LØSNING AF OPENSHOP OG FLOWSHOP PROBLEMER

Susanne Hjorth Tønder Rasmussen

LYNGBY 2001
EKSAMENSPROJEKT
NR. 00/00

IMM

Trykt af IMM, DTU

Forord

Denne rapport er skrevet som afsluttende projekt i ingeniørstudiet for opnåelsen af civilingeniørgraden fra Danmarks Tekniske Universitet. Projektet er udført på Institut for Matematisk Modellering (IMM) med professor Jens Clausen som vejleder.

Projektet er udført for at få et større overblik over hvilke metoder der findes til løsning af openshop- og flowshop-problemer. Derudover bliver der i rapporten kommet ind på hvor lette eller svære programmerne er at implementere og hvor gode de er.

Jeg vil gerne benytte lejligheden til at takke Jens Clausen for hans hjælp, opmuntring, og konstruktive kritik. Desuden vil jeg gerne takke min familie for forståelse og opbakning i perioder med stress og sygdom.

Susanne Rasmussen, IMM, 10 september 2001.

Abstract

This note describes a number of different methods, which alone or combined can be used to solve the open-shop and flow-shop problems. The methods are presented and examples solved in order to compare the degree of difficulty in implementation, the accuracy of the solutions and the needed computational time for the different methods. In most cases the computational time needed in the programs developed, could be reduced by a *computer-wizard*, but since everything was developed by me(an average programmer), the relative times should be accurate enough to show if one method is better or worse than another. Further more the fact that an average programmer developed the programs should bring any implementational problems in the methods to light.

The methods implemented are H1, Simulated Annealing, HFC and H1 adapted to flowshopproblems. The algorithms H2, Branch & Bound and Tabusearch are examined but not implemented.

HFC is found to be the best of those algorithms implemented for solving large flowshopproblems and H1 the best for solving large openshopproblems. This was in both cases due to the speed of the algorithms, the relative accuracy and the ease of implementing them.

KEYWORDS: master's thesis, open-shop, flow-shop, Simulated Annealing, HFC, H1, H2.

Indhold

1 Indledning **11**

1.1 Baggrund 11

1.2 Afgrænsning 12

1.3 Problemformulering 13

2 Shop-scheduling **15**

2.1 Matematisk notation 15

2.2 Matematisk formulering 16

2.3 Open-shop modellen 17

2.3.1 Beskrivelse 17

2.3.2 Matematisk 18

2.3.3 Eksempel 19

2.4 Flow-shop modellen 21

2.4.1 Beskrivelse 21

2.4.2 Matematisk 21

2.4.3 Eksempel 21

2.5 (Job-shop modellen) 24

3 Løsningsmetoder **25**
3.1 Branch & Bound 25
3.2 Heuristikker 30
3.2.1 Metaheuristikker 30
3.2.2 Andre heuristikker 37

4 Løsningsmetoder - openshop **47**
4.1 Simuleret Udglødning 47

5 Løsningsmetoder - flow-shop **57**
5.1 HFC 57

6 Programmer **63**

7 Konklusion **69**

A Notation **71**

B Openshop programmer **73**
B.1 H1 73
B.2 Simuleret Udgloedning 84

C Flowshop programmer **137**
C.1 H1 137
C.2 HFC 147

Tabeller

2.1 Operationernes numre. 16

2.2 Procestider for et 4x4 oss-problem 19

2.3 Starttidspunkt for operationerne i oss-problemet 20

2.4 Procestider for et 4x4 fss-problem 22

2.5 Angiver hvilken maskine, der skal lave hvilken operation i fss-problemet 22

2.6 Starttidspunkt for operationerne i fss-problemet 24

3.1 Simpel LB 26

3.2 Hoveder og haler for operationerne på maskine 3 i et 4x4 fss-problem 27

3.3 Procestider i et 4x4 oss-problem 39

3.4 Resterende maskintid. 39

3.5 Startværdi for hovederne for 4x4 oss-problem. 40

3.6 Hovederne efter 1. iteration. 40

3.7 Hovederne efter 2. iteration. 40

3.8 Ændringer af hovederne for operationerne, når den statiske H1-heuristik gentages. 41

3.9 Hoveder efter 1. løsning vha. H1. 41

3.10 4x4 oss-problem, som ønskes løst vha. H2. 43

4.1 Procestider for et 4x4 oss-problem 50
4.2 Resterende tid på andre maskiner 51

5.1 Procestider for et 4x4 fss-problem 60
5.2 Angiver hvilken maskine, der skal lave hvilken operation i fss-problemet 60
5.3 Tabel til brug v. opbygning af H-matrix 61
5.4 H matrix for HFC-heuristikken 62

6.1 Resultater v. løsning af openshop-problemer 65
6.2 Resultater v. løsning af flowshop-problemer 66

Figurer

2.1 Eksempel på et Gantt-diagram. 18
2.2 Gantt-diagram af løsningen til oss-problemet. 20
2.3 Gantt-diagram af løsningen til fss-problemet. 24

3.1 Eksempel på en disjunktiv graf 28
3.2 Eksempel på anvendelse af Nabolag 2 32
3.3 Gantt-diagram af den opnåede tidsplan efter 1. iteration med H1. 42
3.4 Løsning efter H2's 1. fase. 43
3.5 Samme problem efter H2's 2. fase med GCP(P) 45

4.1 ønsker at vende kanterne (i, j), (FJ(j, v), j) og (i, EJ(i, v)) . 48
4.2 Vendt kanterne (i, j), (FJ(j, v),j) og (i, EJ(i, v)) 48
4.3 Gantt-diagram af startløsningen til oss-problemet. 51
4.4 Graf af startløsningen til oss-problemet med operationernes hoveder og haler. 53
4.5 Graf af løsningen med to vendte kanter. 54

5.1 Gantt-diagram af 1. løsning v. HFC 60

Kapitel 1

Indledning

Formålet med dette projekt er at få et overblik over hvilke løsningsmetoder der findes for openshop og flowshop scheduling problemer og over hvor gode disse er når de skal implementeres, dvs. ikke kun hvor god løsningen er, men også hvor nem den er at lave som program og hvor lang tid computeren er om at beregne løsninger på problemer af forskellig størrelse.

Rapporten er bygget op på følgende måde. I Kapitel 1 bliver det overordnede problem præsenteret og afgrænset. I Kapitel 2 defineres openshop- og flowshop-problemerne og der gives eksempler på problemerne. I Kapitel 3 gennemgås nogle generelle løsningsmetoder, disse bliver benyttet i Kapitel 4 og 5, hvor der også beskrives metoder specielt udviklet til et af problemerne. Nogle af metoderne bliver derefter implementeret og afprøvet i Kapitel 6. Endelig er der en samlet konklusion i Kapitel 7.

1.1 Baggrund

Shopscheduling er et område inden for produktionsplanlægningen. Ved shopscheduling forstås et problem, hvor et antal opgaver skal udføres vha. de samme ressourcer på den kortest mulige tid, billigst muligt eller med et andet optimeringskriterium.

Dette sker ofte i dagligdagen, når det opleves at to personer nærmest slås om badeværelset, om den samme avis osv. og til sidst *enes* om hvem der

skal hvad først. Hvis der nu var flere opgaver, flere ressourcer, der skal benyttes af begge bliver det sværere at blive enige, hvis det hele oven i købet skal ske på den kortest mulige tid(som f.eks. mandag morgen) er dette et shopscheduling problem.

Nu bliver shopscheduling jo sjældent benyttet til at skemalægge mandag morgens hasten fra sengen og ud af døren. Shopscheduling bliver mest benyttet af erhvervslivet til f.eks. at planlægge produktion af flere næsten ens varemærker, da disse både benytter de samme maskiner, de samme råvarer og det samme tidsrum.

I resten af rapporten vil såkaldte prototypeproblemer bliver behandlet, dvs. at de virksomhedsspecifikke detaljer og specialbetingelser er fjernet, så der kun er de grundlæggende problemer tilbage.

1.2 Afgrænsning

Den objektfunktion, der normalt betragtes inden for shop-scheduling, skal minimere den tid, det tager fra den første operation påbegyndes til den sidste afsluttes. Der vil her dog ikke blive set på det mere generelle tilfælde, hvor der er opstillet deadlines d_i, som skal overskrides så lidt som muligt. Hvis alle $d_i = 0$ fås den før nævnte objektfunktion.

En vigtig begrænsning for problemet er, at der kun kan behandles ét job på én maskine på ét tidspunkt. Ligeledes kan ét job kun behandles af én maskine på ét tidspunkt. Dette betyder at to operationer, der tilhører det samme job ikke kan blive udført af to forskellige maskiner på det samme tidspunkt.

En anden vigtig begrænsning er at operationer er sammenhængende og ikke kan afbrydes undervejs. Dermed menes at man ikke kan starte på operation A, afbryde den, udføre operation B, for derefter at færdiggøre operation A.

Endvidere besøger hvert job hver maskine præcist én gang, dette sker evt. ved at oprette pseudo operationer med procestid $p_i = 0$. Der vil ikke her blive set på tilfælde, hvor job kan tænkes at besøge en maskine mere end en gang, da det meste af den forskning der er sket på området er sket under den antagelse at hver maskine besøges netop én gang.

Udover dette er der også andre faktorer, som ikke bliver taget med i de fleste modeller.

Den tid det tager for en maskine at blive omstillet til et nyt job kan både afhænge af hvilket job, som er før omstillingen og hvilket der kommer efter. Ofte er denne omstillingstid regnet indeholdt i procestiden, men dette rummer en antagelse om at omstillingstiden er uafhængig af hvilken operation, der lige er blevet udført.

Når en operation på et job er blevet udført er der ofte ventetid før jobbet kan komme til på den næste maskine, dette medfører at det er nødvendigt med mellemlagre, hvilket ikke altid er muligt, f.eks. på grund af pladsmangel. Normalt antages det at sådanne mellemlagre er store nok, men i virkeligheden er dette sjældent tilfældet. Det kan derfor være nødvendigt at indføre begrænsninger på størrelsen af disse lagre.

Et andet spørgsmål, som kan stilles er om jobbet i det hele taget kan tåle at vente mellem to operationer. Dette kan f.eks. være tilfældet med kemiske blandinger, som ikke kan tåle at blive afbrudt midt i forarbejdningsprocessen. Det kan derfor være nødvendigt at indføre begrænsninger på ventetiden.

Med disse begrænsninger kan følgende problemtyper stilles op OpenShop-Scheduling (OSS), JobShopSchedulig (JSS) og FlowShopScheduling (FSS). Disse problemer er ens bortset fra de bånd der er mht. operationernes rækkefølge dvs. jobbenes opbygning.

1.3 Problemformulering

Det hovedområde, som ønskes belyst, er hvilke metoder, der findes til løsning af openshop- og flowshop-problemer. Disse metoder ønskes nærmere beskrevet og om muligt implementeret. Desuden ønskes det udfra de opnåede resultater bedømt, hvilken løsningsmetode, der er den bedste til løsning af hvert problem.

Kapitel 2

Shop-scheduling

Shopscheduling er kort fortalt planlægning af hvordan et antal opgaver, der alle består af flere delopgaver, der skal bruge de samme ressourcer kan udføres på kortest mulig tid.

2.1 Matematisk notation

I resten af rapporten vil jeg anvende følgende generelle notation. De ressourcer, der benyttes vil blive kaldt for *maskiner* og antallet af disse betegnes med $m \in M$. Begrebet opgaver vil blive kaldt for *jobs* og hver delopgave for en *operation*. Antallet af jobs betegnes med $n \in N$ og antallet af operationer med $o \in O$. Her vil o_{ij} betegne den specifikke operation, der tilhører job i og udføres på maskine j. Det antal operationer, der udføres på maskine j kaldes $m_j \in M_j$ og den maskine, der skal lave operation i betegnes μ_i. Den tid det tager at udføre operation i er denne operations *procestid*, som betegnes med p_i. Starttidspunktet for denne operation angives ved t_i og forfaldstidspunktet (deadline) ved d_i. Endelig vil objektfunktionsværdien for den bedste kendte løsning blive betegnet C_{max} og den optimale løsnings objektfunktionsværdi vil blive angivet ved C^*_{max}. Der vil senere, når det bliver nødvendigt, blive introduceret mere notation. Den anvendte notation kan også ses i Appendiks A.

I eksemplerne vil operationerne blive nummereret som vist i tabel 2.1.

job \ maskine	1	2	3	4
1	1	2	3	4
2	5	6	7	8
3	9	10	11	12
4	13	14	15	16

Tabel 2.1: Operationernes numre.

2.2 Matematisk formulering

For alle shop-problemerne gælder det generelt at:

$$Min \quad C_{max} \tag{2.1}$$

$$C_{Max} \geq t_i + p_i \qquad \forall i \in O \tag{2.2}$$

$$t_i \geq 0 \qquad \forall i \in O \tag{2.3}$$

$$t_j \geq t_i + p_i \quad eller \quad t_i \geq t_j + p_j \qquad \forall i \neq j; i,j \in N_k \tag{2.4}$$

$$t_j \geq t_i + p_i \quad eller \quad t_i \geq t_j + p_j \qquad \forall i \neq j; i,j \in M_k \tag{2.5}$$

Her viser den første ligning at der er tale om et minimerings problem, den anden at objektfunktionsværdien er lig med tidspunktet for den sidste operations afslutning. Den tredje begrænsning kommer af at alle starttidspunkter skal være større end nul og de sidste to sørger for at operationer, der hører til samme job eller som skal udføres på den samme maskine ikke overlapper, dvs. at de ikke bliver lavet på samme tid.

I job- og flow-shop problemerne kan en del af de her nævnte begrænsninger fjernes, da operationernes rækkefølge indenfor et givent job er kendt på forhånd. Der kan laves følgende ændringer:

I formel (2.2) er det kun nødvendigt at medtage en begrænsning for den sidste operation i hvert job, fordi det nu vides hvilken operation, der er den sidste.

Også formel (2.4) kan skiftes ud, idet der her i stedet skal være en lang række uligheder, der beskriver præcedensforholdene mellem operationer, der er del af samme job. For eksempel vil et job, hvor operation a skal laves før operation b og operation b før operation c, give anledning til følgende begrænsninger:

$$\begin{aligned} t_b &\geq t_a + p_a \\ t_c &\geq t_b + p_b \end{aligned}$$

Den nævnte problemformulering kan ved hjælp af nogle simple omskrivningsregler laves om til en Lineær Programmerings model (LP-model), idet der gøres brug af et antal binære variable y_{ij}, som er lig 1 hvis operation i laves før operation j og 0 ellers. Med disse variable kan formel (2.4) og (2.5) skrives om som følger:

$$t_i + p_i \leq t_j + L(1 - y_{ij}) \tag{2.6}$$
$$t_j + p_j \leq t_i + L \cdot y_{ij} \tag{2.7}$$
$$y_{ij} \in \{0, 1\} \tag{2.8}$$

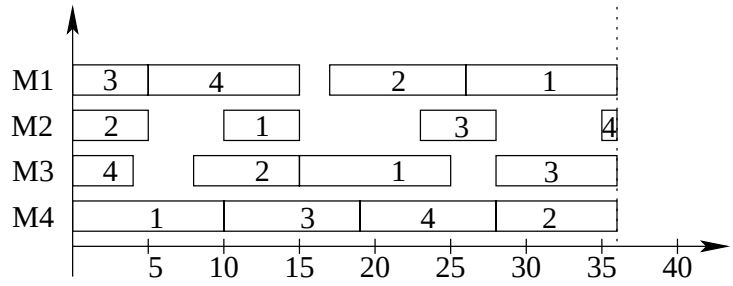
hvor L er en passende stor konstant. Hvis det problem der behandles er et oss-problem, skal både formel (2.4) og (2.5) udskiftes, mens kun formel (2.5) udskiftes ved jss- og fss-problemer.

Da løsningerne til de her nævnte problemer med fordel kan vises i et såkaldt Gantt-diagram, skal dette kort beskrives her. Gantt-diagrammet er en figur, der består af to akser, en vandret akse, som markerer tiden t, og en lodret akse, som viser hvilken maskine, der er tale om. Mellem de to akser er optegnet et antal kasser, hver kasse symboliserer en operation, kassens længde svarer til operationens procestid og kassens nummer viser hvilket job operationen tilhører. Et Gantt-diagram kan f.eks. se ud som i figur 2.1.

2.3 Open-shop modellen

2.3.1 Beskrivelse

Ved Open-shop forstås en type af shop-scheduling, hvor der ikke er nogle bånd der binder mht. den rækkefølge, som operationerne i et job udføres i. Dette er f.eks. tilfældet ved distribution af beskeder fra en satellit, idet der her er tale om en proces, hvor en række afsendingsstationer på jorden



Figur 2.1: Eksempel på et Gantt-diagram.

alle sender beskeder til en række modtagestationer på jorden via en satellit. Denne satellit fungerer nu som en sorteringscentral, idet beskederne modtages, sorteres og sendes til en modtagestation. Da hver modtagekanal på satellitten kun kan modtage en besked ad gangen og hver sendekanal kun sende en ad gangen og man ønsker at den sidste besked modtages så tidligt som muligt på jorden kan problemet formuleres som et oss-problem: Beskederne er operationer, beskeder fra den samme modtagekanal tilhører samme job og afsendingskanalerne er "maskinerne". Da beskederne ikke skal sendes i en bestemt rækkefølge er dette et oss-problem.

2.3.2 Matematisk

Som nævnt kan dette f.eks. formuleres som en LP-model, som den følgende.

$$
\begin{array}{rcl}
Min \quad C_{max} & & \\
C_{Max} & \geq & t_i + p_i \\
t_i & \geq & 0 \\
t_i + p_i & \leq & t_j + L \cdot (1 - y_{ij}) \\
t_j + p_j & \leq & t_i + L \cdot y_{ij} \\
y_{ij} & \in & \{0, 1\}
\end{array}
$$

2.3.3 Eksempel

I det følgende opstilles et oss-problem, som her løses ved hjælp af CPLEX. CPLEX er et program, som kan benyttes til løsning af LP-problemer og blandede heltalsproblemer. CPLEX benytter, som standard, Simplexmetoden til løsning af LP-problemer og Branch & Bound (Se afsnit 3.1) til løsning af heltalsproblemerne. Efter sigende er CPLEX et af de bedste programmer på markedet til dette formål.

Det nævnte problem vil blive benyttet igen senere til at illustrere andre metoder til løsning af open-shop problemer.

job \ maskine	1	2	3	4
1	10	5	10	10
2	9	5	7	8
3	5	5	8	9
4	10	1	4	9

Tabel 2.2: Procestider for et 4x4 oss-problem

Det Lineære Programmeringsproblem, som skal løses (f.eks. med CPLEX) bliver:

$$
\begin{aligned}
Min \quad & C_{Max} & & \\
& t_1 - C_{Max} & \leq & -10 \\
& t_2 - C_{Max} & \leq & -5 \\
& t_3 - C_{Max} & \leq & -10 \\
& \vdots & & \vdots \\
& t_i - C_{Max} & \leq & -p_i \\
& \vdots & & \vdots \\
& t_{15} - C_{Max} & \leq & -4 \\
& t_{16} - C_{Max} & \leq & -9 \\
\\
t_1 - t_2 + 10000 \cdot y_{1,2} & & \leq & 9990 \\
t_2 - t_1 - 10000 \cdot y_{1,2} & & \leq & -5 \\
t_1 - t_3 + 10000 \cdot y_{1,3} & & \leq & 9990
\end{aligned}
$$

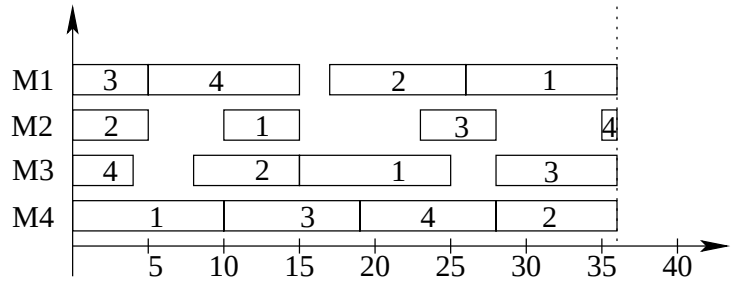
$$
\begin{aligned}
t_3 - t_1 - 10000 \cdot y_{1,3} &\leq -10 \\
&\vdots \quad \vdots \\
t_i - t_j + 10000 \cdot y_{i,j} &\leq 10000 - p_i \\
t_j - t_i - 10000 \cdot y_{i,j} &\leq -p_j \\
&\vdots \quad \vdots \\
t_{15} - t_{16} + 10000 \cdot y_{15,16} &\leq 9996 \\
t_{16} - t_{15} - 10000 \cdot y_{15,16} &\leq -9 \\
\\
t_i &\geq 0 \\
y_{i,j} &\in \{0,1\}
\end{aligned}
$$

Ved brug af CPLEX opnås en løsning med en objektfunktionsværdi på 36 og det tog programmet 0.37s at finde løsningen til:

job \ maskine	1	2	3	4
1	26	10	15	0
2	17	0	8	28
3	0	23	28	10
4	5	35	0	19

Tabel 2.3: Starttidspunkt for operationerne i oss-problemet

Løsningen kan også ses i figur 2.2.



Figur 2.2: Gantt-diagram af løsningen til oss-problemet.

2.4 Flow-shop modellen

2.4.1 Beskrivelse

I modsætning til det netop beskrevne problem er Flow Shop Scheduling det problem, der har de fleste begrænsninger, her er operationernes rækkefølge fastlagt på forhånd, men ikke nok med det, alle job består af de samme operationer i den samme rækkefølge. Dette er tilfældet i et almindeligt produktionssystem, hvor maskinerne er placeret i den rækkefølge, som operationerne skal udføres i, i jobbene. Et eksempel på dette kunne være produktionen af vandhaner, hvor en hane skal støbes, slibes, afprøves, lakeres, pudses og pakkes. Dette kan kun gøres i en rækkefølge. Da der på de samme "maskiner"godt kan laves flere forskellige typer vandhaner og da procestiden på de forskellige maskiner varierer fra hane til hane er dette et flowshopproblem.

2.4.2 Matematisk

Som nævnt kan også flow-shop problemerne formuleres som LP-modeller, som tidligere vist. Dog skal formel (2.5) ved fss-problemer ændres, så følgende model opnås.

$$
\begin{array}{rcl}
Min \quad C_{max} & & \\
C_{Max} \geq t_i + p_i & & \forall i \in O \\
t_i \geq 0 & & \forall i \in O \\
t_j \geq t_i + p_i \quad eller \quad t_i \geq t_j + p_j & & \forall i \neq j; i,j \in N_k \\
t_i + p_i & \leq & t_j + L \cdot (1 - y_{ij}) \\
t_j + p_j & \leq & t_i + L \cdot y_{ij} \\
y_{ij} & \in & \{0,1\}
\end{array}
$$

2.4.3 Eksempel

I det følgende opstilles et fss-problem, som her løses ved hjælp af CPLEX. Som det kan ses ligner problemet umiddelbart det der blev løst under shop-scheduling og procestiderne er da også de samme. Forskellen er som nævnt

at operationernes rækkefølge her er fast og ikke valgfri som ved oss. Det nævnte problem vil blive benyttet igen senere til at illustrere andre metoder til løsning af flow-shop problemer.

job \ maskine	1	2	3	4
1	10	5	10	10
2	9	5	7	8
3	5	5	8	9
4	10	1	4	9

Tabel 2.4: Procestider for et 4x4 fss-problem

job \ operation nr.	1	2	3	4
1	1	2	3	4
2	1	2	3	4
3	1	2	3	4
4	1	2	3	4

Tabel 2.5: Angiver hvilken maskine, der skal lave hvilken operation i fss-problemet

Det Lineære Programmeringsproblem, som skal løses (f.eks. med CPLEX) bliver:

$$
\begin{aligned}
Min \quad & C_{Max} & & \\
t_4 - C_{Max} & \leq & -10 \\
t_8 - C_{Max} & \leq & -8 \\
t_12 - C_{Max} & \leq & -9 \\
t_16 - C_{Max} & \leq & -9 \\
& & \\
t_1 - t_2 & \leq & -10 \\
t_2 - t_3 & \leq & -5 \\
\vdots & & \vdots \\
t_i - t_{i+1} & \leq & -p_i \\
\vdots & & \vdots \\
t_14 - t_15 & \leq & -1
\end{aligned}
$$

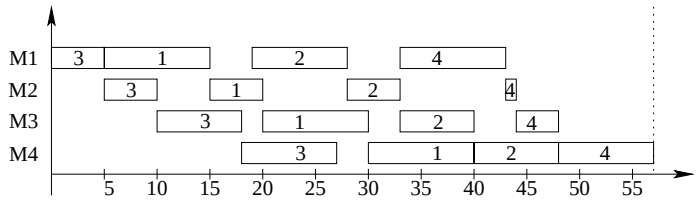
$$
\begin{aligned}
t_15 - t_16 &\leq -4 \\
\\
t_1 - t_5 + 10000 \cdot y_{1,5} &\leq 9990 \\
t_5 - t_1 - 10000 \cdot y_{1,5} &\leq -9 \\
t_1 - t_9 + 10000 \cdot y_{1,9} &\leq 9990 \\
t_9 - t_1 - 10000 \cdot y_{1,9} &\leq -5 \\
t_1 - t_13 + 10000 \cdot y_{1,13} &\leq 9990 \\
t_13 - t_1 - 10000 \cdot y_{1,13} &\leq -10 \\
t_5 - t_9 + 10000 \cdot y_{5,9} &\leq 9991 \\
t_9 - t_5 - 10000 \cdot y_{5,9} &\leq -5 \\
t_5 - t_13 + 10000 \cdot y_{5,13} &\leq 9991 \\
t_13 - t_5 - 10000 \cdot y_{5,13} &\leq -10 \\
t_9 - t_13 + 10000 \cdot y_{9,13} &\leq 9995 \\
t_13 - t_9 - 10000 \cdot y_{13,9} &\leq -10 \\
\vdots &\quad \vdots \\
t_i - t_j + 10000 \cdot y_{i,j} &\leq 10000 - p_i \\
t_j - t_i - 10000 \cdot y_{i,j} &\leq -p_j \\
\vdots &\quad \vdots \\
t_{12} - t_{16} + 10000 \cdot y_{12,16} &\leq 9991 \\
t_{16} - t_{12} - 10000 \cdot y_{12,16} &\leq -9 \\
\\
t_i &\geq 0 \\
y_{i,j} &\in \{0,1\}
\end{aligned}
$$

Ved brug af CPLEX opnås en løsning med en objektfunktionsværdi på 57 og det tog programmet 0.17s at finde løsningen, som kan ses i tabel 2.6.

Løsningen kan også ses i figur 2.3.

job \ maskine	1	2	3	4
1	5	15	20	30
2	19	28	33	40
3	0	5	10	18
4	33	43	44	48

Tabel 2.6: Starttidspunkt for operationerne i fss-problemet



Figur 2.3: Gantt-diagram af løsningen til fss-problemet.

2.5 (Job-shop modellen)

Her har operationerne i de enkelte jobs en fast rækkefølge, men denne rækkefølge har ingen sammenhæng med operationernes rækkefølge i andre jobs. Dette kan f.eks. være pga. at en ting ikke rent fysiskt kan gøres før en anden ting er lavet. Denne problemtype bliver ikke yderligere behandlet i denne tekst, men mange af de berørte metoder kan udemærket benyttes til at løse job shop scheduling problemer.

Kapitel 3

Løsningsmetoder

3.1 Branch & Bound

Den nok mest benyttede eksakte metode til løsning af heltalsmodeller er Branch & Bound. I det følgende er det antaget at det drejer sig om et minimeringsproblem. Metoden består i korte træk af at der gradvist opstilles et søgetræ ved at opspalte løsningsmængden i mindre og mindre delmængder, indtil de mindste delmængder kun består af en løsning. En øvre grænse (UB) angiver, hvor stor den optimale løsning kan være, denne sættes ofte lig den hidtil bedste fundne løsning. Desuden udregnes for en given delmængde en nedre grænse (LB) for hvor god objektfunktionen kan blive. Hvis denne nedre grænse er dårligere, dvs. større, end den øvre grænse vil den optimale løsning ikke kunne findes i denne delmængde og den del af søgetræet skal derfor ikke undersøges yderligere.

Metodens effektivitet afhænger i høj grad af hvordan de følgende tre beslutningsregler opstilles.

1. Separationsregel.
2. Beregning af nedre grænse.
3. Valg af nyt underproblem

Separationsreglen fortæller hvordan en løsningsmængde skal opdeles i disjunkte delmængder. For det meste opdeles i to delmængder, så der opnås et binært søgetræ.

Den nedre grænse er et udtryk for hvor lille den optimale løsning kan blive. Der er gennem årene gjort mange forsøg på at øge denne grænse, så det hurtigere kan afgøres om det er nødvendigt at undersøge en delmængde yderligere.

En simpel måde at finde en nedre grænse på i et shop-problem er ved at finde den største sum af procestider for job og maskiner dvs.

$$LB = max(max_{j \in N}(\sum_{i \in O, N_i = j} p_i), max_{m \in M}(\sum_{i \in O, M_i = m} p_i)) \tag{3.1}$$

Eksempel 3.1.1

job \ maskine	1	2	3	4	LB_{job}
1	10	5	10	10	35
2	9	5	7	8	29
3	5	5	8	9	27
4	10	1	4	9	24
$LB_{maskine}$	34	16	29	36	

Tabel 3.1: Simpel LB

Som det kan ses af Tabel 3.1 bliver LB her

$$LB = max(LB_{job}, LB_{maskine}) = 36$$

En mere kompliceret metode kaldes Primal Jacksons Preemptive Schedule (PJPS). Denne metode bygger på, at kravet om at operationerne ikke må afbrydes, relakseres. Der defineres en tidsoperator $t = 0$, de operationer, der kan startes på dette tidspunkt, findes, og for disse undersøges det hvilken operation der har den længste hale, dvs. hvilken operation, der er i det job, hvor der resterer mest tid. Den fundne operation skemalægges, men afbrydes, når der findes en operation, der på det tidspunkt har en længere hale.

Tilsvarende findes Dual Jacksons Preemptive Schedule (DJPS), som er den samme metode bortset fra at der startes bagfra, dvs. at den operation der først placeres er den der har det største hoved og denne placeres sidst.

Eksempel 3.1.2 *PJPS med det samme fss-problem*

I dette eksempel benyttes det samme problem, som i eksempel 3.1.1, dog skal det her nævnes at der er tale om et flowshop problem og at dette her betyder at alle jobs først skal behandles på maskine 1, derefter på maskine 2 osv. Hvis man nu f.eks. ser på operationerne på maskine 3 kan man opstille følgende tabel (Tabel 3.2).

job	Hoved r_i	Procestid p_i	Hale q_i
1	15	10	10
2	14	7	8
3	10	8	9
4	11	4	9

Tabel 3.2: Hoveder og haler for operationerne på maskine 3 i et 4x4 fss-problem

Den første operation, der kan blive behandlet er operation (3,3), denne bliver sat i gang til tidspunktet t=10. Tiden kører nu indtil det næste interessante tidspunkt, nemlig t=11, men da operation (3,3) har den længste hale får den lov til at fortsætte. Til t=14 er det samme tilfældet, men til t=15 har operation (3,3) ikke længere den længste hale af de operationer, der kan behandles, det har operation (3,1). Denne operation behandles til den er færdig. Derefter genoptages behandlingen af operation (3,3) og den færdigbehandles. Så behandles operation (3,4) og til sidst operation (3,2). Dette giver en nedre grænse på

$LB = 10 + 5 + 10 + 3 + 4 + 7 + 8 = 47.$

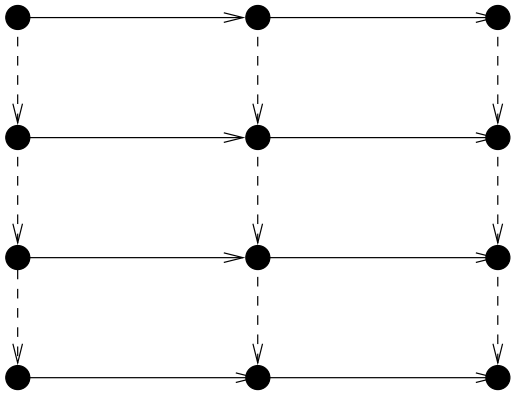
Som det kan ses er den nedre grænse nu blevet øget fra 36 til 47.

Metoden giver ikke en særlig god nedre grænse, men den giver i følge [8] de bedste resultater i forhold til den tid, det tager at benytte den.

Det vigtigste ved valg af nye underproblemer er at disse er strengt mindre end det nuværende problem og at deres foreningsmængde udgør hele løsningsmængden. Hovedideen med Branch & Bound er at opstille præcedensrelationer mellem de forskellige operationer. Dertil kan der iflg. P. Brucker, J. Hurinck, B. Jurisch og B. Wöstmann [1] benyttes en disjunktiv graf $G = (V, D_M \cup D_J)$, hvor

- mængden af knuder V er mængden af operationer og hver knude er kendetegnet ved den tilsvarende operations procestid.
- D_M er mængden af ikke-orienterede kanter, der forbinder operationer, der skal behandles på den samme maskine
- D_J er mængden af kanter der forbinder par af operationer i det samme job.

figur m disjunktiv graf



Figur 3.1: Eksempel på en disjunktiv graf

Skemalægnings problemet kan nu løses ved at orientere de disjunktive kanter. En mængde S af de nu orienterede kanter kaldes et *valg*. Det er klart at S kun er en gyldig løsning hvis alle disjunktive kanter er orienterede og den resulterende graf $G(S) = (V, S)$ er acyklisk, dvs. at der ikke findes kredse i grafen. Hvis S er en gyldig løsning kaldes S et *fuldstændigt valg*. Et fuldstændigt valg giver en mulig løsning ved at definere sluttidspunktet for hver operation som længden af den længste vej i $G(S)$, der ender ved denne operation. Her er længden af en vej lig summen af procestider på alle knuderne på vejen. $C_{max}(S)$ er lig den længste vej i $G(S)$, denne vej kaldes også den kritiske vej. For at løse et shopproblem skal vi finde $G(S)$ således at den kritiske vej bliver mindst mulig.

Dette fører til den kritiske vejs metode. En metode, hvor træet i B&B bliver opdelt ved parvist at ombytte operationer på den kritiske vej. Dette kan iflg. [1] ske ved at finde blokke på den kritiske vej. En blok består her af to eller flere operationer på den kritiske vej der alle skal behandles

på den samme maskine. Desuden er blokken defineret, som værende så stor som mulig. Dette betyder at den operation der kommer umiddelbart før eller efter blokken på den kritiske vej ikke må skulle behandles på den samme maskine. I følge [8] kan man nøjes med at undersøge de disjunktive kanter, der går mellem to operationer i samme blok, hvoraf den ene af disse operationer er enten den første eller den sidste i blokken. Hvis alle mulighederne afprøves vil der enten findes en bedre løsning eller også vil man kunne konstatere at den allerede opnåede løsning er den bedste. Dette vil ganske vist kunne give et temmeligt bredt søgetræ, men er stadig en stor hjælp, da man ellers i princippet skulle prøve at vende alle de disjunktive kanter.

Den kritiske maskines metode: Denne metode er udviklet af J. Carlier og E. Pinson. Første skridt i metoden er at finde den kritiske maskine, dette kan f.eks. gøres ved at finde den maskine, der iflg. PJPS giver den højeste nedre grænse. Udgangspunktet i metoden er også her at anvende en disjunktiv kant til at opsplitte søgetræet. Her findes kanten iflg. [8] ud fra det følgende.

1. Hvis der er ikke-orienterede disjunktive kanter på den kritiske maskine findes den af mængderne E og F, der har den mindste kardinalitet og en ikke-orienteret kant vælges. Her er E mængden af operationer, der kan være input til blokken og F mængden af operationer, der kan være output til blokken. Hvis der er flere mulige kanter, gå til punkt 3.
2. Hvis alle disjunktive kanter er orienterede findes blandt alle mængderne E og F for alle maskinerne den mængde der har den mindste kardinalitet. Der vælges en ikke-orienteret disjunktiv kant mellem to af punkterne i den fundne mængde. hvis der er flere mulige kanter, gå til punkt 3.
3. Hvis der er flere disjunktive kanter at vælge imellem, beregnes disse størrelser:

$$d_{ij} = Max(0, r_i + p_i + q_j - LB) \tag{3.2}$$
$$d_{ji} = Max(0, r_j + p_j + q_i - LB) \tag{3.3}$$
$$a_{ij} = Min(d_{ij}, d_{ji}) \tag{3.4}$$
$$v_{ij} = |d_{ij} - d_{ji}| \tag{3.5}$$
$$\tag{3.6}$$

Den disjunktive kant der har den største v_{ij} vælges, hvis der er flere med samme værdi vælges den der har den største a_{ij}. Udover at få delt træet

yderligere op har metoden også en sidegevinst. Hvis en mængde, det være sig E eller F, har kardinaliteten 0, betyder det at der ikke eksisterer et input/output til mængden, som giver en tilfredsstillende løsning. Knuden kan derfor afskæres fra søgetræet. Hvis en af mængderne har kardinaliteten 1, betyder det at operationen skal vælges som input/output og de relevante kanter kan orienteres.

Med de nævnte beslutningsregler kan Branch & Bound finde den optimale løsning, men ved store problemer bliver søgetræet hurtigt så stort at det tager meget lang tid at finde denne løsning. Derfor er der blevet udviklet en række heuristikker, der relativt hurtigt kan finde en god, men ikke garanteret optimal løsning. Disse heuristikker vil der blive set på i det følgende.

3.2 Heuristikker

3.2.1 Metaheuristikker

Metaheuristikker er en gruppe heuristikker, der modsat andre heuristikker er velegnede til at løse mange *forskellige* problemer. Ideen i en metaheuristik er, at den er en overordnet skabelon for, hvordan løsningsmetoden til et specifikt problem kan se ud. Skabelonen består af en række delelementer og det er disse delelementer, der giver en metaheuristik sit særpræg. Inden for rammerne af denne skabelon kan brugeren selv designe delelementer, så løsningsmetoden tilpasses til det enkelte problem. Dette er en fordel, da metaheuristikkerne kan finde gode løsninger til mange forskellige problemer, men det er også en ulempe, da der i en metaheuristik er mange parametre, som skal indstilles temmelig nøjagtigt hver gang man behandler et ny type problem. I det følgende vil to metaheuristikker, tabusearch og simuleret udglødning blive beskrevet, da disse har vist sig at give bedre løsninger end andre metaheuristikker når de har været anvendt på shop-problemer. I disse metaheuristikker er der nogle fælles elementer, som vil blive beskrevet først.

Nabolag

Et nabolag er den del af løsningsområdet, som betragtes, når man ændrer en løsnings opbygning en lille smule. For et på shop-problem kan en ændring

f.eks. defineres som en ombytning af rækkefølgen af to operationer. Dette kan gøres på mange måder og selvom et godt nabolag ikke garanterer at man finder den optimale løsning, så øger det muligheden for at finde den.

Nabolag 1

En måde bliver beskrevet af Van Larhoven, Aarts og Lenstra [11] og går ud på at de to operationer, der skal ombyttes, begge skal behandles på samme maskine og begge skal ligge på den kritiske vej. Fordelen ved at benytte dette nabolag er at en ændring altid vil give en gyldig løsning. Desuden vil der i dette nabolag kun være medtaget løsninger, som vil kunne medføre forbedringer i forhold til den nuværende løsning. Endelig vil man altid kunne komme til den optimale løsning ved at benytte et endeligt antal af disse ændringer [11].

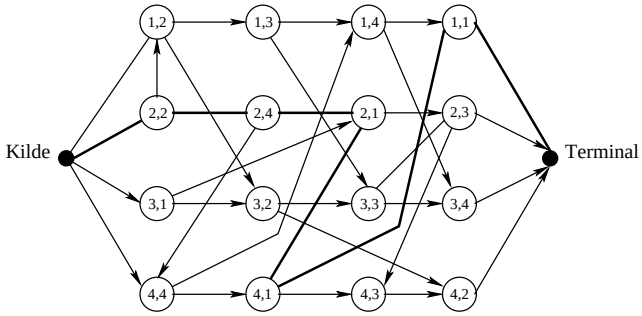
Nabolag 2

Et andet nabolag er blevet nævnt i [9], det opnås ved at bytte to operationer (i, v) og (j, v), der begge ligger på den kritiske vej og skal behandles på den samme maskine, men derudover skal det gælde at en af operationerne skal ligge i starten eller slutningen af en blok, dvs. at enten $EM(j, v)$, dvs. efterfølgeren til operation (j, v) eller $FM(i, v)$, dvs. forgængeren til operation (i, v), ikke ligger på den kritiske vej. Endvidere er der i nabolaget også de løsninger der fremkommer ved kombinere en af de ovenstående ombytninger med en af de følgende. Her er $EJ(i, v)$ efterfølgeren til (i, v) på job i og $FJ(i, v)$ forgængeren til (i, v) på job i. Den første ombytning er $(EJ(i, v), EM(EJ(i, v), x))$, hvor x er den maskine der behandler $EJ(i, v)$. Den anden ombytning er $(FM(FJ(j, v), y), FJ(j, v))$, hvor y er den maskine, der behandler $FJ(j, v)$. På figur 3.2 er vist en løsning til et job-shop problem taget fra [8].

Hvis dette nabolag benyttes på den disjunktive kant fra (2,1) til (4,1) (også kaldet b og e) fås følgende. I kassen er vist hvilke operationer, de forskellige udtryk bliver til.

$EJ(2,1)$	(2,3)
$EM(EJ(2,1),3)$	(4,3)
$FJ(4,1)$	(4,4)
$FM(FJ(4,1),4)$	(2,4)

Heraf ses at nabolaget dannet ud fra denne ene kant bliver på følgende tre elementer.



Figur 3.2: Eksempel på anvendelse af Nabolag 2

- Vend orienteringen på kanten [(2,1),(4,1)]
- Vend orienteringen på kanterne [(2,1),(4,1)] og [(2,3),(4,3)]
- Vend orienteringen på kanterne [(2,1),(4,1)] og [(2,4),(4,4)]

For dette nabolag gælder det *ikke* at man altid kan komme til den optimale løsning ved at benytte et endeligt antal af disse ændringer [8].

Dette nabolag er i [7] yderligere blevet udvidet til også at omfatte de kanter, hvor (i, v) og (i, u) tilhører det samme job. Som før benyttes også kombinationerne med at vende orienteringen på andre kanter, men disse kanter er nu $EM(i, u)$ og $FM(i, v)$.

Nabolag 3

De to første er såkaldte ombytningsnabolag, der som vist er definerede ved at der byttes om på to operationer. Dette tredje nabolag er et flytte-nabolag, som er defineret ved at en operation fjernes fra sin position og indsættes på en anden position.

Anvendelse

Et nabolag kan anvendes på mange måder.

1. Man kan gennemsøge hele nabolaget og finde den bedste løsning og dermed få det bedst mulige udgangspunkt for den næste iteration. Dette tager lang tid, især hvis nabolaget er stort eller løsningsværdien er vanskelig at beregne.
2. Man kan anvende den første løsning i nabolaget, der er bedre end den hidtil bedste. Hvis der ikke er en bedre løsning benyttes den

bedste af dem der er i nabolaget. Fordelen ved dette er, at man ikke behøver at gennemsøge hele nabolaget i hver iteration. Ulempen er at man risikerer at overse den bedste løsning og dermed forlænge søgeproceduren eller tilmed helt ramme forbi den optimale løsning.

3. Man kan vælge at undersøge et fast antal løsninger i nabolaget og så benytte den bedste af disse, i stedet for at benytte den første løsning, der er bedre end den hidtil bedste. Fordele og ulemper er som nævnt under 2, som er et specialtilfælde af dette nabolag.

Iflg. [10] kan det ikke betale sig at undersøge hele nabolaget.

Stopkriterier

Det er nødvendigt at have mindst et stopkriterium i en metaheuristik, da man ellers risikerer at heuristikken vil fortsætte i det uendelige. Dette kunne nemt ske i en metaheuristik, da dårligere løsninger her ofte bliver accepteret, og da den optimale løsninge ikke nødvendigvis opnås. Et godt stopkriterium sørger for, at heuristikken kører længe nok til at man kommer tæt på den optimale løsning, mens den på den anden side forhindrer at heuristikken står næsten stille i mange iterationer. Hvor tæt man vil på den optimale løsning og hvor mange iterationer er en skønssag overladt til den enkelte bruger. Der findes flere forskellige stopkriterier, nedenfor vil nogle af de mest benyttede blive beskrevet. De kan benyttes enkeltvist eller i sammenhæng.

Stopkriterium 1

Dette stopkriterium benyttes i de fleste metaheuristikker. Stopkriteriet er:

> Når søgningen har kørt i x iterationer, så stop.

I Simuleret Udglødning bliver dette ofte erstattet af:

> Når temperaturen er faldet til en bestemt værdi, så stop.

Fordelen ved dette kriterium er at man er sikker på at metaheuristikken stopper. Ulempen er, med et arbitrært valgt x, at en søgning måske stoppes for tidligt, så man kunne have været kommet meget tættere på de optimale løsning, eller for sent, så man har gennemgået en masse unødige iterationer uden at komme tættere på den optimale løsning. Derfor vælges dette x ofte i stedet, idet flere muligheder afprøves, før det bedste vælges. Desuden suppleres dette kriterium ofte med et eller flere af de andre.

Stopkriterium 2

Dette stopkriterium er også brugt i de fleste metaheuristikker. Kriteriet bliver kaldt for $SlidingWindow$ og lyder:

> Hvis den hidtil bedste løsning ikke er blevet forbedret i x iterationer, så stop.

Også denne har en version, der kun bliver benyttet ved Simuleret udglødning.

> Hvis der i de sidste x iterationer er accepteret færre end y procent af naboløsningerne, så stop.

Fordelen ved dette kriterium er at metaheuristikken kører, så længe løsningen bliver bedre. Ulempen er at metaheuristikken kan køre i meget lang tid uden at komme meget tættere på den optimale løsning (hvis forbedringerne er meget små og jævnt fordelte).

Stopkriterium 3

For at kunne benytte dette kriterium er det nødvendigt først at beregne en nedre grænse for problemets objektfunktionsværdi. Stopkriteriet er:

> Hvis løsningsværdien er inden for x % af LB, så stop.

Fordelen er, at det kan spare en del iterationer, hvis brugeren er tilfreds med en god, men ikke nødvendigvis optimal, løsning. Ulempen er, at kriteriet bliver ligegyldigt, og man derfor risikerer at metaheuristikken kører i det uendelige, hvis den nedre grænse ligger langt fra den optimale løsning. For shopproblemer ville det være fristende at bruge dette kriterium, da man for det meste i forvejen skal bruge LB til løsning af problemet, men som nævnt er det svært at finde en LB, der ligger tæt på den optimale løsning. Dette kriterium skal derfor helst ikke anvendes alene.

Stopkriterium 4

Endelig skal dette kriterium nævnes:

> Brugeren kan selv afbryde metaheuristikken.

Dette kriterium kan være fordelagtigt, hvis den tid brugeren har til rådighed er begrænset eller varierer meget. Ulempen er, at det ikke, når brugeren stopper Metaheuristikken, vides, hvor i søgeforløbet metaheuristikken blev stoppet, den kan have været i gang med meget små eller ingen forbedringer, men den kan også have været i gang med meget store forbedringer og der

kunne derfor være en (meget) bedre løsning indenfor meget kort tid. Dette kriterium anvender derved slet ikke den information som stopkriterium 1 gør.

En blanding af stopkriterium 1 og 2 bliver ofte benyttet, da man så kan udnytte fordelene ved dem begge. Man kan f.eks. sætte det første stopkriterium til 10000 og det andet til 500. Derved har man sikret, at der ikke udføres en masse iterationer, der ikke giver nogen forbedring af den hidtil bedste løsning. Samtidig opnås en maksimal grænse for antallet af iterationer, så metaheuristikken ikke vil fortsætte med at frembringe små forbedringer med jævne mellemrum.

Tabusearch

Generelt for metaheuristikken Tabusearch er ideen om at en startløsning forbedres ved at man

- hele tiden finder den bedste løsning i nabolaget (også selv om denne er dårligere end den gamle løsning), idet nabolaget opnås ved at ændre den gamle løsning en lille smule.
- husker den gamle løsning
- erklærer den benyttede ændring eller rettere dens modsætning for *tabu*, dvs. at den ikke må benyttes igen.
- opretter en tabuliste over disse ændringer.
- bestemmer et stopkriterium, f.eks. et antal iterationer, en bestemt afvigelse fra LB eller andet.
- evt. bestemmes et aspirationskriterium, som kan bruges til at afgøre om en ændring kan tillades før den når bunden af tabulisten.

Tabulisten

En tabuliste er en liste over hvilke løsninger, der er tabu. For at få listen til at virke bedst muligt er det nødvendigt at bestemme hvor lang denne liste skal være, dvs. hvor længe en ændring ikke må benyttes. Det er vigtigt at listen hverken er for lang eller for kort. Hvis den er for kort risikerer man at gå i cirkler, hvis den er for lang bliver det sværere at finde den optimale løsning. Den bedste listelængde kommer ofte an på det enkelte problem, hvorfor den tit findes ved at prøve sig frem. Iflg. [8] mener Fred Glover at tabulister virker bedst, hvis de er på mellem 5 og 12 (helst 7) elementer, uafhængigt af det enkelte problem. Modsat bliver det i [Taillard89] fundet

at listen skal være på $\frac{n+m}{2}$ elementer, hvis der er næsten lige mange job og maskiner og på $\frac{o}{2}$ hvis $n >> m$.

Hvad der skal gemmes i tabulisten kræver også lidt omtanke. At gemme hele den løsning der forlades i listen er meget pladskrævende, derfor er der flere andre muligheder. Af disse skal der her nævnes at

- Man kan nøjes med at gemme løsningsværdien. Dette gør at den løsning man forlader forbydes, men det samme gør alle andre løsninger med denne løsningsværdi, også selvom de ikke ligner den forladte løsning ret meget.
- Man kan nøjes med at gemme den ændring, der fører tilbage til den gamle løsning. Dette har den samme fordel og ulempe som før nævnt.

Aspirationskriterier

Et aspirationskriterium er et kriterium, som kan bruges til at afgøre om en ændring kan tillades før den når bunden af tabulisten. Dette kan f.eks. være tilfældet hvis man ved at benytte en ændring, der er på tabulisten, kan opnå en løsningsværdi, der er bedre end den hidtil bedste.

Simuleret udglødning

Generelt for metaheuristikken Simuleret Udglødning er ideen om, at en startløsning forbedres ved at man

- Udvælger løsninger tilfældigt fra nabolaget
- Undersøger dem en ad gangen indtil en accepteres, som ny løsning.
- Altid accepterer en løsning, der er bedre end den hidtil bedste.
- Accepterer en dårligere løsning med en vis sandsynlighed.

Dette betyder at denne metaheuristik sjældent benytter den bedste løsning i nabolaget og endog nogle gange anvender en løsning der er dårligere end den hidtil bedste, selvom der er en eller flere løsninger i nabolaget, der er bedre.

Sandsynligheden for at en dårligere løsning vælges kan i princippet defineres på mange måder, men for det meste benyttes

$$Sandsynlighed = min\{1, exp^{-\frac{\Delta_{ij}}{T}}\} \tag{3.7}$$

Hvor $\Delta_{ij} = C_{max}$ - Ny Løsningsværdi og T er en *temperatur*, der falder med hver iteration.

Som det kan ses vil der med denne acceptsandsynlighed blive accepteret færre dårlige løsninger som tiden går, mens en bedre løsning altid vil blive accepteret.

Desuden er kan det ses af ovennævnte definition af acceptsandsynligheden at starttemperaturen har en stor indflydelse på løsningens kvalitet. Hvis den er for høj kommer man godt nok godt rundt i løsningsrummet, men samtidigt øges beregningstiden betragteligt. Hvis starttemperaturen derimod er for lav er beregningstiden godt nok kort, men man risikerer nemt at komme til at sidde fast i et lokalt minimum.

Oven i dette har også den hastighed, hvormed temperaturen falder en indflydelse på løsningens kvalitet. Hvis temperaturfaldene er små koster det mere beregningstid og løsningen bliver bedre end hvis de er store. Desuden er der spørgsmålet om hvordan faldene i temperatur skal foregå. Skal temperaturen falde jævnt eller trinvist? Og skal den falde hurtigere i starten end til slut eller skal den falde jævnt gennem hele forløbet? Det er spørgsmål, som den enkelte bruger må tage stilling til, hvad der passer bedst til det enkelte problem, det tidsrum der til rådighed og den løsningskvalitet der ønskes.

3.2.2 Andre heuristikker

H1

Heuristikken H1 er blevet foreslået af C. Gueret og C. Prins [3] til at løse openshop-problemet. I artiklen bliver der foreslået to versioner af denne heuristik, en dynamisk og en statisk, der gentages flere gange. For begge versioner gælder at der først opstilles en prioriteret liste og at hovedet for alle operationer beregnes. Hovedet for en operation er det tidligste tidspunkt, som operationen kan startes på, f.eks. fordi andre operationer i jobbet skal færdiggøres først. Herefter skemalægges den af operationerne, der har det mindste hoved, som står først på listen og denne operation slettes fra listen. Endelig opdateres hovederne og i den dynamiske version også listen og næste operation kan skemalægges. I den statiske version opdateres listen først efter alle operationerne er skemalagt. Dette gøres som følger: de operationer, som behandles, mens en maskine er ledig skrives op i en ny liste kaldet $H1_{set}$. Også de operationer, der behandles efter den nedre grænse er nået, skrives op her. Alle operationer, der nu står i $H1_{set}$ skal prioriteres højere i den næste iteration, dette gøres ved at flytte dem alle

en position mod venstre på $H1_{liste}$ (se Eksempel 3.2.2). Derefter bliver alle operationer skemalagt igen. Denne procedure gentages 50 gange, da der iflg. Gueret og Prins ikke opnås signifikante forbedringer af løsningen ved flere gentagelser. Et andet stopkriterium kunne være at der stoppes når man er tilpas nær ved den nedre grænse.

Når listen skal opstilles sammenlignes først operationernes ikke-skemalagte procestid for maskinerne, dvs. den tid maskinerne mindst skal arbejde i for at udføre de ikke skemalagte operationer, og den længste vælges. Hvis disse er ens sammenlignes operationernes procestid, hvoraf den længste vælges. Hvis disse også er ens vælges den første operation.

Algoritme 1 *H1 - Dynamisk version*

- Lav prioriteret liste ($H1_{liste}$)
- Udregn hoveder for alle operationer (i OSS er alle hoveder 0 v. start)

For i=1 til o do

- Skemalæg
  - Operation m. mindst hoved
  - Af disse vælg operation m. højest prioritet.
- Slet valgt operation fra $H1_{liste}$
- Opdater listen
- Opdater hoveder for operationerne i listen

end for

Algoritme 2 *H1 - Statisk version*

- Lav prioriteret liste
- Udregn hoved for alle operationer (v. OSS er alle hoveder 0 v. start)

while (stopkriterium ikke opfyldt)

- for j=1 til listelængde do
  - skemalæg
    - operation m. mindst hoved
    - af disse vælges operation m. højest prioritet
  - slet valgt operation fra listen
  - opdater hoveder for operationerne i listen
  - end for
- find mængden af operationer, der behandles, mens en eller flere maskiner er ledige og operationer der behandles efter den nedre grænse er nået.

- opdater listen ved hjælp af denne mængde

end while

Som stopkriterium kan fx benyttes et fast antal iterationer, en vis afstand fra en kendt nedre grænse osv. på samme måde som nævnt v. metaheuristikkerne.

Der er nedenfor givet et eksempel på den statiske version, da denne iflg. Gueret og Prins er den der har 'de største muskler' og giver de bedste resultater.

Eksempel 3.2.1 *Open-shop løst vha. H1.*

I eksemplet løses det i tabel 3.3 viste 4x4 oss-problem.

job \ maskine	1	2	3	4
1	10	5	10	10
2	9	5	7	8
3	5	5	8	9
4	10	1	4	9

Tabel 3.3: Procestider i et 4x4 oss-problem

Som beskrevet i Algoritme 2 opstilles først en liste vha. den nævnte prioriteringsregel, dvs. den resterende maskintid for alle operationer findes. Disse er vist i tabel 3.4. Her er maskintid den tid, som maskinen skal køre i.

job \ maskine	1	2	3	4
1	24	11	19	26
2	25	11	22	28
3	29	11	21	27
4	24	15	25	27

Tabel 3.4: Resterende maskintid.

Listen kan nu findes vha. tabel 3.4 og tabel 3.3 til $H1_{liste} = \{9, 8, 12, 16, 4, 5, 15, 1, 13, 7, 11, 3, 14, 2, 6, 10\}$ Også hovederne skal beregnes, de er alle lig nul, da der ikke er en fast rækkefølge for de enkelte operationer i et job. Dette skrives i tabel 3.5.

Nu skemalægges den første operation i listen og operationen fjernes fra listen, som nu bliver $H1_{liste} = \{8, 12, 16, 4, 5, 15, 1, 13, 7, 11, 3, 14, 2, 6, 10\}$, hovederne opdateres, som det kan ses i tabel 3.6.

job \ maskine	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0

Tabel 3.5: Startværdi for hovederne for 4x4 oss-problem.

job \ maskine	1	2	3	4
1	5	0	0	0
2	5	0	0	0
3	-	5	5	5
4	5	0	0	0

Tabel 3.6: Hovederne efter 1. iteration.

Den næste operation, der skal skemalægges bliver nu operation nr. 8, da denne er den første på listen og har et af de mindste hoveder, denne operation slettes fra listen, som nu er

$$H1_{liste} = \{12, 6, 4, 5, 15, 1, 13, 7, 11, 3, 14, 2, 6, 10\}$$

og hovederne i tabel 3.6 opdateres, som det kan ses i tabel 3.7.

job \ maskine	1	2	3	4
1	5	0	0	8
2	8	8	8	-
3	-	5	5	8
4	5	0	0	8

Tabel 3.7: Hovederne efter 2. iteration.

Denne procedure gentages til $H1_{liste}$ er tom, idet der hele tiden vælges den operation, der er har et af de mindste hoveder og som er forrest på listen.

I tabel 3.8 ses, hvordan hovederne ændres, idet - viser at operationen er blevet skemalagt.

Da det kan være lidt svært umiddelbart at se den endelige løsning i tabel 3.8 kan dette også ses i tabel 3.9.

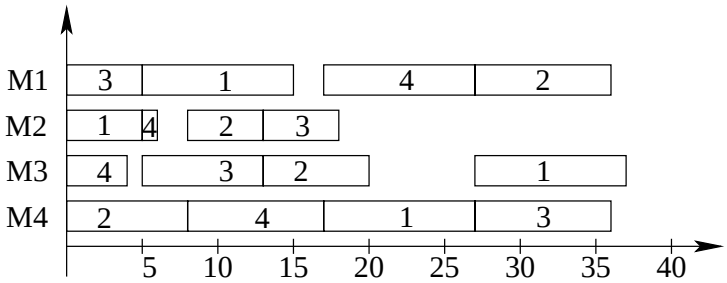
I figur 3.3 ses et Gantt-diagram over den nu opnåede tidsplan.

Operation nr.															
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
5	0	0	0	5	0	0	0	-	5	5	5	5	0	0	0
5	0	0	8	8	8	8	-	-	5	5	8	5	0	0	8
5	0	4	8	8	8	8	-	-	5	5	8	5	4	-	8
5	-	5	8	8	8	8	-	-	5	5	8	5	5	-	8
-	-	15	15	15	8	8	-	-	5	5	8	15	5	-	8
-	-	15	15	15	8	13	-	-	13	-	13	15	5	-	8
-	-	15	15	15	8	13	-	-	13	-	13	15	-	-	8
-	-	15	17	15	8	13	-	-	13	-	17	17	-	-	-
-	-	15	17	15	-	13	-	-	13	-	17	17	-	-	-
-	-	20	17	20	-	-	-	-	13	-	17	17	-	-	-
-	-	20	17	20	-	-	-	-	-	-	18	17	-	-	-
-	-	27	-	20	-	-	-	-	-	-	27	17	-	-	-
-	-	27	-	27	-	-	-	-	-	-	27	-	-	-	-
-	-	27	-	27	-	-	-	-	-	-	-	-	-	-	-
-	-	27	-	-	-	-	-	-	-	-	-	-	-	-	-
-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

Tabel 3.8: Ændringer af hovederne for operationerne, når den statiske H1-heuristik gentages.

job \ maskine	1	2	3	4
1	5	0	27	17
2	27	8	13	0
3	0	13	5	27
4	17	5	0	8

Tabel 3.9: Hoveder efter 1. løsning vha. H1.



Figur 3.3: Gantt-diagram af den opnåede tidsplan efter 1. iteration med H1.

Som det kan ses af figur 3.3 er der flere 'huller' i tidsplanen, steder, hvor planen måske kan gøres mere kompakt, derfor findes mængden $H1_{set} = \{1, 2, 3, 4, 5, 7, 8, 9, 10, 11, 12, 13, 16\}$ til at korrigere $H1_{liste}$ med. Den ny liste bliver nu $H1_{liste} = \{9, 8, 12, 16, 4, 5, 1, 13, 7, 11, 3, 15, 2, 14, 10, 6\}$ og hermed kan de nye hoveder beregnes og dermed den ny tidsplan findes. Der fortsættes med dette indtil stopkriteriet er opfyldt.

Denne heuristik kan også benyttes til al løse flowshop-problemet, idet der dog er et par mindre ændringer:

- hovederne opdateres anderledes, idet der tages hensyn til bånd til andre operationer.
- når H1-sæt findes, tages der ikke hensyn til *huller* før alle maskiner er *startet*.

H2

Denne heuristik er også blevet foreslået af C. Gueret og C. Prins [3], men er meget forskellig fra H1. Heuristikken går i korte træk ud på at tidsplanen findes stykvist, hvor stykkets længde er længden af den operation, der tager længst tid. Hvert stykke består af alle de operationer, der kan blive behandlet på det samme tidspunkt. Derefter sættes disse stykker sammen og man har en mulig tidsplan. H2-heuristikken er opdelt i to faser, i den første findes en mulig tidsplan og i den anden pakkes denne, således at der opnås en bedre plan.

Første fase.

Her findes de ønskede stykker. For at undgå spildtid ønskes det, at operationerne i et 'stykke' er så balancerede, som muligt, dvs. at de stort set har samme procestider. Dette kan f.eks. gøres ved at benytte en to-delt graf $TG(TG_N, TG_M, TG_{kant}, P_{n,m})$. TG_N er en mængde af punkter, hvor hvert punkt er et job. Tilsvarende er TG_M en mængde af punkter, hvor hvert punkt er en maskine. TG_{kant} er mængden af kanter i grafen, hvor hver kant er en operation, dvs. at kanten mellem job i og maskine j er operation (i, j). Hver kant er vægtet med operationens procestid og $P_{m,n}$ er en matrix med disse tider. I [3] bliver der gennemgået en række metoder til at opnå en balanceret parring. Af disse metoder bliver det fundet at den bedste for shop-problemernes vedkommende er min-max metoden. Denne metode går ud på at længden af den længste operation i et stykke minimeres. Når de forskellige stykker er bestemt bliver disse sat sammen, således at de ikke overlapper hinanden.

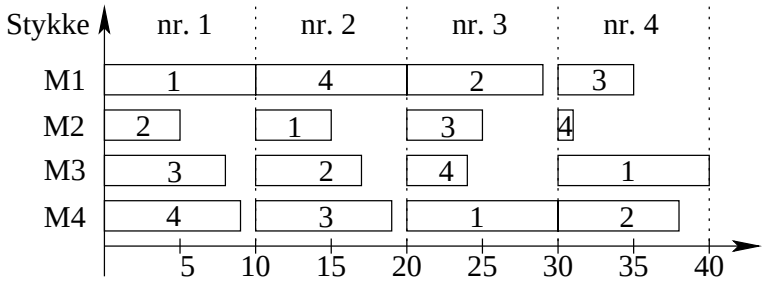
Eksempel 3.2.2 *Eksempel på brug af H2-heuristikken*

I dette eksempel benyttes problemet vist i tabel (3.10).

job \ maskine	1	2	3	4
1	10	5	10	10
2	9	5	7	8
3	5	5	8	9
4	10	1	4	9

Tabel 3.10: 4x4 oss-problem, som ønskes løst vha. H2.

Efter første fase kan tidsplanen i figur 3.4 opnås.



Figur 3.4: Løsning efter H2's 1. fase.

Som det kan ses af denne figur er der flere steder i planen, hvor den umiddelbart kan forbedres ved at blive pakket bedre, dette sker i den anden fase.

Anden fase.

Her bliver tidsplanen pakket bedre. Man kunne være fristet til blot at rykke de enkelte operationer så langt frem i tidsplanen, som deres restriktioner tillader, men dette vil generelt ikke give de store forbedringer. Af denne grund har C. Gueret og C. Prins set på to andre pakkemetoder, der begge er baseret på det de kalder GCP-kriteriet (Greatest Compression Percentage/største sammenpresnings procent). Her er GCP = Produktionstid før pakning/Produktionstid efter pakning. I den første pakkemetode $GCP(P)$ (P for pak) sættes de forskellige stykker sammen på en sådan måde at GCP maksimeres. Dette gøres ved følgende algoritme.

Algoritme 3 *GCP(P)*

- S = tidsplan svarende til det længste stykke.

while(ikke fastlagt stykke eksisterer) do

GCP=0

for(hvert ikke fastlagt stykke *Stykke*) do

for(hver mulig position p i S) do

- fastlæg *Stykke* i position p
- beregn produktionstid før pakning
- pak planen
- beregn produktionstid efter pakning
- beregn GCP_{ny}

if($GCP_{ny} > GCP$)

- $GCP = GCP_{ny}$
- Bedste Placering = *Stykke*
- Bedste Position = p

end if

end for

end for

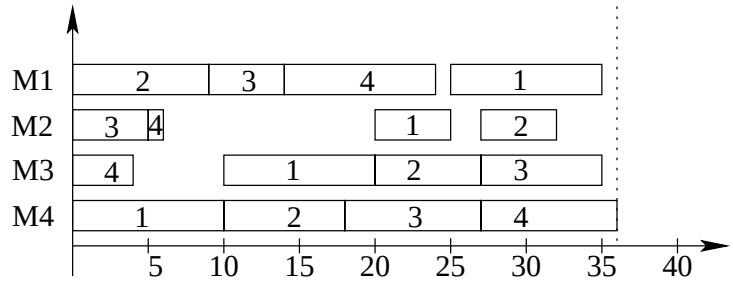
- Modificer S ved at fastlægge Bedste Placering på Bedste Position

end while

- pak S

Eksempel 3.2.3 *GCP(P)*

Hvis GCP(P) benyttes på vores problem findes først at stykke 1 selvfølgelig skal indsættes på position 1, derefter afprøves stykke 2 først på position 1 og dernæst på pos. 2, da der opnås det samme resultat begge steder benyttes den første, så stykke 2 nu er på pos. 1 og stykke 1 på pos. 2. Stykke 3 afprøves nu på alle mulige positioner og indsættes på pos. 1. Til slut afprøves stykke 4 og denne placeres på pos. 2, da dette giver den største GCP. Det færdige resultat kan ses i figur 3.5.



Figur 3.5: Samme problem efter H2's 2. fase med GCP(P)

Som det kan ses af figuren er der i dette tilfælde opnået en optimal løsning, idet $C_{Max} = 36 = LB$.

I den anden pakke metode $GCP(I/P)$ (Indsæt eller Pak) gøres stort set det samme, bortset fra at man før man pakker, skal se om det kan lade sig gøre at placere enkelte af stykkets operationer i et tidligere 'hul', i den plan man er i gang med at lave. Når dette er gjort fortsættes med at pakke som i oven nævnte algoritme.

Eksempel 3.2.4 *GCP(I/P)*

Hvis GCP(I/P) benyttes på vores problem er der på intet tidspunkt mulighed for at indsætte operationer i de tidligere huller, derfor opnås den samme løsning, som tidligere.

Kapitel 4

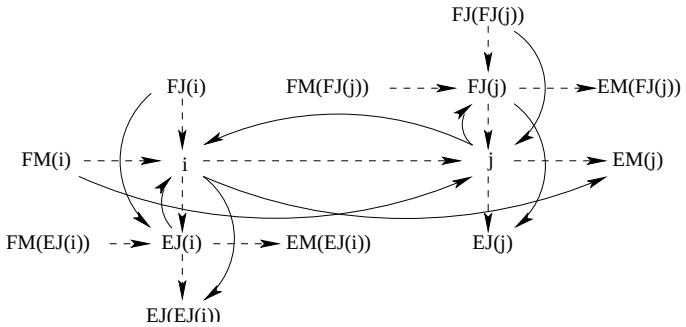
Løsningsmetoder - openshop

I dette kapitel gennemgås en metoder (heuristik) til løsning af openshop problemet. Som nævnt i kapitel 2.3.1 er oss-problemet det problem, der har de færreste bindinger, det er derfor også det problem der har det største løsningsrum og det der sandsynligvist er sværest at finde en optimal løsning til.

4.1 Simuleret Udglødning

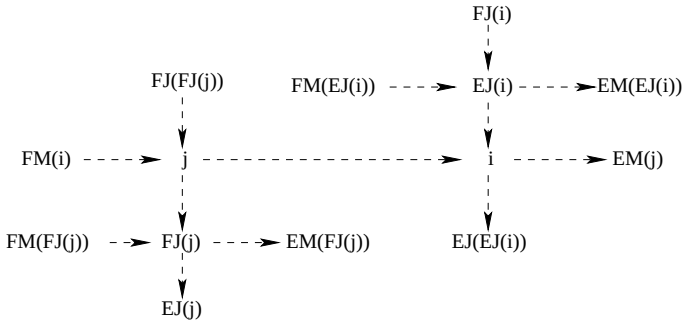
Som nævnt i kapitel 3.2 kan visse heuristikker benyttes til at løse shopproblemerne. I det følgende vil der blive set nærmere på hvordan heuristikken Simuleret Udglødning kan tilpasses til Open-shop problemet.

I følge [7] kan et godt nabolag her findes ved at benytte Nabolag 3, som det er beskrevet i afsnittet om Metaheuristikker. Som det er beskrevet bliver der kun set på operationer på den kritiske vej, dvs. kun operationer, hvorom det gælder at $r_i + p_i + q_i = C_{max}$. Dette benyttes også ved evalueringen af nabolaget, idet der her ses på længden af den længste vej gennem en af $\{i, j, FJ(j, v), EJ(i, v)\}$, hvis i og j behandles på samme maskine. Som nævnt under Nabolag3 kan orienteringen på kanten mellem (i, v) og (j, v) vendes alene eller dette kan kombineres med samtidigt at vende orienteringen på andre kanter. Hvis orienteringen vendes på alle tre kanter bliver følgende operationer berørt, som vist i figur 4.1.



Figur 4.1: ønsker at vende kanterne (i, j), (FJ(j, v), j) og (i, EJ(i, v))

Og figur 4.2 opnås.



Figur 4.2: Vendt kanterne (i, j), (FJ(j, v),j) og (i, EJ(i, v))

Dette giver dog ikke altid en lovlig løsning, da der kan opstå kredse i løsningen, hvis der er en orienteret kant fra $EM(FJ(j))$ til $FM(i)$ eller fra $EM(j)$ til $FM(EJ(i))$ i den løsning, som der skal ændres på. Flytningen giver kun en lovlig løsning, hvis disse kanter ikke findes, dvs.

hvis

$$r_{EM(FJ(j))} + p_{EM(FJ(j))} > r_{FM(i)}$$

og

$$r_{EM(j)} + p_{EM(j)} > r_{FM(EJ(i))}$$

Den nye længste vej gennem mindst en af de nævnte knuder kan nu beregnes ved

$$\begin{aligned}
r_j^* &= \max\{r_{FJ(FJ(j))} + p_{FJ(FJ(j))}, r_{FM^*(j)} + p_{FM^*(j)}\} \\
&\quad \text{hvor} \quad FM^*(j) = FM(i). \\
r_{FJ(j)}^* &= \max\{r_{FM(FJ(j))} + p_{FM(FJ(j))}, r_j^* + p_j\}. \\
r_{EJ(i)}^* &= \max\{r_{FJ^*(EJ(i))} + p_{FJ^*(EJ(i))}, r_{FM(EJ(i))} + p_{FM(EJ((i))}\} \\
&\quad \text{hvor} \quad FJ^*(EJ(i)) = FJ(i). \\
r_i^* &= \max\{r_{EJ(i)}^* + p_{EJ(i)}, r_j^* + p_j\}. \\
q_i^* &= \max\{q_{EJ(EJ(i))} + p_{EJ(EJ(i))}, q_{EM^*(i)} + p_{EM^*(i)}\} \\
&\quad \text{hvor} \quad EM^*(i) = EM(j). \\
q_{EJ(i)}^* &= \max\{q_{EM(EJ(i))} + p_{EM(EJ(i))}, q_i^* + p_i\}. \\
q_{FJ(j)}^* &= \max\{q_{EM(FJ(j))} + p_{EM(FJ(j))}, q_{EJ^*(FJ(j))} + p_{EJ^*(FJ(j))}\} \\
&\quad \text{hvor} \quad EJ^*(FJ(j)) = EJ(j). \\
q_j^* &= \max\{q_i^* + p_i, q_{FJ(j)} + p_{FJ(j)}\}. \\
Z &= \max\{r_i^* + p_i + q_i^*, r_j^* + p_j + q_j^*, r_{EJ(i)}^* + p_{EJ(i)} + q_{EJ(i)}^*, \\
&\quad r_{FJ(j)}^* + p_{FJ(j)} + q_{FJ(j)}^*\}.
\end{aligned}$$

Værdien Z angiver længden af denne nye længste vej og dermed en nedre grænse for C_{max} i den løsning, der opstår med den nævnte ændring.

Når heuristikken Simuleret Udglødning skal benyttes, skal der udover nabolaget også vælges en starttemperatur og en nedkølingshastighed eller plan. Hvis vi igen følger C.-F. Liaw [7], benytter vi en såkaldt geometrisk nedkølingsplan. Denne kan kort beskrives ved en start temperatur T_0, der i hver iteration multipliceres med en reduktionsfaktor.

Endelig skal stopkriteriet bestemmes, idet der naturligvis skal stoppes, hvis den optimale løsning opnås(dvs. hvis $C_{max} = LB$). Derudover ønsker vi at stoppe algoritmen efter et max. antal iterationer. For at undgå unødvendigt mange iterationer, men stadig fortsætte længe nok, foreslår C.-F. Liaw [7] at tælleren nulstilles, hver gang der benyttes en bedre løsning og tæller op når der benyttes en dårligere og $pct < min\_pct$. Her er min_pct en på forhånd fastsat faktor og pct procentdelen af accepterede løsninger indenfor de sidste L iterationer ved samme temperatur. Alt dette kan sammenfattes til følgende algoritme.

Algoritme 4 *Simuleret udglødning for openshop-scheduling*

1. Find en startløsning s og en nedre grænse LB.

2. Bestem en start temperatur T=T_0, en $max\_tæller$, en min_pct, en str_faktor og en red_faktor.
3. Sæt $L = str\_faktor \cdot n \cdot m$ og $tæller = 0$.
4. While($tæller < max\_tæller$ & & $C_{max} > LB$)
   4.1 for(1;L)
      4.1.1 Generer en nabo s^* til s
      4.1.2 Lad $\Delta C = C_{max}(s^*) - C_{max}(s)$.
      4.1.3 if($\Delta C < 0$) $s = s^*$
      else generer tilfældigt tal $R \in$ [0,1]
      4.1.3.1 if $\left(R < \exp\left(\frac{-\Delta C}{T}\right)\right)$ $s = s^*$
   4.2 Sæt $T = T \cdot red\_faktor$.
   4.3 Sæt pct = procendel accepterede løsninger i de sidste L iterationer
   4.4 if(ny bedste løsning blev fundet) $tæller = 0$
   else
      if($pct < min\_pct$) $tæller = tæller + 1$
5. Returner bedste fundne løsning

Startløsningen kan f.eks. findes vha. LTRPOM (Longest Total Remaining Processing Time on Other Machine First), som vil blive benyttet i det følgende eksempel. Denne regel betyder : når en maskine er ledig, findes den operation, der mangler mest tid på andre maskiner. Hvis der ikke er en sådan operation, står maskinen ledig indtil en operation bliver færdig på en anden maskine. Hvis der er mere end en operation med den samme resterende procestid på andre maskiner vælges den, der mangler mest tid i jobbet, hvis disse også er ens vælges den første.

Eksempel 4.1.1 *Simuleret udglødning for oss.*

I det følgende gennemgås proceduren i et taleksempel, her benyttes samme 4x4 oss-problem som tidligere, med de i tabel 4.1 viste procestider.

job \ maskine	1	2	3	4
1	10	5	10	10
2	9	5	7	8
3	5	5	8	9
4	10	1	4	9

Tabel 4.1: Procestider for et 4x4 oss-problem

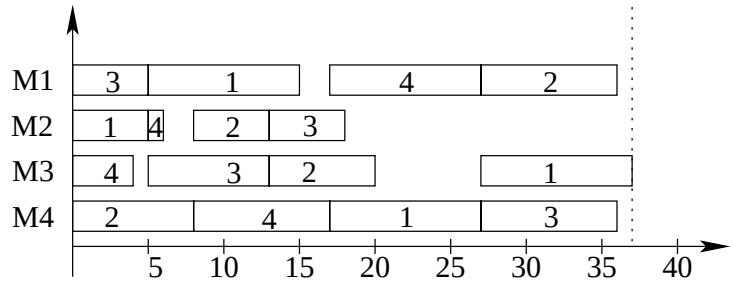
1. Find en startløsning s og en nedre grænse LB.

For at finde en startløsning, skal vi bruge den resterende tid på andre maskiner for alle operationer. Disse kan ses i tabel 4.2.

job \ maskine	1	2	3	4
1	24	11	19	26
2	25	11	22	28
3	29	11	21	27
4	24	15	25	27

Tabel 4.2: Resterende tid på andre maskiner

Heraf kan ses at den første skemalagte operation på maskine 1 bliver operation nr. 9, som tilhører job 3. Ligeledes kan det ses, at operation 15, 8 og 1 umiddelbart kan skemalægges. Maskine 3 bliver først færdig, men der er ingen mulige operationer, så maskinen må vente til maskine 1 bliver færdig, til dette tidspunkt bliver også maskine 2 færdig og de resterende tider på andre maskiner undersøges, dette giver, at maskine 1 udfører operation 1, maskine 2 operation 14 og maskine 3 operation 11. Der fortsættes på denne måde til alle operationer er blevet skemalagt. Startløsningen kan ses i Gantt-diagrammet i figur 4.3.



Figur 4.3: Gantt-diagram af startløsningen til oss-problemet.

Som det kan ses af figur 4.3 er $C_{max} = 37$ for denne startløsning.

Den nedre grænseværdi kan simpelt findes til $LB = 36$, (hvilket også tidligere er vist er C_{max} for problemets optimale løsning).

2. Bestem en start temperatur T=T_0, en *max_tæller*, en *min_pct*, en *str_faktor* og en *red_faktor*.

Dette er det punkt hvor der skal justeres en del på værdierne for at få de helt rigtige. I dette lille eksempel bliver der dog ikke kommet nærmere ind på dette. Her benyttes

$$\begin{aligned} T_0 &= 100 \\ max\_tæller &= 5 \\ min\_pct &= 50\% \\ str\_faktor &= 0.2 \\ red\_faktor &= 0.995 \end{aligned}$$

Samtidigt nulstilles *tæller*.

3. Sæt $L = str\_faktor \cdot n \cdot m$ og $tæller = 0$.

Med den valgte str_faktor bliver $L = 0.5 \cdot n \cdot m = 0.2 \cdot 4 \cdot 4 = 3.2 \approx 3$.

4. While($tæller < max\_tæller \quad \& \& \quad C_{max} > LB$)
   4.1 for(1;L)
      4.1.1 Generer en nabo s^* til s

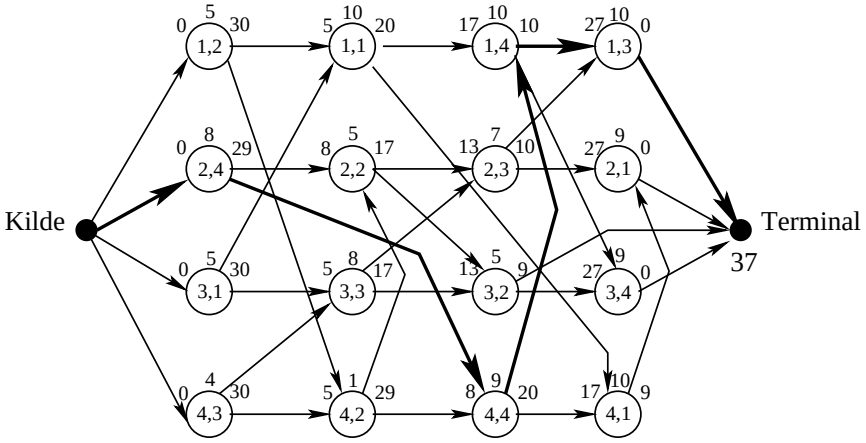
For at finde en nabo til s undersøges s, så den kritiske vej findes, samtidig findes hoveder og haler for operationerne. Dette kan ses i figur 4.4. I figurens knuder kan operationens (jobnr, maskinnr) ses.

Som det kan ses består den kritiske vej af to blokke, én på maskine 4 bestående af operation {(2,4), (4,4), (1,4)} og én på job 1, bestående af operation {(1,4), (1,3)}.

Naboer kan nu dannes ud fra operation (2,4), (1,4) og (1,3), da disse er enten først eller sidst i en blok. Idet der tages hensyn til at operationerne begge skal tilhøre den kritiske vej og skal være naboer på denne, kan følgende nabolag findes (idet orienteringen på de nævnte kanter skal vendes). Dette sker vha. tabel 4.1.

For (1) giver dette anledning til følgende mulige ændringer og dermed naboer

a. ((2,4), (4,4)),
b. ((2,4), (4,4)), ((2,4), (2,2)),
c. ((2,4), (4,4)), ((4,2), (4,4)),
d. ((2,4), (4,4)), ((2,4), (2,2)), ((4,2), (4,4))



Figur 4.4: Graf af startløsningen til oss-problemet med operationernes hoveder og haler.

(i,j)	((2,4), (4,4))	(1)
$(i,EJ(2,4))$	((2,4), (2,2))	
$(FJ(4,4),j)$	((4,2), (4,4))	
(i,j)	((4,4), (1,4))	(2)
$(i,EJ(4,4))$	((4,4), (4,1))	
$(FJ(1,4),j)$	((1,1), (1,4))	
(i,j)	((1,4), (1,3))	(3)
$(i,EM(1,4))$	((1,4), (3,4))	
$(FM(1,3),j)$	((2,3), (1,3))	

Før en nabo benyttes, skal det først undersøges om disse ændringer er lovlige, dvs. om ændringerne giver anledning til kredse i grafen. Dette er dog kun nødvendigt, hvis der vendes tre kanter, dvs. kun for d.-ændringerne. For disse betyder dette, at det undersøges, om der eksisterer en kant fra $EM(FJ(j))$ til $FM(i)$ eller en kant fra $EM(j)$ til $FM(EJ(i))$. For (3) skal dette omformuleres, således at der for (3 d.) i stedet søges efter en kant fra $EJ(FM(j))$ til $FJ(i)$ eller en kant fra $EJ(j)$ til $FJ(EM(i))$.

For f.eks. (1 d.) vil undersøgelsen se ud, som følger.

Her kan det nu undersøges om der eksisterer en kant fra I til II (hvilket

(i,j)	((2,4), (4,4))	(1)
$EM(FJ(4,4))$	(2,2)	I
$FM(2,4)$	(,)	II
$EM(4,4)$	(1,4)	III
$FM(EJ(2,4)$	(4,2)	IV

ikke er muligt her, da II er Kilde i grafen) eller en kant fra III til IV, hvilket, der ikke er.

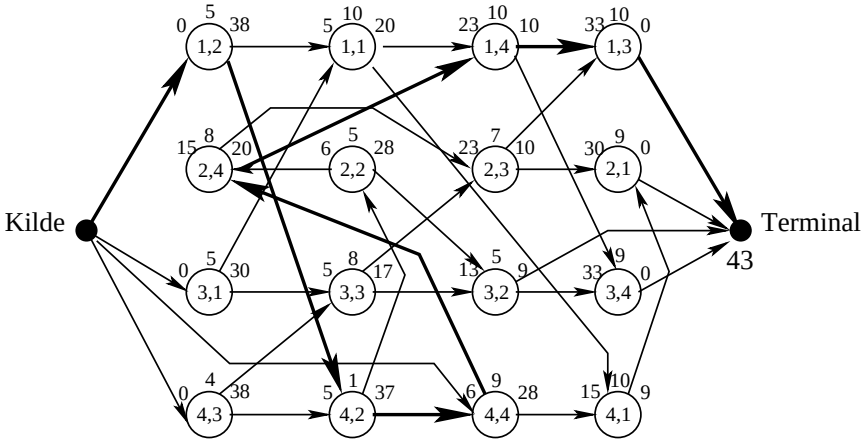
Eller dette kan undersøges matematisk ved

$$r_{(1,4)} + p_{(1,4)} \quad = \quad 17 + 10 \quad = \quad 27 \qquad 29 \quad = \quad r_{(4,2)}$$

Dette betyder at denne ændring (1 d.) er lovlig.

Den nabo, der skal undersøges, vælges tilfældigt mellem de lovlige naboer. I dette eksempel blev dette nabo (1 b.)

Med to vendte kanter (1 b.) ser grafen over løsningen ud som vist i figur 4.5. Denne løsning har en længste vej på 40.



Figur 4.5: Graf af løsningen med to vendte kanter.

Dette resultat kan her aflæses af figuren, men da dette ikke vil være helt så nemt ved større problemer vises det her, hvordan resultatet kunne have været opnået.

$$
\begin{aligned}
r_j^* &= \max\{r_{FJ(FJ(j))} + p_{FJ(FJ(j))}, r_{FM(i)} + p_{FM(i)}\} \\
&\quad \max\{5+1, /\} \\
&= 6 \\
r_{EJ(i)}^* &= \max\{r_{FJ(i)} + p_{FJ(i)}, r_{FM(EJ(i))} + p_{FM(EJ((i))}\} \\
&\quad \max\{/, 5+1\} \\
&= 6 \\
r_i^* &= \max\{r_{EJ(i)}^* + p_{EJ(i)}, r_j^* + p_j\} \\
&\quad \max\{6+5, 6+9\} \\
&= 15 \\
q_i^* &= \max\{q_{EJ(EJ(i))} + p_{EJ(EJ(i))}, q_{EM(j)} + p_{EM(j)}\} \\
&\quad \max\{10+7, 10+10\} \\
&= 20 \\
q_{EJ(i)}^* &= \max\{q_{EM(EJ(i))} + p_{EM(EJ(i))}, q_i^* + p_i\} \\
&\quad \max\{9+5, 20+8\} \\
&= 28 \\
q_j^* &= \max\{q_i^* + p_i, q_{EJ(j)} + p_{EJ(j)}\} \\
&\quad \max\{20+8, 10+9\} \\
&= 28 \\
Z &= \max\{r_i^* + p_i + q_i^*, r_j^* + p_j + q_j^*, r_{EJ(i)}^* + \\
&\qquad + p_{EJ(i)} + q_{EJ(i)}^*\} \\
&= \max\{15+8+20, 6+9+28, 6+5+28\} \\
&= \max\{43, 43, 39\} \\
&= 43
\end{aligned}
$$

hvor

- / betyder at det ikke er muligt at beregne denne værdi, da den berørte operation er enten kilde eller terminal.
- Z er længste vej, dvs. $Z = C_{max}$.
- $*$ er den nye værdi.

Da denne løsning er dårligere en den allerede opnåede afprøves det, om den alligevel skal accepteres. Dette bestemmes ved at sammenligne et tilfældigt genereret tal og en beregnet værdi for derefter at sammenligne disse.

$$R < \exp\left(\frac{-\Delta C}{T}\right)$$

med $R \quad = 0.2505$

$\Delta C = 40 - 37 = 3$
$T \quad = 100$

$$R = 0.2505 < 0.9704 = \exp\left(\frac{-3}{100}\right) = \exp\left(\frac{-\Delta C}{T}\right)$$

Hvilket betyder at denne løsning accepteres (den gamle, bedre løsning huskes naturligvis for det tilfælde at der ikke findes en endnu bedre løsning).

Igen findes løsningens naboer, først ved at opdele den kritiske vej i blokke {(1,2), (4,2)}, {(4,2), (4,4)}, {(4,4), (2,4), (1,4)}, {(1,4), (1,3)}, derefter findes de mulige i og/eller j, som første henholdsvis sidste operation i en blok, dvs. naboer kan dannes ud fra kanter fra i = {(1,2), (4,2), (4,4), (1,4)} eller kanter til j = {(4,2), (4,4), (1,4), (1,3)}. Dette betyder at der kan dannes nabolag ud fra 5 kanter, hvilket igen betyder at nabolaget bliver på 20 elementer (selvom det ikke er sikkert at alle giver lovlige løsninger).

En tilfældig nabo udvælges, her bliver dette nabo (4 a.), hvilket betyder at orienteringen på kant (1,4) - (1,3) skal vendes. Dette giver en ny C_{max} = 59.

$$R = 0.9133 \not< 0.8270 = \exp\left(\frac{-19}{100}\right) = \exp\left(\frac{-\Delta C}{T}\right)$$

Dette betyder at denne løsning ikke accepteres og der kigges derfor på en ny nabo. Ved at tage en tilfældig nabo findes nabo (3 b.), Denne består af at vende orienteringen på kant (4,4) - (2,4) og kant (4,4) - (4,1). For denne nabo bliver C_{max}=55, da denne løsning er bedre end den der haves i øjeblikket benyttes den som ny løsning.

Da der nu er gået 3 iterationer reduceres T til $T = 0.995 \cdot 100 = 99.95$. Samtidigt sættes *pct* til $2/3 = 66\%$, da der i de sidste 3 iterationer er blevet accepteret 2 løsninger. Da der ikke er blevet fundet en ny "bedste løsning" undersøges det om $pct < min\_pct$, da dette ikke er tilfældet ændres der ikke på *tæller*.

Nu følger igen 3 iterationer, hvor der ledes efter naboløsninger, der kan accepteres, derefter reduceres T yderligere, *pct* findes og sammenlignes evt. med *min_pct*. Dette gøres indtil en løsning har $C_{max} = LB$ eller indtil der er benyttet det under 2. bestemte antal iterationer.

Den fundne løsning er på dette tidspunkt den, der er gemt som "bedste løsning".

Kapitel 5

Løsningsmetoder - flow-shop

I dette kapitel gennemgås en metode (heuristik) til løsning af flow-shop problemet. Som nævnt i kapitel 2.4 er fss-problemet det problem, der har de fleste bindinger, det er derfor også det problem der har det mindste løsningsrum og det der i princippet burde være det nemmeste at finde en optimal løsning til.

5.1 HFC

Som nævnt i kapitel 3.2 kan visse heuristikker benyttes til at løse shopproblemerne. I det følgende vil der blive set nærmere på HFC-heuristikken, så vidt vides er denne den bedste af et fåtal af fss-heuristikker. Her skal det dog nævnes, at der findes en del heuristikker til løsning af det nærliggende pfss-problem(permuteret fss).

Heuristic Flowshop scheduling with C_{max}-objective blev i 1998 foreslået af C. Koulamas. Heuristikken går i korte træk ud på at problemet opdeles i $\frac{m(m-1)}{2}$ 2-maskine problemer, vha. disse små problemer findes en foreløbig løsning, denne løsning ændres derefter ved at bytte om på nabooperationer i løsningen, idet der kun byttes, hvis det giver en bedre løsning.

HFC fase 1.

Som nævnt opdeles et fss-problem først i $\frac{m(m-1)}{2}$ 2-maskine problemer, for et 3-maskine problem ville dette betyde at de 3 små problemer, som der

opdeles i består af maskine (1,2), (2,3) og (1,3). For hvert af disse findes den bedste rækkefølge af jobbene, hvis job i i flertallet af problemerne bliver lavet før job j, bliver i skemalagt før j. Dette giver en 1. løsning og en matrix H, som senere kan benyttes til muligvis at forbedre denne.

C. Koulamas foreslår i [5] den følgende algoritme for fase 1.

Algoritme 5 *HFC (fase 1)*

0. Sæt $I_i = 0 \qquad i = 1, \ldots, n$
1. for$(i = 1, \ldots, n-1)$
   {
   for$(j = i+1, \ldots, n)$
   {
   for$(k = 1, \ldots, m-1)$
   {
   for$(l = k+1, \ldots, m)$
   {
   if$(min\{p_{ik}, p_{jl}\} < min\{p_{il}, p_{jk}\})$
   {
   $I_i = I_i - 1$
   $I_j = I_j + 1$
   $H((i,j),(k,l)) = -1$
   }
   else if$(min\{p_{ik}, p_{jl}\} > min\{p_{il}, p_{jk}\})$
   {
   $I_i = I_i + 1$
   $I_j = I_j - 1$
   $H((i,j),(k,l)) = 1$
   }
   $l = l + 1$
   }
   $k = k + 1$
   }
   $j = j + 1$
   }
   $I = i + 1$
   }
2. Sorter I_i i ikke-faldende orden og skemalæg jobbene i denne rækkefølge.
3. Lad $\sigma(1), \ldots, \sigma(n)$ være denne rækkefølge; beregn C_{max}.

HFC fase 2.

H-matricen benyttes i heuristikkens 2. del, som grundlag for den førnævnte ombytning. Denne ombytning foregår som beskrevet i Algoritme 6.

Algoritme 6 *HFC (fase 2)*

for$(i = 1, \ldots, n-1)$
  Lad k være den række i H, der svarer til job-parret $\sigma(i), \sigma(i+1)$
  for$(j = 2, \ldots, m-1)$
    $T_1 = 0$
    for$(q = 1, \ldots, j-1)$
      for$(r = q+1, \ldots, j)$
        Lad l være den søjle i H, der svarer til maskin-parret q, r.
        $T_1 = T_1 + H(k, l)$
      $r = r + 1$
    $q = q + 1$
    if$(|T_1| < (j(j-1))/2)$
      $j = j + 1$
    $T_2 = 0$
    for$(q = j+1, \ldots, m-1)$
      for $(r = q+1, \ldots, m)$
        Lad l være den søjle i H, der svarer til maskin-parret q, r.
        $T_2 = T_2 + H(k, l)$
      $r = r + 1$
    $q = q + 1$
    if$(T_2 \neq -T_1)$
      $j = j + 1$
    Beregn C_{max} for σ^*, hvor $\sigma^*(k) = \sigma(k)$ for alle $k = 1, \ldots,$
    $i-1, i+2, \ldots, n$ og hvor $\sigma^*(i) = \sigma(i)$, $\sigma^*(i+1) = \sigma(i+1)$
    på maskine $1, \ldots, j$, og hvor $\sigma^*(i) = \sigma(i+1)$ og $\sigma^*(i+1) = \sigma(i)$
    på maskine $j+1, \ldots, m$.
    if$(C_{max}(\sigma^*) < C_{max}(\sigma)$
      $\sigma = \sigma^*$
  j = j+1
i = 1+1

Eksempel 5.1.1 *HFC for 4x4 fss*

I eksemplet benyttes følgende 4x4 fss-problem

Som nævnt benyttes Algoritme 5 og tabel 5.3 opstilles

job \ maskine	1	2	3	4
1	10	5	10	10
2	9	5	7	8
3	5	5	8	9
4	10	1	4	9

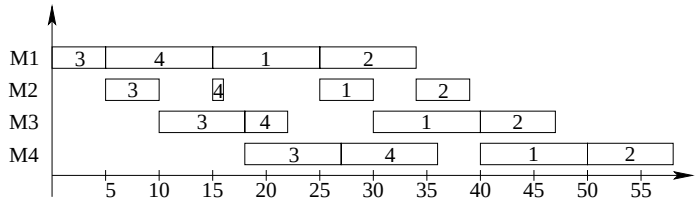
Tabel 5.1: Procestider for et 4x4 fss-problem

job \ operation nr.	1	2	3	4
1	1	2	3	4
2	1	2	3	4
3	1	2	3	4
4	1	2	3	4

Tabel 5.2: Angiver hvilken maskine, der skal lave hvilken operation i fss-problemet

Dette giver H-matricen vist i tabel 5.4.

Dette giver $I_1 = 2$, $I_2 = 4$, $I_3 = -4$ og $I_4 = -2$, og dermed bliver jobrækkefølgen $3 \rightarrow 4 \rightarrow 1 \rightarrow 2$. Denne løsning har $C_{max} = 58$ og kan ses i figur 5.1



Figur 5.1: Gantt-diagram af 1. løsning v. HFC

Da fase 2 ikke giver en bedre løsning i dette tilfælde er en gennemgang af denne ikke vist her.

Som det kan ses opnås den optimale løsning ikke i dette tilfælde.

i	j	k	l	p_{ik}	p_{jl}	p_{il}	p_{jk}	$A < B$	$A > B$	$A = B$
1	2	1	2	10	5	5	9			x
1	2	1	3	10	7	10	9	x		
1	2	1	4	10	8	10	9	x		
1	2	2	3	5	7	10	5			x
1	2	2	4	5	8	10	5			x
1	2	3	4	10	8	10	7		x	
1	3	1	2	10	5	5	5			x
1	3	1	3	10	8	10	5		x	
1	3	1	4	10	9	10	5		x	
1	3	2	3	5	8	10	5			x
1	3	2	4	5	9	10	5			x
1	3	3	4	10	9	10	8		x	
1	4	1	2	10	1	5	10	x		
1	4	1	3	10	4	10	10	x		
1	4	1	4	10	9	10	10	x		
1	4	2	3	5	4	10	1		x	
1	4	2	4	5	9	10	1		x	
1	4	3	4	10	9	10	4		x	
2	3	1	2	9	5	5	5			x
2	3	1	3	9	8	7	5		x	
2	3	1	4	9	9	8	5		x	
2	3	2	3	5	8	7	5			x
2	3	2	4	5	9	8	5			x
2	3	3	4	7	9	8	8	x		
2	4	1	2	9	1	5	10	x		
2	4	1	3	9	4	7	10	x		
2	4	1	4	9	9	8	10		x	
2	4	2	3	5	4	7	1		x	
2	4	2	4	5	9	8	1		x	
2	4	3	4	7	9	8	4		x	
3	4	1	2	5	1	5	10	x		
3	4	1	3	5	4	7	10	x		
3	4	1	4	5	9	8	10	x		
3	4	2	3	5	4	7	1		x	
3	4	2	4	5	9	8	1		x	
3	4	3	4	8	9	8	4		x	

Tabel 5.3: Tabel til brug v. opbygning af H-matrix

jobpar \ maskinpar	1-2	1-3	1-4	2-3	2-4	3-4
1-2	0	-1	-1	0	0	1
1-3	0	1	1	0	0	1
1-4	-1	-1	-1	1	1	1
2-3	0	1	1	0	0	-1
2-4	-1	-1	-1	1	1	1
3-4	-1	-1	-1	1	1	1

Tabel 5.4: H matrix for HFC-heuristikken

Kapitel 6

Programmer

For at kunne sammenligne de metoder, der er blevet nævnt i de forrige kapitler har jeg implementeret:

- For openshop
  - H1
  - Simuleret Udglødning (SA)
- For flowshop
  - HFC
  - H1

Desuden benyttes CPLEX til at finde løsninger og løsningstider v. brug af LP og Branch & Bound. Den CPLEX version, der benyttes er 'CPLEX Linear Optimizer 6.6.1 with Mixed Integer & Barrier Solvers Copyright (c) ILOG 1997-2000'.

I følge litteraturen er disse metoder nemlig de bedste og derfor de mest interessante at holde op mod hinanden. Det bliver dog i [8] nævnt at heuristikken H2 er bedre end H1, alligevel er det her valgt at implementere H1 af 2 årsager: H1 er nemmere at implementere og kunne forholdsvist nemt også benyttes til flowshop, hvilket så interessant ud.

Koden i C for de nævnte programmer, undtagen CPLEX, kan findes i Appendiks.

Om implementeringen.

Mens H1-algoritmen var relativt nem at implementere for openshop viste det sig at der var flere problemer end forventet ved flowshop. $H1_{set}$ fandt her ofte alle operationer, hvilket betød at Slutløsningen altid blev den samme som startløsningen, andre løsninger blev nemlig ikke undersøgt. Dette problem blev forsøgt afhjælpet ved kun at medtage operationer påbegyndt efter det sidste job er startet, men med begrænset succes. Simuleret udglødning bød ikke på de store overraskelser, men tog lang alligevel relativt lang tid at implementere pga. de mange særtilfælde der kan opstå ved ombytning af en eller flere operationers rækkefølge. HFC var den algoritme, der var nemmest at implementere.

For alle programmer bortset fra CPLEX og HFC er tiderne fundet som gennemsnittet ved 100 kørsler. For CPLEX er oplyst tiden ved en kørsel og for HFC gennemsnitstid v. 10 kørsler.

Resultaterne af kørsler af disse programmer kan ses i tabel 6.1 og tabel6.2.

Tegn i tabellen:

- * løsningen er optimal.
- ** det resultat CPLEX nåede til da den meddelte 'out of memory'.
- \- værdien er ikke blevet fundet eller tiden kan ikke umiddelbart sammenlignes med resten, da løsningen er blevet fundet på en anden computer.

De nævnte afvigelser er for openshop den procentvise afvigelse fra C^*_{max} eller hvis denne er ukendt afvigelsen fra LB. For flowshop vises den procentvise afvigelse fra den bedste kendte løsning (dvs. den bedste af de 3 fundne).

Bemærkninger til openshop programmerne.

Som det kan ses er CPLEX bedst til løsning af små openshopproblemer når det gælder objektfunktionsværdien, mht. den tid det tager at opnå en løsning er både H1-open og SA hurtigere, men for det meste med dårligere løsninger. Når det kommer til de større problemer er CPLEX imidlertid nødt til at give op pga. den størrelse Branch & Bound træet kommer op på. For de større problemer er både SA og H1-open hurtige, og selv om loesningernes kvalitet er svingende, er de rimelige for H1 og selv om de ikke er særlig gode for Simuleret Udgloedning kan dette skyldes at der ikke er justeret nok på parametrene.

Openshop										
		CPLEX			H1			SA		
	LB	C_{max}	Afv. (s)	Tid	C_{max}	Afv. (ms)	Tid	C_{max}	Afv. (ms)	Tid
5x 5 - 1	401*	401	0	1,87	450	12,2	0,12	449	12	0,23
5x 5 - 2	381*	381	0	2,95	385	1,1	0,14	-	-	-
5x 5 - 3	343*	343	0	11,57	343	0,0	0,15	353	3	0,19
5x 5 - 4	354*	354	0	4,47	419	16,1	0,16	411	16	0,27
5x 5 - 5	390*	390	0	2,83	390	0,0	0,30		-	-
10x10 - 1	711	933**	31	6329	733	3,1	0,75	-	-	-
10x10 - 2	551	957**	73	-	596	8,2	1,32	596	8,2	0,14
10x10 - 3	691	-	-	-	722	4,5	1,61	722	4,5	0,26
10x10 - 4	637	-	-	-	651	2,2	0,76	651	2,2	0,25
10x10 - 5	707	-	-	-	754	6,7	0,30	-	-	-
20x20 - 1	1266	-	-	-	1319	4,2	4,81	1361	7,5	17,1
20x20 - 2	1219	-	-	-	1288	1,7	5,28	1448	14,4	20,3
20x20 - 3	1248	-	-	-	1333	5,3	4,81	1400	10,6	-
20x20 - 4	1183	-	-	-	1215	2,7	5,12	-	-	-
20x20 - 5	1286	-	-	-	1344	4,5	5,12	1409	9,6	-

Tabel 6.1: Resultater v. løsning af openshop-problemer

Flowshop	Flowshop								
Flowshop	CPLEX			HFC			H1		
Sæt nr.	C_{max}	Afv.	Gns. tid(ms)	C_{max}	Afv.	Gns. tid	C_{max}	Afv.	Gns. tid
5x 5 - 1	525*	0	0,37	525	0	0,19	593	13	-
5x 5 - 2	514*	0	0,42	556	6	0,25	514	0	-
5x 5 - 3	426*	0	0,18	490	15	0,03	455	7	-
5x 5 - 4	450*	0	0,11	459	2	0,08	459	2	-
5x 5 - 5	574*	0	0,36	574	0	0,22	574	0	-
10x10 - 1	1262**	2,27	4409	1234	0	0,53	1253	1,54	1,71
10x10 - 2	957**		2071	958	0,74	0,39	951	0	1,24
10x10 - 3	-	-	-	1111	0	0,50	1209	8,82	2,64
10x10 - 4	-	-	-	1118	0	0,33	1132	1,25	-
10x10 - 5	-	-	-	1214	6,49	0,57	1140	0	-
20x20 - 1	-	-	-	2475	0	3,88	2567	3,72	31,1
20x20 - 2	-	-	-	2548	0,67	4,04	2531	0	59,3
20x20 - 3	-	-	-	2503	0	3,88	2570	2,68	58,1
20x20 - 4	-	-	-	2429	1,89	4,19	2384	0	36,3
20x20 - 5	-	-	-	2626	2,42	4,19	2543	0	47,2

Tabel 6.2: Resultater v. løsning af flowshop-problemer

Bemærkninger til flowshop programmerne.

Som det kan ses er CPLEX bedst til løsning af små flowshopproblemer både mht. objektfunktionsværdien og mht. den tid det tager at opnå en løsning. Når det kommer til de større problemer er CPLEX imidlertid nødt til at give op pga. den størrelse Branch & Bound træet kommer op på. For de større problemer er HFC hurtig og giver rimelig gode løsninger, mens H1 giver gode, men forholdsvis langsomme resultater.

Kapitel 7

Konklusion

I denne tekst er der blevet set på forskellige metoder til løsning af openshop- og flowshopproblemer. Fire af disse metoder er blevet implementeret og afprøvet på et antal testproblemer. Ud fra resultaterne af disse testkørsler kan det ses at CPLEX, som kombinerer Lineær Programmering med Branch & Bound, er bedst til små problemer pga. sin nøjagtighed. CPLEX giver altid optimale løsninger, men kræver mere hukommelse for computeren end de andre programmer og tager desuden mere tid. Ved små problemer er dette ikke et problem, men ved større problemer løber computeren tør for hukommelse og det tager desuden meget lang tid. De andre metoder, der er blevet gennemgået er alle heuristikker, som på forholdsvis kort tid finder en løsning tæt på den optimale. Af disse er H1 den bedste til openshop, med mindre man tager sig tiden til at justere på parametrene til de bliver helt rigtige. Til flowshop vil jeg anbefale HFC, idet den giver hurtigere løsninger end H1 til flowshop og da løsningerne fra de to metoder er omtrent lige gode.

De ovennævnte metoder er ifølge de fremkomne resultater de bedste af de implementerede metoder givet de nævnte begrænsninger. Det bør dog nævnes at det under udarbejdningen af denne tekst blev klart at det også kunne være interessant at implementere nogle af de andre gennemgåede alternativer, selvom disse ikke ser videre lovende ud på forhånd. F.eks. kunne Simuleret Udglødning også implementeres for flowshop. Dette ligger dog udenfor tidsrammen for dette projekt.

Appendiks A

Notation

C_{max}	Objektfunktionsværdi for den bedste kendte løsning.
C^*_{max}	Objektfunktionsværdi for den optimale løsning.
$C_{max}(U)$	Objektfunktionsværdi for den fuldstændige udvælgelse U.
d_i	Forfaldstidspunkt for operation i.
D_J	Mængde af kanter, der forbinder par af operationer, som tilhører samme job.
D_M	Mængde af ikke-orienterede kanter, der forbinder par af operationer, som behandles på samme maskine.
E	Mængde af operationer, der kan være input til blok.
$EJ(i,v)$	Efterfølgeren til operation (i,v) på job i.
$EM(i,v)$	Efterfølgeren til operation (i,v) på maskine v.
F	Mængde af operationer, der kan være output fra en blok.
$FJ(i,v)$	Forgænger til operation (i,v) på job i.
$FM(i,v)$	Forgænger til operation (i,v) på maskine v.
G	En graf.
$G(U)$	En graf for U.
$H1_{liste}$	Liste over prioritering af endnu ikke planlagte operationer i $H1$-algoritmen.
$H1_{set}$	Liste over operationer der skal opprioriteres i $H1$-algoritmen.
(i,v)	Operation i på maskine v.

L	"straf" for at bryde begrænsning.
LB	Nedre grænse.
m	Antallet af maskiner i problemet.
m_j	Antal operationer, som maskine j skal lave.
M	Mængden af maskiner i problemet.
M_j	Mængden af operationer, som skal laves på maskine j.
n	Antal jobs i problemet.
n_i	Antal operationer i job i.
N	Mængden af jobs i problemet.
N_i	Mængden af operationer, der skal laves på job i.
o	Antal operationer, der indgår i problemet.
o_{ij}	Operation i job i på maskine j.
O	Mængden af operationer i problemet.
p_i	Procestid for operation i.
$P_{m,n}$	Matrix med alle procestiderne.
q_i	Hale for operation i.
r_i	Hoved for operation i.
S	Mængde af orienterede kanter, også kaldt et valg.
t	Tidsoperator.
T	Temperatur.
$TG(a,b,c,d)$	Todelt graf, a og b er punktmængder, c er en kantmængde og d er en matrix med kanternes vægte.
TG_{kant}	Mængde af kanter i en todelt graf.
TG_N	Mængde af punkter. Hvert punkt er et job, der er derfor n punkter i mængden.
TG_M	Mængde af punkter. Hvert punkt er en maskine, der er derfor m punkter i mængden.
t_i	Starttidspunkt for operation i.
U	En fuldstændig udvælgelse.
UB	øvre grænse.
V	Mængde af knuder i graf.
y_{ij}	Binær variabel lig 1 for $t_i < t_j$.
Z	Længden af en længste vej.
Δ_{ij}	Difference mellem nuværende og tidligere objektfunktionsværdi.
μ_i	Den maskine, der skal udføre operation i.

Appendiks B

Openshop programmer

B.1 H1

```
/* Program der implementerer den dynamiske H1-heuristik
   foreslaaet af Gueret og Prins '98
   Begyndt d.5.April 2000. Sidst AEndret d. 30.November 2000.

   Definer MaxJ, MaxM, MaxOP og P[]
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MaxJ   5          /*Antal job                      */
#define MaxM   5          /*Antal maskiner                 */
#define MaxOp 25          /*Antal operationer i alt ( M*J )*/

int J = MaxJ;
int M = MaxM;

/* Procestider                                             */
```

```
int P[MaxOp] = {
 54,  34,  61,   2,   9,
 39,  66,  89,  91,  53,
 27,  46,  28,  40,  64,
 98,  74,  96,  53,  80,
 73,  93,  50,   3,  49};

/*Tidligste start for operationer (hoveder) ved open shop
  saettes alle til 0                                      */

int hoved[MaxOp];

/*Hvilken maskine laver hvilken operation                 */

int MO[MaxOp];

void Orden()
{
   int i,j;
   int PKopi[MaxOp];

   for(i=0; i<M*J; i++)
     PKopi[i]=P[i];

  for(i=0; i<J; i++)
    for (j=0; j<M; j++)
      P[i*M + MO[i*M+j] - 1]=PKopi[i*M+j];

}


void SkemaByg( int H1Liste[2][MaxOp], int Skema[2][MaxOp] )
{
  int i, j;
  int MinHovedPos = 0, MinHoved = 0;

  /*Finder den operation, der skal skemalaegges           */
  for (j=0; j<J*M; j++)
  {
```

```
MinHoved = hoved[H1Liste[1][0]-1];
MinHovedPos = 0;
for (i=0; i<J*M; i++)
{
  if (hoved[H1Liste[1][i]-1]
     < MinHoved && H1Liste[1][i] != -1)
  {
    MinHoved = hoved[H1Liste[1][i]-1];
    MinHovedPos = i;
  }
}

/*Skemalaegger den valgte operation                    */
Skema[0][j] = H1Liste[1][MinHovedPos];
Skema[1][j] = MinHoved;

/*Sletter den valgte operation fra h1-listen og       */
for (i=MinHovedPos; i<J*M-1;i++) /*trunkerer denne,   */
{              /*saetter ikke benyttede pladser til -1*/
  H1Liste[0][i] = H1Liste[0][i+1];
  H1Liste[1][i] = H1Liste[1][i+1];
  H1Liste[0][i+1] = -1;
  H1Liste[1][i+1] = -1;
}

for (i=0; i<J; i++)               /*Opdaterer hovederne*/
{
  if (i!=(Skema[0][j]-1)/M)       /*Foerst lodret      */
  {
    if (hoved[(Skema[0][j]-1)%M+i*M]
         <hoved[Skema[0][j]-1] + P[Skema[0][j]-1])
      hoved[(Skema[0][j]-1)%M+i*M]
          = hoved[Skema[0][j]-1] + P[Skema[0][j]-1];
  }
}

for (i=0; i<M; i++)               /*Og saa vandret     */
{
  if (i!=(Skema[0][j]-1)%M)
```

```
      {
        if (hoved[((Skema[0][j]-1)/M)*M+i]
             < hoved[Skema[0][j]-1] + P[Skema[0][j]-1])
          hoved[((Skema[0][j]-1)/M)*M+i]
              = hoved[Skema[0][j]-1] + P[Skema[0][j]-1];
      }
    }
  }
  for(i=0; i<M*J; i++) /*Nulstiller hoveder(skal aendres   */
    hoved[i]=0;        /*hvis ikke openshop)              */
}

void SetByg(int LB, int Skema[2][MaxOp], int H1set[MaxOp])
{
  int i,j,k,l;
  int MaskinListe[2][MaxOp], Hul[2];

  k = 0;
  for (i=0; i<M; i++) /*Finder operationernes raekkefoelge */
  {                   /*paa maskinerne                     */
    for (j=0; j<J*M; j++)
    {
      if ((Skema[0][j]-1)%M == i)
      {
        MaskinListe[0][i*J+k] = Skema[0][j];
        MaskinListe[1][i*J+k] = Skema[1][j];
        k++;
      }
    }
    k = 0;
  }

  for(k=0; k<J-1; k++)            /*Finder H1-set          */
  {
    for(i=0; i<M; i++)            /*Foerst findes hullerne */
    {
    Hul[0] = MaskinListe[1][k+i*J]
          + P[MaskinListe[0][k+J*i]-1]; /*starttid for hul */
    Hul[1] = MaskinListe[1][k+i*J+1];   /*sluttid for hul  */
```

```
    if ( Hul[0] < Hul[1] )
    {
     for( j=0; j<M; j++)
     {
      if ( j != i )
      {
       /*Finder operationer der bearbejdes mens der er hul */
       for ( l=0; l<J; l++)
       {
        if((MaskinListe[1][j*J+l]<Hul[0]
          && P[MaskinListe[0][j*J+l]-1]
               +MaskinListe[1][ j*J+l]>Hul[0])
          || (MaskinListe[1][j*J+l]<Hul[1]
          && P[MaskinListe[0][j*J+l]-1]
               +MaskinListe[1][ j*J+l]>Hul[1]))
        {
         /*=1 hvis operationen skal prioriteres hoejere    */
         H1set[MaskinListe[0][j*J+l]-1] = 1;
        }
       }
      }
     }
    }
   }
  }
/*Finder de operationer der bearbejdes efter den nedre     */
  for ( i = 0; i < M*J; i++)          /* graense er naaet  */
  {
    if (MaskinListe[1][i] + P[MaskinListe[0][i]-1] > LB)
      H1set[MaskinListe[0][i]-1] = 1;
  }
}

int Sammenlign(int Skema[2][MaxOp], int BedstSkema[2][MaxOp])
{
  int i, tmp = 1;

  for (i = 0; i < M*J; i++)
    if (Skema[0][i]!=BedstSkema[0][i]
```

```
        ||Skema[1][i]!=BedstSkema[1][i])
      tmp = 0;

  return tmp;
}

void main2()
{
  int Nummer = 1, MaxRest = 0, Raekke = 1, Position = 1;
  int RestMaskintid[MaxOp], MaskinSum[MaxM];
  int Skema[2][MaxOp], BedstSkema[2][MaxOp];
  int H1set[MaxOp], GammelSkema[2][MaxOp];
  int H1Liste[2][MaxOp], H1ListeKopi[MaxOp], AntalIt=1;
  int JobSum[MaxJ], LB = 0, SlutTid = 0, BedstSlutTid = -1;
  int i, j, k, Temp;

  char name[13]="H1_out.txt";
  FILE *fid;
  fid = fopen(name,"w");

  for(i=0; i<M*J; i++)
hoved[i]=0;

for(i=0; i<J; i++)
for(j=0; j<M; j++)
MO[i*M+j]=j+1;

  Orden();

  for(i=0; i<M; i++)
   MaskinSum[i]=0;

  /*Finder den nedre graense for hver maskine              */
  for (i=0; i<M; i++)
  {
    for (j=0; j<J; j++)
    {
      MaskinSum[i] += P[j*M+i];
    }
```

```
  }

  for (i=0; i<M; i++)   /*Finder den stoerste af disse     */
  {
    if (MaskinSum[i] > LB)
      LB = MaskinSum[i];
  }

  for(i=0; i<J; i++)
   JobSum[i]=0;

  /*Finder den nedre graense for hvert job                 */
  for (i=0; i<J; i++)
  {
    for (j=0; j<M; j++)
    {
      JobSum[j] += P[j*M+i];
    }
  }


  for (i=0; i<J; i++)   /*Finder den nedre graense LB      */
  {
    if (JobSum[i] > LB)
      LB = JobSum[i];
  }

  for(i=0; i<J*M; i++) /*Finder den minimale resterende   */
  {                     /*arbejdstid for hver maskine      */
    RestMaskintid[i]=0;
    for(j=0; j<J; j++)
      if( j!=i/M )
        RestMaskintid[i] = RestMaskintid [i] + P[i%M +j*M];
  }

  for(i=0; i<J*M; i++) /*Nulstiller/initialiserer H1listen*/
  {
    H1Liste[0][i]=0;
H1set[i]=0;
```

```
}

for(i=0; i<J*M; i++)    /*Laver H1-listen                */
{
  j = 0;

  /*Her findes den position i H1-listen hvor en operation*/
  while(RestMaskintid[i]<H1Liste[0][j]) /*skal indsaettes*/
    j++;

  k = j;
  /*Her undersoeges det hvilken af to operationer med    */
  while (RestMaskintid[i] == H1Liste[0][j] && k == j )
  {                                 /*samme restmaskintid*/
    if (P[i] <= P[H1Liste[1][j]-1]) /*der skal staa      */
      j++;                          /*foerst  paa listen */
    k++;
  }

  /*Flytter allerede indsatte operationer efter denne    */
  for(k=i-1; k>=j; k--) /*position en plads ned ad listen*/
  {                     /*(saa der opstaar et hul)       */
    if(k+1!=J*M)
    {
      H1Liste[0][k+1] = H1Liste[0][k];
      H1Liste[1][k+1] = H1Liste[1][k];
    }
  }

  /*Indsaetter den nye operation paa sin plads           */
  H1Liste[0][j] = RestMaskintid[i];
  H1Liste[1][j] = i+1;
}

for(i=0; i<M; i++)
SlutTid=0;
for (i = 0; i<M; i++)
  SlutTid += MaskinSum[i];
```

```
do
{
  /*Laver en kopi af listen til senere brug              */
  for (i=0; i<J*M; i++)
    H1ListeKopi[i]=H1Liste[1][i];
   SkemaByg(H1Liste, Skema);

  /*Opdaterer bedste kendte loesning                     */
  if (BedstSlutTid == -1 || SlutTid<= BedstSlutTid)
  {
    BedstSlutTid = SlutTid;
    for (i=0; i<M*J; i++)
    {
      BedstSkema[0][i] = Skema[0][i];
      BedstSkema[1][i] = Skema[1][i];
    }
  }

  for (i=0; i<M*J; i++)
  {
    GammelSkema[0][i] = Skema[0][i];
    GammelSkema[1][i] = Skema[1][i];
  }

  /*SkemaByg(H1Liste, Skema);   */

  SetByg(LB, Skema, H1set);

  SlutTid = 0;
  for (i=0; i<M*J; i++)  /*Finder Cmax                  */
  {
    if (SlutTid < Skema[1][i] + P[Skema[0][i]-1])
      SlutTid = Skema[1][i] + P[Skema[0][i]-1];
  }

  fprintf(fid,"SlutTid [%d] = %d\n",AntalIt, SlutTid);

  k=0;
```

```
  for (i=0; i<M*J; i++)  /*Finder ny H1-liste            */
  {
    if (H1set[H1ListeKopi[i]-1] == 0 && k == 0)
    {
      Temp = H1ListeKopi[i];
      k=1;
    }
    else if (H1set[H1ListeKopi[i]-1] == 0 && k == 1)
         {
           H1Liste[1][i-1] = Temp;
           Temp = H1ListeKopi[i];
         }
    else if (H1set[H1ListeKopi[i]-1]==1 && k==0)
         {
           H1Liste[1][i] = H1ListeKopi[i];
         }
    else if (H1set[H1ListeKopi[i]-1]==1 && k==1)
         {
           H1Liste[1][i-1] = H1ListeKopi[i];
         }
  }
  if (H1Liste[1][M*J-1]==-1)
    H1Liste[1][M*J-1] = Temp;

  AntalIt++;
}while(Sammenlign(Skema, GammelSkema) == 0
   && AntalIt < 1000 );
 /*Kan evt. supp. med afstand fra bedst kendte loesning  */

fprintf(fid,"Bedste Sluttid %d\n SlutTid=%d\n",
                BedstSlutTid, SlutTid);

fprintf(fid,"LB = %d\n", LB);

for(i=0; i<J*M; i++)\
  fprintf(fid,"Bedste Skema[%d] = (%d, %d)\n",i,
                BedstSkema[0][i], BedstSkema[1][i]);

fprintf(fid,"AntalIt = %d\n",AntalIt);
```

```
   printf("Cmax=%d, LB=%d \n", BedstSlutTid, LB);
  fclose(fid);
}

void main()
{
  time_t Tidstart,Tidslut, Tid;
int i,j, Antal=100;
for (j = 0;j<1;j++)
{
Tidstart=clock();
for(i=0; i<Antal; i++)
  main2();
Tidslut = clock();
Tid= (Tidslut - Tidstart)*1000/CLOCKS_PER_SEC;
printf("Tid = %i ved %d koersler \n",Tid, Antal);
}
}
```

B.2 Simuleret Udgloedning

```
/* Program der implementerer heuristikken simuleret
   udgloedning for et openshopproblem

   Begyndt d. 17.Maj 2001. Sidst AEndret d. 1.September 2001.

   Definer Antal job MaxJ, antal maskiner MaxM, antal
   operationer MaxOP, starttemperatur T0, min_pct,
   str_faktor, red_faktor, procestiderne P[], operationernes
   tidligste starttidspkt. Eventuelt hoved[] og
   hvilken maskine, der behandler hvilken operation M0[]
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

#define MaxJ   5          /*Antal job                      */
#define MaxM   5          /*Antal maskiner                 */
#define MaxOp 25          /*Antal operationer i alt ( M*J )*/

double T = 100;
int max_tal =   5;
double min_pct = 0.9;
double str_faktor = 0.2;
double red_faktor = 0.995;

int J = MaxJ;
int M = MaxM;

/* Procestider                                             */

int P[MaxOp] = {
54,  34,  61,   2,   9,
 39,  66,  89,  91,  53,
 27,  46,  28,  40,  64,
 98,  74,  96,  53,  80,
```

```
 73,  93,  50,   3,  49};

/*Tidligste start og slut for operationer                  */

int hoved[MaxOp+1];
int hale[MaxOp];
int Liste[2][MaxOp];
int Mliste[2][MaxOp];

/*Hvilken maskine laver hvilken operation                  */

int MO[MaxOp];

/*Operationer paa den kritiske vej*/
int KritVej[2][MaxOp];

void Orden()
{
  int PKopi[MaxOp];
  int i, j;

  for(i=0; i<M*J; i++)
    PKopi[i]=P[i];

  for(i=0; i<J; i++)
{
    for (j=0; j<M; j++)
      P[i*M + MO[i*M+j] - 1]=PKopi[i*M+j];
}
}

int Minimum()
{
  int MaskinSum[MaxM], JobSum[MaxJ], LB=0;
  int i, j;

  for (i=0; i<M; i++) /*Nulstiller maskinsum     */
    MaskinSum[i] = 0;
```

```
  for (i=0; i<J; i++) /*Nulstiller jobsum        */
    JobSum[i] = 0;

  /*Finder nedre graense for hver maskine                  */
  for (i=0; i<M; i++)
  {
    for (j=0; j<J; j++)
    {
      MaskinSum[i] += P[j*M+i];
    }
  }

  for (i=0; i<M; i++)   /*Finder den stoerste af disse     */
  {
    if (MaskinSum[i] > LB)
      LB = MaskinSum[i];
  }

  /*Finder den nedre graense for hvert job                 */
  for (i=0; i<J; i++)
  {
    for (j=0; j<M; j++)
    {
      JobSum[j] += P[j*M+i];
    }
  }

  for (i=0; i<J; i++)   /*Finder den nedre graense LB      */
  {
    if (JobSum[i] > LB)
      LB = JobSum[i];
  }
  return LB;
}

void LTRPOM(int RestTid[MaxOp])
{
int tjob[MaxJ], tmaskine[MaxM], JobNr=0, Antal=M*J;
```

```
int i, j, k, a, b, OpNr, t=0, RestT=0;

for(i=0; i<J; i++)
tjob[i]=0;

for(i=0; i<M; i++)
tmaskine[i]=0;

do
{
for(i=0; i<J; i++)
{
      for(j=0; j<M; j++)
{
  if(tjob[i]<=0 && tmaskine[j]<=0)
{
if(RestTid[i*M+j]>RestT)
{
  RestT=RestTid[i*M+j];
  OpNr=i*M+j;
a=i; b=j;
}
}
}
}

    if(RestT!=0)
{
  hoved[OpNr]=t;
RestTid[OpNr]=-1;
  tjob[OpNr/M]=P[OpNr];
  tmaskine[OpNr%M]=P[OpNr];
  RestT=0;
Antal-=1;
      Liste[1][a*M+b]=JobNr;

JobNr+=1;
}
  else
```

```
{
for(k=0; k<J; k++)
  tjob[k]-=1;

  for(k=0; k<M; k++)
  tmaskine[k]-=1;

  t++;
}
}while(Antal>0);
}

int Partition(int I[2][MaxOp], int p, int r)
{
int x = I[1][p - 1];
int i = p - 1;
int j = r + 1;
int tmp, tmp2;
int keyl = p;

while (i < j)
{
do
{
j--;
} while (I[1][j - 1] > x);
do
{
i++;
} while (I[1][i - 1] < x);
if (i < j)
{
tmp = I[1][i - 1];
tmp2 = I[0][i - 1];
I[1][i - 1] = I[1][j - 1];
I[0][i - 1] = I[0][j - 1];
I[1][j - 1] = tmp;
I[0][j - 1] = tmp2;
}
```

```
}
return j;
}

void QuickSort( int I[2][MaxOp], int p, int r )
{
int q;
if (p < r)
{
q = Partition(I,p,r);
QuickSort(I,p,q);
QuickSort(I,q+1,r);
}
}

int FindVej(int Cmax)
{
int i, j=0;

for(i=0; i<M*J; i++)
{
if(hoved[i]+P[i]+hale[i]==Cmax)
{
KritVej[0][j]=i;
j++;
}
}
return(j);
}

int FindPunkt(int Punkt[2][MaxOp], int Laengde)
{
int j=0, OpNr=0, link=2;

Punkt[0][0]=KritVej[0][0];
Punkt[1][0]=0;
Punkt[0][1]=KritVej[0][1];
Punkt[1][1]=1;
if(Punkt[0][0]/M==Punkt[0][1]/M)
```

```
link=0; /*link=0 => samme job*/
else
link=1; /*link=1 => samme maskine*/

j=1;
for(OpNr=2; OpNr<Laengde; OpNr++)
{
if(KritVej[0][OpNr]%M==Punkt[0][j]%M && link==0)
j++;
else if(KritVej[0][OpNr]/M==Punkt[0][j]/M && link==1)
j++;
else;

Punkt[0][j]=KritVej[0][OpNr];
Punkt[1][j]=OpNr;
}
return(j);
}

void updhov(int element, int cmax, int ny_hoved[MaxOp+1])
{
int t,p,q,c = element,i;

if (ny_hoved[Liste[0][c]] == -1)
{
if (c%M > 0)
{
updhov(c-1,cmax,ny_hoved);
p = ny_hoved[Liste[0][c-1]]+P[Liste[0][c-1]];

for (i = 0; i <M*J; i++)
{
if (Mliste[0][i] == Liste[0][c])
t = Mliste[0][i - 1];
}

for (i = 0; i <M*J; i++)
{
if (Liste[0][i] == t)
```

```
{
t = i;
i=M*J;
}
}

updhov(t,cmax,ny_hoved);
q = ny_hoved[Liste[0][t]]+P[Liste[0][t]];

if(p>q)
ny_hoved[Liste[0][c]] = p;
else
ny_hoved[Liste[0][c]] = q;
}
else
ny_hoved[Liste[0][c]] = 0;
}
}

void updhal(int element, int ny_hale[MaxOp+1])
{
int t,p,q,c = element,i;

if (ny_hale[Liste[0][c]] == -1)
{
if (c%M != J - 1)
{
updhal(c+1,ny_hale);
p = ny_hale[Liste[0][c+1]]+P[Liste[0][c+1]];
}
else
p = -1;

t = -1;
for (i = 0; i <M*J; i++)
{
if (Mliste[0][i]==Liste[0][c]
  && Liste[0][c]%M==Mliste[0][(i+1)]%M)
t = Mliste[0][i + 1];
```

```
}
if (t != -1)
{
for (i = 0; i <M*J; i++)
{
if (Liste[0][i] == t)
{
t = i;
i=M*J;
}
}

updhal(t,ny_hale);
q = ny_hale[Liste[0][t]]+P[Liste[0][t]];
}
else
q = -1;

if(p>q)
ny_hale[Liste[0][c]] = p;
else if(q>p)
ny_hale[Liste[0][c]] = q;
else if(p == -1 && q == -1)
ny_hale[Liste[0][c]] = 0;
else
ny_hale[Liste[0][c]] = p;
}
}

int Nabo(int OpNr, int NaboType, int Cmax, int VejLaengde,
     double R)
{
  int ny_hale[MaxOp], ny_hoved[MaxOp], legal=1, ny_Cmax;
  int EMj, EJi, EJj, FMi, FJj, FJi; /*samme maskine*/
  int EMFJj, FMEJi, EJEJi, EMEJi, FMFJj, FJFJj;
  /*ekstra v samme job*/
  int EMi, FMj, EJFMj, FJEMi, EMEMi, EJEMi, FJFMj, FMFMj;
  int temp_a1, temp_a2, temp_a3, temp_b1, temp_b2, temp_b3;
  int temp_c1, temp_c2, temp_c3, temp_d1, temp_d2, temp_d3;
```

```
  int temp_e1, temp_e2, temp_e3, temp_f1, temp_f2, temp_f3;
  int i, j, k; /*taellere*/

for(i=0; i<M*J; i++)
{
ny_hoved[i]=0;
ny_hale[i]=0;
}/*end of for*/

switch(NaboType)
{
case 0: /*(i,j)*/
{
if(OpNr==0)
{
i=KritVej[0][0];
j=KritVej[0][1];
}
else if(OpNr==VejLaengde)
{
i=KritVej[0][VejLaengde-2];
j=KritVej[0][VejLaengde-1];
}
else
{
if(OpNr%2==0)
{
i=KritVej[0][OpNr-1];
j=KritVej[0][OpNr];
}
else
{
i=KritVej[0][OpNr];
j=KritVej[0][OpNr+1];
}
}

if(i%M==j%M)/*samme maskine*/
{
```

```
EJi=M*J;
hoved[EJi]=Cmax;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]>hoved[i] && hoved[EJi]>hoved[k])
EJi=k;
}

FJj=M*J;
   hoved[FJj]=0;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]<hoved[j] && hoved[FJj]<hoved[k])
FJj=k;
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<i%M; k++)
{
if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
FMi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
EJj=k;
}

EMj=M*J;
hoved[EJj]=Cmax;
for(k=j%M; k<j%M; k++)
{
if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
EMj=k;
}
```

```
FJi=M*J;
   hoved[FJi]=0;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
FJi=k;
}

/*Obj.vaerdi findes hvis lovlig*/
if(FJj!=M*J && FMi!=M*J && FJi!=M*J && EJi!=M*J && EMj!=M*J
&& EJj!=M*J)
{
if(hoved[FJj]+P[FJj]>hoved[FMi]+P[FMi])
  ny_hoved[j]=hoved[FJj]+P[FJj];
   else if(hoved[FJj]+P[FJj]<=hoved[FMi]+P[FMi])
   ny_hoved[j]=hoved[FMi]+P[FMi];

if(hoved[FJi]+P[FJi]>ny_hoved[j]+P[j])
   ny_hoved[i]=hoved[FJi]+P[FJi];
  else
   ny_hoved[i]=ny_hoved[j]+P[j];

   if(hale[EJi]+P[EJi]>hale[EMj]+P[EMj])
   ny_hale[i]=hale[EJi]+P[EJi];
  else if(hale[EJi]+P[EJi]<=hale[EMj]+P[EMj])
   ny_hale[i]=hale[EMj]+P[EMj];

  if(ny_hale[i]+P[i]<=hale[EJj]+P[EJj])
  ny_hale[j]=hale[EJj]+P[EJj];
   else
   ny_hale[j]=ny_hale[i]+P[i];

/*ny_Cmax=*/
if(ny_hoved[i]+P[i]+ny_hale[i]>=
                  ny_hoved[j]+P[j]+ny_hale[j])
  ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
else
  ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];
```

```
if(ny_Cmax<Cmax ||R<1)
{
for(k=0; k<M*J; k++)
{
if(Mliste[0][k]==i)
{
temp_a1=k; temp_a2=i; temp_a3=Mliste[1][k];
}
if(Mliste[0][k]==j)
{
temp_b1=k; temp_b2=j; temp_b3=Mliste[1][k];
}
}
Mliste[0][temp_a1]=temp_b2;
Mliste[1][temp_a1]=temp_b3;
Mliste[0][temp_b1]=temp_a2;
Mliste[1][temp_b1]=temp_a3;
}

if(ny_Cmax>=Cmax && R<1)
ny_Cmax=0;
else if(ny_Cmax<Cmax)
ny_Cmax=1;
else
ny_Cmax=-1;
}
else
ny_Cmax=-1;
}
else/*samme job*/
{
EMi=M*J;
hoved[EMi]=Cmax;
for(k=i%M; k<M*J; k=k+M)
{
if(hoved[k]>hoved[i] && hoved[EMi]>hoved[k])
EMi=k;
}
```

```
FMj=M*J;
hoved[FMj]=0;
for(k=j%M; k<M*J; k=k+M)
{
if(hoved[k]<hoved[j] && hoved[FMj]<hoved[k])
FMj=k;
}

EMj=M*J;
hoved[EMj]=Cmax;
for(k=j%M; k<M*J; k=k+M)
{
if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
EMj=k;
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<M*J; k=k+M)
{
   if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
FMi=k;
}

FJi=M*J;
hoved[FJi]=0;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
FJi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
EJj=k;
```

```
}

/*Obj.vaerdi findes, hvis den er legal*/
if(FMj!=M*J && FJi!=M*J && FMi!=M*J && EMi!=M*J && EJj!=M*J
&& EMj!=M*J)
{
if(hoved[FMj]+P[FMj]>hoved[FJi]+P[FJi])
  ny_hoved[j]=hoved[FMj]+P[FMj];
   else if(hoved[FMj]+P[FMj]<=hoved[FJi]+P[FJi])
   ny_hoved[j]=hoved[FJi]+P[FJi];

  if(hoved[FMi]+P[FMi]>ny_hoved[j]+P[j])
   ny_hoved[i]=hoved[FMi]+P[FMi];
  else
   ny_hoved[i]=ny_hoved[j]+P[j];

  if(hale[EMi]+P[EMi]>hale[EJj]+P[EJj])
   ny_hale[i]=hale[EMi]+P[EMi];
  else if(hale[EMi]+P[EMi]<=hale[EJj]+P[EJj])
   ny_hale[i]=hale[EJj]+P[EJj];

  if(ny_hale[i]+P[i]<=hale[EMj]+P[EMj])
  ny_hale[j]=hale[EMj]+P[EMj];
  else
  ny_hale[j]=ny_hale[i]+P[i];

/*ny_Cmax=*/
if(ny_hoved[i]+P[i]+ny_hale[i]>=
                        ny_hoved[j]+P[j]+ny_hale[j])
  ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
else
  ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];

if(ny_Cmax<Cmax || R<1)
{
for(k=0; k<M*J; k++)
{
if(Liste[0][k]==i)
{
```

```
temp_a1=k; temp_a2=i; temp_a3=Liste[1][k];
}
if(Liste[0][k]==j)
{
temp_b1=k; temp_b2=j; temp_b3=Liste[1][k];
}
}
Liste[0][temp_a1]=temp_b2;
Liste[1][temp_a1]=temp_b3;
Liste[0][temp_b1]=temp_a2;
Liste[1][temp_b1]=temp_a3;
}

if(ny_Cmax>=Cmax && R<1)
ny_Cmax=0;
else if(ny_Cmax<Cmax)
ny_Cmax=1;
else
ny_Cmax=-1;
}
else
ny_Cmax=-1;
}/*end of if samme job*/
};break;
case 1: /*(i,j), (i,EJi)||  (a,b), (a,EM)*/
{
if(OpNr==0)
{
i=KritVej[0][0];
j=KritVej[0][1];
}
else if(OpNr==VejLaengde)
{
i=KritVej[0][VejLaengde-2];
j=KritVej[0][VejLaengde-1];
}
else
{
if(OpNr%2==0)
```

```
{
i=KritVej[0][OpNr-1];
j=KritVej[0][OpNr];
}
else
{
i=KritVej[0][OpNr];
j=KritVej[0][OpNr+1];
}
}

if(i%M==j%M)/*samme maskine*/
{
EJi=M*J;
hoved[EJi]=Cmax;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]>hoved[i] && hoved[EJi]>hoved[k])
EJi=k;
}

FJj=M*J;
   hoved[FJj]=0;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]<hoved[j] && hoved[FJj]<hoved[k])
FJj=k;
}

EMEJi=M*J;
hoved[EMEJi]=Cmax;
for(k=(EJi/M)*M; k<(EJi/M)*M+M; k++)
{
if(hoved[k]>hoved[EJi] && hoved[EMEJi]>hoved[k])
EMEJi=k;
}

FMEJi=M*J;
hoved[FMEJi]=0;
```

```
for(k=(EJi/M)*M; k<(EJi/M)*M+M; k++)
{
if(hoved[k]<hoved[EJi] && hoved[FMEJi]<hoved[k])
FMEJi=k;
}

EJEJi=M*J;
hoved[EJEJi]=Cmax;
for(k=(EJi/M)*M; k<(EJi/M)*M+M; k++)
{
if(hoved[k]>hoved[EJi] && hoved[EJEJi]>hoved[k])
EJEJi=k;
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<i%M; k++)
{
if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
FMi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
EJj=k;
}

EMj=M*J;
hoved[EJj]=Cmax;
for(k=j%M; k<j%M; k++)
{
if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
EMj=k;
}

FJi=M*J;
```

```
   hoved[FJi]=0;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
FJi=k;
}

/*Obj.vaerdi findes hvis lovlig*/
if(EJi!=M*J && FJj!=M*J && FMi!=M*J && FJi!=M*J && FMEJi!=M*J
&& EJEJi!=M*J && EMj!=M*J && EJj!=M*J && EMEJi!=M*J)
{
if(hoved[FJj]+P[FJj]>hoved[FMi]+P[FMi])
  ny_hoved[j]=hoved[FJj]+P[FJj];
   else if(hoved[FJj]+P[FJj]<=hoved[FMi]+P[FMi])
   ny_hoved[j]=hoved[FMi]+P[FMi];

  if(hoved[FJi]+P[FJi]>hoved[FMEJi]+P[FMEJi])
  ny_hoved[EJi]=hoved[FJi]+P[FJi];
else if(hoved[FJi]+P[FJi]<=hoved[FMEJi]+P[FMEJi])
  ny_hoved[EJi]=hoved[FMEJi]+P[FMEJi];

   if(ny_hoved[EJi]+P[EJi]>ny_hoved[j]+P[j] )
   ny_hoved[i]=ny_hoved[EJi]+P[EJi];
  else
   ny_hoved[i]=ny_hoved[j]+P[j];

   if(hale[EJEJi]+P[EJEJi]>hale[EMj]+P[EMj])
   ny_hale[i]=hale[EJEJi]+P[EJEJi];
  else if(hale[EJEJi]+P[EJEJi]<=hale[EMj]+P[EMj])
   ny_hale[i]=hale[EMj]+P[EMj];

  if(hale[EMEJi]+P[EMEJi]>ny_hale[i]+P[i])
  ny_hale[EJi]=hale[EMEJi]+P[EMEJi];
  else
  ny_hale[EJi]=ny_hale[i]+P[i];

if(ny_hale[i]+P[i]<=hale[EJj]+P[EJj])
  ny_hale[j]=hale[EJj]+P[EJj];
   else
```

```
   ny_hale[j]=ny_hale[i]+P[i];

/*ny_Cmax=*/
if(ny_hoved[i]+P[i]+ny_hale[i]>=
                         ny_hoved[j]+P[j]+ny_hale[j]
  && ny_hoved[i]+P[i]+ny_hale[i]>=
                   ny_hoved[EJi]+P[EJi]+ny_hale[EJi])
  ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
else if(ny_hoved[j]+P[j]+ny_hale[j]>=
                         ny_hoved[i]+P[i]+ny_hale[i]
  && ny_hoved[j]+P[j]+ny_hale[j]>=
                   ny_hoved[EJi]+P[EJi]+ny_hale[EJi])
  ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];
else
  ny_Cmax=ny_hoved[EJi]+P[EJi]+ny_hale[EJi];

if(ny_Cmax<Cmax ||R<1)
{
for(k=0; k<M*J; k++)
{
if(Mliste[0][k]==i)
{
temp_a1=k; temp_a2=i; temp_a3=Mliste[1][k];
}
if(Mliste[0][k]==j)
{
temp_b1=k; temp_b2=j; temp_b3=Mliste[1][k];
}
if(Liste[0][k]==EJi)
{
temp_c1=k; temp_c2=EJi; temp_c3=Liste[1][k];
}
if(Liste[0][k]==i)
{
temp_d1=k; temp_d2=i; temp_d3=Liste[1][k];
}
}
Mliste[0][temp_a1]=temp_b2;
Mliste[1][temp_a1]=temp_b3;
```

```
Mliste[0][temp_b1]=temp_a2;
Mliste[1][temp_b1]=temp_a3;
Liste[0][temp_c1]=temp_d2;
Liste[1][temp_c1]=temp_d3;
Liste[0][temp_d1]=temp_c2;
Liste[1][temp_d1]=temp_c3;
}

if(ny_Cmax>=Cmax && R<1)
ny_Cmax=0;
else if(ny_Cmax<Cmax)
ny_Cmax=1;
else
ny_Cmax=-1;
}
else
ny_Cmax=-1;
}
else/*samme job*/
{
EMi=M*J;
hoved[EMi]=Cmax;
for(k=i%M; k<M*J; k=k+M)
{
if(hoved[k]>hoved[i] && hoved[EMi]>hoved[k])
EMi=k;
}

FMj=M*J;
hoved[FMj]=0;
for(k=j%M; k<M*J; k=k+M)
{
if(hoved[k]<hoved[j] && hoved[FMj]<hoved[k])
FMj=k;
}

EJEMi=M*J;
hoved[EJEMi]=Cmax;
for(k=(EMi/M)*M; k<(EMi/M)*M+M; k++)
```

```
{
if(hoved[k]>hoved[EMi] && hoved[EJEMi]>hoved[k])
EJEMi=k;
}

EMEMi=M*J;
hoved[EMEMi]=Cmax;
for(k=EMi%M; k<M*J; k=k+M)
{
if(hoved[k]>hoved[EMi] && hoved[EMEMi]>hoved[k])
EMEMi=k;
}

FJEMi=M*J;
hoved[FJEMi]=0;
for(k=(EMi/M)*M; k<(EMi/M)*M+M; k++)
{
if(hoved[k]<hoved[EMi] && hoved[FJEMi]<hoved[k])
FJEMi=k;
}

EMj=M*J;
hoved[EMj]=Cmax;
for(k=j%M; k<M*J; k=k+M)
{
if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
EMj=k;
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<M*J; k=k+M)
{
   if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
FMi=k;
}

FJi=M*J;
hoved[FJi]=0;
```

```
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
FJi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
EJj=k;
}

/*Obj.vaerdi findes, hvis den er legal*/
if(EMi!=M*J && FMj!=M*J && FJi!=M*J && FMi!=M*J && FJEMi!=M*J
&& EMEMi!=M*J && EJj!=M*J && EJEMi!=M*J && EMj!=M*J)
{
if(hoved[FMj]+P[FMj]>hoved[FJi]+P[FJi])
  ny_hoved[j]=hoved[FMj]+P[FMj];
  else if(hoved[FMj]+P[FMj]<=hoved[FJi]+P[FJi])
   ny_hoved[j]=hoved[FJi]+P[FJi];

   if(hoved[FMi]+P[FMi]>hoved[FJEMi]+P[FJEMi])
   ny_hoved[EMi]=hoved[FMi]+P[FMi];
   else if(hoved[FMi]+P[FMi]<=hoved[FJEMi]+P[FJEMi])
   ny_hoved[EMi]=hoved[FJEMi]+P[FJEMi];

  if(ny_hoved[EMi]+P[EMi]>ny_hoved[j]+P[j])
   ny_hoved[i]=ny_hoved[EMi]+P[EMi];
  else
   ny_hoved[i]=ny_hoved[j]+P[j];

  if(hale[EMEMi]+P[EMEMi]>hale[EJj]+P[EJj])
   ny_hale[i]=hale[EMEMi]+P[EMEMi];
  else if(hale[EMEMi]+P[EMEMi]<=hale[EJj]+P[EJj])
   ny_hale[i]=hale[EJj]+P[EJj];

  if(hale[EJEMi]+P[EJEMi]>ny_hale[i]+P[i])
```

```
  ny_hale[EMi]=hale[EJEMi]+P[EJEMi];
  else
  ny_hale[EMi]=ny_hale[i]+P[i];

  if(ny_hale[i]+P[i]<=hale[EMj]+P[EMj])
  ny_hale[j]=hale[EMj]+P[EMj];
  else
  ny_hale[j]=ny_hale[i]+P[i];

/*ny_Cmax=*/
if(ny_hoved[i]+P[i]+ny_hale[i]>=
                          ny_hoved[j]+P[j]+ny_hale[j]
  && ny_hoved[i]+P[i]+ny_hale[i]>=
                    ny_hoved[EMi]+P[EMi]+ny_hale[EMi])
  ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
else if(ny_hoved[j]+P[j]+ny_hale[j]>=
                          ny_hoved[i]+P[i]+ny_hale[i]
  && ny_hoved[j]+P[j]+ny_hale[j]>=
                    ny_hoved[EMi]+P[EMi]+ny_hale[EMi])
  ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];
else
  ny_Cmax=ny_hoved[EMi]+P[EMi]+ny_hale[EMi];

if(ny_Cmax<Cmax ||R<1)
{
for(k=0; k<M*J; k++)
{
if(Liste[0][k]==i)
{
temp_a1=k; temp_a2=i; temp_a3=Liste[1][k];
}
if(Liste[0][k]==j)
{
temp_b1=k; temp_b2=j; temp_b3=Liste[1][k];
}
if(Mliste[0][k]==EMi)
{
temp_c1=k; temp_c2=EMi; temp_c3=Mliste[1][k];
}
```

```
if(Mliste[0][k]==i)
{
temp_d1=k; temp_d2=i; temp_d3=Mliste[1][k];
}
}
Liste[0][temp_a1]=temp_b2;
Liste[1][temp_a1]=temp_b3;
Liste[0][temp_b1]=temp_a2;
Liste[1][temp_b1]=temp_a3;
Mliste[0][temp_c1]=temp_d2;
Mliste[1][temp_c1]=temp_d3;
Mliste[0][temp_d1]=temp_c2;
Mliste[1][temp_d1]=temp_c3;
}
if(ny_Cmax>=Cmax && R<1)
ny_Cmax=0;
else if(ny_Cmax<Cmax)
ny_Cmax=1;
else
ny_Cmax=-1;
}
else
ny_Cmax=-1;
}/*end of if samme job*/
};break;
case 2: /*(i,j), (FJj,j)||  (a,b), (FM,b)*/
{
if(OpNr==0)
{
i=KritVej[0][0];
j=KritVej[0][1];
}
else if(OpNr==VejLaengde)
{
i=KritVej[0][VejLaengde-2];
j=KritVej[0][VejLaengde-1];
}
else
{
```

```
if(OpNr%2==0)
{
i=KritVej[0][OpNr-1];
j=KritVej[0][OpNr];
}
else
{
i=KritVej[0][OpNr];
j=KritVej[0][OpNr+1];
}
}

if(i%M==j%M)/*samme maskine*/
{
EJi=M*J;
hoved[EJi]=Cmax;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]>hoved[i] && hoved[EJi]>hoved[k])
EJi=k;
}

FJj=M*J;
   hoved[FJj]=0;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]<hoved[j] && hoved[FJj]<hoved[k])
FJj=k;
}

FJFJj=M*J;
   hoved[FJFJj]=0;
for(k=(FJj/M)*M; k<(FJj/M)*M+M; k++)
{
if(hoved[k]<hoved[FJj] && hoved[FJFJj]<hoved[k])
FJFJj=k;
}

EMFJj=M*J;
```

```
hoved[EMFJj]=Cmax;
for(k=(FJj/M)*M; k<(FJj/M)*M+M; k++)
{
if(hoved[k]>hoved[FJj] && hoved[EMFJj]>hoved[k])
EMFJj=k;
}

FMFJj=M*J;
hoved[FMFJj]=0;
for(k=(FJj/M)*M; k<(FJj/M)*M+M; k++)
{
if(hoved[k]<hoved[FJj] && hoved[FMFJj]<hoved[k])
FMFJj=k;
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<i%M; k++)
{
if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
FMi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
EJj=k;
}

EMj=M*J;
hoved[EJj]=Cmax;
for(k=j%M; k<j%M; k++)
{
if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
EMj=k;
}
```

```
FJi=M*J;
   hoved[FJi]=0;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
FJi=k;
}

/*Obj.vaerdi findes hvis lovlig*/
if(FJj!=M*J && FJi!=M*J && FJFJj!=M*J && FMi!=M*J
&& FMFJj!=M*J && EJi!=M*J && EMj!=M*J && EMFJj!=M*J
&& FJj!=M*J)
{
if(hoved[FJFJj]+P[FJFJj]>hoved[FMi]+P[FMi])
  ny_hoved[j]=hoved[FJFJj]+P[FJFJj];
   else if(hoved[FJFJj]+P[FJFJj]<=hoved[FMi]+P[FMi])
   ny_hoved[j]=hoved[FMi]+P[FMi];

if(hoved[FMFJj]+P[FMFJj]>ny_hoved[j]+P[j])
ny_hoved[FJj]=hoved[FMFJj]+P[FMFJj];
else
ny_hoved[FJj]=ny_hoved[j]+P[j];

   if(hoved[FJi]+P[FJi]>ny_hoved[j]+P[j])
   ny_hoved[i]=hoved[FJi]+P[FJi];
  else
   ny_hoved[i]=ny_hoved[j]+P[j];

   if(hale[EJi]+P[EJi]>hale[EMj]+P[EMj])
   ny_hale[i]=hale[EJi]+P[EJi];
  else if(hale[EJi]+P[EJi]<=hale[EMj]+P[EMj])
   ny_hale[i]=hale[EMj]+P[EMj];

  if(hale[EMFJj]+P[EMFJj]>hale[EJj]+P[EJj])
  ny_hale[FJj]=hale[EMFJj]+P[EMFJj];
  else
  ny_hale[FJj]=hale[EJj]+P[EJj];

if(ny_hale[i]+P[i]<=ny_hale[FJj]+P[FJj])
```

```
   ny_hale[j]=ny_hale[FJj]+P[FJj];
   else
   ny_hale[j]=ny_hale[i]+P[i];

/*ny_Cmax=*/
if(ny_hoved[i]+P[i]+ny_hale[i]>=
                         ny_hoved[j]+P[j]+ny_hale[j]
  && ny_hoved[i]+P[i]+ny_hale[i]>=
                   ny_hoved[FJj]+P[FJj]+ny_hale[FJj])
  ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
else if(ny_hoved[j]+P[j]+ny_hale[j]>=
                         ny_hoved[i]+P[i]+ny_hale[i]
  && ny_hoved[j]+P[j]+ny_hale[j]>=
                   ny_hoved[FJj]+P[FJj]+ny_hale[FJj])
  ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];
else
  ny_Cmax=ny_hoved[FJj]+P[FJj]+ny_hale[FJj];

if(ny_Cmax<Cmax ||R<1)
{
for(k=0; k<M*J; k++)
{
if(Mliste[0][k]==i)
{
temp_a1=k; temp_a2=i; temp_a3=Mliste[1][k];
}
if(Mliste[0][k]==j)
{
temp_b1=k; temp_b2=j; temp_b3=Mliste[1][k];
}
if(Liste[0][k]==FJj)
{
temp_c1=k; temp_c2=FJj; temp_c3=Liste[1][k];
}
if(Liste[0][k]==j)
{
temp_d1=k; temp_d2=j; temp_d3=Liste[1][k];
}
}
```

```
Mliste[0][temp_a1]=temp_b2;
Mliste[1][temp_a1]=temp_b3;
Mliste[0][temp_b1]=temp_a2;
Mliste[1][temp_b1]=temp_a3;
Liste[0][temp_c1]=temp_d2;
Liste[1][temp_c1]=temp_d3;
Liste[0][temp_d1]=temp_c2;
Liste[1][temp_d1]=temp_c3;
}
if(ny_Cmax>=Cmax && R<1)
ny_Cmax=0;
else if(ny_Cmax<Cmax)
ny_Cmax=1;
else
ny_Cmax=-1;
}
else
ny_Cmax=-1;
}
else/*samme job*/
{
EMi=M*J;
hoved[EMi]=Cmax;
for(k=i%M; k<M*J; k=k+M)
{
if(hoved[k]>hoved[i] && hoved[EMi]>hoved[k])
EMi=k;
}

FMj=M*J;
hoved[FMj]=0;
for(k=j%M; k<M*J; k=k+M)
{
if(hoved[k]<hoved[j] && hoved[FMj]<hoved[k])
FMj=k;
}

FMFMj=M*J;
hoved[FMFMj]=0;
```

```
for(k=FMj%M; k<M*J; k=k+M)
{
if(hoved[k]<hoved[FMj] && hoved[FMFMj]<hoved[k])
FMFMj=k;
}

EJFMj=M*J;
hoved[EJFMj]=Cmax;
for(k=(FMj/M)*M; k<(FMj/M)*M+M; k++)
{
if(hoved[k]>hoved[FMj] && hoved[EJFMj]>hoved[k])
EJFMj=k;
}

FJFMj=M*J;
hoved[FJFMj]=0;
for(k=(FMj/M)*M; k<(FMj/M)*M+M; k++)
{
if(hoved[k]<hoved[FMj] && hoved[FJFMj]<hoved[k])
FJFMj=k;
}

EMj=M*J;
hoved[EMj]=Cmax;
for(k=j%M; k<M*J; k=k+M)
{
if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
EMj=k;
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<M*J; k=k+M)
{
   if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
FMi=k;
}

FJi=M*J;
```

```
hoved[FJi]=0;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
FJi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
EJj=k;
}

/*Obj.vaerdi findes, hvis den er legal*/
if(FMj!=M*J && FMi!=M*j && FMFMj!=M*J && FJi!=M*J
&& FJFMj!=M*J && EMi!=M*J && EJj!=M*J && EJFMj!=M*J
&& EMj!=M*J)
{
   if(hoved[FMFMj]+P[FMFMj]>hoved[FJi]+P[FJi])
   ny_hoved[j]=hoved[FMFMj]+P[FMFMj];
   else if(hoved[FMFMj]+P[FMFMj]<=hoved[FJi]+P[FJi])
   ny_hoved[j]=hoved[FJi]+P[FJi];

if(hoved[FJFMj]+P[FJFMj]>ny_hoved[j]+P[j])
ny_hoved[FMj]=hoved[FJFMj]+P[FJFMj];
else
ny_hoved[FMj]=ny_hoved[j]+P[j];

   if(hoved[FMi]+P[FMi]>ny_hoved[j]+P[j])
   ny_hoved[i]=hoved[FMi]+P[FMi];
  else
   ny_hoved[i]=ny_hoved[j]+P[j];

  if(hale[EMi]+P[EMi]>hale[EJj]+P[EJj])
   ny_hale[i]=hale[EMi]+P[EMi];
  else if(hale[EMi]+P[EMi]<=hale[EJj]+P[EJj])
   ny_hale[i]=hale[EJj]+P[EJj];
```

```
  if(hale[EJFMj]+P[EJFMj]>hale[EMj]+P[EMj])
  ny_hale[FMj]=hale[EJFMj]+P[EJFMj];
  else
  ny_hale[FMj]=hale[EMj]+P[EMj];

if(ny_hale[i]+P[i]<=ny_hale[FMj]+P[FMj])
  ny_hale[j]=ny_hale[FMj]+P[FMj];
  else
  ny_hale[j]=ny_hale[i]+P[i];

/*ny_Cmax=*/
if(ny_hoved[i]+P[i]+ny_hale[i]>=
                         ny_hoved[j]+P[j]+ny_hale[j]
  && ny_hoved[i]+P[i]+ny_hale[i]>=
                   ny_hoved[FMj]+P[FMj]+ny_hale[FMj])
  ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
else if(ny_hoved[j]+P[j]+ny_hale[j]>=
                         ny_hoved[i]+P[i]+ny_hale[i]
  && ny_hoved[j]+P[j]+ny_hale[j]>=
                      hoved[FMj]+P[FMj]+ny_hale[FMj])
  ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];
else
  ny_Cmax=hoved[FMj]+P[FMj]+ny_hale[FMj];

if(ny_Cmax<Cmax ||R<1)
{
for(k=0; k<M*J; k++)
{
if(Liste[0][k]==i)
{
temp_a1=k; temp_a2=i; temp_a3=Liste[1][k];
}
if(Liste[0][k]==j)
{
temp_b1=k; temp_b2=j; temp_b3=Liste[1][k];
}
if(Mliste[0][k]==FMj)
{
```

```
temp_c1=k; temp_c2=FMj; temp_c3=Mliste[1][k];
}
if(Mliste[0][k]==i)
{
temp_d1=k; temp_d2=i; temp_d3=Mliste[1][k];
}
}
Liste[0][temp_a1]=temp_b2;
Liste[1][temp_a1]=temp_b3;
Liste[0][temp_b1]=temp_a2;
Liste[1][temp_b1]=temp_a3;
Mliste[0][temp_c1]=temp_d2;
Mliste[1][temp_c1]=temp_d3;
Mliste[0][temp_d1]=temp_c2;
Mliste[1][temp_d1]=temp_c3;
}
if(ny_Cmax>=Cmax && R<1)
ny_Cmax=0;
else if(ny_Cmax<Cmax)
ny_Cmax=1;
else
ny_Cmax=-1;
}
else
ny_Cmax=-1;
}/*end of if samme job*/
};break;
  case 3: /*(i,j),(i,EJi),(FJj,j) || (a,b), (a,EM), (FM,b) */
    {
      if(OpNr==0)
      {
i=KritVej[0][0];
j=KritVej[0][1];
      }
      else if(OpNr==VejLaengde)
      {
i=KritVej[0][VejLaengde-2];
j=KritVej[0][VejLaengde-1];
      }
```

```
      else
      {
if(OpNr%2==0)
{
i=KritVej[0][OpNr-1];
j=KritVej[0][OpNr];
}
else
{
i=KritVej[0][OpNr];
j=KritVej[0][OpNr+1];
}
      }

      if(i%M==j%M)/*samme maskine*/
      {
EJi=M*J;
hoved[EJi]=Cmax;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
if(hoved[k]>hoved[i] && hoved[EJi]>hoved[k])
EJi=k;
}

FJj=M*J;
   hoved[FJj]=0;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]<hoved[j] && hoved[FJj]<hoved[k])
FJj=k;
}

FJFJj=M*J;
        hoved[FJFJj]=0;
for(k=(FJj/M)*M; k<(FJj/M)*M+M; k++)
{
if(hoved[k]<hoved[FJj] && hoved[FJFJj]<hoved[k])
FJFJj=k;
}
```

```
EMFJj=M*J;
hoved[EMFJj]=Cmax;
for(k=(FJj/M)*M; k<(FJj/M)*M+M; k++)
{
if(hoved[k]>hoved[FJj] && hoved[EMFJj]>hoved[k])
EMFJj=k;
}

EMEJi=M*J;
hoved[EMEJi]=Cmax;
for(k=(EJi/M)*M; k<(EJi/M)*M+M; k++)
{
if(hoved[k]>hoved[EJi] && hoved[EMEJi]>hoved[k])
EMEJi=k;
}

FMEJi=M*J;
hoved[FMEJi]=0;
for(k=(EJi/M)*M; k<(EJi/M)*M+M; k++)
{
if(hoved[k]<hoved[EJi] && hoved[FMEJi]<hoved[k])
FMEJi=k;
}

FMFJj=M*J;
hoved[FMFJj]=0;
for(k=(FJj/M)*M; k<(FJj/M)*M+M; k++)
{
if(hoved[k]<hoved[FJj] && hoved[FMFJj]<hoved[k])
FMFJj=k;
}

EJEJi=M*J;
hoved[EJEJi]=Cmax;
for(k=(EJi/M)*M; k<(EJi/M)*M+M; k++)
{
if(hoved[k]>hoved[EJi] && hoved[EJEJi]>hoved[k])
EJEJi=k;
```

```
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<i%M; k++)
{
if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
FMi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
EJj=k;
}

EMj=M*J;
hoved[EJj]=Cmax;
for(k=j%M; k<j%M; k++)
{
  if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
    EMj=k;
}

FJi=M*J;
   hoved[FJi]=0;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
  if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
    FJi=k;
}

/*Obj.vaerdi findes hvis lovlig                    */
if(EJi!=M*J && FJFJj!=M*J && FMi!=M*J && FMFJj!=M*J
 && FJi!=M*J && FMEJi!=M*J && EJEJi!=M*J && EMj!=M*J
 && EMEJi!=M*J && EMFJj!=M*J && EJj!=M*J && FJj!=M*J)
{
```

```
if(hoved[EMFJj]+P[EMFJj]<=hale[FMi]
                 && hoved[EMj]+P[EMj]<=hale[FMEJi])
{
  if(hoved[FJFJj]+P[FJFJj]>hoved[FMi]+P[FMi])
    ny_hoved[j]=hoved[FJFJj]+P[FJFJj];
  else if(hoved[FJFJj]+P[FJFJj]<=hoved[FMi]+P[FMi])
    ny_hoved[j]=hoved[FMi]+P[FMi];

  if(hoved[FMFJj]+P[FMFJj]>ny_hoved[j]+P[j])
    ny_hoved[FJj]=hoved[FMFJj]+P[FMFJj];
  else
    ny_hoved[FJj]=ny_hoved[j]+P[j];

  if(hoved[FJi]+P[FJi]>hoved[FMEJi]+P[FMEJi])
    ny_hoved[EJi]=hoved[FJi]+P[FJi];
  else if(hoved[FJi]+P[FJi]<=hoved[FMEJi]+P[FMEJi])
    ny_hoved[EJi]=hoved[FMEJi]+P[FMEJi];

  if(ny_hoved[EJi]+P[EJi]>ny_hoved[j]+P[j] )
    ny_hoved[i]=ny_hoved[EJi]+P[EJi];
  else
    ny_hoved[i]=ny_hoved[j]+P[j];

  if(hale[EJEJi]+P[EJEJi]>hale[EMj]+P[EMj])
    ny_hale[i]=hale[EJEJi]+P[EJEJi];
  else if(hale[EJEJi]+P[EJEJi]<=hale[EMj]+P[EMj])
    ny_hale[i]=hale[EMj]+P[EMj];

  if(hale[EMEJi]+P[EMEJi]>ny_hale[i]+P[i])
    ny_hale[EJi]=hale[EMEJi]+P[EMEJi];
  else
    ny_hale[EJi]=ny_hale[i]+P[i];

  if(hale[EMFJj]+P[EMFJj]>hale[EJj]+P[EJj])
    ny_hale[FJj]=hale[EMFJj]+P[EMFJj];
  else if(hale[EMFJj]+P[EMFJj]<=hale[EJj]+P[EJj])
    ny_hale[FJj]=hale[EJj]+P[EJj];

  if(ny_hale[i]+P[i]>ny_hale[FJj]+P[FJj])
```

```
      ny_hale[j]=ny_hale[FJj]+P[FJj];
    else
      ny_hale[j]=ny_hale[i]+P[i];

    /*ny_Cmax=*/
    if(ny_hoved[i]+P[i]+ny_hale[i]>=
                         ny_hoved[j]+P[j]+ny_hale[j]
&& ny_hoved[i]+P[i]+ny_hale[i]>=
           ny_hoved[EJi]+P[EJi]+ny_hale[EJi]
&& ny_hoved[i]+P[i]+ny_hale[i]>=
           ny_hoved[FJj]+P[FJj]+ny_hale[FJj])
      ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
    else if(ny_hoved[j]+P[j]+ny_hale[j]>=
                         ny_hoved[i]+P[i]+ny_hale[i]
&& ny_hoved[j]+P[j]+ny_hale[j]>=
           ny_hoved[EJi]+P[EJi]+ny_hale[EJi]
&& ny_hoved[j]+P[j]+ny_hale[j]>=
           ny_hoved[FJj]+P[FJj]+ny_hale[FJj])
      ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];
    else if(ny_hoved[EJi]+P[EJi]+ny_hale[EJi]>=
                         ny_hoved[i]+P[i]+ny_hale[i]
&& ny_hoved[EJi]+P[EJi]+ny_hale[EJi]>=
                 ny_hoved[j]+P[j]+ny_hale[j]
&& ny_hoved[EJi]+P[EJi]+ny_hale[EJi]>=
           ny_hoved[FJj]+P[FJj]+ny_hale[FJj])
      ny_Cmax=ny_hoved[EJi]+P[EJi]+ny_hale[EJi];
    else
      ny_Cmax=ny_hoved[FJj]+P[FJj]+ny_hale[FJj];

    if(ny_Cmax<Cmax ||R<1)
    {
      for(k=0; k<M*J; k++)
      {
if(Mliste[0][k]==i)
{
  temp_a1=k;
  temp_a2=i;
  temp_a3=Mliste[1][k];
}
```

```
if(Mliste[0][k]==j)
{
  temp_b1=k;
  temp_b2=j;
  temp_b3=Mliste[1][k];
}
if(Liste[0][k]==FJj)
{
  temp_c1=k;
  temp_c2=FJj;
  temp_c3=Liste[1][k];
}
if(Liste[0][k]==i)
{
  temp_d1=k;
  temp_d2=i;
  temp_d3=Liste[1][k];
}
if(Liste[0][k]==j)
{
  temp_e1=k; temp_e2=j; temp_e3=Liste[1][k];
}
if(Liste[0][k]==EJi)
{
  temp_f1=k;
  temp_f2=EJi;
  temp_f3=Liste[1][k];
}
      }
      Mliste[0][temp_a1]=temp_b2;
      Mliste[1][temp_a1]=temp_b3;
      Mliste[0][temp_b1]=temp_a2;
      Mliste[1][temp_b1]=temp_a3;
      Liste[0][temp_c1]=temp_d2;
      Liste[1][temp_c1]=temp_d3;
      Liste[0][temp_d1]=temp_c2;
      Liste[1][temp_d1]=temp_c3;
      Liste[0][temp_e1]=temp_f2;
      Liste[1][temp_e1]=temp_f3;
```

```
      Liste[0][temp_f1]=temp_e2;
      Liste[1][temp_f1]=temp_e3;
    }
    if(ny_Cmax>=Cmax && R<1)
      ny_Cmax=0;
    else if(ny_Cmax<Cmax)
      ny_Cmax=1;
            else
      ny_Cmax=-1;
          }
          else
    ny_Cmax=-1;
        }
        else
  ny_Cmax=-1;
      }
      else/*samme job*/
      {
EMi=M*J;
hoved[EMi]=Cmax;
for(k=i%M; k<M*J; k=k+M)
{
  if(hoved[k]>hoved[i] && hoved[EMi]>hoved[k])
    EMi=k;
}

FMj=M*J;
hoved[FMj]=0;
for(k=j%M; k<M*J; k=k+M)
{
  if(hoved[k]<hoved[j] && hoved[FMj]<hoved[k])
    FMj=k;
}

FMFMj=M*J;
hoved[FMFMj]=0;
for(k=FMj%M; k<M*J; k=k+M)
{
  if(hoved[k]<hoved[FMj] && hoved[FMFMj]<hoved[k])
```

```
    FMFMj=k;
}

EJFMj=M*J;
hoved[EJFMj]=Cmax;
for(k=(FMj/M)*M; k<(FMj/M)*M+M; k++)
{
  if(hoved[k]>hoved[FMj] && hoved[EJFMj]>hoved[k])
    EJFMj=k;
}

EJEMi=M*J;
hoved[EJEMi]=Cmax;
for(k=(EMi/M)*M; k<(EMi/M)*M+M; k++)
{
  if(hoved[k]>hoved[EMi] && hoved[EJEMi]>hoved[k])
    EJEMi=k;
}

EMEMi=M*J;
hoved[EMEMi]=Cmax;
for(k=EMi%M; k<M*J; k=k+M)
{
  if(hoved[k]>hoved[EMi] && hoved[EMEMi]>hoved[k])
    EMEMi=k;
}

FJEMi=M*J;
hoved[FJEMi]=0;
for(k=(EMi/M)*M; k<(EMi/M)*M+M; k++)
{
  if(hoved[k]<hoved[EMi] && hoved[FJEMi]<hoved[k])
    FJEMi=k;
}

FJFMj=M*J;
hoved[FJFMj]=0;
for(k=(FMj/M)*M; k<(FMj/M)*M+M; k++)
{
```

```
  if(hoved[k]<hoved[FMj] && hoved[FJFMj]<hoved[k])
            FJFMj=k;
}

EMj=M*J;
hoved[EMj]=Cmax;
for(k=j%M; k<M*J; k=k+M)
{
  if(hoved[k]>hoved[j] && hoved[EMj]>hoved[k])
    EMj=k;
}

FMi=M*J;
hoved[FMi]=0;
for(k=i%M; k<M*J; k=k+M)
{
     if(hoved[k]<hoved[i] && hoved[FMi]<hoved[k])
    FMi=k;
}

FJi=M*J;
hoved[FJi]=0;
for(k=(i/M)*M; k<(i/M)*M+M; k++)
{
  if(hoved[k]<hoved[i] && hoved[FJi]<hoved[k])
    FJi=k;
}

EJj=M*J;
hoved[EJj]=Cmax;
for(k=(j/M)*M; k<(j/M)*M+M; k++)
{
  if(hoved[k]>hoved[j] && hoved[EJj]>hoved[k])
    EJj=k;
}

/*Obj.vaerdi findes, hvis den er legal*/
if(FMFMj!=M*J && FJi!=M*J && FJFMj!=M*J && FMi!=M*J
 && FJEMi!=M*J && EMi!=M*J && EMEMi!=M*J && EJj!=M*J
```

```
 && EJEMi!=M*J && EJFMj!=M*J && EMj!=M*J && FMj!=M*J)
{
  if(hoved[EJFMj]+P[EJFMj]<=hale[FJi]
    && hoved[EJj]+P[EJj]<=hale[FJEMi])
  {
    if(hoved[FMFMj]+P[FMFMj]>hoved[FJi]+P[FJi])
      ny_hoved[j]=hoved[FMFMj]+P[FMFMj];
    else if(hoved[FMFMj]+P[FMFMj]<=hoved[FJi]+P[FJi])
      ny_hoved[j]=hoved[FJi]+P[FJi];

    if(hoved[FJFMj]+P[FJFMj]>ny_hoved[j]+P[j])
      ny_hoved[FMj]=hoved[FJFMj]+P[FJFMj];
    else
      ny_hoved[FMj]=ny_hoved[j]+P[j];

    if(hoved[FMi]+P[FMi]>hoved[FJEMi]+P[FJEMi])
      ny_hoved[EMi]=hoved[FMi]+P[FMi];
    else if(hoved[FMi]+P[FMi]<=hoved[FJEMi]+P[FJEMi])
      ny_hoved[EMi]=hoved[FJEMi]+P[FJEMi];

    if(ny_hoved[EMi]+P[EMi]>ny_hoved[j]+P[j])
      ny_hoved[i]=ny_hoved[EMi]+P[EMi];
    else
      ny_hoved[i]=ny_hoved[j]+P[j];

    if(hale[EMEMi]+P[EMEMi]>hale[EJj]+P[EJj])
      ny_hale[i]=hale[EMEMi]+P[EMEMi];
    else if(hale[EMEMi]+P[EMEMi]<=hale[EJj]+P[EJj])
      ny_hale[i]=hale[EJj]+P[EJj];

    if(hale[EJEMi]+P[EJEMi]>ny_hale[i]+P[i])
      ny_hale[EMi]=hale[EJEMi]+P[EJEMi];
    else
      ny_hale[EMi]=ny_hale[i]+P[i];

    if(hale[EJFMj]+P[EJFMj]>hale[EMj]+P[EMj])
      ny_hale[FMj]=hale[EJFMj]+P[EJFMj];
    else if(hale[EJFMj]+P[EJFMj]<=hale[EMj]+P[EMj])
      ny_hale[FMj]=hale[EMj]+P[EMj];
```

```
    if(ny_hale[i]+P[i]<=ny_hale[FMj]+P[FMj])
      ny_hale[j]=ny_hale[FMj]+P[FMj];
    else
      ny_hale[j]=ny_hale[i]+P[i];

    /*ny_Cmax=*/
    if(ny_hoved[i]+P[i]+ny_hale[i]>=
                          ny_hoved[j]+P[j]+ny_hale[j]
&& ny_hoved[i]+P[i]+ny_hale[i]>=
           ny_hoved[EMi]+P[EMi]+ny_hale[EMi]
&& ny_hoved[i]+P[i]+ny_hale[i]>=
           ny_hoved[FMj]+P[FMj]+ny_hale[FMj])
      ny_Cmax=ny_hoved[i]+P[i]+ny_hale[i];
    else if(ny_hoved[j]+P[j]+ny_hale[j]>=
                          ny_hoved[i]+P[i]+ny_hale[i]
&& ny_hoved[j]+P[j]+ny_hale[j]>=
           ny_hoved[EMi]+P[EMi]+ny_hale[EMi]
&& ny_hoved[j]+P[j]+ny_hale[j]>=
           ny_hoved[FMj]+P[FMj]+ny_hale[FMj])
      ny_Cmax=ny_hoved[j]+P[j]+ny_hale[j];
    else if(ny_hoved[EMi]+P[EMi]+ny_hale[EMi]>=
                          ny_hoved[i]+P[i]+ny_hale[i]
        && ny_hoved[EMi]+P[EMi]+ny_hale[EMi]>=
                          ny_hoved[j]+P[j]+ny_hale[j]
&& ny_hoved[EMi]+P[EMi]+ny_hale[EMi]>=
           ny_hoved[FMj]+P[FMj]+ny_hale[FMj])
      ny_Cmax=ny_hoved[EMi]+P[EMi]+ny_hale[EMi];
    else
              ny_Cmax=hoved[FMj]+P[FMj]+ny_hale[FMj];

    if(ny_Cmax<Cmax ||R<1)
    {
      for(k=0; k<M*J; k++)
      {
if(Liste[0][k]==i)
{
  temp_a1=k;
  temp_a2=i;
```

```
  temp_a3=Liste[1][k];
}
if(Liste[0][k]==j)
{
  temp_b1=k;
  temp_b2=j;
  temp_b3=Liste[1][k];
}
if(Mliste[0][k]==FMj)
{
  temp_c1=k;
  temp_c2=FMj;
  temp_c3=Mliste[1][k];
}
if(Mliste[0][k]==i)
{
  temp_d1=k;
  temp_d2=i;
  temp_d3=Mliste[1][k];
}
if(Mliste[0][k]==j)
{
  temp_e1=k; temp_e2=j;
  temp_e3=Mliste[1][k];
}
if(Mliste[0][k]==EMi)
{
  temp_f1=k;
  temp_f2=EMi;
  temp_f3=Mliste[1][k];
}
      }
      Liste[0][temp_a1]=temp_b2;
      Liste[1][temp_a1]=temp_b3;
      Liste[0][temp_b1]=temp_a2;
      Liste[1][temp_b1]=temp_a3;
      Mliste[0][temp_c1]=temp_d2;
      Mliste[1][temp_c1]=temp_d3;
      Mliste[0][temp_d1]=temp_c2;
```

```
      Mliste[1][temp_d1]=temp_c3;
      Mliste[0][temp_e1]=temp_f2;
      Mliste[1][temp_e1]=temp_f3;
      Mliste[0][temp_f1]=temp_e2;
      Mliste[1][temp_f1]=temp_e3;
            }
    if(ny_Cmax>=Cmax && R<1)
      ny_Cmax=0;
    else if(ny_Cmax<Cmax)
      ny_Cmax=1;
    else
      ny_Cmax=-1;
  }/*end of legal*/
  else
            ny_Cmax=-1;
        }
        else
          ny_Cmax=-1;
      }/*end of if samme job*/
    };/*end of case 3*/
    break;
    default:
    {
      ny_Cmax=-1;
    };/*end of default*/
    break;
  }/*end of switch */
  return(ny_Cmax);
}/*end of Nabo*/

void main2()
{
  int LB, Cmax=0, Resttid[MaxOp], NaboType, OpNr;
  int L, tal=0, VejLaengde, VejLgd, Punkt[2][MaxOp], Valg;
  int BedstCmax, accept=0, BedstHoved[MaxOp];
  int i, j, k;

  double random_tal, OK=0, pct;
```

```
  char name[13]="SA_out.txt";
  FILE *fid;
  fid = fopen(name,"w");

  srand((unsigned)time(NULL));

  for(i=0; i<J; i++)
    for(j=0; j<M; j++)
      MO[i*M+j]=j+1;

  for(i=0; i<M*J; i++)
  {
    Resttid[i]=0;
    hoved[i]=0;
  }

  for(i=0; i<M; i++)
  {
    for(j=0; j<J; j++)
    {
      Liste[1][j*M+i]=0;
      Liste[0][j*M+i]=j*M+i;
      BedstHoved[j*M+i]=hoved[j*M+i];
    }
  }

  L=(int)(str_faktor*M*J);

  Orden();   /*Ordner Matricen P,saa maskine 1 staar i*/
     /*foerste kolonne, maskine 2 i anden osv.*/

  LB = Minimum(); /*Finder den nedre graense for problemet.*/

  fprintf(fid,"LB = %d\n", LB);
  fprintf(fid,"\n");

/*Simuleret udgloedning*/

/*Foerste loesning*/
```

```
  for (i=0; i<M*J; i++)
  {
    for(j=0; j<J; j++)
    {
      if(j!=i/M)
Resttid[i]=Resttid[i]+P[(i%M)+j*M];
    }
  }

  LTRPOM(Resttid);

  for(i=0; i<J; i++)
  {
    for(j=0; j<M; j++)
    {
      Mliste[0][i*J+j]=Liste[0][M*j+i];
      Mliste[1][i*J+j]=Liste[1][M*j+i];
    }
  }

  for(i=0; i<M*J; i++)
  {
    if(hoved[i]+P[i]>Cmax)
      Cmax=hoved[i]+P[i];
  }

  BedstCmax=Cmax;

  for(i=0; i<M*J; i++)
    BedstHoved[i]=hoved[i];

  /*operationers raekkefoelge paa job*/
  for(i=0; i<M; i++)
    QuickSort(Liste,1+i*J,J*(i+1));

  /*operationers raekkefoelge paa maskiner*/
  for(i=0; i<J; i++)
    QuickSort(Mliste,1+i*M,M*(i+1));
```

```
  for ( i = 0; i <M*J; i++)
    hale[i] = -1;

  for (i = 0; i<M*J; i = i + M)
    updhal(i,hale);

/*Soegen efter en bedre loesning*/
  while(tal<max_tal && Cmax>LB)
  {
    j=0;

    for(i=0; i<M*J; i++)
    {
      KritVej[0][i]=0;
      KritVej[1][i]=0;
    }

    for(i=0; i<M*J; i++)
    {
      if(hoved[i]+P[i]+hale[i]==Cmax)
      {
KritVej[0][j]=i;
KritVej[1][j]=hoved[i];
j++;
      }
    }

    VejLaengde=j;

    QuickSort(KritVej,1,VejLaengde);

    VejLgd=FindPunkt(Punkt, VejLaengde)+1;

    for(i=0; i<L; i++) /*find naboloesning*/
    {
      random_tal=(double)(rand()%1000)/1000;
      j=(int)(random_tal*(VejLgd-1));

      OpNr=Punkt[1][j];
```

```
      NaboType=(int)(random_tal*3);

      OK=(double)(rand()%1000)/1000;

      Valg=Nabo(OpNr, NaboType, Cmax, VejLaengde-1, OK);

      if(Valg==0)
T=T*red_faktor;

      pct=accept/L;

      if(Valg==1 || Valg==0)
      {
accept++;

for(k=0; k<M*J; k++)
  hoved[k]=-1;

for(k=M-1; k<M*J; k=k+M)
  updhov(k, Cmax, hoved);

for ( k = 0; k <M*J; k++)
  hale[k] = -1;

for (k = 0; k<M*J; k = k + M)
  updhal(k,hale);

for(k=0; k<M*J; k++)
{
  if(hoved[k]+P[k]>Cmax)
    Cmax=hoved[k]+P[k];
}

if(Cmax<BedstCmax)
{
  BedstCmax=Cmax;

  for(k=0; k<M*J; k++)
```

```
    BedstHoved[k]=hoved[k];
}
      }

      if(Valg==1)
        i=L;

      tal++;
    }
  }

/*Bedste loesning*/

  for(i=0; i<M*J; i++)
    hoved[i]=BedstHoved[i];

  for ( k = 0; k <M*J; k++)
    hale[k] = -1;

  for (k = 0; k<M*J; k = k + M)
    updhal(k,hale);

  for(k=0; k<M*J; k++)
  {
    if(hoved[k]+P[k]>Cmax)
      Cmax=hoved[k]+P[k];
  }

  fprintf(fid, "op  hoved  P  hale \n");
  for(i=0; i<J*M; i++)
    fprintf(fid,"[%d] = %d   %d   %d\n",
               i+1, BedstHoved[i], P[i], hale[i]);
  fprintf(fid,"\n");

  fprintf(fid,"Cmax=%d\n",Cmax);

  fclose(fid);
}
```

```
void main()
{
  time_t Tidstart,Tidslut, Tid;
int i,j, Antal=10;
for (j = 0;j<1;j++)
{
Tidstart=clock();
for(i=0; i<Antal; i++)
  main2();
Tidslut = clock();
Tid= (Tidslut - Tidstart)*1000/CLOCKS_PER_SEC;
printf("Tid = %i ved %d koersler \n",Tid, Antal);
}
}
```

Appendiks C

Flowshop programmer

C.1 H1

```
/* Program der implementerer den dynamiske H1-heuristik
   foreslaaet af Gueret og Prins '98
   Begyndt d. 10. April 2001. Sidst AEndret d. 5.August 2001.

   Definer MaxJ, MaxM, MaxOP og P[]
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MaxJ   5          /*Antal job                      */
#define MaxM   5          /*Antal maskiner                 */
#define MaxOp 25          /*Antal operationer i alt ( M*J )*/

int J = MaxJ;
int M = MaxM;

/* Procestider                                             */
```

```
int P[MaxOp] = {
 54,  34,  61,   2,   9,
 39,  66,  89,  91,  53,
 27,  46,  28,  40,  64,
 98,  74,  96,  53,  80,
 73,  93,  50,   3,  49};

/*Tidligste start for operationer (hoveder), ved open shop
saettes alle til 0                                        */

int hovedres[MaxOp];

int hoved[MaxOp];

/*Hvilken maskine laver hvilken operation                  */

int MO[MaxOp];

void Orden()
{
  int i,j;
  int PKopi[MaxOp];

  for(i=0; i<M*J; i++)
    PKopi[i]=P[i];

  for(i=0; i<J; i++)
    for (j=0; j<M; j++)
      P[i*M + MO[i*M+j] - 1]=PKopi[i*M+j];

}


void SkemaByg( int H1Liste[2][MaxOp], int Skema[2][MaxOp] )
{
  int i, j, k;
  int MinHovedPos, MinHoved, OpNr;
  int Job[MaxJ];
```

```
  for(i=0; i<M*J; i++)
    hoved[i]=hovedres[i];

  /*Finder den operation, der skal skemalaegges  */
  for (j=0; j<J*M; j++)
  {
    for(i=0; i<J; i++)
      Job[i]=0;

    MinHoved = hoved[H1Liste[1][0]-1];
    MinHovedPos = 0;
    OpNr=H1Liste[1][0]-1;

    for (i=0; i<J*M; i++)
    {
      if (hoved[H1Liste[1][i]-1] < MinHoved
          && H1Liste[1][i] != -1)
      {
        MinHoved = hoved[H1Liste[1][i]-1];
        MinHovedPos = i;
OpNr=H1Liste[1][i]-1;
      }
    }

    /*Skemalaegger den valgte operation                    */
    Skema[0][j] = H1Liste[1][MinHovedPos];
    Skema[1][j] = MinHoved;

    /*Sletter valgt operation fra h1-listen og trunkerer   */
    for (i=MinHovedPos; i<J*M-1;i++) /*denne, saetter ikke */
    {                           /*benyttede pladser til -1 */
      H1Liste[0][i] = H1Liste[0][i+1];
      H1Liste[1][i] = H1Liste[1][i+1];
      H1Liste[0][i+1] = -1;
      H1Liste[1][i+1] = -1;
    }

    hoved[OpNr]=-1;
```

```
    /*Opdaterer hovederne                                    */
    for(i=OpNr%M; i<M*J; i=i+M)
      if(hoved[i]!=-1 && hoved[i]<MinHoved+P[OpNr])
hoved[i]=MinHoved+P[OpNr];

    for(i=0; i<J; i++)
      for(k=1; k<M; k++)
if(hoved[i*M+k]<hoved[i*M+k-1]+P[i*M+k-1]
  && hoved[i*M+k-1]!=-1 && hoved[i*M+k]!=-1)
  hoved[i*M+k]=hoved[i*M+k-1]+P[i*M+k-1];
  }
}

void SetByg(int LB, int Skema[2][MaxOp], int H1set[MaxOp])
{
  int i,j,k,l;
  int MaskinListe[2][MaxOp], Hul[2];

  k = 0;
  /*Finder operationernes raekkefoelge paa maskinerne      */
  for (i=0; i<M; i++)
  {
    for (j=0; j<J*M; j++)
    {
      if ((Skema[0][j]-1)%M == i)
      {
        MaskinListe[0][i*J+k] = Skema[0][j];
        MaskinListe[1][i*J+k] = Skema[1][j];
        k++;
      }
    }
    k = 0;
  }

  for(k=0; k<J-1; k++)         /*Finder H1-set             */
  {
   for(i=0; i<M; i++)          /*Foerst findes hullerne    */
   {
    /*starttid for hul                                     */
```

```
    Hul[0] = MaskinListe[1][k+i*J]
             + P[MaskinListe[0][k+J*i]-1];

    /*sluttid for hul                                        */
    Hul[1] = MaskinListe[1][k+i*J+1];
    if ( Hul[0] < Hul[1] )
    {
     for( j=0; j<M; j++)
     {
      if ( j != i )
      {
       /*Finder operationer der bearbejdes mens der er hul */
       for ( l=0; l<J; l++)
       {
        if((MaskinListe[1][j*J+l] < Hul[0]
         && P[MaskinListe[0][j*J+l]-1]
               +MaskinListe[1][j*J+l] > Hul[0])
         || (MaskinListe[1][j*J+l] < Hul[1]
         && P[MaskinListe[0][j*J+l]-1]
               +MaskinListe[1][j*J+l] > Hul[1]) )
        {
        /*=1 hvis operationen skal prioriteres hoejere     */
         H1set[MaskinListe[0][j*J+l]-1] = 1;
        }
       }
      }
     }
    }
   }
  }

/*Finder operationer der bearbejdes efter den nedre graense*/
  for ( i = 0; i < M*J; i++)  /*er naaet                    */
  {
    if (MaskinListe[1][i] + P[MaskinListe[0][i]-1] > LB)
      H1set[MaskinListe[0][i]-1] = 1;
  }
}
```

```
int Sammenlign(int Skema[2][MaxOp],int BedstSkema[2][MaxOp])
{
  int i, tmp = 1;

  for (i = 0; i < M*J; i++)
    if ( Skema[0][i] != BedstSkema[0][i]
      || Skema[1][i] != BedstSkema[1][i] )
      tmp = 0;

  return tmp;
}

void main2()
{
  int Nummer = 1, MaxRest = 0, Raekke = 1, Position = 1;
  int RestMaskintid[MaxOp], MaskinSum[MaxM];
  int Skema[2][MaxOp], BedstSkema[2][MaxOp];
  int H1set[MaxOp], GammelSkema[2][MaxOp], AntalIt=1;
  int H1Liste[2][MaxOp], H1ListeKopi[MaxOp];
  int JobSum[MaxJ], LB = 0, SlutTid = 0, BedstSlutTid = -1;
  int i, j, k, Temp;

  char name[13]="H1_out.txt";
  FILE *fid;
  fid = fopen(name,"w");

  for(i=0; i<J; i++)
    for(j=0; j<M; j++)
      MO[i*M+j]=j+1;

  Orden();

  for(i=0; i<M*J; i=i+M)
    hovedres[i]=0;

  for(i=0; i<M*J; i++)
    if(i%M!=0)
      hovedres[i]=hovedres[i-1]+P[i-1];
```

```
  for (i=0; i<M; i++) /*Finder nedre graense for hver    */
    for (j=0; j<J; j++) /*maskine                         */
      MaskinSum[i] += P[j*M+i];

  for (i=0; i<M; i++)   /*Finder den stoerste af disse     */
    if (MaskinSum[i] > LB)
      LB = MaskinSum[i];

  for (i=0; i<J; i++) /*Finder nedre graense for hvert job*/
    for (j=0; j<M; j++)
      JobSum[j] += P[j*M+i];

  for (i=0; i<J; i++)   /*Finder den nedre graense LB      */
    if (JobSum[i] > LB)
      LB = JobSum[i];

  for(i=0; i<J*M; i++) /*Finder den minimale resterende   */
  {                    /*arbejdstid for hver maskine      */
    RestMaskintid[i]=0;
    for(j=0; j<J; j++)
      if( j!=i/M )
        RestMaskintid[i] = RestMaskintid [i] + P[i%M +j*M];
  }

  for(i=0; i<J*M; i++)  /*Nulstiller/initialiserer H1listen*/
  {
    H1Liste[0][i]=0;
H1set[i]=0;
  }

  for(i=0; i<J*M; i++)                   /*Laver H1-listen*/
  {
    j = 0;

    /*Her findes den position i H1-listen hvor en operation*/
    while(RestMaskintid[i]<H1Liste[0][j]) /*skal indsaettes*/
      j++;

    k = j;
```

```
  /*Her undersoeges det hvilken af to operationer med*/
  while (RestMaskintid[i] == H1Liste[0][j] && k == j )
  {                                /*samme restmaskintid*/
    if (P[i] <= P[H1Liste[1][j]-1]) /*der skal staa forst*/
      j++;                         /*paa listen         */
    k++;
  }

  /*Flytter allerede indsatte operationer efter denne    */
  for(k=i-1; k>=j; k--) /*position en plads ned ad listen*/
  {                     /*(saa der opstaar et hul)       */
    if(k+1!=J*M)
    {
      H1Liste[0][k+1] = H1Liste[0][k];
      H1Liste[1][k+1] = H1Liste[1][k];
    }
  }

  /*Indsaetter den nye operation paa sin plads           */
  H1Liste[0][j] = RestMaskintid[i];
  H1Liste[1][j] = i+1;
}

for (i = 0; i<M; i++)
  SlutTid += MaskinSum[i];

do
{
  /*Laver en kopi af listen til senere brug              */
  for (i=0; i<J*M; i++)
    H1ListeKopi[i]=H1Liste[1][i];

  /*Opdaterer bedste kendte loesning                     */
  if (BedstSlutTid == -1 || SlutTid<= BedstSlutTid)
  {
    BedstSlutTid = SlutTid;
    for (i=0; i<M*J; i++)
    {
      BedstSkema[0][i] = Skema[0][i];
```

```
      BedstSkema[1][i] = Skema[1][i];
    }
  }

  for (i=0; i<M*J; i++)
  {
    GammelSkema[0][i] = Skema[0][i];
    GammelSkema[1][i] = Skema[1][i];
  }

  SkemaByg(H1Liste, Skema);

  SetByg(LB, Skema, H1set);

  SlutTid = 0;
  for (i=0; i<M*J; i++)       /*Finder Cmax              */
  {
    if (SlutTid < Skema[1][i] + P[Skema[0][i]-1])
      SlutTid = Skema[1][i] + P[Skema[0][i]-1];
  }

  k=0;

  for (i=0; i<M*J; i++) /*Finder ny H1-liste       */
  {
    if (H1set[H1ListeKopi[i]-1] == 0 && k == 0)
    {
      Temp = H1ListeKopi[i];
      k=1;
    }
    else if (H1set[H1ListeKopi[i]-1] == 0 && k == 1)
         {
           H1Liste[1][i-1] = Temp;
           Temp = H1ListeKopi[i];
         }
    else if (H1set[H1ListeKopi[i]-1]==1 && k==0)
         {
           H1Liste[1][i] = H1ListeKopi[i];
         }
```

```
      else if (H1set[H1ListeKopi[i]-1]==1 && k==1)
           {
             H1Liste[1][i-1] = H1ListeKopi[i];
           }
    }
    if (H1Liste[1][M*J-1]==-1)
      H1Liste[1][M*J-1] = Temp;

    AntalIt++;
  }while(Sammenlign(Skema, GammelSkema) == 0
        && AntalIt < 1000 );  /*Kan evt. supp. med afstand */
         /*fra bedst kendte loesning  */

  fprintf(fid,"Bedste Sluttid %d\n",BedstSlutTid);

 /* fprintf(fid,"LB = %d\n", LB);*/

  for(i=0; i<J*M; i++)\
    fprintf(fid,"Bedste Skema[%d] = (%d, %d)\n",i,
       BedstSkema[0][i], BedstSkema[1][i]);

  fprintf(fid,"AntalIt = %d\n",AntalIt);
  fclose(fid);
}

void main()
{
  time_t Tidstart,Tidslut, Tid;
  int i,j, Antal=10;
  for (j = 0;j<1;j++)
  {
    Tidstart=clock();
    for(i=0; i<Antal; i++)
      main2();
    Tidslut = clock();
    Tid= (Tidslut - Tidstart)*1000/CLOCKS_PER_SEC;
    printf("Tid = %i ved %d koersler\n",Tid, Antal);
  }
}
```

C.2 HFC

```
/* Program der implementerer HFC-heuristikken foreslaaet af
   C. Koulamas
   Begyndt d. 5. April 2001. Sidst aendret d. 5. Maj 2001.

   Definer MaxJ, MaxM, MaxOp, Max2M og P[] evt. ogsaa MO
*/

#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MaxJ  5   /*Antal job                                */
#define MaxM  5   /*Antal maskiner                          */
#define MaxOp 25  /*Antal operationer ialt                  */
#define Max2M 100 /*Antal 2-maskineprob.(M*(M-1)*M*(M-1)/4)*/

int J = MaxJ;
int M = MaxM;

/* Procestider                                              */

int P[MaxOp] = {
 54,  34,  61,   2,   9,
 39,  66,  89,  91,  53,
 27,  46,  28,  40,  64,
 98,  74,  96,  53,  80,
 73,  93,  50,   3,  49};

/*Hvilken maskine laver hvilken operation                   */

int MO[MaxOp];

/*Tidligste start for operationer (hoveder) saettes normalt*/
/*alle til 0                                               */

int hoved[MaxOp];
```

```
void Orden()
{
  int i,j;
  int PKopi[MaxOp];

  for(i=0; i<M*J; i++)
    PKopi[i]=P[i];

  for(i=0; i<J; i++)
    for (j=0; j<M; j++)
      P[i*M + MO[i*M+j] - 1]=PKopi[i*M+j];
}

void SkemaByg( int I[2][MaxJ])
{
  int i, j;
  int tminj[MaxJ], tminm[MaxM];

  for (i=0; i<J; i++)
    tminj[i]=0;

  for (j=0; j<M; j++)
    tminm[j]=0;

  for (j=0; j<M; j++)
  {
    for (i=0; i<J; i++)
    {
      if(tminj[I[0][i]]<tminm[j])
      {
        hoved[I[0][i]*M+j]=tminm[j];
tminm[j]=tminm[j]+P[I[0][i]*M+j];
tminj[I[0][i]]=tminm[j];
      }
      else
      {
hoved[I[0][i]*M+j]=tminj[I[0][i]];
tminj[I[0][i]]=tminj[I[0][i]]+P[I[0][i]*M+j];
tminm[j]=tminj[I[0][i]];
```

```
      }
    }
  }
}

int Index(int a, int b)
{
  int rk=0, i;

  for (i=0; i<a; i++)
    rk = rk + J-1-i;

  rk = rk + b - a - 1;

  return rk;
}

int Partition(int I[2][MaxJ], int p, int r)
{
  int x = I[1][p - 1];
  int i = p - 1;
  int j = r + 1;
  int tmp, tmp2;
  int keyl = p;

  while (i < j)
  {
    do
    {
      j--;
    } while (I[1][j - 1] > x);
    do
    {
      i++;
    } while (I[1][i - 1] < x);
    if (i < j)
    {
      tmp = I[1][i - 1];
      tmp2 = I[0][i - 1];
```

```
      I[1][i - 1] = I[1][j - 1];
      I[0][i - 1] = I[0][j - 1];
      I[1][j - 1] = tmp;
      I[0][j - 1] = tmp2;
    }
  }
  return j;
}

void QuickSort( int I[2][MaxJ], int p, int r )
{
  int q;
  if (p < r)
  {
    q = Partition(I,p,r);
    QuickSort(I,p,q);
    QuickSort(I,q+1,r);
  }
}

void SkemaByg2( int I[MaxJ][MaxM])
{
  int i, j;
  int tminj[MaxJ], tminm[MaxM];

  for (i=0; i<J; i++)
    tminj[i]=0;

  for (j=0; j<M; j++)
    tminm[j]=0;

  for (j=0; j<M; j++)
  {
    for (i=0; i<J; i++)
    {
      if(tminj[I[i][j]]<tminm[j])
      {
        hoved[I[i][j]*M+j]=tminm[j];
tminm[j]=tminm[j]+P[I[i][j]*M+j];
```

```
tminj[I[i][j]]=tminm[j];
      }
      else
      {
hoved[I[i][j]*M+j]=tminj[I[i][j]];
tminj[I[i][j]]=tminj[I[i][j]]+P[I[i][j]*M+j];
tminm[j]=tminj[I[i][j]];
      }
    }
  }
}

void main2()
{
  int Nummer = 1, MaxRest = 0, Raekke = 1, Position = 1;
  int JobSum[MaxJ], MaskinSum[MaxM], A=0, B=0;
  int LB = 0, SlutTid = 0, Cmax=0, Cmax_ny=0, cmaxbedst=0;
  int I[2][MaxJ], Ibedst[MaxJ][MaxM], H[Max2M];
  int i, j, k, l, r, q, tempj, tempm, Hnum=0;
  int rho[MaxJ][MaxM], T1, T2;

  char name[13]="HFC_out.txt";
  FILE *fid;
  fid = fopen(name,"w");

  for(i=0; i<J; i++)
    for(j=0; j<M; j++)
      MO[i*M+j]=j+1;

  Orden();

  for(i=0; i<M*J; i++)
    hoved[i]=0;

  /*Finder den nedre graense LB*/
  for (i=0; i<M; i++) /*for hver maskine                */
  {
    MaskinSum[i]=0;
    for (j=0; j<J; j++)
```

```
    {
      MaskinSum[i] += P[j*M+i];
    }
  }

  for (i=0; i<M; i++)   /*Finder den stoerste af disse     */
  {
    if (MaskinSum[i] > LB)
      LB = MaskinSum[i];
  }

  for (i=0; i<J; i++)
    JobSum[i]=0;

  for (i=0; i<J; i++) /*Finder nedre graense for hvert job */
  {
    for (j=0; j<M; j++)
    {
      JobSum[j] += P[j*M+i];
    }
  }

  for (i=0; i<J; i++)          /*Finder nedre graense LB   */
  {
    if (JobSum[i] > LB)
      LB = JobSum[i];
  }


  fprintf(fid,"LB = %d\n", LB);
  fprintf(fid, "\n");

/*HFC fase 1*/
  for (i=0; i<J; i++)          /*Nummererer og nulstiller I*/
  {
    I[0][i]=i;
    I[1][i]=0;
  }
```

```
for(i=0; i<M*(M-1)*M*(M-1)/4; i++)
  H[i]=0;

for (i=0; i<J-1; i++)        /*Finder I                    */
{
  for (j=i+1; j<J; j++)
  {
    for (k=0; k<M-1; k++)
    {
      for (l=k+1; l<M; l++)
      {
        if(P[k+i*M] < P[l+j*M])
          A=P[k+i*M];
        else
          A=P[l+j*M];

        if(P[l+i*M] < P[k+j*M])
          B=P[l+i*M];
        else
          B=P[k+j*M];

        if(A < B)
        {
          I[1][i]=I[1][i]-1;
          I[1][j]=I[1][j]+1;
          H[Hnum]=-1;
        }
        else if(B < A)
        {
          I[1][i]=I[1][i]+1;
          I[1][j]=I[1][j]-1;
          H[Hnum]=1;
        }
        Hnum=Hnum+1;
      }
    }
  }
}
```

```
  //sortering

  QuickSort(I,1,MaxJ);


  SkemaByg(I);

  Cmax = hoved[M*I[0][J-1]+M-1] + P[M*I[0][J-1]+M-1];

  for(j=0; j<M; j++)
    for(i=0; i<J; i++)
      Ibedst[i][j]=I[0][i];

/*HFC 2.fase*/
  for (i=0; i<J-1; i++)
  {
    if (I[0][i]<I[0][i+1])
      k=Index(I[0][i],I[0][i+1]);
    else
      k=Index(I[0][i+1],I[0][i]);

    for (j=0; j<M; j++)
    {
      T1=0;

      for (q=0; q<j-1; q++)
      {
for (r=q+1; r<j; r++)
{
l=Index(q,r);
T1=T1+H[k*M*(M-1)/2+l];
}
      }

      if (T1<j*(j-1)/2)
j++;

      T2=0;
```

```
      for (q=j+1; q<M-1; q++)
      {
for (r=q+1; r<M; r++)
{
l=Index(q,r);
T2=T2+H[k*M*(M-1)/2+l];
}
      }

      if (T2!=-T1)
j++;

      for (tempm=0; tempm<M; tempm++)
      {
for (tempj=0; tempj<J; tempj++)
{
  if (tempm<j)
    rho[tempj][tempm] = I[0][tempj];
  else
  {
    if (tempj<i || tempj>i+1)
      rho[tempj][tempm] = I[0][tempj];
    else if (tempj == i)
      rho[tempj][tempm] = I[0][i + 1];
    else if (tempj == i+1)
      rho[tempj][tempm] = I[0][i];
         }
       }
      }

      SkemaByg2(rho);

      for(r=0; r<M*J; r++)
{
      if (hoved[r]+P[r]>Cmax_ny)
Cmax_ny=hoved[r]+P[r];
}

      if (Cmax_ny<Cmax)
```

```
      {
for(r=0; r<J; r++)
{
  for(q=0; q<M; q++)
    Ibedst[r][q]=rho[r][q];
}

Cmax=Cmax_ny;
      }
    }
  }

  SkemaByg2(Ibedst);

  fprintf(fid, "Bedste loesning \n");

  for(i=0; i<J; i++)
  {
    fprintf(fid, "Hoved = ");
    for(j=0; j<M; j++)
      fprintf(fid, "%d ", hoved[i*M+j]);
    fprintf(fid,"\n");
  }

  fprintf(fid, "Cmax= %d",Cmax);

  fclose(fid);
}

void main()
{
  time_t Tidstart,Tidslut, Tid;
  int i,j, Antal=100;
  for (j = 0;j<10;j++)
  {
    Tidstart=clock();
    for(i=0; i<Antal; i++)
      main2();
    Tidslut = clock();
```

```
    Tid= (Tidslut - Tidstart)*1000/CLOCKS_PER_SEC;
    printf("Tid = %i ved %d koersler \n",Tid, Antal);
  }
}
```

Litteratur

[1] P. Brucker, J. Hurinck, B. Jurisch & B. Wö stmann, *A branch & bound algorithm for the open-shop problem*, Discrete Applied Mathematics 76, side 43-59, 1997.

[2] J. Dorn, M. Girsch, G. Skele & W. Slany, *Comparison of iterative improvement techniques for schedule optimization*, European Journal of Operational Research 94, side 349-361, 1996.

[3] C. Guéret & C. Prins, *Classical and new heuristics for the open-shop problem: A computational evaluation*, European Journal of Operational Research 107, side 306-314, 1998.

[4] S. K. Jacobsen, *Heltalsprogrammering, noter til 0431, Del 3*, IMSOR, 1990.

[5] C. Koulamas, *A new constructive heuristic for the flowshop scheduling problem*, European Journal of Operational Research 105, side 66-71, 1998.

[6] C.-F. Liaw, *A tabu search algorithm for the open shop scheduling problem*, Computers & Operations Research 26, side 109-126, 1999.

[7] C.-F. Liaw, *Applying simulated annealing to the open shop scheduling problem*, HE Transactions 31, side 457-465, 1999.

[8] M. G. Madsen, *Løsningsmetoder til Shop Scheduling*, IMM, 1999.

[9] H. Matsuo, C. J. Suh & R. S. Sullivan, *A controlled search simulated annealing method for the general jobshop scheduling problem*, Internt arbejdspapir nr. 03-44-88 fra Graduate school of Business, University of Texas, 1988.

[10] J. V. Mocellin & M. S. Nagano, *Evaluating the performance of tabu search procedures for flow shop sequencing*, Journal of Operational Research society vol 49 no 12, side 1296-1302, 1998.

[11] P. J. M. V. Larhoven, E. H. L. Aarts & J. K. Lenstra, *Job shop scheduling by simmulated annealing*, Operations Research Society of America

vol 40 no 1, side 113-126, 1992.