 0$ en levertijd d_i gegeven. Het completeringstijdstip C_i van een taak op positie k is gelijk aan de som van de bewerkingstijden van de taken van positie 1 tot en met positie k. Uit deze completeringstijdstippen kan voor elke taak de levertijdoverschrijding berekend worden: $T_i = \max(0, C_i - d_i)$. Het doel is een bewerkingsvolgorde te vinden die de totale gewogen levertijdoverschrijding ($\sum w_i T_i$) minimaliseert.
Voor het speciale geval, waarbij de gewichten van alle taken aan elkaar gelijk zijn, is door Potts en Van Wassenhove [16] een decompositie algoritme ontwikkeld. Met deze methode kunnen problemen tot 100 taken binnen aanvaardbare rekentijd optimaal opgelost worden. Voor het algemene probleem met gewichten hebben Potts en Van Wassenhove [17] een branch and bound algoritme ontwikkeld dat problemen met een aantal taken kleiner dan 40 bevredigend oplost.
De benodigde rekentijd voor deze optimale procedures is echter vrij groot zodat in praktijk dikwijls beroep gedaan wordt op eenvoudige prioriteitsregels. De eenvoudigste regels zijn de SWPT regel, waarbij de taken geordend worden in stijgende p_i/w_i verhouding, en de EDD regel, waarbij de taken volgens stijgende levertijden d_i gesorteerd worden.
In de AU regel (Morton *et al.* [15]) wordt zowel met de levertijden als met de p/w verhoudingen rekening gehouden. De taken worden geordend volgens dalende urgentie waarbij de urgentie berekend wordt als:

$$u_i = (w_i/p_i)\exp(-\max\{0, d_i - t - p_i\}/k\bar{p})$$

waarbij t de starttijd van taak i is, k een parameter afhankelijk van de gemiddelde vertragingsfaktor, en $\bar{p}$ de gemiddelde bewerkingstijd.
Uit de rekenresultaten zal blijken dat deze eenvoudige regels dikwijls resulteren in oplossingen die vrij ver verwijderd zijn van de optimale oplossing. Anderzijds vraagt de optimale oplossingsprocedure soms teveel rekentijd, voornamelijk wanneer het aantal taken groot is. Vandaar de zoektocht naar goede en robuuste heuristieken. Een gedetailleerde beschrijving van de verschillende meta-heuristieken kan gevonden worden in Potts en Van Wassenhove [18] en Crauwels *et al* [1]. Hieronder worden enkel de belangrijkste punten kort beschreven.

De omgeving.

Voor dit één machine volgordeprobleem zijn verschillende definities voor de omgeving mogelijk. De meest eenvoudige bestaat erin de twee taken in naast elkaar gelegen posities om te wisselen. Het aantal elementen in de omgeving is dan $n-1$. Dit type van omgeving gaf echter geen goede resultaten bij de verschillende algoritmen. De reden is waarschijnlijk

dat deze omgeving te beperkend is om het zoeken te diversifiëren naar verschillende regio's in de oplossingsruimte.
Een meer algemene omgeving kan gedefinieerd worden door in de volgorde twee posities u en v $(1 \leq u < v \leq n)$ te kiezen en de taken in deze posities om te wisselen. Deze omgeving bevat $n(n-1)/2$ elementen. Het is deze omgeving die in onze implementatie voor het *descent* algoritme gebruikt is. In tegenstelling tot een strikte descent, worden in onze methode ook steeds de nul-wisselingen (het omwisselen van twee taken die op tijd zijn) doorgevoerd. Een gevolg hiervan is dat de procedure niet gestopt wordt wanneer geen beter element meer kan gevonden worden in de omgeving. Zelfs wanneer alleen nul-wisselingen doorgevoerd zijn, loont het misschien de moeite om de omgeving van de nieuwe oplossing verder te doorzoeken. Er is dus een expliciet stopcriterium nodig, bijvoorbeeld een maximum aantal iteratieniveaus.
Ook voor *simulated annealing* en *threshold accepting* wordt deze omgeving gebruikt.
Bij *tabu search* bleek deze omgeving echter te groot te zijn om tijdens elke iteratiestap volledig te doorzoeken naar het beste element. Daarom wordt uit de volledige omgeving op basis van het toeval een deelverzameling van n elementen gekozen. Uit deze n elementen wordt het beste niet-taboe element gekozen. Het karakteristiek kenmerk van de omwisseling van de jobs in posities u en v, dat op de taboe lijst bijgevoegd wordt, bevat alleen de identifikatie van de job op positie v. Omdat hierdoor niet alleen de vorige oplossing maar ook een aantal andere mogelijke oplossingen taboe worden, is een aspiratie criterium ingebouwd. Taboe oplossingen met een betere doelfunctiewaarde worden toch geaccepteerd. Bij dit probleem is het ook mogelijk dat gedurende een aantal iteratiestappen geen verandering in de doelfunctiewaarde optreedt: de beste beweging in de deelverzameling van de omgeving is steeds het omwisselen van twee taken die op tijd zijn. Een meer diversifiërende beweging is dan nodig door de u, v posities verder uit elkaar te nemen: u in de eerste $n/4$ en v in de laatste $n/4$ posities.
In deze drie meta-heuristieken is ook een bijkomend intensifiërend element ingebracht. Telkens een daling van de doelfunctiewaarde volgt op een geaccepteerde toename, wordt een strikt descent proces, gebaseerd op het omwisselen van naast elkaar gelegen taken, uitgevoerd.

Genetische algoritmen.

In onze implementatie van een genetisch algoritme, werden twee verschillende voorstellingen voor de oplossing uitgeprobeerd. In een eerste codering wordt de oplossing voorgesteld als een permutatie van de n taken. Elk chromosoom in de populatie bevat dus een mogelijke volgorde. De volgorden in de initiële populatie worden door toeval gekozen op twee na. Het eerste element bevat de betere van de EDD of de SWPT volgorde; het tweede element is de volgorde op basis van de AU regel. Bij elke evaluatie wordt zo'n volgorde verbeterd met een strikt descent proces, gebaseerd op het omwisselen van naast elkaar gelegen taken. Als crossover-operator wordt PMX gebruikt waarbij het snijpunt door toeval gekozen wordt en de lengte van de deelvolgorde, die uitgewisseld wordt, een invoerparameter is. Bij mutatie worden twee taken in naast elkaar gelegen posities omgewisseld.
De tweede codering is binair, per taak is een bit voorzien die aangeeft of de taak op tijd of te laat verondersteld wordt. De verzameling van op tijd veronderstelde taken wordt realistisch gemaakt door een aantal hiervan bij de te late taken te voegen op basis van de EDD volgorde. Dan kunnen de twee verzamelingen omgevormd worden naar

een volgorde met behulp van een heuristiek. Eerst worden de vroege taken in EDD volgorde geplaatst. Dan worden, waar mogelijk, te late taken tussen deze vroege taken geplaatst waarbij er opgelet wordt deze taken niet te laten eindigen voor hun levertijd. Vervolgens wordt de rest van de taken op het einde van de volgorde toegevoegd in SWPT volgorde. De resulterende volgorde wordt tenslotte onderworpen aan een strikt descent proces, gebaseerd op het omwisselen van naast elkaar gelegen taken. Bij crossover worden een aantal stukjes tussen twee chromosomen omgewisseld en bij mutatie wordt één of andere bit van waarde veranderd.

Rekenresultaten.

Om tot een vergelijking van de verschillende methodes te komen, werden twee reeksen van elk 125 problemen gegenereerd. De eerste reeks bevat problemen met $n = 20$ jobs, bij de tweede is n gelijk aan 40. Voor elke taak i is een bewerkingstijd p_i gegenereerd uit de uniforme distributie $[1, 100]$ en een gewicht w_i uit de uniforme distributie $[1, 10]$. De levertijd d_i wordt genomen uit de uniforme distributie $[P(1 - \tau - \rho/2), P(1 - \tau + \rho/2)]$ waarbij P gelijk is aan de totale bewerkingstijd $(\sum_{i=1}^{n} p_i)$. τ en ρ zijn twee parameters die de moeilijkheidsgraad van een probleem bepalen. τ is de gemiddelde vertragingsfactor $(1 - \bar{d}/P)$ en ρ de gemiddelde spreiding van de levertijden $((\max(d_i) - \min(d_i))/P)$. Elk van deze twee parameters heeft de waarde 0.2, 0.4, 0.6, 0.8 of 1.0 gekregen zodat er in het totaal 25 mogelijke combinaties zijn. Voor elke combinatie van ρ en τ zijn 5 problemen gegenereerd.
Door toepassing van het branch and bound algoritme van Potts en Van Wassenhove [17] werden de optimale oplossingswaarden berekend. De gemiddelde uitvoeringstijd per probleem voor de reeks met $n = 20$ is gelijk aan 24 msec. Voor de reeks met $n = 40$ bedraagt dit ongeveer 6 sec; er blijven hier echter 4 problemen onopgelost omdat meer dan 1 min rekentijd nodig is. Uit de resultaten van het branch and bound algoritme blijkt dat problemen met een τ waarde ongeveer gelijk aan 0.6 de moeilijkste zijn.

In tabel 1 worden resultaten van volgende heuristieken weergegeven.

AU: de AU regel;

DSN: descent met nul-wisselingen; stopcriterium: L iteratieniveaus;

SA: simulated annealing: $K_0 = 0.99$, $K_f = 0.001$, L iteratieniveaus, met convex dalende kansfunctie;

TA: threshold accepting: $T_0 = 0.1F(x_1)$, $T_f = 0.0001F(x_1)$, L iteratieniveaus, met convex dalende drempelfunctie;

TS: tabu search: lengte van de taboe lijst is 7, nL iteraties;

GAP: genetisch algoritme met permutatie voorstelling: populatiegrootte $= n$, $L/5$ iteraties, geen schaling; PMX: één sectie met lengte $n/5$ met crossoverkans $= 0.8$; mutatiekans $= 0.1$.

GAB: genetisch algoritme met binaire voorstelling: populatiegrootte $= n$, L iteraties, waarbij voortijdig gestopt wordt wanneer geen verbetering optreedt gedurende $L/2$ iteraties; lineaire schaling van fitheidswaarden met $C_{mult} = 2.0$; crossover: twee secties met lengte $n/5$ met crossoverkans $= 1.0$; mutatiekans $= 0.001$.

De vergelijking is gebaseerd op het aantal maal dat de desbetreffende methode de optimale oplossing vindt (NO), de gemiddelde procentuele afwijking (ARPD), en de maximale procentuele afwijking (MRPD). Ook wordt de gemiddelde uitvoeringstijd (ACT) in milliseconden op een HP9000-G50 voor één probleem weergegeven.

Tabel 1. Resultaten voor 20-job en 40-job problemen

		ARPD			MRPD			NO			ACT		
n	L	10	30	50	10	30	50	10	30	50	10	30	50
20	AU		16.24			220.58			23			4	
	DSN	0.29	0.29	0.29	9.47	9.47	9.47	101	101	101	16	32	56
	SA	0.05	0.02	0.05	3.90	2.20	3.90	120	123	121	24	48	80
	TA	0.08	0.03	0.05	3.90	2.20	3.90	116	122	123	16	40	64
	TS	0.07	0.04	0.00	2.20	1.88	0.35	111	120	123	24	56	96
	GAP	1.56	0.73	0.13	46.74	29.78	6.93	93	107	115	24	48	72
	GAB	0.05	0.03	0.03	1.68	1.68	1.68	115	118	118	16	40	48
40	AU		12.78			169.18			24			8	
	DSN	0.63	0.61	0.61	19.27	19.27	19.27	76	79	79	72	216	360
	SA	0.86	0.23	0.17	41.65	8.72	6.08	79	98	100	120	320	512
	TA	0.39	0.24	0.21	9.87	8.72	8.72	84	97	101	104	296	480
	TS	0.24	0.15	0.14	6.70	6.70	6.70	83	97	101	120	320	520
	GAP	3.20	1.50	1.00	42.03	27.59	20.27	53	64	65	128	296	448
	GAB	0.06	0.02	0.02	1.21	0.64	0.64	87	102	105	104	256	384

De eenvoudige AU-prioriteitsregel kan alleen relatief eenvoudige problemen (bijv. met $\tau = 0.2$) exact oplossen. Voor het overgrote deel wordt een vrij slechte oplossing gevonden wat in een grote gemiddelde afwijking van meer dan 10 % resulteert. Het enige positieve is dat voor het toepassen van deze regel weinig rekentijd nodig is.
Het uitvoeren van nul-wisselingen in DSN is niet voldoende diversifiërend om niet in een lokaal optimum vast te raken. Wanneer meer iteratieniveaus uitgevoerd worden, verbetert de kwaliteit nagenoeg niet.
Voor de drie omgevingszoekheuristieken zorgt de verhoging van L van 10 naar 30 wel voor een gevoelige verbetering. Het aantal exacte oplossingen bijvoorbeeld, stijgt bij $n = 40$ met meer dan 10 in de drie gevallen. Bij $n = 20$ krijgen we voor $L = 50$ het eigenaardige effect dat de kwaliteit van de oplossing verslechtert bij SA en TA. Dit heeft te maken met het feit dat bij $L = 50$ de aanvaardingskansen en de drempels andere waarden hebben dan bij $L = 30$. Daardoor kan het zoekproces een "verkeerde" richting uitgestuurd worden. Van deze drie zoekheuristieken presteert TS het beste maar deze heuristiek heeft ook de meeste rekentijd nodig.
Globaal gezien geeft GAB de beste resultaten: een klein gemiddelde en maximale afwijking, het grootste aantal exacte oplossingen (bij $n = 40$), en de gemiddelde rekentijd is kleiner dan bij de andere methodes. Voornamelijk de kleine maximale afwijking wijst erop dat alle problemen vrij behoorlijk opgelost worden en dat dit genetisch algoritme dus een robuuste oplossingsmethode voor het gewogen levertijdoverschrijdingsprobleem is.
De resultaten van het andere genetische algoritme (GAP) zijn echter slechter dan die van de vier zoekmethodes. Het belangrijkste verschil tussen de genetische algoritmen is de manier waarop de oplossing gecodeerd wordt in een chromosoom. Bij GAB met een binaire codering kan de crossover operator goede elementen uit twee oplossingen samenbrengen

zodat een chromosoom met een betere waarde ontstaat. De PMX operator in GAP slaagt hier duidelijk minder in.

8. Conclusies.

In deze tekst werden verschillende nieuwe zoektechnieken kort besproken en toegepast op een klassiek machinevolgordeprobleem, het *minimaliseren van de totale gewogen levertijdoverschrijding.*
Met behulp van de AU-prioriteitsregel kan snel een oplossing berekend worden maar in de meeste gevallen is die niet van een goede kwaliteit. De resultaten verkregen met een descent algoritme, kunnen door de drie zoekheuristieken gevoelig verbeterd worden. Tabu search scoort hierbij het beste. Uit de resultaten kan men niet concluderen dat threshold accepting beter zou zijn dan simulated annealing. Het genetisch algoritme blijkt zeer goed te functioneren, waarschijnlijk dankzij een juiste codering voor de voorstelling van een oplossing.
Meer algemeen kan men stellen dat deze meta-heuristieken niet zomaar direct leiden naar een efficiënte oplossingsprocedure. Bij de omgevingszoekheuristieken moet een goede omgeving gedefinieerd worden, waarbij best enkele specifieke elementen van het op te lossen probleem in rekening gebracht worden. Voor een genetisch algoritme is de juiste codering van een oplossing uiterst belangrijk. Ook hier kan weer beroep gedaan worden op specifieke voorkennis die men omtrent het op te lossen probleem heeft.

Referenties.

1. H.A.J. Crauwels, C.N. Potts, L.N. Van Wassenhove, 1994. Local search heuristics for single machine tardiness sequencing, unpublished report.

2. L. Davis, 1991. *Handbook of Genetic Algorithms,* Van Nostrand Reinhold, New York.

3. G. Dueck and T. Scheuer, 1990. Threshold Accepting: A General Purpose Optimization Algorithm Appearing Superior to Simulated Annealing, *Journal of Computational Physics 90,* 161–175.

4. R.W. Eglese, 1990. Simulated annealing: a tool for operational research, *European Journal of Operations Research 46,* 271–281.

5. Ernst & Young, 1994. Finite Schedulers: State-of-the-Market.

6. F. Glover, 1986. Future paths for integer programming and links to artificial intelligence, *Computer. Oper. Res. 13,* 533–549.

7. F. Glover, E. Taillard and D. de Werra, 1993. A user's guide to tabu search, *Annals of Operations Research 41,* 3–28.

8. D.E. Goldberg, 1989. *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Reading, MA.

9. E. Goldratt, 1990. *Theory of Constraints,* North River Press.

10. J.H. Holland, 1975. *Adaptation in Natural and Artificial Systems,* Ann Arbor: The University of Michigan Press.

11. D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, 1989. Optimization by simulated annealing: an experimental evaluation; part I, graph partitioning, *Operations Research 37,* 865–892.

12. D.S. Johnson, C.R. Aragon, L.A. McGeoch and C. Schevon, 1991. Optimization by simulated annealing: an experimental evaluation; part II, graph coloring and number partitioning, *Operations Research 39,* 378–405.

13. A. Kirkpatrick, C.D. Getlatt Jr. and M.P. Vechi, 1983. Optimization by simulated annealing, *Science 220,* 671–680.

14. C. Koulamas, S.R. Antony and R. Jaen, 1994. A Survey of Simulated Annealing Applications to Operations Research Problems, *Omega 22, No. 1,* 41–56.

15. T.E. Morton, R.M. Rachamadugu and A. Vepsalainen, 1984. Accurate Myopic Heuristics for Tardiness Scheduling, GSIA Working Paper No. 36-83-84, Carnegie-Mellon University, PA.

16. C.N. Potts and L.N. Van Wassenhove, 1982. A Decomposition Algorithm for the Single Machine Total Tardiness Problem, *Operations Research Letters 1,* 177–181.

17. C.N. Potts and L.N. Van Wassenhove, 1985. A Branch and Bound Algorithm for the Total Weighted Tardiness Problem, *Operations Research 33,* 363–377.

18. C.N. Potts and L.N. Van Wassenhove, 1991. Single Machine Tardiness Sequencing Heuristics, *IIE Transactions 23,* 346–354.

19. P.J.M. Van Laarhoven and E.H.L. Aarts, 1987. *Simulated Annealing: Theory and Applications,* D. Reidel, Dordrecht.

De auteurs wensen hierbij prof. C. Potts te danken voor zijn ideeën en constructieve opmerkingen bij het uitvoeren van deze studie.

Ontvangen: 06-01-1995
Geaccepteerd: 16-01-1997