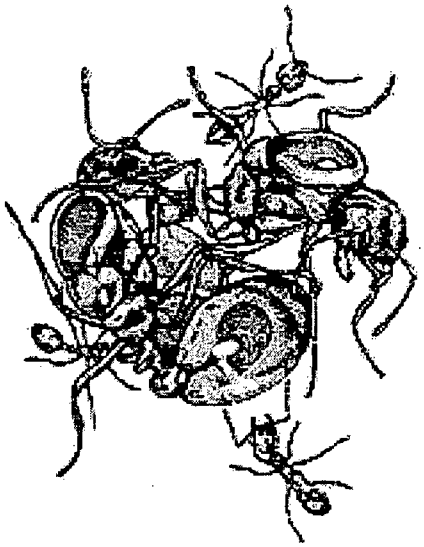
TEKST NR 426 2004

Myreintelligens

Distribuering af Ant Colony System Traveling Salesman Problem



Uffe Thomas Volmer Jankvist
Magnus Kaas Meinild

Vejleder: Keld Helsgaun

TEKSTER fra

IMFUFA ROSKILDE UNIVERSITETSCENTER
INSTITUT FOR STUDIET AF MATEMATIK OG FYSIK SAMT DERES FUNKTIONER I UNDERVISNING, FORSKNING OG ANVENDELSER

IMFUFA - Roskilde Universitetscenter - Postboks 260 - DK 4000
Roskilde Tlf.: 46742263 - Fax: 46743020 - Mail: imfufa@ruc.dk
Uffe Thomas Volmer Jankvist og Magnus Kaas Meinild
'Myreintelligens – Distribuering af Ant Colony System
Traveling Salesman Problem'
IMFUFA tekst nr. 426 - 136 sider - ISBN 0106-6242

Abstract

I dette projekt implementeres først et sekventielt Ant Colony System til løsning af Traveling Salesman Problem (ACS-TSP). Dernæst distribueres dette system i overenstemmelse med paralleliseringsstrategien *Parallelle myrer* på et netværk af pc'er.

Efterfølgende testes de to implementationer af ACS-TSP, den sekventielle og den distribuerede, på otte udvalgte problemtilfælde fra TSPLIB. De opnåede resultater fra de to udgaver af ACS-TSP sammenlignes dels indbyrdes og dels med resultater fra litteraturen. Ydermere diskuteres speedup'et for det distribuerede system.

Det konkluderes, at paralleliseringsstrategien *Parallelle myrer* ikke egner sig til distribuering på et netværk af pc'er.

Forord

Nærværende IMFUFA-tekst er en lettere revideret udgave af en projektrapport, der beskæftiger sig med en af ACO-heuristikkerne, nærmere betegnet heuristikken ACS samt distribuering af denne.

Projektet, der ligger til grund for IMFUFA-teksten, indgik som en del af overbygningsuddannelsen på Datalogi ved Roskilde Universitetscenter i foråret 2003. Projektet hørte under 2. modul og talte $^1/_2$ semesterværk (15 ECTS). Rapporten er skrevet i LaTeX.

Vi vil gerne takke vores vejleder Keld Helsgaun, for et konstruktivt og yderst engageret samarbejde. Vi vil også gerne takke lektor Jørgen Larsen (IMFUFA) for hans utrættelige besvarelser af LaTeX-relaterede spørgsmål.

Uffe Thomas Volmer Jankvist
Magnus Kaas Meinild

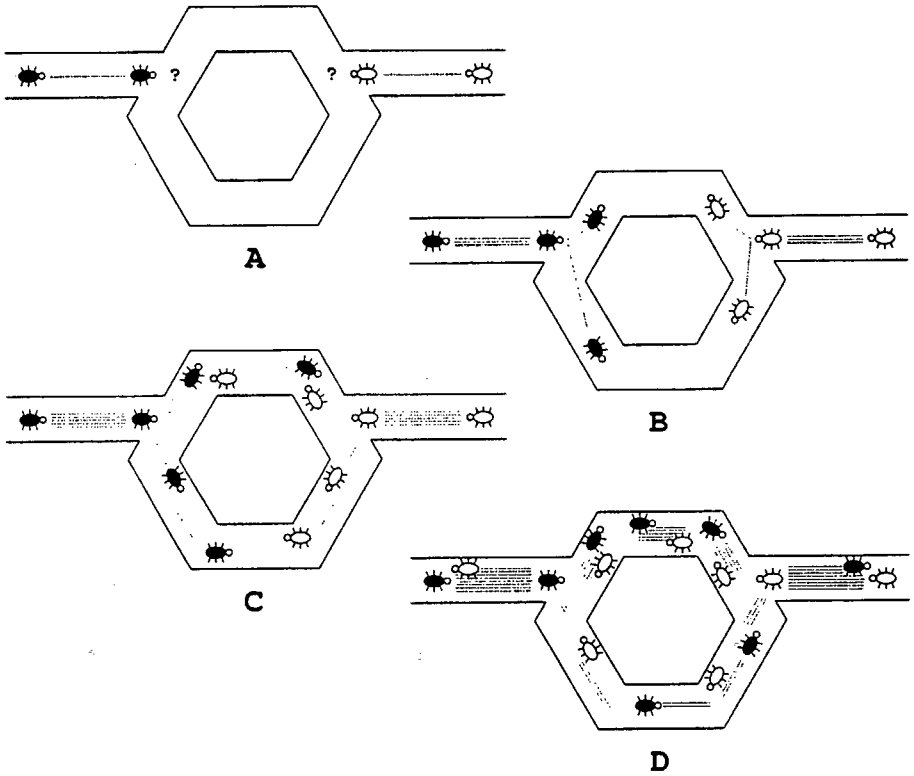
Datalogi OB, Roskilde, den 21. februar 2004

Indhold

1 Indledning

I dette projekt har vi arbejdet med implementeringen og distribueringen af en algoritme, der tager sit udgangspunkt i sværmintelligensen. De senere år har man, når man har skullet løse problemer – optimering og lignende – inden for datalogien, i højere og højere grad ladet sig inspirere af fænomener i naturen. Simuleret udglødning, genetiske algoritmer, sværmintelligens, og til en vis grad neurale netværk, er eksempler på dette. Sværmintelligensen beskæftiger sig med insekter. Det enkelte insekt, det være sig en bi eller en myre, er i sig selv ikke videre intelligent, men når insekterne er tilstrækkeligt mange, det vil sige er en sværm eller en tue, udgør de tilsammen en vis form for intelligens. Det er denne form for intelligens, man betegner som sværmintelligens. Sværmen eller tuen er således som selvstændig enhed betragtet i stand til at løse langt mere komplicerede problemer end det enkelte insekt er i stand til.

Heuristikker, som tager deres udgangspunkt i myretuens 'intelligens' til løsning af optimeringsproblemer, kaldes overordnet for ACO-heuristikker, ACO er en forkortelse af *Ant Colony Optimization*. Den ACO-heuristik, vi har beskæftiget os med i dette projekt, er en heuristik kaldet *Ant Colony System* (ACS). ACS gør, ligesom andre ACO-heuristikker, brug af, at myrerne efterlader duftstoffer (pheromon) på deres vej, når de bevæger sig i et terræn. Det er disse duftstoffer, som myrerne benytter til at kommunikere indbyrdes med – myrerne vælger nemlig at bevæge sig ad den vej, hvorpå der er ophobet mest pheromon. På denne måde er myrerne i stand til at finde den korteste vej imellem to punkter (for eksempel ind- og udgang i en labyrint), figur 1.1 viser dette. Myrerne er interesseret i at finde den korteste vej imellem de to indgange i 'labyrinten'. Det vil sige at de sorte myrer fra venstre har som mål at finde den korteste vej til den indgang, de hvide myrer fra højre benytter, og omvendt. Figur 1.1 A: Myrer ankommer til en skillevej i terrænet, hvor de skal vælge om de vil gå til højre eller venstre. I og med at de ingen ide har om, hvilken vej der er den mest fordelagtige, vælger de tilfældigt. Det kan antages, at halvdelen af myrerne i gennemsnit vil gå til højre, og den anden halvdel til venstre. Figur 1.1 B og 1.1 C viser, hvad der sker umiddelbart efter situationen i figur 1.1 A, når det antages at myrer bevæger sig med stort set samme hastighed. Antallet af linier er omtrent proportionalt med den ophobede mængde af pheromon på de pågældende vejstrækninger. I og med at den øverste sti i 'labyrinten' er kortere end den nederste, vil der i gennemsnit være flere myrer, der bevæger sig af denne, hvorfor pheromon også hurtigere ophobes her. Efter en kort forbigående periode vil pheromonmængderne på de to stier være tilstrækkelig store, og derfor være i stand til at påvirke de i systemet nyintroducerede myrers valg. Dette fremgår af figur 1.1 D. Nye myrer i systemet vil fra nu af vælge den øverste og kortere sti, i og med at de ved skillevejen vil blive udsat for større pheromonmængder fra denne sti. Dette har den selvforstærkende effekt, at flere og flere myrer



Figur 1.1 Hvordan myrer i naturen finder korteste vej [Dorigo and Gambadella, 1997]. (A) Myrerne ankommer til en skillevej. (B) Myrerne vælger tilfældigt at gå ad enten den øverste eller nederste sti. (C) I og med at myrerne stort set bevæger sig med konstant hastighed, vil de myrer som vælger den øverste – og kortere – sti nå deres bestemmelsessted før de myrer der vælger den nederste – og længere – sti. (D) Pheromonen, illustreret ved linierne, ophobes hurtigere på den korte sti, hvilket har den selvforstærkende effekt, at flere myrer vil gå ad denne.

efterhånden vil vælge den øverste og korteste sti. Snart vil alle myrer bevæge sig ad denne vej. I stedet for ind- og udgang i en labyrint vil de to punkter, som myrer ønsker at finde den korteste vej imellem, ofte være givet ved et bestemt maddepot og myretuen. I dette tilfælde er situationen den samme som den ovenfor beskrevne, da myrerne her jo både har en ud- og hjemtur. I det tilfælde hvor der er flere maddepoter placeret omkring en tue, vil myrerne ved hjælp af pheromoner være i stand til at lokalisere det nærmeste depot, tømme dette, derefter det næstnærmeste, tømme dette, og så videre [Bonabeau et al., 1999].

Ved at simulere myrer og deres adfærd har man en tilgang til at løse en række problemer inden for datalogien og andre fagområder. Myrealgoritmer kan som ovenfor nævnt benyttes til finde korteste vej imellem to punkter i en graf, for eksempel to datamater i et netværk. Ligeledes kan de benyttes til at finde en tur imellem n punkter i en graf. I dette projekt har vi taget vores udgangspunkt i den rejsende sælgers problem, forkortet TSP. TSP går, kort fortalt, ud på, at

en sælger skal besøge n byer og derefter vende tilbage til sit udgangspunkt, blot skal dette gøres således, at rejseafstanden bliver mindst mulig. Det skal i denne forbindelse nævnes, at TSP formentlig ikke er det bedste problem at anvende myrealgoritmer på, da der findes langt smartere algoritmer, som løser dette problem til næroptimalitet. Men det er et illustrativt prototype-problem, som kan vise, hvor god en given algoritme er. Tilmed kan algoritmer og teknikker udviklet til løsning af TSP ofte også overføres til andre kombinatoriske optimeringsproblemer [Helsgaun, 2003a]. Der findes til TSP også en lang række veldefinerede og kendte problemtilfælde, som man kan teste sin algoritme på.

Antallet af ture for et TSP er et eksempel på en såkaldt kombinatorisk eksplosion (illustreret ved tabel 1.1). Til et symmetrisk TSP[1] med n byer findes der

$$\frac{(n-1)\cdot(n-2)\cdot\ldots\cdot 2\cdot 1}{2} = \frac{(n-1)!}{2}$$

mulige ture.

n	antal ture
3	1
4	3
5	12
6	60
7	360
8	2.520
9	20.160
10	181.440
11	1.814.400
12	19.958.400
13	239.500.000
14	3.113.500.000
15	43.589.000.000
17	10.461.000.000.000
19	3.201.200.000.000.000
21	1.216.500.000.000.000.000
23	562.000.000.000.000.000.000
25	310.220.000.000.000.000.000.000
30	4.420.900.000.000.000.000.000.000.000.000
35	147.620.000.000.000.000.000.000.000.000.000.000.000
40	10.019.900.000.000.000.000.000.000.000.000.000.000.000.000.000

Tabel 1.1 Antallet af mulige ture for et symmetrisk TSP med n byer [Helsgaun, 2003a]. Et eksempel på en kombinatorisk eksplosion.

[1] Et symmetrisk TSP er et, hvor der er lige langt fra by i til by j og fra by j til by i.

For store n bliver antallet af mulige ture ubehageligt stort. Et TSP med blot 62 byer vil have cirka 2,5 gange så mange mulige ture, som der er atomer i universet (10^{83}). En konsekvens af dette er at det tager uforholdsmæssig lang tid at udregne samtlige ture for et givet TSP. Antages det at det tager et nanosekund at bestemme en mulig tur, vil tidforbruget selv for små værdier af n blive stort (se tabel 1.2). Grundet dette faktum benytter man sig oftest ikke af eksakte algoritmer, men derimod af approximative.

n	antal ture	tidsforbrug
20	$\approx 10^{17}$	≈ 2 år
25	$\approx 10^{24}$	$\approx 10^{7}$ år
30	$\approx 10^{31}$	$\approx 10^{15}$ år

Tabel 1.2 Tidsforbruget for beregning af samtlige ture i et givet TSP med n byer [Helsgaun, 2003a].

ACS er et eksempel på en approximativ algoritme der kan finde forholdsvis fornuftige løsninger på givne TSP-tilfælde. Designet af ACS medfører, at antallet af beregninger, der skal foretages for et TSP-tilfælde med n byer i værste tilfælde er proportionalt med n^2. Denne kvadratiske stigning i antallet af beregninger begrænser størrelsen, med hensyn til n, af de TSP-tilfælde, det er muligt for ACS at løse inden for en overskuelig tidsperiode. Hvis man distribuerer ACS over flere maskiner, er en mulig tese, at man således vil være i stand til at løse større problemer inden for en overskuelig tidsramme. Mere præcist er vores hypotese, at køretiden er omvendt proportional med antallet af maskiner. I fagtermer er dette ensbetydende med at have et *speedup* på over 1. (Speedupet beskriver kort fortalt, hvor mange gange hurtigere et distibueret/parallelt system er i forhold til et lignende sekventielt system.)

Vi har i dette projekt først implementeret og kørt en sekventiel udgave af ACS og derefter en distribueret udgave af denne. Der findes forskellige strategier til at distribuere/parallelisere ACS. Vi vil dog i denne rapport hovedsageligt beskæftige os med én bestemt strategi, nemlig den der kaldes *Parallelle myrer*. Denne strategi går i grove træk ud på, at der er én myre (en *mester*), der 'dirigerer' et antal andre myrer (*slaver*) rundt i et givet TSP-tilfældes graf. Mesteren og de enkelte slaver er tilknyttet hver deres processor. Slaverne traverserer grafen og finder således mulige ture til TSP-tilfældet. På baggrund af disse ture opdaterer mesteren pheromonniveauet på grafens kanter. Slaverne bliver løbende informeret om dette pheromonniveau, så de kan bygge deres ture ud fra dette. Slaverne udgør således de enkelte uintelligente myrer, hvorfor mesteren, lidt groft sagt, gør det ud for tuens samlede intelligens, da det jo er den der kontrollerer pheromonværdierne.

Med udgangspunkt i de to implementationer, henholdsvis den sekventielle og den distribuerede udgave af ACS, ønsker vi i dette projekt, at undersøge den ovenfor formulerede hypotese, om hvorvidt køretiden af et distribueret ACS vil være omvendt proportionalt med antallet af processorer. Eller med andre ord, hvorvidt et sådant distribueret system vil have et speedup på over 1.

I det efterfølgende gives vores motivation for at arbejde med ACO-heuristikker og distribution af disse. Derefter præsenteres problemformuleringen samt en

uddybning og afgrænsning af denne. Til sidst gives en beskrivelse af rapportens opbygning.

1.1 Motivation

Årsagen til at vi har valgt at arbejde med distribuering af ACO skyldes hovedsageligt to ting. For det første havde vi et ønske om at arbejde med algoritmer. For det andet ønskede vi også at blive klogere på, hvad det vil sige at arbejde med datamater i et netværk.

Vi var fra første færd fascinerede af sværmintelligensen og brugen af denne indenfor algoritmedesign. Vi lagde os derfor fast på at arbejde med ACO-heuristikker. Da vi senere fandt ud af, at der var folk, der arbejdede med parallelle ACO-heuristikker øjnede vi chancen for, at kombinere vores to interessefelter ved at distribuere ACS på et netværk af datamater.

1.2 Problemformulering

Vi ønsker i projektet at:

- Implementere en sekventiel udgave af ACS-TSP i overenstemmelse med Bonabeau et al.
- Med udgangspunkt i Randall og Lewis at distribuere ACS-TSP asynkront på et netværk af datamater ved hjælp af strategien *Parallelle myrer*.
- Teste vores egen distribuerede udgave af ACS-TSP på udvalgte TSP-tilfælde.
- Undersøge speedup'et for vores distribuerede system og sammenligne dette med speedup'et for Randall og Lewis' system.

1.2.1 Uddybning og afgrænsning

Udgangspunktet for vores distribuering af ACS-TSP og vores videre undersøgelser i denne rapport er en artikel fra 2002 skrevet af Randall og Lewis [Randall and Lewis, 2002]. Denne artikel har været en hovedkilde igennem hele projektforløbet. Det er også resultaterne i Randall og Lewis' artikel vi ønsker at sammenligne vores resultater med.

For at have et referenceprogram for vores distribuerede udgave af ACS har vi implementeret en sekventiel udgave af ACS. Til implementeringen af sekventiel ACS har vi taget udgangspunkt i en fremstilling af Bonabeau, Dorigo og Theraulaz fra 1999 [Bonabeau et al., 1999]. Også dette arbejde må siges at have været en hovedkilde for os igennem projektforløbet.

Vi har for overskuelighedens skyld begrænset os til udelukkende at behandle symmetriske TSP i vores implementeringer. Både den sekventielle og den distribuerede udgave af ACS er skrevet i Java, da det er det programmeringssprog

vi er vant til at arbejde med. I kraft af at koden for implementationen i Randall og Lewis' artikel ikke er tilgængelig – kun pseudo-koden er tilgængelig – vil vores implementering af distibueret ACS sandsynligvis afvige en smule fra denne. Det samme gør sig gældende for vores implementering af sekventiel ACS, i og med at der hos Bonabeau et al. ligeledes kun forelægger pseudo-kode.

1.3 Rapportens opbygning

Rapporten består af to dele. Første del omhandler den sekventielle udgave af ACS og anden del den distribuerede. I anden del trækkes så vidt muligt på allerede introducerede begreber fra første del. Efter anden del bringes diskussionen samt konklusionen.

I kapitel 2 gives først en formel beskrivelse af den rejsende sælgers problem. Dernæst gennemgås ACS, herunder de forskellige opdateringsregler samt brugen af kandidatlister.

I kapitel 3 gives en beskrivelse af, samt dokumentationen for vores implementering af sekventiel ACS.

Kapitel 4 indeholder en beskrivelse af de forsøg, vi har udført med sekventiel ACS samt resultaterne af disse.

Kapitel 5 beskriver først klassificeringen af paralleldatamater. Dernæst beskrives teknikker til distribuering af algoritmer. Til sidst introduceres visse nøglebegreber og målestokke til brug i den senere analyse af vores distribuering.

I kapitel 6 bringes en gennemgang af de forskellige paralleliseringsstrategier, herunder *Parallelle myrer* som er den vi benytter. Til sidst præsenteres Randall og Lewis' pseudo-kode for parallel ACS.

I kapitel 7 gives en beskrivelse af, samt dokumentationen for vores implementering af den distibuerede udgave af ACS.

Kapitel 8 indeholder beskrivelsen af forsøgene udført med den distribuerede udgave af ACS samt resultaterne af disse. Det er også her at sammenligningen af vores resultater med relevante resultater fra litteraturen finder sted.

I kapitel 9 forefindes rapportens egentlige diskussion samt en løbende opsummering af de problemstillinger der er blevet rejst i rapporten. En kort perspektivering forefindes også her.

Kapitel 10 indeholder rapportens konklusion. Vi forholder os her til vores problemformulering samt visse hovedpointer fra diskussionen.

I appendiks A findes et udvalg af relevant grafteori. Det er især formuleringen af den rejsende sælgers problem der trækker på de grafteoretiske definitioner i dette appendiks.

Appendiks B indeholder kildekoden for vores implementeringer. Afsnit B.1 indeholder al ACS-relateret funktionalitet for den sekventielle såvel som for den distribuerede udgave. En trådet udgave af sekventiel ACS findes også her. Afsnit B.2 indeholder den del af koden der sørger for kommunikationen i den

distribuerede udgave. Afsnit B.3 indeholder koden for den del af programmet der sørger for indlæsningen af problemtilfældene.

I appendiks C findes de gennemsnitlige køretider samt standard afvigelser for de forsøg vi har udført med henholdsvis den sekventielle og den distibuerede udgave af ACS.

Appendiks D indeholder grafer for det relative speedup, dels som funktion af antallet af byer i problemtilfældet, dels som funktion af antallet af processorer.

Litteraturlisten indeholder samtlige kilder anvendt i rapporten.

Kildehenvisninger i rapporten laves på følgende måder: Når der er tale om en specifik information kommer henvisningen i forbindelse med denne eller i umiddelbart forlængelse. Når kilder har været benyttet som grundlag for hele afsnit, kommer henvisningerne til disse kilder i starten af de pågældende afsnit. Kildehenvisninger ser iøvrigt ud på følgende måde: [forfatter(e), udgivelsesår].

Del I

Sekventiel ACS

2 Ant Colony Optimization

I dette kapitel gives først en mere formel beskrivelse af den rejsende sælgers problem, da det som tidligere nævnt er dette problem, vi beskæftiger os med. Med udgangspunkt i denne beskrivelse forklares ACO-algoritmen ACS.

2.1 Den rejsende sælgers problem

Traveling Salesman Problem (TSP) består i, at en sælger skal besøge en række byer og derefter returnere til sit udgangspunkt. Givet et antal byer og et antal veje, som forbinder byerne, skal sælgeren finde den korteste rute rundt til alle byerne uden at besøge samme by mere end én gang. Til beskrivelse af den rejsende sælgers problem har vi taget udgangspunkt i følgende kilder: [Lawler et al., 1985], [Dolan and Aldous; 1995], [Reinelt, 1994] og [Minieka, 1978]. For en kort gennemgang af et udvalg af grafteori se appendiks A.

Betragtes TSP som et grafteoretisk problem, kan situationen beskrives ved en simpel, komplet og vægtet graf $G = (V, E, D)$. $V = \{1, 2, \ldots, n\}$ er mængden af byer. $E = \{(i, j) \mid i, j \in V\}$ er mængden af alle uordnede par af elementer fra V, for eksempel repræsenterer parret (i, j) vejen fra by nummer i til by nummer j. I og med at G er komplet, kan hver enkelt knude parres med alle andre knuder, og da G også er simpel følger at $(i, j) = (j, i)$, med $i \neq j$. D er en $n \times n$ matrix hvor d_{ij} er vægten af kanten (i, j), som beskriver omkostningen af en rejse fra i til j. Til hver kant knyttes altså en funktion $d : E \rightarrow \mathbf{R}$ kaldet en *omkostningsfunktion*, som til hver kant $(i, j) \in E$ knytter en vægt d_{ij}. Den samlede vægt af en delmængde af kanter $F \subseteq E$ er defineret som

$$d(F) = \sum_{(i,j) \in F} d_{ij}. \tag{2.1}$$

Man kan sige, at TSP består i at finde den *korteste* Hamiltoniske kreds, idet en *Hamiltonisk kreds* er en cykel af længde n i en graf med n knuder, det vil sige $V_n = V$.

Et TSP siges at være *symmetrisk*, hvis det for alle i og j gælder, at $d_{ij} = d_{ji}$. Matricen D er da en symmetrisk matrix. Et asymmetrisk TSP adskiller fra et symmetrisk ved at være beskrevet ved en digraf, altså en retningsorienteret graf. For dette problem behøver matricen D ikke være symmetrisk.

En matematisk definition af et symmetrisk TSP kunne lyde:

> *Givet en simpel, komplet og vægtet graf $G = (V, E, D)$, er det symmetriske TSP at finde den korteste Hamiltoniske kreds på G* [Reinelt, 1994].

Hvis et symmetrisk TSP opfylder *trekantsuligheden*, altså

$$d_{ij} \leq d_{il} + d_{lj} \tag{2.2}$$

for alle knuder $i, j, l \in V$, kaldes det et *metrisk* TSP. I disse problemer svarer knuderne til punkter i et metrisk rum, og kanternes vægte er afstandene imellem disse givet ved den til rummet hørende metrik. For eksempel er et *euklidisk* TSP defineret ved en mængde af punkter i planet, hvor kanternes vægte er givet ved den euklidiske afstand imellem to punkter.

2.2 Ant Colony System

Der findes adskillige ACO-heuristikker, som for eksempel *Ant System* (AS), *MAX-MIN Ant System* og *Ant Colony System* (ACS). Vi skal i denne rapport kun beskæftige os med den sidstnævnte, nemlig ACS. Ant Colony System er en forbedring af Ant System (se [Bonabeau et al., 1999]). ACS blev introduceret af Dorigo og Gambardella grundet AS's manglende evne til at finde gode løsninger på større problemer inden for en acceptabel tidsramme. Nedenstående formulering af ACS bygger på beskrivelserne i [Bonabeau et al., 1999] og [Randall and Lewis, 2002].

Betragt et TSP med n byer, hvor afstanden (omkostningen) imellem by i og by j er benævnt med d_{ij}. Fordel m myrer tilfældigt på de n byer ($m \leq n$). I diskrete tidsskridt t tillades det en myre at traversere en kant, indtil alle byer er besøgt. Hver myre indeholder en hukommelse (også kaldet en tabuliste) som holder styr på, hvilke byer myren allerede har besøgt. For hver myre k definerer denne hukommelse en mængde J_i^k af byer, som den pågældende myre stadig skal besøge, når den står i by i (til at begynde med indeholder J_i^k alle byer på nær i). Ved at udforske J_i^k kan en myre undgå at besøge den samme by to gange. Myrerne efterlader en substans kaldet *pheromon*. Denne substans benyttes til at kommunikere med andre myrer fra kolonien om de enkelte kanters anvendelighed. Den opsamlede mængde af pheromon på en kant fra by i til by j benævnes med τ_{ij}. Pheromon på en kant skal repræsentere den tillærte ønskelighed af at vælge by j efter by i. I modsætning til afstand er pheromon en mere global form for information. Pheromon opdateres løbende under løsningen af problemet for således at reflektere den erfaring myrerne har opnået under problemløsningen.

Såvel pheromonværdi τ_{ij} som hukommelsen J_i^k er vigtige elementer i den nedenfor forklarede *overgangsregel*[1]. Et tredje element i overgangsreglen er den inverse af afstanden imellem to byer i og j,

$$\eta_{ij} = \frac{1}{d_{ij}}, \tag{2.3}$$

også kaldet *synligheden*. Synligheden beskriver den heuristiske ønskelighed af at vælge by j efter by i og baserer sig udelukkende på lokal statisk information. Synligheden benyttes til at guide en myre i sin søgning, på trods af at synligheden i sig selv ikke er nogen god metode til konstruktion af ture.

[1] Dansk oversættelse af *transition rule*.

2.2.1 Overgangsregel

Ved begyndelsen af hvert tidsskridt t vælger en myre k, placeret på by i, den næste by j i turen på baggrund af følgende regel:

$$j = \begin{cases} \arg\max_{u \in J_i^k} \{[\tau_{iu}(t)] \cdot [\eta_{iu}]^\beta\} & \text{hvis } q \leq q_0 \\ J & \text{hvis } q > q_0, \end{cases} \tag{2.4}$$

hvor q er en ligelig fordelt tilfældighedsvariabel tilhørende intervallet $[0, 1]$, q_0 er en justerbar parameter ($0 \leq q_0 \leq 1$), og $J \in J_i^k$ er en by der vælges tilfældigt i overenstemmelse med følgende sandsynlighed

$$p_{iJ}^k(t) = \frac{[\tau_{iJ}(t)] \cdot [\eta_{iJ}]^\beta}{\sum_{l \in J_i^k} [\tau_{il}(t)] \cdot [\eta_{il}]^\beta}. \tag{2.5}$$

Når $J \notin J_i^k$ sættes $p_{iJ}^k(t) = 0$. Parameteren β vælges som regel til at være en positiv konstant, da dette favoriserer kortere kanter.

Ligning (2.4) er en yderst grådig uvælgelsesstrategi, der favoriserer byer, som besidder den bedste kombination af korte afstande og høje pheromonværdier. Ligning (2.5) er til for at udglatte denne grådighed en smule ved at tillade en sandsynlighedsbaseret udvælgelse af den næste by. I den oprindelige AS-algoritme oppererer man også med en α-parameter på τ, hvilket gør det muligt at justere endnu mere på forholdet mellem afstanden og pheromonværdien. Det er vigtigt at bemærke at selv om ligning (2.5) forbliver uændret i ét tidsskridt, kan værdien af sandsynligheden $p_{iJ}^k(t)$ meget vel ændre sig for to myrer på samme by i, da $p_{iJ}^k(t)$ er en funktion af J_i^k – det vil sige den delvise løsning bygget af myre k.

2.2.2 Lokal pheromonopdateringsregel

Den lokale opdatering af pheromonværdierne foregår samtidigt med at en myre er ved at bygge en tur. Når myre k er i by i og vælger by $j \in J_i^k$, opdateres pheromonkoncentrationen ved brug af følgende regel:

$$\tau_{ij}(t) \leftarrow (1 - \rho) \cdot \tau_{ij}(t) + \rho \cdot \tau_0, \tag{2.6}$$

hvor ρ er en parameter der sikrer at mængden af pheromon på de enkelte kanter henfalder over tid ($0 < \rho < 1$) og τ_0 er den initiale pheromonværdi på kanterne. Dorigo og Gambardella har fundet, at man ved at sætte $\tau_0 = (n \cdot L_{nn})^{-1}$, hvor L_{nn} er længden af en tur T_{nn}, produceret ved hjælp af nærmeste nabo heuristikken, kan opnå gode resultater [Dorigo and Gambadella, 1997]. Pseudokoden for nærmeste nabo heuristikken gennemgås i algoritme 2.1.

Når en myre traverserer en kant, vil anvendelsen af den lokale opdateringsregel bevirke, at pheromonværdien på den pågældende kant bliver formindsket. Effekten af dette er, at kanter, der besøges ofte, bliver mindre og mindre attraktive, hvilket indirekte favoriserer udforskningen af endnu ikke besøgte kanter. En konsekvens af dette er at myrerne ikke finder de samme ture. Dette gør det mere sandsynligt at en af dem vil finde en forbedret tur, i modsætning til hvis alle

```
sæt første_by til en tilfældig by;
aktuelle_by ← første_by;
for(antallet af byer minus én)
  sæt næste_by til den by der er nærmest
aktuelle_by
  og som endnu ikke har været besøgt;
  aktuelle_by ← næste_by;
end for;
gå til første_by;
return T_nn og L_nn;
```

Algoritme 2.1 Pseudo-kode for nærmeste nabo heuristik.

myrernes ture konvergerede mod een og samme tur, hvilket jo ville gøre det formålsløst at benytte m myrer. Den lokale opdateringsregel gør det altså muligt at tidligt besøgte byer i en myres tur først bliver udforsket senere i en anden myres tur. Med andre ord er effekten af den lokale opdateringsregel at få den føromtalte tillærte ønskelighed til at ændre sig dynamisk i løbet af problemløsningen.

2.2.3 Global pheromonopdateringsregel

Efter en endt iteration, det vil sige efter at samtlige m myrer har konstrueret en tur, foretages en global opdatering af kanternes pheromonværdier. Kanterne tilhørende den hidtil bedste løsning belønnes med en forøgelse af deres pheromonværdier. Denne forøgelse bestemmes ved brug af følgende regel:

$$\tau_{ij}(t) \leftarrow (1-\gamma) \cdot \tau_{ij}(t) + \gamma \cdot \Delta\tau_{ij}(t), \tag{2.7}$$

hvor γ er en global henfaldsparameter ($0 < \gamma < 1$) og hvor $\Delta\tau_{ij}(t)$ benyttes til at forstærke pheromonværdierne i løsningen i overenstemmelse med følgende formel:

$$\Delta\tau_{ij}(t) = \frac{Q}{L^+} \quad \text{hvis } (i,j) \in T^+. \tag{2.8}$$

Her er T^+ den hidtil bedste tur, L^+ er længden (omkostningen) af denne, og Q er en problemafhængig konstant. Randall og Lewis sætter $Q = 1$, hvorimod Bonabeau et al. har sat $Q = 100$.

2.2.4 Kandidatliste

ACS kan benytte en kadidatliste[2]. Kandidatlister benyttes ofte når man arbejder med store TSP-tilfælde. En kandidatliste er en datastruktur, som indeholder en liste af fortrukne byer man ønsker at gå til, når man står i en given by. Det vil sige at man, i stedet for at undersøge alle de mulige byer, man kan gå til fra en given by, nu kun skal undersøge byerne i kandidatlisten. Først når kandidatlisten ikke indeholder flere ikke-besøgte byer, tages byer, som ikke er i kandidatlisten

[2] Randall og Lewis benytter ikke kandidatlister i deres implementation, men det gør derimod Bonabeau et al.

```
initialiser pheromon på alle kanter;
for(antallet af tidsskridt t_max)
  placer hver myre på en tilfældig by således, at
  to myrer ikke er på samme by;
  for(antallet af myrer m)
    if(kandidatlisten indeholder ikke-besøgte byer)
      vælg næste_by der skal besøges iblandt de cl
      byer i kandidatlisten ved at benytte
      overgangsreglen ((2.4) og (2.5));
    else
      vælg nærmeste ikke-besøgte by;
    end if;
    for(hver myre)
      opdater hver kants pheromon i overenstemmelse
      med den lokale opdateringsregel ((2.6));
    end for;
  end for;
  if(L < L+)
    L+ ← L;
  end if;
  forøg pheromonværdien på kanterne tilhørende
  T+ ved hjælp af den globale opdateringsregel
  ((2.7) og (2.8));
end for;
return T+ og L+;
```

Algoritme 2.2 Pseudo-kode for ACS anvendt på TSP. Brugen af kandidatliste stammer fra [Bonabeau et al., 1999]. Det at undgå placering af to myrer på samme by stammer fra [Randall and Lewis, 2002].

i betragtning. Kandidatlisten til en given by indeholder cl byer. Byerne i en kandidatliste er ordnet efter prioritet, for eksempel afstand. Det vil sige den by med højst prioritet er den der først tages i betragtning.

Brug af kandidatliste ved ACS-TSP virker på følgende vis: En myre begrænser sit valg af næste by til kandidaterne i den bys kandidatliste, hvor myren er placeret. Først når myren befinder sig i en by med en liste, der kun indeholder allerede besøgte byer, tager den byer som ikke er indeholdt i kandidatlisten i betragtning. Hvis kandidatlisten indeholder ikke-besøgte byer vælges næste by i overenstemmelse med overgangsreglen (ligningerne (2.4) og (2.5)), ellers vælges den på anden vis, for eksempel ved brug af nærmeste nabo heuristikken.

Kandidaterne i en kandidatliste kan være givet efter forskellige kriterier. En almindelig tilgang er nærmeste nabo, det vil sige at listen indeholder de cl nærmeste naboer til den givne by. Denne tilgang benyttes af Bonabeau et al. med $cl = 15$ samt den restriktion, at når kandidatlisten kun indeholder besøgte byer vælges den nærmeste ikke-tidligere besøgte by.

En anden tilgang til udvælgelse af kandidater er kvadrantmetoden. Denne går ud på at opfatte hver by som origo i et koordinatsystem, og så vælge de s nærmeste byer fra hvert kvadrant. Er $s = 4$ fås således $|cl| = 16$. Denne tilgang

kan ofte vise sig at være mere fordelagtig end nærmeste nabo, specielt ved større problemtilfælde[3].

[3]For mere avancerede strategier til udvælgelse af kandidater se [Helsgaun, 2000].

3 Implementation

I dette kapitel analyseres og beskrives vores implementation af den sekventielle udgave af ACS.

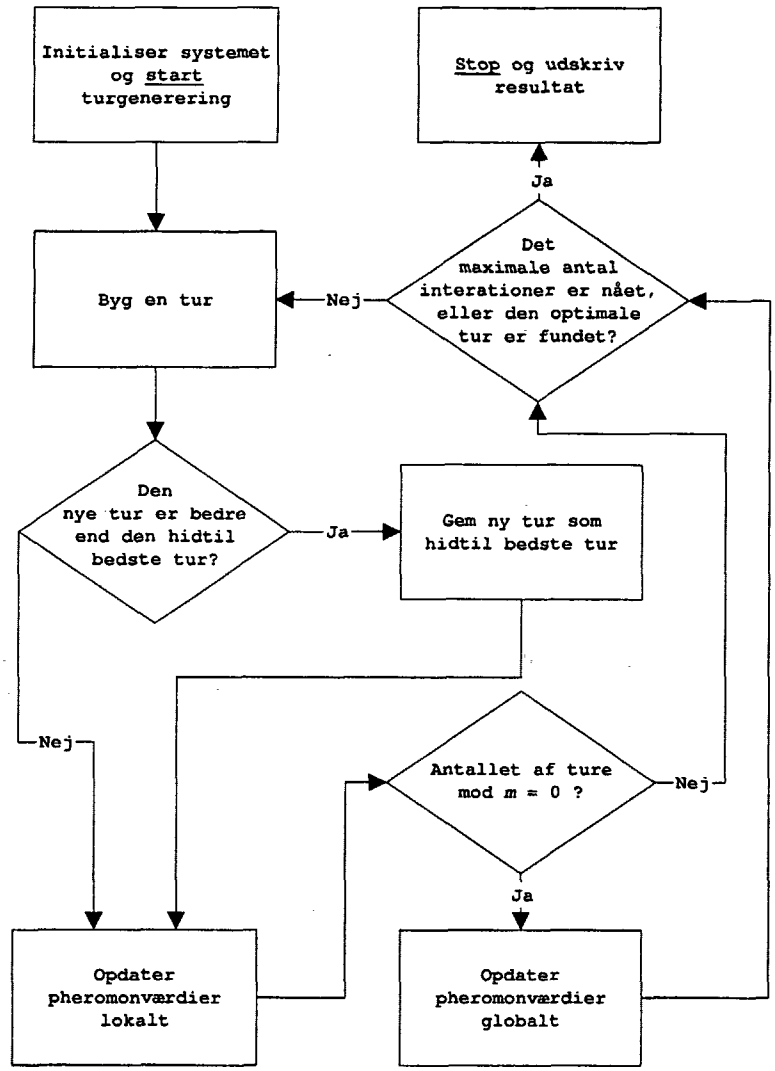
3.1 Analyse

Den sekventielle udgave af ACS er udviklet med det formål at have en simpel implementation af ACS til at tage udgangspunkt i, så vi i første omgang kunne koncentrere os om at få implementationen af ACS-heuristikken til at fungere korrekt. Den simple implementation skal kunne bygge ture og udføre lokal henholdsvis global pheromonopdatering på korrekt vis.

3.1.1 Central funktionalitet

- At en myre kan finde en lovlig tur ved hjælp af overgangsreglen (ligning (2.4) og (2.5)).
- At systemet kan opdatere pheromonværdier ved hjælp af den lokale opdateringsregel (ligning (2.6)) efter hver ny tur.
- At systemet kan opdatere pheromonværdier ved hjælp af den globale opdateringsregel (ligning (2.7) og (2.8)) til sidst i hver iteration, efter at alle m myrer har bygget en tur.
- At systemet kan holde styr på, hvilken tur der er den bedste på et givet tidspunkt.

På figur 3.1 findes en illustration af, hvordan de ovennævnte centrale dele i programmet samarbejder.

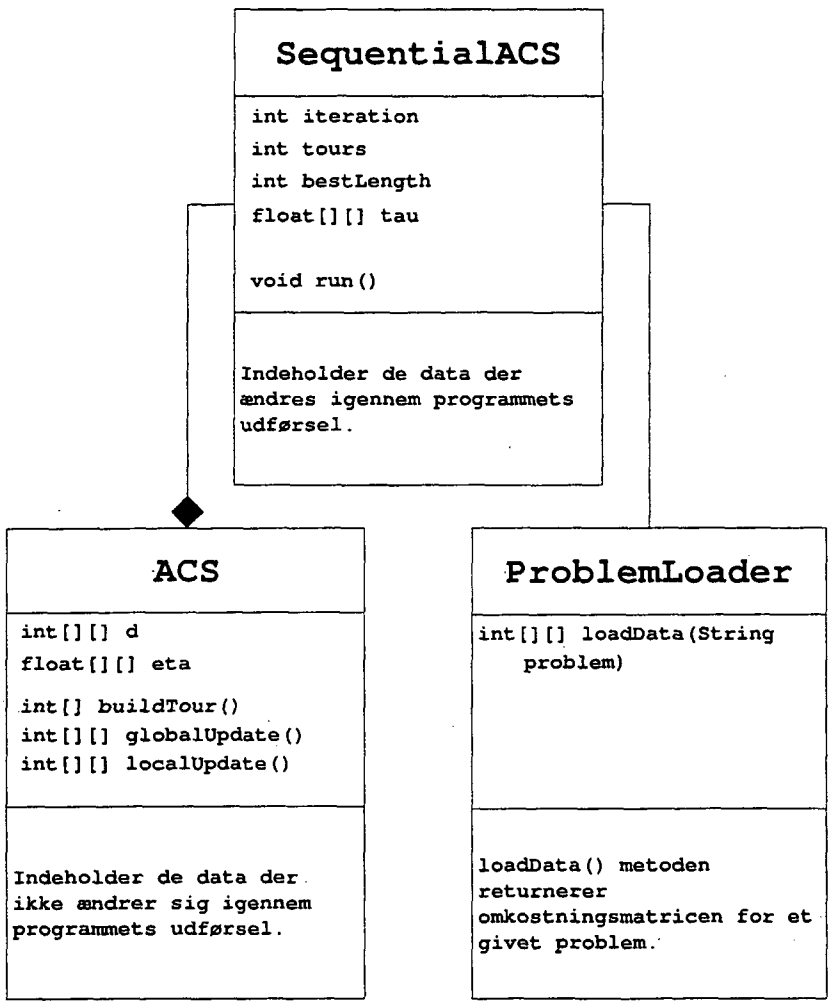


Figur 3.1 Illustration af programmets forløb med hensyn til de vigtigste dele af programmet. Forløbet starter i kassen øverst til venstre. Forløbet stopper i kassen øverst til højre.

3.1.2 Centrale datastrukturer

For at holde programmet forholdsvist enkelt har vi i videst mulig udstrækning søgt at holde os til primitive datatyper. Således er en by repræsenteret ved en Java `int`. En tur er et array af `int` typen, hvor array'ets orden bestemmer rækkefølgen, byerne besøges i. Pheromonværdierne τ_{ij} er repræsenteret ved en matrix af typen `float`, hvor index (i, j) i matricen `tau` angiver pheromonniveauet på kanten (i, j). På samme måde er omkostningen d_{ij} af en given kant (i, j) angivet ved index (i, j) i en matrix `d` af typen `int`. Til sidst er synlighe-

den η_{ij} af en by j fra en by i angivet ved index (i, j) i en matrix `eta` af typen `float`. I vores program indeholder `eta` dog η_{ij}^{β}, af hensyn til køretiden. Hvis vi ikke havde beregnet alle η_{ij}^{β} på forhånd, ville programmet skulle have foretaget denne potensberegning omtrent $n \cdot cl \cdot m$ gange i hver iteration. På figur 3.2 findes en illustration af hvordan ovennævnte optræder i programmet.



Figur 3.2 Illustration af hvordan de forskellige dataobjekter indgår i programmet. Dobbeltarrayet `tau` er pheromonmatricen. Dobbeltarrayet `d` er omkostningsmatricen. Dobbeltarrayet `eta` er synlighedsmatricen.

3.2 Afvigelser fra hovedkilder

Vores implementering af sekventiel ACS adskiller sig på visse punkter en smule fra versionerne i henholdsvis [Bonabeau et al., 1999] og [Randall and Lewis,

2002]. Som tidligere nævnt benytter Randall og Lewis ikke kandidatlister og Bonabeau et al. tager ikke højde for, at to myrer ikke placeres i samme startby. Vi tager, som beskrevet i pseudo-koden (algoritme 2.2), højde for begge dele. Til beregning af kandidatlisterne benytter vi i overenstemmelse med Bonabeau et al. nærmeste nabo. Vi benytter os ligeledes af en global pheromonopdateringsregel, der er lig den som Bonabeau et al. benytter (det er denne, der er beskrevet i afsnit 2.2.3). Randall og Lewis benytter en anden version (se afsnit 6.2).

Ydermere benytter vi os i vores implementation af sekventiel ACS også af sæd i vores tilfældighedsgenerator. Dette gør det muligt af reproducere tidligere resultater fundet ved brug af algoritmen.

Med hensyn til parameterværdierne i vores implementation af sekventiel ACS holder vi os til Bonabeau et al., hvilket gør det muligt for os at sammenligne vores resultater med deres (se afsnit 4.2). Værdierne af vores parametre kan ses i tabel 4.2 i afsnit 4.2.

3.3 Design og afprøvning

Selve udviklingen af programmet er foregået løbende, i og med at vi har ønsket at vores sekventielle udgave og vores distribuerede udgave af ACS har mest mulig kode tilfælles. Derfor skrev vi først det sekventielle program som et helt selvstændigt program bestående af næsten udelukkende procedural kode. Den del af programmet, der stod for indlæsning af data, valgte vi dog fra starten at trække ud i en separat klasse for på den måde at holde selve ACS-koden for sig selv.

Den endelige version anvender den for både den sekventielle og den distribuerede udgave af ACS fælles klasse `ACS`. Dette fremgår af figur 3.2. I denne klasse ligger al den funktionalitet, der ikke har med programmets forløb at gøre. Dette er blandt andet konstruktionen af ture samt begge metoder til pheromonopdatering. Det vil sige, at kun den del af programkoden, der kontrollerer programmets forløb, findes i selve `SequentialACS`-klassen. Udførelsen af den endelige udgave af det sekventielle program foregår dog stadig helt sekventielt.

Denne sammenblanding af de to ellers separate programmer har vi valgt at foretage for at sikre, at vores distribuerede program er så tæt på det sekventielle som muligt.

Afprøvningen af det sekventielle program har i praksis bestået af en visuel overvågning af phermomonmatricens udvikling igennem en programkørsel, samt en kontrol af, om de fundne løsninger var lovlige.

Skulle man foretage en helt indiskutabel verifikation af vores programs korrekthed, ville man være nødt til at sammenligne alle felter i pheromonmatricen for hver iteration med en håndberegnet udgave af samme. Endvidere skulle det kontrolleres, at alle byer indgik i en myres tur én gang og kun én gang.

En enklere, men dog stadig ganske troværdig undersøgelse, kunne bestå i at undersøge pheromonmatricens værdier imellem to givne iterationer for et meget lille TSP-tilfælde, samt de generede ture for den første af de to iterationer. På

bagrund af en sådan undersøgelse ville det være muligt at slutte, hvorvidt programmet var korrekt for et vilkårligt problem, imellem to på hinanden følgende vilkårlige iterationer.

4 Forsøg og resultater

I dette kapitel gives først en kort beskrivelse af de problemtilfælde, vi har testet vores implementation på. Dernæst bringer vi vores resultater for forsøgskørsler på de pågældende problemtilfælde. Vi sammenligner vores resultater med resultater fra litteraturen. Til sidst beskrives nogle småforsøg vi har lavet med forskellige implementeringer af sekventiel ACS.

4.1 Problemtilfælde

Problemtilfældene vi har benyttet til vores forsøg stammer alle fra TSPLIB[1], hvilket er en database over en række prædefinerede TSP-instanser. Tabel 4.1 er en liste over de tilfælde vi anvender i vores forsøg med sekventiel ACS samt de optimale løsninger til disse.

problem	antal byer	optimum
gr24	24	1272
st70	70	675
kroA100	100	21282
kroA200	200	29368
lin318	318	42029
pcb442	442	50778
rat575	575	6773
d657	657	48912

Tabel 4.1 De optimale løsninger til de otte pågældende problemtilfælde fra TSPLIB.

De ovenstående problemtilfælde er valgt af forskellige årsager. For det første er det de tilfælde, som Randall og Lewis tester deres distribuerede udgave af ACS på. Dette betyder at vi, når vi senere skal have fat i de samme tilfælde til vores distribuerede udgave af ACS, kan genbruge den indlæsningsmetode, som vi allerede har implementeret til sekventiel ACS. For det andet tester Bonabeau et al. deres implementation på tilfældet kroA100, det bliver vi selvfølgelig også nødt til, hvis vi vil kunne sammenligne vores resultater med deres. Bonabeau et al. tester dog også på to andre tilfælde kaldet eil50 og eil75, som tilsyneladende er nogle varianter af TSPLIB-tilfældene eil51 og eil76, blot med én by mindre i hver. Det har desværre ikke været os muligt at fremskaffe disse to 'uautoriserede'

[1] `http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/`

TSP-instanser. For det tredje har tilfælde gr24 og st70 den fordel, at de er forholdsvis små, hvorfor de undervejs i implementeringen af programmet har været hurtige at afprøve programmet på.

Alle tilfældene er givet ved euklidiske koordinater, på nær gr24, hvis omkostninger imellem byerne er givet ved en nedre trekantsmatrix. Af denne årsag har vi i vores implementering af `ProblemLoader`-klassen kun behøvet at koncentrere os om to forskellige indlæsningsmetoder.

4.2 Forsøgsbeskrivelse og resultater

Til forsøgene med sekventiel ACS har vi for hvert af de pågældende problemtilfælde foretaget 10 kørsler, hver med 5000 iterationer (det vil sige $t_{\max} = 5000$). Sæden for tilfældighedsgeneratoren er for den første kørsel sat til 1, for den anden til 2, og så videre op til 10 for den tiende kørsel. Forsøgene er udført med parameterværdierne fra tabel 4.2.

parametre	værdier
β	2
γ	0,1
ρ	0,1
m	20
Q	1
q_0	0,9
cl	15

Tabel 4.2 Parameterindstillinger benyttet i vores implementering af ACS-TSP.

Tabel 4.3 viser for hvert problemtilfælde den bedst fundne længde, den gennemsnitlige længde samt den procentmæssige afvigelse fra optimum over de 10 kørsler. Ydermere vises den gennemsnitlige køretid i sekunder for de 10 kørsler. Denne er bestemt ved at måle tiden på systemets hardware når programmet starter og når programmet stopper, og derefter beregne det tidsinterval programmet har kørt over.

Bonabeau et al. henviser til nogle forsøg udført af Dorigo og Gambardella [Dorigo and Gambadella, 1997]. I disse forsøg er der foretaget 15 kørsler med hver 1250 iterationer og 20 myrer. Dorigo og Gambardella opgiver sammen med længden af deres løsninger det antal ture, der er beregnet, før end optimum blev fundet. I én iteration beregnes 20 ture (da $m = 20$), hvilket svarer til at Dorigo og Gambardella i hver kørsel beregner 25.000 ture. Dorigo og Gambardella finder optimum for kroA100 efter 4820 ture.

problem	optimum	bedste	gen. længde	gen. %-af.	gen. køretid
gr24	1272	**1272**	1278,4	0,50	0,62
st70	675	**675**	679,1	0,61	10,59
kroA100	21282	**21282**	21440,3	0,74	17,74
kroA200	29368	29527	29766,0	1,36	15,66
lin318	42029	42889	43643,3	3,84	25,49
pcb442	50778	52816	55370,4	9,00	139,37
rat575	6773	6991	7085,6	4,62	172,29
d657	48912	55763	56814,9	16,16	278,82

Tabel 4.3 1. søjle i tabellen er navnet på problemtilfældet fra TSPLIB. 2. søjle er de optimale løsninger. 3. søjle den bedst fundne længde, er optimum fundet er denne markeret med fed. 4. søjle er gennemsnittet over ti kørsler. 5. søjle er den gennemsnitlige procentvise afvigelse fra optimum. 6. søjle er den gennemsnitlige køretid i sekunder over de 10 kørsler. Kørslerne er foretaget på en Dell Pentium 4, 2,4 GHz maskine med 522.232 KB RAM.

4.3 Yderligere forsøg

I forbindelse med implementeringen af den sekventielle udgave af ACS er vi i litteraturen stødt på forskellige versioner af denne.

Blandt andet er, der som tidligere nævnt (se kapitel 3), to forskellige versioner af den globale opdateringsregel. Vi fandt ved testkørsler, at versionen som beskrevet i Bonabeau et al. (jævnfør afsnit 2.2.3, kapitel 2) gav bedre resultater end Randall og Lewis' version (se afsnit 6.2).

Vi forsøgte os også med en skridtvis, henholdsvis en turvis, lokal pheromonopdatering. I en skridtvis opdatering drager hver myre hurtigere fordel af de andre myrers skridt i og med, at den lokale opdatering foregår efter, at de m myrer hver har taget ét skridt, det vil sige er gået til en ny by i opbygningen af deres tur. En skridtvis lokal opdatering vil formentlig stemme mere overens med den måde, hvorpå myrer gør brug af pheromon i naturen. En turvis lokal opdatering vil derimod betyde langt mindre kommunikation i forbindelse med en senere distribuering, da den lokale opdatering her kun finder sted, hver gang en myre har fuldendt en tur. Vi fandt, at der ikke var nogen synlig forskel, med hensyn til løsningskvaliteten, på at benytte den skridtvise henholdsvis den turvise lokale opdatering.

Del II

Distribueret ACS

5 Datamatarkitektur og distribuering

Dette kapitel tjener som en indføring i terminologien om, samt principperne bag parallelitet, det vil sige paralleldatamater samt parallelle algoritmer og distribuering. Først dog en definition af, hvad man kan forstå ved en paralleldatamat.

> *A parallel computer is one that consists of a collection of processing units, or processors, that cooperate to solve a problem by working simultaneously on different parts of that problem* [Aki, 1989].

Vi vil i kapitlet først beskrive Flynns klassifisering af datamater. Til denne beskrivelse tager vi fortrinsvist udgangspunkt i [Tanenbaum, 1990] og [Nielsen et al., 1994]. Dernæst vil vi beskrive teknikker til distribuering af algoritmer. Til sidst bringes en række målestokke for hastighedsforøgelse.

5.1 Flynns klassifikation

Flynn opstillede i 1966 en klassifikation, også kendt som Flynns taksonomi, for datamater. Flynn tager i sin klassifikation udgangspunkt i sammenhængen imellem datamatens behandling af strømme af instruktioner og de strømme af data datamaten skal behandle. Mere præcist er det antallet af strømme, af henholdsvis instruktioner og data, der kan behandles simultant i datamaten, som Flynn skelner imellem. Med en *instruktionsstrøm* menes de forskellige instruktioner – for eksempel algoritmer – som bliver udført simultant af processorerne.

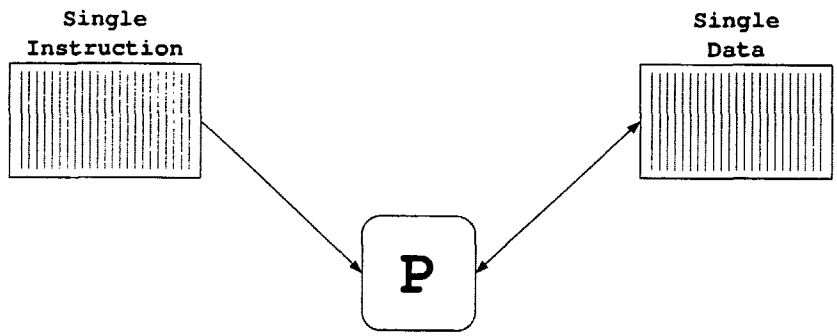
	Single Data	*Multi Data*
Single Instruction	**SISD**	**SIMD**
Multi Instruction	**MISD**	**MIMD**

Figur 5.1 Flynns klassificering af parallelle datamater [Nielsen et al., 1994].

Med en *datastrøm* menes de forskellige dele af data, som processorerne udfører instruktionerne på til et givet tidspunkt. Med andre ord underkastes strømmen af data, strømmen af instruktioner i datamaten. Når man således, som Flynn gør det, skelner imellem enkelte og multiple datastrømme, samt imellem enkelte og multiple instruktionsstrømme, opnår man fire mulige kombinationer (se figur 5.1). I de efterfølgende afsnit beskrives disse kombinationer, samtidig med, at der gives eksempler på problemer, som kan løses ved brug af de respektive datamattyper.

5.1.1 SISD

Single Instruction Single Data, eller SISD-datamater, er datamater beskrevet ved den klassiske sekventielle von Neumann model. Det vil sige, at instruktioner og data er lagret i samme lager, og én processor henter instruktioner fra lageret og udfører dem én ad gangen på ét dataelement. At sige at instruktioner udføres på ét dataelement er dog en sandhed med modifikationer, idet der i mange tilfælde anvendes flere dataelementer, for eksempel må additioner nødvendigvis have mere end ét dataelement for at kunne udføres. Det væsentlige er dog at dataelementerne læses sekventielt, det vil sige de læses i én strøm.



Figur 5.2 Single Instruction Single Data [Nielsen et al., 1994].

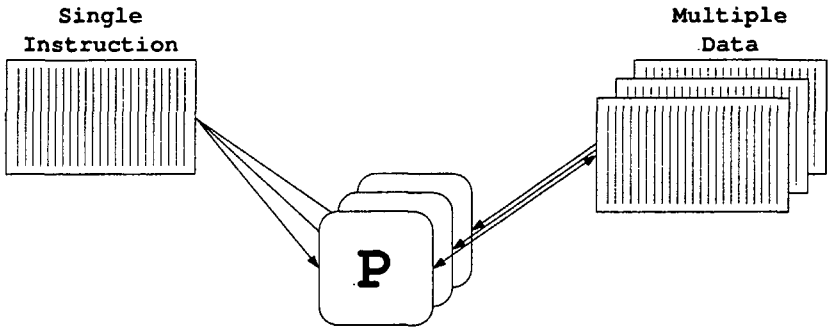
Eksempel

Som ovenfor antydet kan SISD løse problemer omhandlende addition. For at udregne summen af n tal $a_1, a_2, \ldots, a_n$ skal processoren modtage ét tal (dataelement) ad gangen, n på hinanden følgende gange. Processoren skal $n-1$ på hinanden følgende gange modtage instruktionen om at addere det nye tal til summen af de forrige. Tidskompleksiteten af denne denne beregning er $O(n)$.

Almindelige sekventielle datamater så som pc'ere og arbejdsstationer er eksempler på SISD-datamater.

5.1.2 SIMD

Single Instruction Multiple Data eller SIMD-datamater udfører én og samme instruktion, men udfører den simultant på flere strømme af dataelementer, altså synkront. SIMD-datamater udfører i princippet instruktionens strøm på samme måde som de sekventielle datamater, blot foregår instruktionens strøm her på p processorer samtidigt, altså udføres instruktionen på p strømme af dataelementer, som derfor også læses simultant.



Figur 5.3 Single Instruction Multiple Data [Nielsen et al., 1994].

Eksempel

Addition af to matricer er et eksempel på et problem der kan løses ved hjælp af SIMD. Antag at vi har to matricer A og B begge af orden n og at vi har n^2 processorer til vores rådighed. Summen C af de to matricer er givet ved:

$$\begin{matrix} c_{11} = a_{11} + b_{11} & \cdots & c_{1n} = a_{1n} + b_{1n} \\ \vdots & & \vdots \\ c_{n1} = a_{n1} + b_{n1} & \cdots & c_{nn} = a_{nn} + b_{nn}. \end{matrix}$$

Den samme instruktion (addition) gives til samtlige n^2 processorer, som udfører den simultant på hver deres strøm af data. I dette eksempel, i modsætning til det forrige, indeholder hver af de n^2 datastrømme kun ét dataelement, nemlig a_{ij}, b_{ij}. Tidskompleksiteten for matrixaddition udført på denne måde er konstant, $O(1)$, hvorimod den ville have været $O(n^2)$, hvis den var blevet udført på en SISD-datamat.

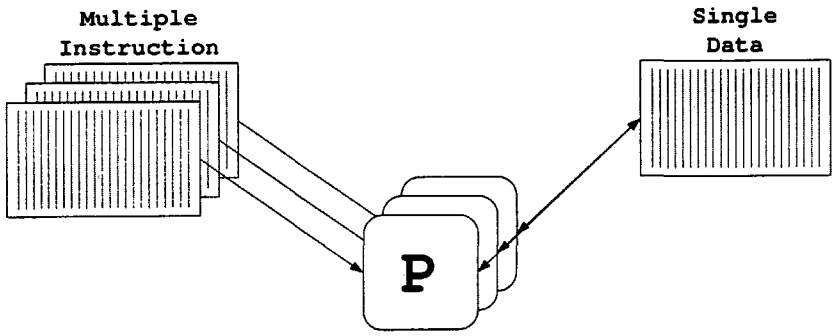
Et andet eksempel kan være kontrol af, hvorvidt et tal z er et primtal eller ej. Lad X være mængden af ulige heltal større end 1 og mindre end z, det vil sige $X = \{3, 5, \ldots\}$[1], og antag at vi har $|X|$ processorer til vores rådighed. Processorerne tildeles nu hver et tal $x_k \in X$ samt tallet z – multiple data – og den enkelte instruktion: Divider z med x_k. Dette kan således foretages i $|X|$ parallelle skridt, hvorfor tidskompleksiteten også her er konstant.

En vektordatamat er et eksempel på en SIMD-datamat.

[1] Kender man alle primtal mindre end $\sqrt{z}$ behøver man kun lade X være mængden af disse.

5.1.3 MISD

Den næste i rækken er *Multiple Instruction Single Data*, MISD-datamater. Disse udfører samtidigt flere instruktioners strømme på én strøm af dataelementer.



Figur 5.4 Multiple Instruction Single Data [Nielsen et al., 1994].

Ifølge Tanenbaum er det ikke helt klart, om der eksisterer MISD-datamater eller ej. Nielsen et al. kender ligeledes ingen eksempler på datamater af denne type, men foreslår dog, at sådanne datamater eventuelt kunne anvendes som en form for kontrol-datamater. Flere forskellige processorer skulle således regne på samme data, og hvis resultaterne stemte overens kunne beregningerne siges at være 'rigtige'. Helsgaun nævner systoliske maskiner som et eksempel på MISD-datamater [Helsgaun, 2003b].

Eksempel

MISD kan for eksempel løse problemer så som matrixmultiplikation. Antag at vi har to matricer A og B begge af orden n og at vi skal finde produktet $C = A \cdot B$. De enkelte elementer i produktmatricen C er givet ved

$$c_{ij} = \sum_k a_{ik} b_{kj}.$$

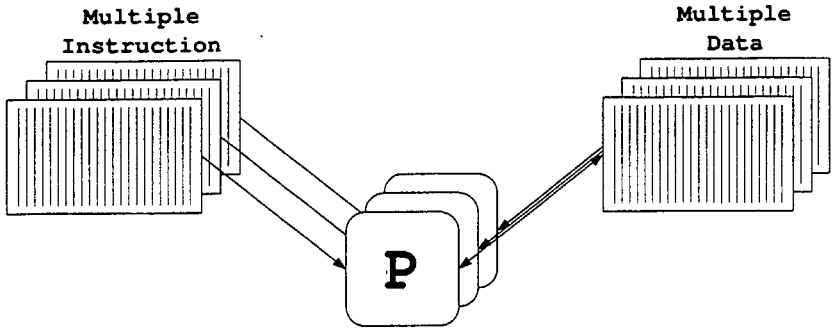
Matrixmultiplikation på en SISD-datamat vil have en tidskompleksitet på $O(n^3)$, da der for hvert af de n^2 elementer i C-matricen skal udføres n multiplikationer (og $n-1$ additioner). Antages det derimod, at vi på vores MISD-datamat har n^2 processorer til vores rådighed, kan matrixmultiplikation udføres i lineær tid, $O(n)$. Hver processor modtager et a_{ik} og et b_{kj}, hvorefter den i ét skridt multiplicer disse og lægger produktet til c_{ij} (ved start er $c_{ij} = 0$). Det tager således n parallelle skridt at udregne C.

Her er altså tale om n^2 strømme af instruktioner, som i princippet kunne være forskellige, men i dette tilfælde er ens. Disse instruktioner udføres på én strøm af dataelementer, nemlig elementerne fra de to matricer A og B.

Det bør dog bemærkes at problemet med matrixmultiplikation lige så vel kan løses ved brug af en SIMD. Her er så tale om én instruktionsstrøm $(\cdot, +, \cdot, +, \cdot, \ldots)$. De multiple datastrømme består her hver af n a_{ik}'er og b_{kj}'er, hvorimod der i MISD kun var én datastrøm bestående af n^2 a_{ik}'er og b_{kj}'er.

5.1.4 MIMD

Multiple Instruction Multiple Data, eller MIMD-datatmater udfører samtidigt flere instruktioners strømme på flere strømme af dataelementer. MIMD-datamater siges at arbejde asynkront.

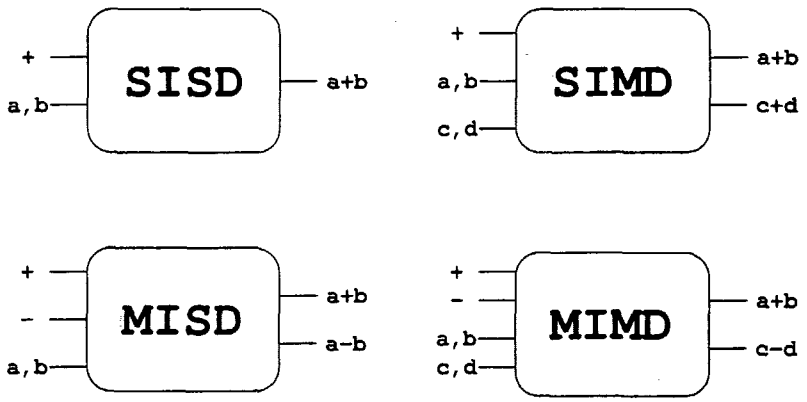


Figur 5.5 Multiple Instruction Multiple Data [Nielsen et al., 1994].

Principielt kan MIMD-datamater simulere de andre typer af datamater, men oftest distribueres dog både en større blok af instruktioner samt en større blok af data til en processer. Det er således vigtigt, at man ved distribuering tager højde for, at problemet skal opsplittes og bearbejdes på forskellige processorer, samt at disse skal arbejde samtidigt og ikke vente på hinanden. Hver enkelt processor i en MIMD-datamat kan dog betragtes som en SISD-datamat, og tankegangen bag visse dele af programudviklingen til MIMD-datamater kan også sammenlignes med tankegangen bag SISD-datamater, men distributionen af data- og instruktionsstrømme skal tage højde for at alle processorer skal arbejde samtidigt.

Eksempel

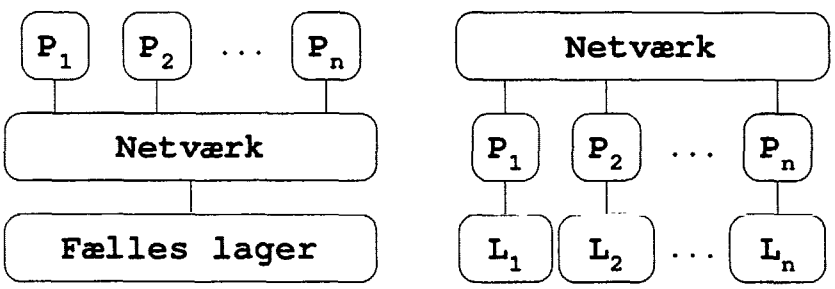
Som ovenfor nævnt kan MIMD-datamater simulere de andre datamattyper. Det vil sige at MIMD-datamater kan foretage såvel matrixaddition som matrixmultiplikation. MIMD kan dog også udføre flere forskellige instruktionsstrømme på flere forskellige datastrømme, hvilket fremgår af figur 5.6. Figur 5.6 eksemplificerer også de tre andre datamattyper i Flynns taksionomi.



Figur 5.6 Eksemplificering af de fire datamattyper: SISD, SIMD, MISD og MIMD [Barr and Hickman, 1993].

Eksempler på MIMD-datamater er for eksempel forbundne datamater i et netværk, såkaldte *clusters*.

Som det fremgår af Flynns klassifikation, findes der to grundlæggende typer paralleldatamater, nemlig SIMD- og MIMD-datamater. MIMD-kategorien kan deles ind i to dele, (1) maskiner med fælles lager (multiprocessorer) og (2) maskiner uden, det vil sige datamater med privat lager. Maskiner, der anvender fælles lager, kendes også som stramt koblede systemer og maskiner anvendende private lagere som løst koblede systemer. Figur 5.7 illustrerer dette. SIMD-datamater kan på ligende vis gøre brug af fælles lager.



Figur 5.7 Et stramt koblet system med fælles lager og et løst koblet system, hvor hver processor har sit eget private lager [Barr and Hickman, 1993].

Når samtlige processorer i et system arbejder på samme opgave skal der oftest udveksles informationer imellem processorerne undervejs. I et stramt koblet system klares dette forholdsvist nemt, da alle processorerne skriver til og læser fra samme lager. I et løst koblet system er man derimod nødt til at håndtere dette på anden vis, da processorerne jo har hver deres private lager. En måde at løse problemet på, er ved at sende beskeder imellem processorerne.

5.2 Implementering af parallelle systemer

Den måde, en implementering af et parallelt system foretages på, afhænger af, hvilken type af system, der er tale om. Hvis man arbejder med stramt koblede systemer, findes der en række muligheder for at uddelegere arbejdet på hver tilkoblet processor. Ligeledes findes der en række muligheder for at uddelegere arbejdet imellem de tilkoblede processorer i et løst koblet system. Der er dog et vist overlap imellem de muligheder systemtyperne kan benytte sig af.

De to følgende afsnit beskriver, hvilke muligheder hver af de to typer systemer har til sin rådighed. Vi har i disse to afsnit taget udgangspunkt i [Farley, 1998] samt et udvalg af internetsider. Internetsidernes adresser vil være angivet i en fodnote første gang, de optræder.

5.2.1 Stramt koblede systemer

I et stramt koblet system kan hver processor tilgå al data med samme hastighed. Hver processor styres af samme operativsystem. Derfor vil man i denne type system typisk benytte sig af, at moderne operativsystemer tillader udførelse af programmer i flere separate tråde af eksekveringer. De fleste moderne programmeringssprog kan udnytte denne mulighed. For eksempel C++[2] og Java[3] giver mulighed for at lade en eller flere dele af et program udføres i separate tråde. Af operativsystemer, der understøtter denne form for parallelisering, kan nævnes Windows2000, Solaris[4] eller Linux SMP[5]. Der findes andre sprog, som for eksempel NESL[6] eller ZPL[7], der tillader en syntaktisk mere naturlig tilgang til, hvordan det angives, hvilke dele af programmet, der skal udføres på hver deres processorer. I NESL angives således blot ved krøllede paranteser omkring et udtryk, at dette udtryk ønskes udført i parallel. ZPL er et array orienteret sprog som det, der anvendes i MATLAB[8].

5.2.2 Løst koblede systemer

Løst koblede systemer er en lidt mere diffus størrelse, idet der findes løst koblede systemer opbygget af flere separate stramt koblede systemer, ligesom der findes løst koblede systemer opbygget over et netværk af pc'er med hver deres operativsystem.

Et eksempel på det første er SGIs Altix[9] multiprocessorsystem, som er et cluster af flerprocessormaskiner, hvor der dog er simuleret fælles hukommelse. Et eksempel på det andet er et netværk af pc'er på for eksempel et universitet, hvor der kører et seperat program på hver maskine, som i en eller anden form samarbejder med en eller flere andre maskiner i netværket om at udføre et stykke

[2] http://www.research.att.com/~bs/C++.html
[3] http://java.sun.com/
[4] http://wwws.sun.com/software/solaris/
[5] http://www.linux.org.uk/SMP/title.html
[6] http://www-2.cs.cmu.edu/~scandal/nesl.html
[7] http://www.cs.washington.edu/research/zpl/
[8] http://www.mathworks.com/products/matlab/
[9] http://www.sgi.com/servers/altix/

arbejde. Til at understøtte den sidst nævnte mulighed kan man benytte sig af forskellige værktøjer til kommunikation over netværk. Der findes for eksempel et system der hedder mpC[10], hvor en applikation definerer et abstrakt netværk, og hvor mpC afviklingsmiljøet ved udførslen af programmet finder ud af den optimale fordeling af data og kodeafvikling.

Et andet alternativ er Javas indbyggede understøttelse af distribuering af data og kode, pakken `java.rmi`. *Remote Method Invocation* (RMI) er et system der tillader et program at kalde metoder i et andet program på en fjern vært. RMI anvender et register over objekter, der er distribuerede. Dette register kører som et separat program på den maskine, der ønsker at tilbyde delte objekter, og det er dette register, som varetager håndteringen af netværksforbindelser og som tillader fjerne klienter at spørge efter et delt objekt på den givne server. Så snart en given implementation af et delt objekt er registreret, er det tilgængeligt på fjerne værter ved opslag i registreret.

Java tilbyder også at arbejde direkte med netværkssockets gennem pakken `java.net`. Opretter man en netværkssocket i Java imellem to værter i et netværk, kan man kommunikerer mellem dem ved at anvende de samme klasser i Java, som man anvender til øvrig IO i Java. For eksempel kan man sende et Java objekt direkte imellem to værter, uden at skulle bekymre sig om, hvordan man får konverteret objektet til en strøm af bytes. Det klarer Java-afviklingsmiljøet for en. Over disse sockets kan man implementere et system, der tillader flere pc'er i et netværk at samarbejde om at løse et problem ved at sende beskeder i form af objekter imellem hver af de enkelte deltagere i systemet.

5.3 Hastighedsforøgelse

Når man beskæftiger sig med parallelitet/distibuering og parallelle/distribuerede algoritmer må man have en måde at måle disses hastighed på i forhold til ikke-parallelle/ikke-distribuerede algoritmer. Nedenfor beskrives, med udgangspunkt i [Metaxas, 1997], [Barr and Hickman, 1993] og [Karp and Flatt, 1990], en række formler og begreber, som benyttes til dette.

5.3.1 Speedup

Den mest gængse metode til at måle hastigheden af parallelle algoritmer er givet ved *speedup*, det vil sige hvor meget hurtigere den parallelle algoritme kører i forhold til en sekventiel.

Reelt speedup

Lad køretiden for en parallel algoritme være $T(p)$, når denne køres på p processorer på et bestemt problem π, og lad den sekventielle køretid være $T(s)$ for det samme problem. Da er speedup'et $S(p)$ givet ved

$$S(p) = \frac{T(s)}{T(p)}. \tag{5.1}$$

[10] `http://www.ispras.ru/~mpc/`

Hvis $S(p) = p$ siges algoritmen at have et *perfekt speedup*.

Ovenstående definition af speedup kaldes for det *reelle speedup*[11]. Her er $T(s) = T^*$, køretiden for den *bedste* algoritme for problemet π kørt på en sekventiel maskine. Det reelle speedup er en god, men hård, målestok for et parallelt program, i og med at det jo sammenlignes med den bedste tilgængelige sekventielle løsning.

Relativt speedup

En anden definition af speedup er det *relative speedup*. Her er $T(s) = T(1)$, det vil sige køretiden af det parallelle program kørt på en enkelt processor på samme parallelle maskine. Ifølge Metaxas regnes relativt speedup ikke for en særlig god målestok for et parallelt program, da programmet nødvendigvis må indeholde kode beregnet til processorhåndtering, hvilket ikke behøves, når det køres på kun én processor.

Absolut speedup

En tredje definition af speedup er det *absolutte speedup*. Her er $T(s) = T^*(1)$, hvilket vil sige køretiden for det bedste sekventielle program, når det køres på den samme parallelle maskine, hvorimod algoritmen ved det reelle speedup blev kørt på to forskellige maskiner. Absolut speedup anses også for at være en god målestok for parallelle programmer.

De tre ovenstående definitioner af speedup tager alle udgangspunkt i faktiske implementeringer af algoritmer. Når man analyserer parallelle algoritmer teoretisk anvender man en anden definition kaldet *asymtotisk reelt speedup*, hvilket vi dog ikke vil komme nærmere ind på her[12].

Det reele speedup må forventes at ligge mellem 1 og antallet af processorer p: $1 \leq S(p) \leq p$. $S(p) = 1$ forekommer når den parallelle algoritme ikke kører hurtigere end den sekventielle. $S(p) = p$, et perfekt speedup, forekommer når anvendelsen af samtlige p processorer er optimal. Rent logisk (og intuitivt) bør det reelle speedup ikke kunne overskride p. Imidlertid findes der eksempler på det modsatte. Disse afvigelser fra normalen skyldes, at parallelle maskiner har mere cache end sekventielle maskiner. Processorcachen er hurtigere end den almindelige hukommelse, hvorfor en sekventiel maskine, som ofte skal læse fra den almindelige hukommelsen, nødvendigvis vil blive overhalet af en parallel, som ikke behøver læse fra den almindelige hukommelsen lige så tit.

5.3.2 Effektivitet

Speedup'et fortæller os hvor god den parallelle algoritme er i forhold til den sekventielle, men det tager ikke antallet af processorer, den parallelle algoritme

[11] Fra engelsk: *Real speedup*.

[12] For definitionen af asymtotisk reelt speedup samt en videre diskussion af de tre ovenfor beskrevne definitioner af speedup se [Metaxas, 1997].

anvender, i betragtning. Det gør derimod *effektiviteten*. Ved effektiviteten af en parallel algoritme forstås speedup'et over antallet af processorer

$$E(p) = \frac{S(p)}{p} = \frac{T(s)}{T(p) \cdot p}. \tag{5.2}$$

Bemærk at $0 \leq E(p) \leq 1$. Når effektiviteten af en parallel algoritme er tæt på 1, betyder det at udbyttet af at anvende flere processorer er tæt ved 100%. En effektivitet tæt på 1 er selvfølgelig derfor at foretrække.

Da definitionen af effektivitet tager sit udgangspunkt i speedup'et, taler man også om tre forskellige definitioner af effektivitet, nærmere betegnet reel, relativ og absolut effektivitet.

Ifølge Metaxas er der blandt forskellige forfattere uenighed omkring definitionerne af speedup og effektivitet, hvilket man bør være opmærksom på hvis man ønsker at sammenligne resultater.

5.3.3 Skalerbarhed

Det kan være nyttigt at vide, om en algoritme har en øvre grænse for parallelisering. Det vil sige, om algoritmen arbejder lige godt på større problemer, afhængig af om den eksekveres på flere processorer eller ej. Antag for eksempel at vi er i stand til at få et speedup med en bestemt algoritme kørt på problemet π_1 med p processorer. Antag dernæst at vi har et nyt problem π_2 dobbelt så stort som π_1, og at den samme algoritme køres på dette problem med $2p$ processorer. Hvis vi stadig er i stand til at konstatere et speedup, siges algoritmen at *skalere*. Er det modsatte derimod tilfældet, siges algoritmen at være *ikke-skalerbar*.

Lad os, som eksempel på hvornår en algoritme er ikke-skalerbar, antage følgende: At vi har en parallel algoritme, som er sammensat af et antal af faser, og at en af disse faser ikke kan paralleliseres. Lige meget hvor meget speedup vi kan opnå fra de resterende faser, og lige meget, hvor lidt tid den ikke-paralleliserbare fase tager, vil der altid eksistere et eller andet stort problem, for hvilket den ikke-paralleliserbare fase vil dominere køretiden, lige meget hvor mange processorer, der er til rådighed.

5.3.4 Amdahls lov

Som antydet i afsnittet ovenfor kan tilstedeværelsen af en sekventiel fase i et parallelt program begrænse de opnåelige speedup betydeligt. Denne ide blev først fremsat af Gene Amdahl i 1967 og kendes i dag som Amdahls lov. Ethvert program, der kan køres på parallelle maskiner, vil have en sekventiel del og en parallel del. Spørgsmålet er dog, hvordan størrelsen af hver af disse dele påvirker det speedup, man kan opnå.

Amdahls lov siger, at den samlede køretid for et parallelt program, aldrig vil kunne blive mindre end den tid det tager at udføre den sekventielle del af programmet. I sin simpleste form kan Amdahls lov opskrives som

$$T(p) = T_s + \frac{T_p}{p}, \tag{5.3}$$

hvor T_s er den del af den samlede køretid $T(p)$ på p processorer, der optages af den sekventielle del og T_p den del der optages af den parallelle del. Bemærk at $T(1) = T_s + T_p$, det vil sige den totale køretid af både den sekventielle og den parallelle del af programmet, når dette køres på én processor.

Antag nu at tiden optaget af den sekventielle del af programmet er givet ved

$$T_s = f \cdot T(1). \tag{5.4}$$

Ved at indsætte ligning (5.4) i Amdahls lov (ligning (5.3)) samt ved at bruge at $T_p = T(1) - T_s$, fås

$$T(p) = T(1) \cdot f + \frac{T(1) \cdot (1 - f)}{p}, \tag{5.5}$$

hvilket er det samme som

$$\frac{T(p)}{T(1)} = f + \frac{(1 - f)}{p}. \tag{5.6}$$

Det observeres nu at venstresiden i ovenstående udtryk er det reciprokke af speedup'et $S(p) = T(1)/T(p)$, hvorfor vi får

$$\frac{1}{S(p)} = f + \frac{(1 - f)}{p}. \tag{5.7}$$

Når f isoleres i denne ligning fås

$$f = \frac{1/S(p) - 1/p}{1 - 1/p}. \tag{5.8}$$

Med ligning (5.8) kan man nu eksperimentelt, når blot man kender speedup'et, bestemme tiden optaget af den sekventielle fase i et parallelt program. Bemærk at tiden optaget af den parallelle fase ligeledes kan bestemmes, blot udtrykkes denne ved brug af $1 - f$.

Karp og Flatt anvender dette udtryk for f, også kaldet *den eksperimentelt bestemte sekventielle del*[13], som en selvstændig målestok. Det ses klart, at hvis $f = 1$, vil der ikke blive udført noget i parallel, hvorfor der dermed ikke kan drages nogen fordel af at køre programmet på flere processorer. Kort sagt, jo mindre f er for et givet problem, jo bedre egner problemet sig til at parallelisere.

[13] Fra engelsk: *The experimentally determined serial fraction.*

6 Parallel Ant Colony System

I dette kapitel gennemgås først de forskellige paralleliseringsstrategier for optimeringsheuristikken ACS. Dernæst bringes pseudo-koden for Randall og Lewis' udgave af parallel ACS.

6.1 Paralleliseringsstrategier for ACS

ACS kan paralleliseres – og dermed distribueres – på forskellig vis, for eksempel nævner Randall og Lewis fem forskellige strategier for parallelisering af ACS. I det nedenstående vil vi med udgangspunkt i [Randall and Lewis, 2002] gennemgå disse fem strategier, men dog lægge mest vægt på den, som Randall og Lewis selv anvender i deres artikel, nemlig *Parallelle myrer*.

De første strategier kan betegnes som værende forholdsvis grovkornede, hvorimod de sidste bliver mere og mere finkornede. Denne finkornethed kommer til udtryk ved, at metoderne uddelegerer flere og flere opgaver til processorerne, samt at disse opgaver omhandler mindre og mindre delproblemer.

Randall og Lewis pointerer at samtlige af de fem metoder tager udgangspunkt i et distribueret lager frem for et fælles lager, i og med at dette er det mest almindelige. De fortsætter:

> *However, as ACO systems typically use global memory structures (such as the pheromone matrix), a shared memory machine would mean a lot less communication and a potential corresponding increase in parallel performance* [Randall and Lewis, 2002].

Vi vil fremover ofte referere til en myre som en *slave* og til myrernes chef (serveren) som *mesteren*[1].

6.1.1 Parallelle uafhængige myrekolonier

I denne, den mest grovkornede, tilgang køres et antal sekventielle ACS-søgninger på en mængde ledige processorer (kolonier). De enkelte kolonier holdes adskilt ved hjælp af værdier af deres nøgleparametre. Selv om samtlige parametre i princippet kan benyttes, er sæden for tilfældighedsgeneratoren dog den mest oplagte. Fordelen ved denne metode er, at der ikke kræves nogen som helst kommunikation imellem de enkelte processorer. Denne tilgang kan implementeres som en række af sekventielle programmer (se algoritme 2.2) på en MIMD-datamat. Randall og Lewis fremhæver dog, at denne tilgang må betragtes som noget naiv.

[1] Fra engelsk: *Slave and master.*

6.1.2 Parallelle vekselvirkende myrekolonier

Denne metode er omtrent magen til den ovenfor beskrevne, blot med den forskel, at der nu til givne iterationer udveksles informationer imellem kolonierne undervejs. Pheromonmatricen fra kolonien med de bedste løsninger kopieres til de andre kolonier.

På trods af at *Parallelle vekselvirkende myrekolonier* er en forholdsvis grovkornet metode for parallelisering af ACS, må den dog betegnes som mere finkornet end den foregående metode. Dette fremgår af, at det nu pålægges processorerne at kommunikere med hinanden.

6.1.3 Parallelle myrer

Her får hver myre (slave) tildelt en seperat processor p, hvormed den skal bygge sin løsning. I tilfældet hvor $m > p$ må hver processor tildeles et variabelt antal myrer. Mesterprocessoren (serveren) er ansvarlig for modtagelse af inddata fra brugeren, udregning af τ_0, placering af myrer på startbyer, foretagelse af den globale pheromonopdatering samt at producere uddata. Ifølge Randall og Lewis kan det under visse omstændigheder være en fordel, at mesteren også optræder som slave. Ofte vil man nemlig komme ud for at man spilder processortid på mesterprocessoren, da denne ikke har lige så beregningstunge opgaver som slaveprocessorerne. Ved således at lade mesterprocessoren udfylde rollen som både mester og slave, kan man eventuelt sikre en mere effektiv implementation.

Denne strategi har et moderat kommunikationsniveau, fordi det, der kræver mest kommunikation, er opretholdelsen af pheromonstrukturerne. Specielt den lokale opdatering af pheromonværdierne kræver en del kommunikation imellem mesteren og slaverne (se algoritmerne 6.1 og 6.2).

Også denne metode kan betegnes som mere finkornet end de foregående, da kommunikationsniveauet imellem processorerne igen forhøjes.

6.1.4 Parallel evaluering af løsningselementer

Med et løsningselement forstås i denne sammenhæng en mulig næste by i den tur, myren er ved at opbygge. I den foregående metode undersøger hver myre samtlige tilgængelige byer – løsningselementer – i hvert skridt, før end den vælger en. Dette kan være en noget beregningstung tilgang, især hvis der er flere betingelser, der skal opfyldes. I og med at hvert af løsningselementerne er uafhængige af hinanden, kan disse dog evalueres parallelt. En anden og endnu mere finkornet metode er derfor at lade de forskellige slaveprocessorer blive tildelt en lige stor delmængde af løsningselementerne, som de så skal evaluere. Ifølge Randall og Lewis er denne tilgang bedst egnet til problemer, der er underlagt mange betingelser.

6.1.5 Parallel kombination af de to foregående strategier

Denne tilgang, som er en kombination af *Parallelle myrer* og *Parallel evaluering af løsningselementer*, kræver ofte et stort antal processorer. Her tildeles hver

myre et lige stort antal af processorer, også kaldet en gruppe. I hver gruppe er en mester ansvarlig for at konstruere myrens tur samt at uddelegere evalueringen af løsningselementerne til gruppens slaver. Er der for eksempel givet 20 myrer og to processorer per myregruppe, er der således ialt brug for 40 processorer. Denne tilgang må betegnes som den mest finkornede af de fem.

6.2 Randall og Lewis' parallelisering

Randall og Lewis vælger at anvende *Parallelle myrer* som strategi for deres parallelisering af ACS. Deres argumentation for dette valg er som følger:

> *It is believed that the communication overhead of* Parallel Interacting Ant Colonies[2] *will be too large for TSP.* [...] *The* Parallel Evaluation of Solution Elements[3] *technique (and hence the* Parallel Combination of Ants and Evaluation of Solution Elements[4]*) is only effective if the cost of the evaluation of an element is high (e.i., the computation is expensive and/or there are numerous and difficult constraints to evaluate), which is not the case for the TSP* [Randall and Lewis, 2002].

Algoritmerne 6.1 og 6.2 beskriver henholdsvis mesterens og slavernes aktiviteter samt kommunikation for strategien *Parallelle myrer*. I denne tilgang simulerer hver processor en myre, og denne myre indeholder selv sig egen kopi af pheromonmatricen, som opdateres løbende igennem hver kørsel. Randall og Lewis skriver, at dette gøres for at sikre et minimalt kommunikationsniveau, selv om det dog stadig antages, at det er netop pheromonopdateringerne der optager størstedelen af kommunikationen.

Det skal nævnes, at den globale pheromonopdateringsregel, som Randall og Lewis anvender ikke stemmer helt overens med den i afsnit 2.2 beskrevne. I den globale opdateringregel hos Randall og Lewis er $\Delta\tau_{ij}(t)$ givet på anden vis, nemlig ved

$$\Delta\tau_{ij}(t) = \begin{cases} Q/L^+ & \text{hvis } (i,j) \in T^+ \\ 0 & \text{ellers,} \end{cases} \tag{6.1}$$

hvor T^+ den hidtil bedste tur, L^+ er længden af denne, og Q er den problemafhængige konstant.

[2] *Parallelle vekselvirkende myrekolonier.*
[3] *Parallel evaluering af løsningselementer.*
[4] Kombinationen af *Parallelle myrer* og *Parallel evaluering af løsningselementer.*

```
modtag brugerparametre (β, q0, γ, ρ, sæd);
send (β, q0, γ, ρ, sæd) til hver myre;
beregn nærmeste nabo omkostning Lnn;
τ0 ← (n · Lnn)^-1;
send τ0 til hver myre;
send afstandsmatricen samt n til hver myre;
while(stopkriterie ikke er mødt)
  placer hver myre på en tilfældig by således at to
  myrer ikke er på samme by;
  send start_by til hver myre;
  for(hver myre)
    modtag hver myres næste_by og tilføj den til
    koloniløsningen;
    opdater pheromonmatricen ved at benytte den
    lokale opdateringsregel ((2.6));
    send m pheromonopdateringer (i, j, τij) til hver
    myre;
  end for;
  modtag L for hver myres løsning;
  bestem bedste L fra den pågældende kolonis
  løsninger;
  if(L < L+)
    L+ ← L;
  end if;
  opdater pheromonmatricen i overenstemmelse med den
  globale opdaterngsregel ((2.7) og (6.1));
  send n pheromonopdateringer til hver myre;
  undersøg om stopkriteriet er mødt og giv hver myre
  besked;
end while;
return T+ og L+;
```

Algoritme 6.1 Pseudo-kode for mestermyrerens processor i ACS anvendt på TSP [Randall and Lewis, 2002].

Ovenstående formulering af den globale pheromonopdateringsregel resulterer i en formindskning af værdierne på samtlige kanter i pheromonmatricen (første del af ligning (2.7)) samt en forøgelse af pheromonværdierne udelukkende på kanterne tilhørende den hidtil bedste tur (anden del af ligning (2.7)). Den globale opdateringsregel for ACS i afsnit 2.2.3 stammer fra Bonabeau et al., og her opdateres udelukkende på de kanter, som tilhører den hidtil bedste løsning – altså ingen formindskning af pheromonværdierne på de øvrige kanter.

```
modtag (β, q0, γ, ρ, sæd) fra mesteren;
modtag τ0 fra mesteren;
modtag afstandsmatricen og n fra mesteren;
initialiser pheromonmatricen ved hjælp af τ0;
while(stopkriterie ikke er mødt)
  modtag start_by fra mestern;
  start_by ← by;
  for(hver by)
    vælg næste_by i overenstemmelse med
    overgangsreglen ((2.4) og (2.5));
    send næste_by til mesteren;
    omkostning ← omkostning + d_naeste_by,start_by;
    by ← næste_by;
  end for;
  send omkostning til mesteren;
  modtag pheromonopdatering fra mesteren;
  modtag stopkriterieinformation fra mesteren;
end while
```

Algoritme 6.2 Pseudo-kode for slavemyrenes processorer i ACS anvendt på TSP [Randall and Lewis, 2002].

I kapitel 7 gives en beskrivelse af, hvorledes vi har implementeret vores distribuering af ACS, samt hvilke ændringer, vi har foretaget i forhold til ovenstående gennemgang, og hvorfor disse ændringer er foretaget.

7 Implementation

I dette kapitel analyseres og beskrives vores implementation af den distribuerede udgave af ACS med fokus på, hvordan det sekventielle program distribueres ved hjælp af strategien *Parallelle myrer*.

7.1 Analyse

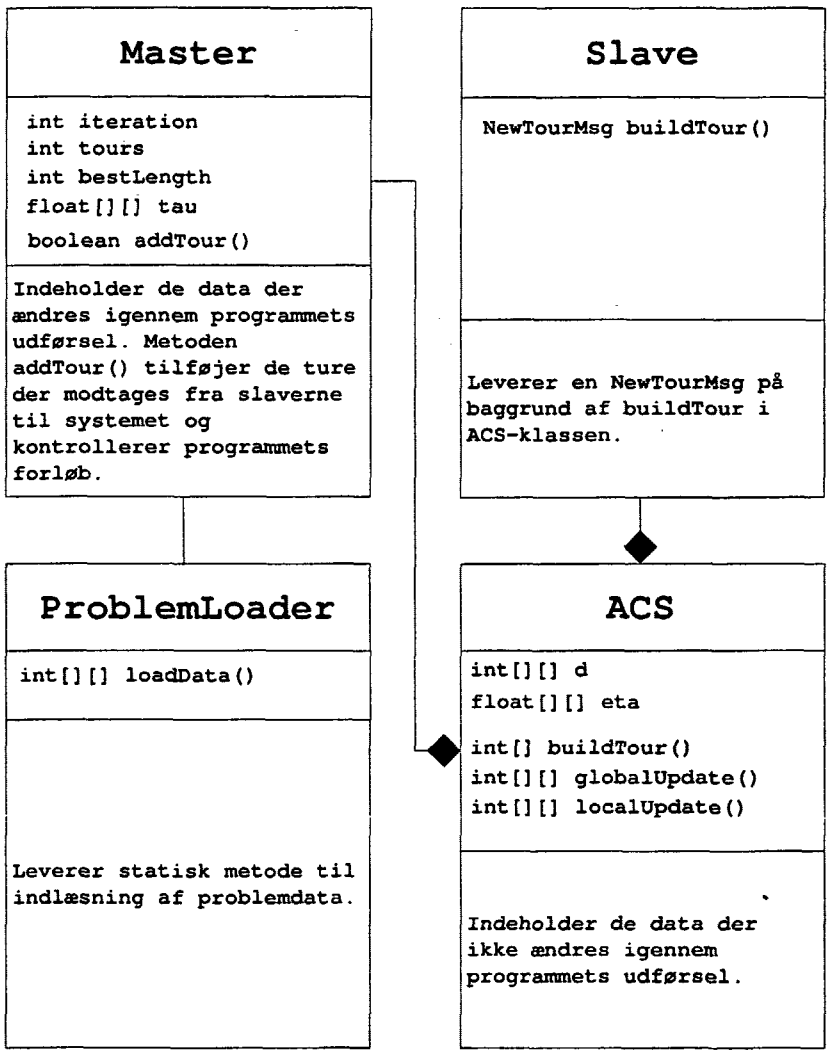
Den distribuerede udgave af ACS er udviklet med det formål, at parallelisere vores sekventielle ACS-program over et almindelig netværk af pc'er, ved hjælp af paralleliseringsstrategien *Parallelle myrer*. Dette betyder at vi skal trække de dele af det sekventielle program, der skal udføres i parallel, ud i ét program, og trække de dele der skal udføres sekventielt ud i et andet. Den sekventielle del der skal udføres centralt af en server kalder vi *myremesteren*. Den parallelle del der skal udføres parallelt af klienterne, kalder vi for *myreslaver* eller blot *slaver*. Altså skal et i mesterprogrammet defineret, men vilkårligt, antal slaver (processorer) via mesteren udgøre et ACS. Dette ACS skal generere løsninger til et givet problem, der er omtrent lige så gode, som dem det sekventielle program finder.

Da vi er af den opfattelse, at det simpleste kommunikationssystem er et der baserer sig på udveksling af beskeder, har vi i vores implementation af den distribuerede udgave af ACS besluttet, at kommunikationssystemet skal tage udgangspunkt i et sådant.

Som ovenfor nævnt, havde vi på forhånd vedtaget at systemet skulle kunne afvikles på et netværk bestående af et ikke på forhånd kendt, men begrænset, antal pc'er. Denne restriktion har betydet, at vi ikke kan operere med én processor per myre, men må fordele antallet af myrer ud over det antal af slaver, der er tilrådighed ved den enkelte kørsel. Ved at kigge på ACS-algoritmen (jævnfør afsnit 2.2) fandt vi, at det eneste antallet af myrer har indflydelse på, er: Antallet af lokale pheromonopdateringer, der foretages imellem hver global pheromonopdatering, samt hvor mange forskellige positioner (byer) der startes i, i hvert tidsskridt t.

Denne observation giver mulighed for programmeringsmæssigt at abstrahere fra begrebet myre. Vi har således blot valgt at lade slaveprogrammet være en turgenerator uden et myre-ID. Når programmet startes finder mesteren m forskellige startpositioner imellem 0 og n, hvor m er antallet af myrer og n er antallet af byer. Mesteren sætter den første tilgængelige slave igang med at lave en tur med den første af de m tilfældige byer som startposition. Derefter sætter mesteren den næste tilgængelige slave igang, og så fremdeles, indtil alle slaver er igangsat. Hvis antallet af slaver overstiger antallet af myrer i ACS'et øges

iterationsnummeret, i, med 1 inden den $m + 1$'te slave sættes igang. Herefter foretager mesteren sig ikke noget, før den modtager en tur fra en slave. Når en ny tur modtages af mesteren, tilføjer den denne tur til systemet og svarer tilbage med en ordre til slaven om at gå i gang med at lave en ny tur. Hver gang mesteren har modtaget m ture, genereres der igen m forskellige startpositioner imellem 0 og n. Den enkelte slave tildeles en af disse startpositioner samtidig med, at mesteren beordrer den til at bygge en ny tur. Den til slaven givne startposition afhænger af den rækkefølge, hvori slaverne svarer tilbage til mesteren. På denne måde kan vi have et vilkårligt antal slaver tilknyttet et ACS med et fast antal myrer.



Figur 7.1 De klasser, der indgår i det distribuerede ACS. Dobbeltarrayet `tau`, i `Master`-klassen, er pheromonmatricen. Dobbeltarrayet `d`, i `ACS`-klassen, er omkostningsmatricen. Dobbeltarrayet `eta`, i `ACS`-klassen, er synlighedsmatricen. Slaven og mesteren anvender hver deres forskellige instanser af `ACS`-klassen.

7.2 Design

I dette afsnit beskrives designet af det distribuerede program.

7.2.1 Opdeling af sekventielt program

Strategien *Parallelle myrer*, medfører implicit en opdeling af programmet. I denne opdeling er en enkelt myre ikke i stand til andet end at finde en tur for et givet TSP. Vi har derfor med dette udgangspunkt, blot konstrueret en opdeling af programmet mellem mester og slave, hvor mesteren foretager alt arbejde med pheromonopdatering, kontrol af antallet af iterationer samt kontrol af konstanter vedrørende ACS'et, herunder antallet af slaver. Slavernes eneste job er at konstruere nye ture og levere disse til mesteren. På figur 7.1 ses en skitse af klasserne i det distribuerede system, samt deres indbyrdes realationer. Nedenfor findes en opsummering af de vigtigste elementer i hver del.

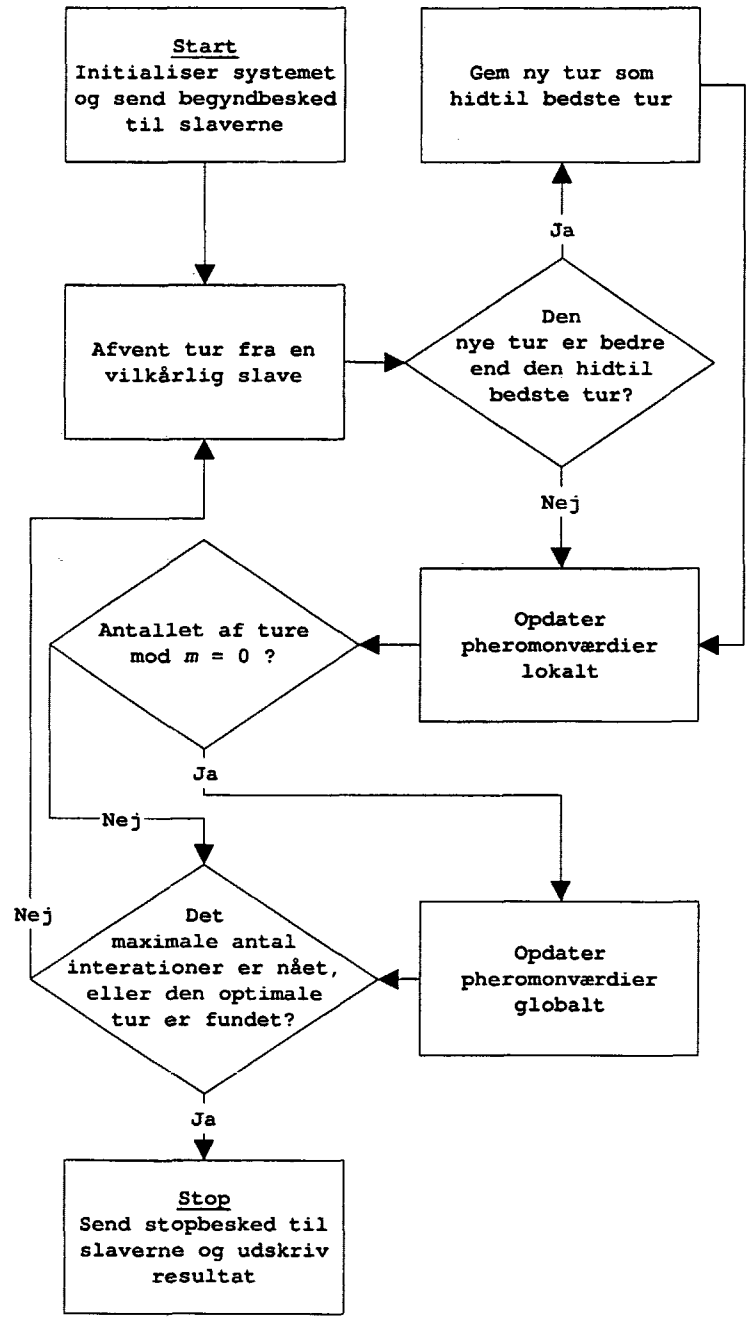
7.2.2 Central funktionalitet i mesteren

- Opdatering af pheromonværdier efter hver ny tur ved hjælp af den lokale opdateringsregel.
- Opdatering af pheromonværdier ved hjælp af den globale opdateringsregel hver gang alle m myrer har bygget deres tur, altså til sidst i hver iteration.
- At holde styr på, hvilken tur, der er den hidtil bedste på et givet tidspunkt.

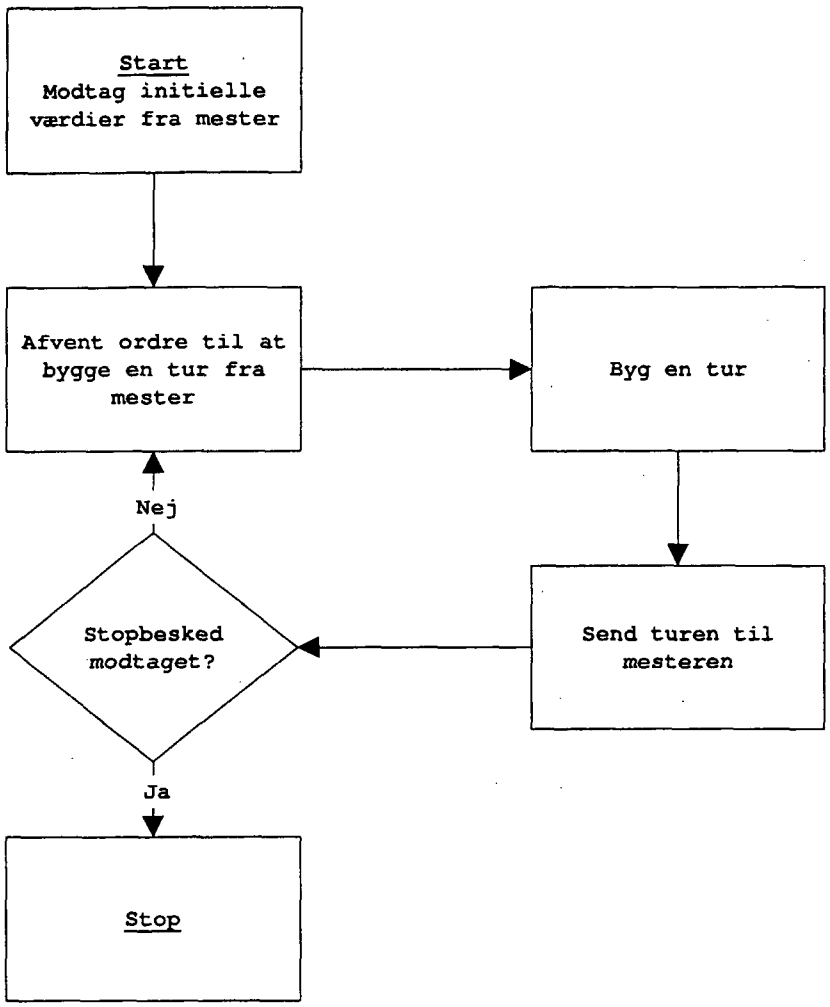
7.2.3 Central funktionalitet i slaven

- At en myre kan finde en lovlig tur ved hjælp af overgangsreglen (ligning (2.4) og (2.5)).

Figurene 7.2 og 7.3 viser det distribuerede programs forløb for henholdsvis mesteren og slaven.



Figur 7.2 Viser mesterprogrammets forløb.



Figur 7.3 Viser slaveprogrammets forløb.

7.2.4 Central funktionalitet i kommunikationssystemet

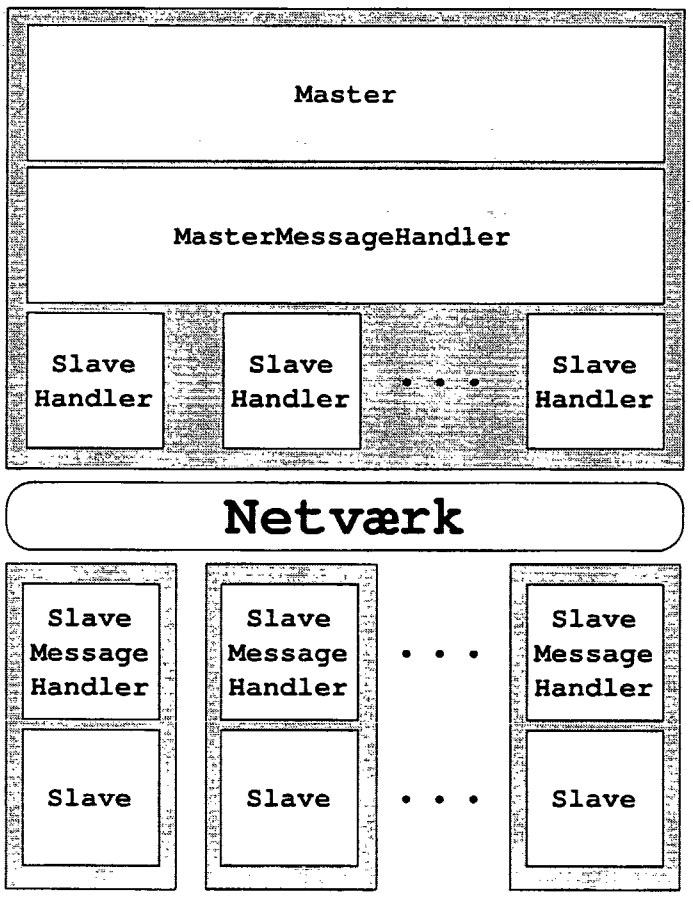
- At mesteren kan sende til og modtage beskeder fra en vilkårlig slave til et vilkårligt tidspunkt.
- At slaverne kan sende og modtage beskeder fra mesteren uafhængigt af hinanden.

Da den distribuerede ACS-kode var færdig, gik vi igang med at udvikle vores eget beskedomdelingssystem inspireret af Farley, men reduceret med hensyn til både funktionalitet og omfang.

Farleys system har følgende egenskaber: En besked kan indeholde et vilkårligt antal objekter. En tekststreng angiver beskedens type. Beskedtyper kan tilfø-

jes under kørsel. Objektet der håndterer kommunikationen med beskeder for enten slave eller mester kan udskiftes under kørsel. Der kan tilføjes yderligere objekter til håndtering af kommunikation. Et sådant objekt til håndtering af kommunikation definerer hvilke beskedtyper, det selv håndterer og samarbejder, efter at være blevet tilføjet, med de eksisterende objekter til håndtering af kommunikation. Alle deltagere i systemet kan sende til hinanden.

Vores reducerede system har følgende egenskaber: En besked består altid af kun ét objekt. Objektets datatype angiver beskedens type. Mængden af beskeder er fast og tilpasset ACS. Der er ét objekt til håndtering af kommunikation for hver daltager i systemet. Dette kan ikke udskiftes, og skal passe til den på forhånd bestemte mængde af beskeder. Der kan kun sendes imellem mester og slave.



Figur 7.4 Kommunikationssystemets opbygning. På denne figur ses det, hvordan det at sende og modtage beskeder er adskilt fra det at fortolke beskeder. `MasterMessageHandler` er det objekt, der fortolker beskeder og reagerer afhængigt af, hvilken besked der modtages. `SlaveHandler` er det objekt, der tilknyttes en instans af til hver slave og som står for den egentlig kommunikation af beskeder. I slaven varetages begge funktioner af samme objekt, `SlaveMessageHandler`, idet slaven ikke skal kommunikere med mere end én mester.

Resultatet af denne måde at opbygge ACS'et på, er en langt mere overskuelig programkode samt et bedre overblik over programmet som helhed. Udfald af en enkelt slave vil ikke betyde noget for, hvorvidt systemet er i stand til at færdiggøre kørslen af programmet. Mesteren vil blot forsætte med udførelsen på de resterende slaver. En slave, der går i stå i et stykke tid og derefter forstætter med at fungere, vil kunne betyde at en 'gammel' tur bliver tilføjet systemet. Den resulterende afvigelse i pheromonværdier vil dog være så lille, at den reelt set ikke har nogen betydning for slutresultatet. Dog må vi antage, at slutresultatet vil blive forringet, hvis et overtal af slaverne er ustabile.

Figur 7.4 viser et diagram over kommunikationssystemet i det distribuerede program.

7.3 Afvigelser fra hovedkilder

Ligesom vores implementering af sekventiel ACS afviger vores implementering af den distribuerede udgave også en smule fra vores kilder, hvilket i dette tilfælde vil sige [Randall and Lewis, 2002].

For det første benytter vi – grundet vores tidligere nævnte resultater med sekventiel ACS (jævnfør afsnit 4.2) – ikke den samme globale opdateringsregel som Randall og Lewis. Ligeledes har vi – også på grund af resultaterne med sekventiel ASC – valgt at implementere den turvise lokale opdatering, i modsætning til den skridtvise, som Randall og Lewis benytter sig af. Dette valg er truffet på baggrund af overvejelser omkring pakkeforsinkelse og sendehastighed i netværket. Den turvise opdatering formindsker nemlig kommunikationen imellem mester og slaver med en faktor n (antallet af byer). På grund af denne ændring har vi iøvrigt valgt at holde alle pheromonopdateringer i mesteren, også den lokale som foretages på baggrund af de ture, der modtages fra slaverne. Med hensyn til parametrene har vi i den parallelle udgave fastholdt de samme værdier som vi benyttede i den sekventielle, det vil sige dem i tabel 4.3 i afsnit 3.2. Dette er gjort af hensyn til den indbyrdes sammenligning af de to ACS-versioners resultater. Randall og Lewis har kun tilknyttet én myre til hver processor. Vi har ligesom Bonabeau et al. valgt at adskille antallet af processorer fra antallet af myrer m. Bonabeau et al. har et variabelt antal myrer på én processor, vi har et variabelt antal myre på p procesorer.

Også Randall og Lewis afviger på et punkt fra de mere gængse fremstillinger af ACS-TSP, idet de ikke opererer direkte med synligheden η_{ij} (se ligning (2.3) i afsnit 2.2). Dette medfører, at de må lade deres β-parameter i overgangsreglens ligninger (2.4) og (2.5) være negativ for at opnå de ønskede resultater. Vi har i vores implementeringer af ACS-TSP benyttet os af definitionen af synligheden.

En sidste vigtigt bemærkning gælder forskellen på det sekventielle program og det distribuerede program med hensyn til, hvorvidt resultater kan reproduceres. I det sekventielle program er det, som nævnt i afsnit 3.2 i første del af rapporten, muligt at reproducere resultater. Dette er imidlertid ikke gældende for det distribuerede program. Årsagen til dette er det faktum at den hastighed hvormed en slave genererer ture varierer en smule fra kørsel til kørsel. Denne varians er nok til at give små variationer i løsningerne.

7.4 Afprøvning

Afprøvningen af det distribuerede program er foregået på samme måde som for det sekventielle program. Vi finder at denne metode vil være tilstrækkelig til også at verificere, at kommunikationen fungerer tilfredsstillende, idet fejl opstået under kommunikation imellem deltagere vil resulterer i fejl i opdateringen af pheromonmatricen eller i ulovlige ture. Dertil sammenholder vi i næste kapitel (afsnit 8.2.1) kvaliteten af løsninger fundet af det distribuerede system med løsninger fundet af det sekventielle program. Denne test foretages for at sikre, at den valgte parallelliseringsstrategi ikke forringer det distribuerede systems evne til at løse et givet problem. Resultatet af denne test viser ikke nogen afgørende forskel i løsningernes kvalitet.

Vi har envidere foretaget en test af, hvorvidt kommunikationssystemet rent faktisk fordeler arbejdet ligeligt imellem de slaver, der er tilknyttet en given kørsel. Dette har vi gjort ved at aflæse den CPU-tid, hver slave havde brugt fra vi startede systemet og til systemet var afviklet. CPU-tid skal forstås som et mål, for den mængde arbejde CPU'en har udført samlet igennem et programs udførsel. Hvis en CPU har arbejdet en tiendedel af tiden i en samlet udførselstid på 5 minutter, vil denne CPU have opnået en CPU-tid på 30 sekunder. Testen af kommunikationssystemet vil vi ikke bringe resultater for, da CPU-tiden for et program i Windows2000 kun kan ses imens programmet kører og derfor er svær at bestemme helt nøjagtigt. Vi var dog i stand til at overvåge programmets CPU-tidsforbrug visuelt, og testen viste at hver slave lavede en ens mængde arbejde for alle problemtilfælde, uafhængigt af antallet af iterationer og slaver.

Med hensyn til stabiliteten af systemet ville det være nødvendigt at teste systemets respons på udfald af en eller flere slaver, systemets respons på en eller flere slaver, der leverer fejlagtigt data, og lignende fejl. Denne form for afprøvning har vi ikke foretaget, da vores program må betragtes som en prototype, hvis målgruppe er personer der er i stand til at læse og forstå selve programkoden. Vi vil dog gerne fremhæve vigtigheden af at have krav til og afprøvning af stabiliteten af et program, hvis målgruppe er slutbrugere.

8 Forsøg og resultater

I dette kapitel gives først en kort beskrivelse af de problemtilfælde, vi har testet vores implementation på. Dernæst beskrives de forsøg, som vi har underkastet vores implementation, samt resultaterne for forsøgskørslerne.

8.1 Problemtilfælde

Problemtilfældene anvendt til distribueret ACS er de samme, som vi anvendte til den sekventielle udgave af ACS (jævnfør tabel 4.1 i afsnit 4.1).

8.2 Forsøgsbeskrivelse og resultater

I dette afsnit beskrives først kvaliteten af de løsninger, det distribuerede program finder. Dernæst præsenteres speedup og effektivitet for programmet, samtidig med at der gives en beskrivelse af de forsøg, vi har udført. Til sidst diskuteres vores resultater i forhold til Randall og Lewis' resultater.

8.2.1 Afvigelser fra optimum

Tabel 8.1 viser for hvert af de otte problemtilfælde et gennemsnit af de fundne løsninger samt et gennemsnit af den procentvise afvigelse fra optimum over 10 kørsler. Ligesom i forsøget med sekventiel ACS er der i hver af de 10 kørsler foretaget 5000 iterationer, og sæden for tilfældighedsgeneratoren er i første kørsel sat til 1, i anden til 2, og så fremdeles. Forsøgene er udført med parameterværdierne fra tabel 4.2.

Det ses at kvaliteten af løsningerne fundet ved hjælp af den distribuerede udgave af ACS er omtrent lige så gode og i nogle tilfælde bedre end løsningerne fundet ved hjælp af den sekventielle udgave (tabel 4.3 i afsnit 4.2). For både pcb442 og rat575 finder det distribuerede program løsninger af en lidt bedre kvalitet end det sekventielle. Dette må antages at være et resultat af, at vores gennemsnit ikke er beregnet over et tilstrækkeligt stort antal kørsler. Overordnet lader det dog ikke til, at der er nogen afgørende forskel på kvaliteten af løsningerne fundet af den distribuerede udgave af ACS-TSP i forhold til løsningerne fundet af den sekventielle udgave.

problem	optimum	gen. længde	gen. %-af.	sekr. %-af.
gr24	1272	1272,0	0,00	0,05
st70	675	678,9	0,58	0,61
kroA100	21282	21458,1	0,83	0,74
kroA200	29368	29713,3	1,18	1,36
lin318	42029	43520,6	3,55	3,84
pcb442	50778	53587,2	5,14	9,00
rat575	6773	6961,6	2,79	4,62
d657	48912	56640,5	15,79	16,16

Tabel 8.1 1. søjle i tabellen er navnet på problemtilfældet fra TSPLIB. 2. søjle er de optimale løsninger. 3. søjle er gennemsnittet over de ti kørsler. 4. søjle er den gennemsnitlige procentvise afvigelse fra optimum. 5. søjle er det sekventielle programs gennemsnitlige procentmæssige afvigelse fra optimum.

8.2.2 Speedup og effektivitet

I forbindelse med udregningen af speedup og effektivitet i vores resultatbehandling afviger vi en smule fra Randall og Lewis, der i deres resultatbehandling tager udgangspunkt i [Barr and Hickman, 1993]. Ifølge Barr og Hickman bør man ikke benytte gennemsnit i forbindelse med udregningen af speedup'et (ligning 5.1). Som en konsekvens af dette foretager Randall og Lewis kun én kørsel per problemtilfælde og anvender derfor også kun én sæd.

Vi har trods ovenstående alligevel valgt at udregne speedup'et som et gennemsnit over flere kørsler. Dette valg har vi truffet på baggrund af, at køretiderne for vores program ofte svinger så meget inden for det samme sæt af kørsler, at vi ikke mener, at en enkelt kørsel vil kunne give et fornuftigt billede af speedup'et og dettes udvikling.

Udregning af speedup og effektivitet

Det relative speedup er givet ved den gennemsnitlige køretid med det distribuerede program for $p = 1$, divideret med den gennemsnitlige køretid for den distribuerede kode med p processorer. Ligeledes er det reelle speedup givet ved den gennemsnitlige køretid med det sekventielle program, divideret med den gennemsnitlige køretid for den distribuerede kode med p processorer.

Effektiviteten udregnes som det gennemsnitlige speedup over værdien af p. Der forelægger både en relativ og en reel effektivitet.

De gennemsnitlige køretider er taget over 10 kørsler med et varierende antal af iterationer for de enkelte problemtilfælde (se tabel 8.2).

Noter til målemetoder

Det er ikke muligt at måle CPU-tiden for det distribuerede program, da det kører på separate maskiner. Endvidere skal ventetid introduceret på grund af synkronisering mellem mesteren og slaverne også regnes med i den tid, som det

problemtilfælde	antal iterationer
gr24	1000
st70	500
kroA100	500
kroA200	250
lin318	100
pcb442	50
rat575	30
d657	20

Tabel 8.2 Antallet af iterationer kørt på de pågældende TSP-tilfælde.

tager at afvikle det distribuerede system. Derfor måler Randall og Lewis i deres udregning af speedup-brøken tælleren i CPU-tid og nævneren i 'wall clock time'.

Alle vores køretider er målt ved at måle tiden på systemets hardware, når programmet starter, og når programmet stopper, og derefter beregne det tidsinterval programmet kørte over. På denne måde måles også eventuelle pauser forårsaget af operativsystemet. Vi mener ikke, at vi ville få mere præcise resultater af at anvende CPU-tid for det sekventielle program, da den målte tid for det distribuerede program alligevel introducerer en usikkerhed i resultatet for speedup'et. Derfor mener vi, at det rigtige at gøre er, at anvende den samme målestok for både det sekventielle og for det distribuerede program.

Med hensyn til vores angivelse af værdier i tabellerne i dette afsnit, skal det nævnes at systemtiden på en almindelig pc ikke kan aflæses med større præcision end ± 10 millisekunder. Det betyder, at vores tider i sekunder ikke kan angives præcist med mere end to decimaler, altså ned til en hundrededel af et sekund. Vores angivelse af samtlige speedup reflekterer denne præcision, idet alle speedup og effektiviteter er angivet med to decimaler. Som følge af dette, vil den reelle effektivitet i tabel 8.4 i mange tilfælde være opgivet til at være nul, da der måske kun er tale om en effektivitet på for eksempel en titusindedel.

Forsøgsbeskrivelse

Vores forsøg med det distribuerede program er foretaget på samme måde, som forsøgene for det sekventielle program (se afsnit 4.2). De gennemsnitlige køretider for problemtilfældene, det vil sige dem der ligger til grund for nedenstående beregninger, kan findes i appendiks C.

Antallet af iterationer programmet er blevet kørt med på et givet problem, er afpasset efter størrelsen af problemet. Dette er gjort ved eksperimentelt at finde et antal iterationer til hvert problemtilfælde, der gav en køretid på et sted imellem 30 sekunder og nogle minutter. Jævnfør tabel 8.2 for antallet af iterationer for de pågældende problemtilfælde.

I forbindelse med udførelsen af vores forsøg observerede vi, at tiderne for de første kørsler i et sæt af kørsler ofte kunne være markant længere end de efterfølgende kørselstider. Især for de mindre problemtilfælde gjorde dette sig gældende. Det lader altså til at man, om man så må sige, først skal 'vække' netværket med

et par kørsler. Vi har derfor i vores forsøg så vidt muligt sørget for at tage højde for dette ved at køre nogle 'dummy' kørsler, inden vi begyndte at måle køretiderne. Mere præcist består hvert forsøg af ialt 15 kørsler, hvoraf de første 5 ikke tæller med. Kørsel nummer 6 er således den kørsel, der starter med sæd lig 1.

Iøvrigt skal det nævnes, at vores distribuerede system består af den samme slags maskiner, nemlig Dell Pentium 4 maskiner med 2,4 GHz og 522.232 KB RAM, hvorfor vores reelle speedup og effektivitet med rette også kan betragtes som værende henholdsvis absolut speedup og effektivitet (jævnfør afsnit 5.3.1).

Afslutningsvist undersøgte vi konsekvensen af at halvere den mængde af data, der skal kommunikeres hver gang en slave skal sættes igang med at konstruere en tur. Dette gjorde vi ved at undersøge forskellen i køretid mellem to forskellige versioner af programmet. Det ene anvendte en pheromonmatrix af typen `double`, som i Java er på 8 bytes. Det andet program anvendte en pheromonmatrix af typen `float`, som i Java er på 4 bytes. Resultatet af denne undersøgelse var, at programmet, der anvendte floats kørte tæt på dobbelt så hurtigt som det, der anvendte doubles. Også det sekventielle program fik udskiftet sine doubles med floats og resultatet af dette var en hastighedsforøgelse på cirka 10%.

Resultater

I tabel 8.3 og tabel 8.4 ses vores egne resultater. Tabel 8.3 er en tabel over det relative speedup samt den relative effektivitet. Tabel 8.4 er en tabel over det reelle speedup samt den reelle effektivitet (jævnfør afsnit 5.3.1 og afsnit 5.3.2).

Tabellerne indeholder for hvert problemtilfælde to værdier, angivet som funktion af antallet af processorer. Disse værdier er henholdsvis det gennemsnitlige speedup samt effektiviteten. Værdierne er angivet i den nævnte rækkefølge.

problem	$p = 2$	$p = 4$	$p = 6$	$p = 8$
gr24	1,12	1,13	1,00	0,89
	0,56	0,28	0,17	0,11
st70	0,71	1,47	1,44	0,87
	0,35	0,37	0,24	0,11
kroA100	1,14	1,15	1,15	0,65
	0,57	0,29	0,19	0,08
kroA200	0,97	1,99	1,99	1,05
	0,48	0,50	0,33	0,13
lin318	1,21	1,22	0,66	0,62
	0,61	0,30	0,11	0,08
pcb442	1,51	1,52	0,83	0,77
	0,75	0,38	0,14	0,10
rat575	1,07	2,19	1,13	2,19
	0,53	0,55	0,19	0,27
d657	1,04	1,05	0,57	1,06
	0,52	0,26	0,09	0,13

Tabel 8.3 Tabel over det relative speedup og den relative effektivitet. Første søjle er navnet på det pågældende problemtilfælde. Rækkerne i de næste søjler indeholder for hvert problemtilfælde det gennemsnitlige speedup samt effektiviteten ved distribuering over p processorer.

problem	$p = 2$	$p = 4$	$p = 6$	$p = 8$
gr24	0,03	0,03	0,02	0,02
	0,01	0,01	0,00	0,00
st70	0,02	0,03	0,03	0,02
	0,01	0,01	0,00	0,00
kroA100	0,02	0,02	0,02	0,01
	0,01	0,00	0,00	0,00
kroA200	0,01	0,01	0,01	0,01
	0,00	0,00	0,00	0,00
lin318	0,01	0,01	0,01	0,00
	0,00	0,00	0,00	0,00
pcb442	0,01	0,01	0,01	0,01
	0,01	0,00	0,00	0,00
rat575	0,01	0,01	0,01	0,01
	0,00	0,00	0,00	0,00
d657	0,02	0,02	0,01	0,02
	0,01	0,00	0,00	0,00

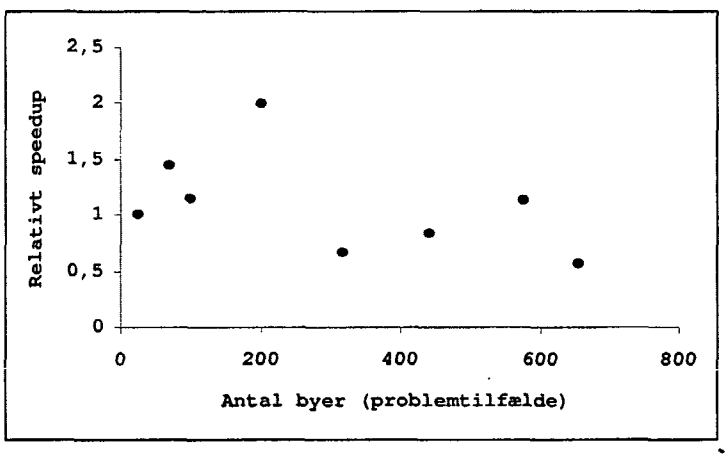
Tabel 8.4 Tabel over det reelle speedup og den reelle effektivitet. Første søjle er navnet på det pågældende problemtilfælde. Rækkerne i de næste søjler indeholder for hvert problemtilfælde det gennemsnitlige speedup samt effektiviteten ved distribuering over p processorer.

Opsummering

Først og fremmest skal det siges, at kigger man på det reelle speedup bliver det hurtigt klart, at denne form for distribuering af ACS ikke er nogen god ide. Netværket begrænser ganske enkelt hastigheden af en så kommunikationsintensiv paralleliseringsstrategi som *Parallelle myrer*. Derfor er det sekventielle program simpelthen langt hurtigere end det distribuerede.

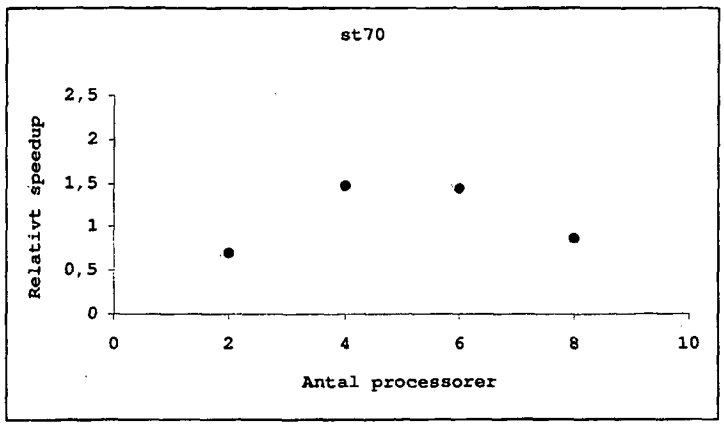
Med hensyn til, hvordan det distribuerede systems speedup, som funktion af antallet af processorer, udvikler sig relativt til hastigheden af systemet, selv med 1 slave, må det konstateres at et speedup kan opnås. Dette speedup er dog kraftigt påvirket af at kommunikationsbehovet for systemet stiger hurtigere end køretiden for den procedure der bygger en tur i slaven. Kommunikationsbehovet stiger proportionalt med kvadratet på antallet af byer i problemtilfældet, altså $O(n^2)$. Køretiden for den procedure, der konstruerer ture, stiger proportionalt med antallet af byer i problemtilfældet, mere præcist $O(cl \cdot n)$, hvor cl er antallet af kandidater i kandidatlisten.

På figur 8.1 ses en af graferne over speedup'ets udvikling som funktion af antallet af byer med 6 processorer. Her ses det tydeligt at der ikke vindes noget nævneværdigt ved at anvende flere processorer, uanset problemstørrelsen. Dog ses i et enkelt tilfælde et speedup på cirka 2. Vi må antage, at det er denne værdi af n (200), der giver den bedste balance imellem kommunikationsbehov og køretid for proceduren, der bygger ture. Grafer over speedups for de andre antal af processorer kan findes i appendix D.1.



Figur 8.1 Det gennemsnitlige relative speedup som funktion af antallet af byer i problemtilfældet for $p = 6$.

Kigger man på figur 8.2, ses det, hvordan køretiden, som funktion af antallet af processorer, udvikler sig lidt underligt. Der opnås typisk et speedup ved 2 til 4 processorer, men ved 4 til 8 processorer falder dette igen, i stedet for at holde sig nogenlunde konstant. Dette gælder alle problemtilfældene, men rat575 ser specielt usædvanlig ud og følger ikke et mønster. Grafer over speedups for de øvrige problemer forefindes i appendix D.2.



Figur 8.2 Det gennemsnitlige relative speedup, som funktion af antallet af processorer for TSP-tilfældet st70.

For en god ordens skyld bringer vi et uddrag af den tabel over speedup og effektivitet som Randall og Lewis præsenterer i deres artikel. Randall og Lewis benytter som tidligere nævnt ikke gennemsnit i deres beregninger af speedup'et. De beregner det reelle speedup samt den reelle effektivitet for én kørsel af 1000 iterationer på værdier af p op til 8. De for os relevante resultater fra deres forsøg fremlægges i tabel 8.5.

problem	$p = 2$	$p = 4$	$p = 6$	$p = 8$
gr24	0,08	0,07	0,07	0,06
	0,04	0,02	0,01	0,01
st70	0,27	0,23	0,23	0,21
	0,13	0,06	0,04	0,03
kroA100	0,41	0,37	0,37	0,33
	0,20	0,09	0,06	0,04
kroA200	0,82	0,85	0,92	0,90
	0,41	0,21	0,15	0,11
lin318	1,20	1,44	1,61	1,58
	0,60	0,36	0,27	0,20
pcb442	1,42	1,93	2,31	2,35
	0,71	0,48	0,38	0,29
rat575	1,56	2,10	2,77	3,08
	0,78	0,52	0,46	0,38
d657	1,67	2,32	3,25	3,30
	0,83	0,58	0,54	0,41

Tabel 8.5 Randall og Lewis' resultater for $p = 2, 4, 6$ og 8 [Randall and Lewis, 2002]. Første søjle er navnet på det pågældende problemtilfælde. De næste søjler viser for antallet af processorer skiftevis det reelle speedup og den reelle effektivitet for de respektive problemtilfælde.

Randall og Lewis udfører deres forsøg på en MIMD-datamat med fælles lager, nærmere betegnet en IBM SP2 bestående af 18 RS6000 processorer med højst 8 tilgængelige processorer for parallelberegninger. Det omtales dog ikke i Randall og Lewis' artikel, hvorvidt der benyttes en seperat processer til mesteren (ligesom i vores forsøg), eller om denne kører på en af slavernes processorer. Endvidere anvender Randall og Lewis som tidligere nævnt ikke kandidatlister i deres system. Dette betyder, at køretiden af algoritmen udvikler sig med samme hastighed som kommunikationsbehovet. Begge udvikler sig proportionalt med kvadratet på antallet af byer, altså $O(n^2)$.

Randall og Lewis beregner iøvrigt den eksperimentelt bestemte sekventielle del f (se ligning (5.8)). Denne størrelse angiver, som nævnt i afsnit 5.3.4, den andel af programmets køretid der stammer fra sekventielt udført kode. Denne størrelse bruger de til at finde det teoretiske maksimale speedup for deres system. At bestemme f for vores resultater giver dog ikke mening, idet denne størrelse bliver større end 1, hvis speedup'et er under 1.

8.2.3 Forsøg uden brug af kandidatliste

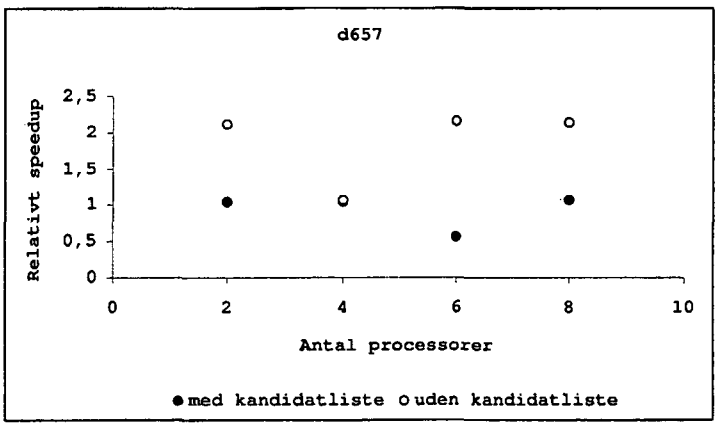
Grundet de forholdsvist lave speedup vi finder, har vi forsøgt os med at køre systemet uden brug af kandidatlister. Distribueret ACS uden kandidatliste medfører, at slaverne skal foretage flere beregninger, hvilket bør resultere i, at køretiderne for programmet bliver tilsvarende længere. Kørslerne uden brug af kandidatliste må således formodes, at have et højere speedup end de tilsvarende kørsler uden kandidatliste.

Vi har kørt d657 uden kandidatliste for de samme værdier af p, som d657 blev kørt med, ved brug af kandidatliste. Det relative speedup samt den relative effektivitet for begge disse forsøg kan ses i table 8.6.

d657	$p = 2$	$p = 4$	$p = 6$	$p = 8$
med kandidatliste	1,04	1,05	0,57	1,06
	0,52	0,26	0,09	0,13
uden kandidatliste	2,11	1,07	2,15	2,14
	1,06	0,27	0,36	0,27

Tabel 8.6 Tabel over det relative speedup og den relative effektivitet for problemtilfældet d657 kørt med og uden kanditatliste.

I figur 8.3 er speedup for henholdsvis kørslerne med og uden kandidatliste afbilledet i samme graf. Bortset fra de to afvigelser, ses det, at niveauet for speedup'et uden brug af kandidatliste ligger højere end speedup-niveauet uden brug af kandidatliste. Dette var, hvad vi havde forventet.



Figur 8.3 *Det gennemsnitlige relative speedup, som funktion af antallet af processorer for TSP-tilfældet d657, kørt med og uden kandidatliste.*

Det må også i forsøgene uden brug af kandidatliste konstateres at speedup'et udvikler sig noget underligt, som funktion af antallet af processorer. For d657 uden brug af kandidatliste ses et pludseligt fald i speedup ved 4 processorer.

8.3 Fejlkilder

Forsøgene er udført på maskiner med minimal aktivitet ud over den aktivitet ACS programmerne genererer. Dog vil uundgåelig systemaktivitet muligvis have en lille indflydelse på de målte køretider.

Netværket, som forsøgene er udført på, er et almindeligt 100M bit ethernet netværk. Vi har dog ikke været i stand til at kontrollere netværksaktivitet genereret af udenforstående maskinel, idet netværket vi har anvendt er en del af det almindelig campusnetværk på instituttet for Datalogi, Journalistik og Kommunikation ved RUC. Forsøgene er dog udført om natten, og vi har løbende lavet stikprøver på køretiden for et givet problem med et givet antal processorer ved at køre problemet først på ét sæt af maskiner og derefter på et andet sæt af maskiner. Disse stikprøver viste ikke nogen nævneværdige forskelle i køretiderne.

9 Diskussion

Dette kapitel indeholder en opsummering af de i rapporten tidligere nævnte problemstillinger sammen med en løbende diskussion af disse og de resultater, vi har opnået. Hele diskussionen bør ses i lyset af vores problemformulering. Vi vil også i diskussionen søge at forholde os kritisk til de kilder, hvori vi har taget vores udgangspunkt.

9.1 Opsummering og diskussion

9.1.1 Sekventiel ACS

I første punkt i vores problemformulering sætter vi som et mål for projektet at implementere en sekventiel udgave af ACS-TSP i overenstemmelse med den, der findes i Bonabeau et al., hvis resultater iøvrigt stammer fra Dorigo og Gambardella. Vi mener at have nået dette mål i den forstand, at vi med vores sekventielle udgave er i stand til at finde optimum på kroA100, hvilket de også er. Vi baserer således afgørligheden af, om vores program er 'godt nok' på sammenligninger af resultater fundet for et eneste probelmtilfælde. Dette forekommer måske nok at være et noget spinkelt grundlag for en sådan afgørelse, men i kraft af Dorigo og Gambardellas resultatfremstilling kan det ikke være anderledes. Dorigo og Gambardella kører måske nok 15 kørsler, men der forelægger i deres artikel ikke et gennemsnit af de fundne længder, ej heller en gennemsnitlig procentmæssig afvigelse fra optimum. Det eneste man kan udlede af deres resultater er, at de på et eller andet tidspunkt i løbet af deres 15 kørsler har fundet optimum. Iøvrigt synes vi at deres opgivelse af, hvor mange ture der er beregnet, før end de finder optimum er en smule mærkelig, idet de ikke opgiver noget sæd for deres kørsler. De kan således, blot ved at afprøve programmet med en masse forskellige sæd, nøjes med at opgive den mest fordelagtige kørsel og derved præsentere et resultat, der på ingen måde afspejler, hvad den pågældende implementering er i stand til at præstere.

9.1.2 Distribueret ACS

Andet punkt i vores problemformulering omhandler distribueringen af ACS-TSP efter Randall og Lewis' strategi *Parallelle myrer*. Vi mener at opfylde de krav, der stilles af denne strategi i den sproglige formulering. Vi har dog ingen mulighed for at kontrollere dette, da der ikke er angivet psuedo-kode for, hvordan denne strategi implementeres i et asynkront løst koblet distribueret system.

9.1.3 Løsningskvalitet for distribueret ACS

I tredje punkt af problemformuleringen omtaler vi testningen af vores distribuerede ACS-system på udvalgte TSP-tilfælde fra TSPLIB. Vi finder, at løsningskvaliteten af det distribuerede system er omtrent lige så god, som løsningskvaliteten af det sekventielle program. Dette må siges at understøtte ovenstående antagelse om, at vores implementation af parallelliseringsstrategien *Parallelle myrer* er foretaget korrekt.

Med hensyn til den generelle kvalitet af løsningerne fra begge implementationer må det dog konkluderes, at ingen af disse kan måle sig med velkendte metoder som for eksempel Lin-Kernighans λ-opt. ACS-TSP vil ofte ikke finde den optimale længde på selv små problemer som kroA100. Dog er kvaliteten af løsningerne ganske god når man betragter ACS som en kontruktionsheuristik.

9.1.4 Speedup for distribueret ACS

Fjerde punkt i problemformuleringen omhandler en undersøgelse af speedup'et for vores distribuerede system samt en sammenligning af dette med speedup'et for Randall og Lewis' system. Den på dette tidspunkt præsenterede hypotese om, at køretiden skulle være omvendt propotional med antallet af processorer må vi klart afvise for et system af denne type. Der ses et relativt speedup i visse situationer som vist i tabel 8.3, men som antallet af processorer vokser falder dette speedup igen og resulterer faktisk i visse tilfælde i et 'speeddown'. Endvidere var standardafvigelsen for de målte værdier meget svingende.

Dette mønster i speedup'ets udvikling som funktion af antallet processorer undrede os. Derfor gennemgik vi koden og fandt, at det var muligt for mesteren at læse pheromonmatricens værdier i en tråd, mens de er ved at blive opdateret i en anden tråd. Dette må betragtes som en fejl, som vi derfor rettede. Helt præcist drejede det sig om følgende linie kode fra `Master`-klassen:

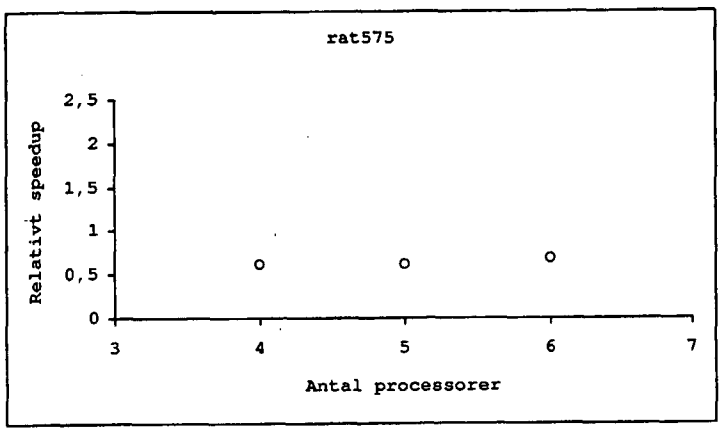
```
public void updateSlave(int id) {
```

som burde hedde:

```
public synchronized void updateSlave(int id) {
```

Derfor er alle resultater i afsnit 8.2 foretaget uden synkronisering af `updateSlave`-metoden. I kildekoden i appendiks B er fejlen dog rettet.

På det opdaterede program udførte vi en lille serie af forsøg, for henholdsvis 4, 5 og 6 processorer, på problemet rat575. Vi undersøgte speedup'et for 5 processorer for at sikre at der for det opdaterede program, ikke foregik noget usædvanligt imellem 4 og 6 processorer. Resultatet af dette ekstra forsøg viste, at der i det opdaterede program ikke foregår noget usædvanligt med 5 processorer, samt at den ovennævnte fejl medførte at programmet kunne udføres hurtigere end ellers. Dette kan ses ved at speedup'et for rat575 med fejlen rettet generelt ligger lavere, men at speedup'et ved hvert antal af processorer er det samme,



Figur 9.1 Det gennemsnitlige relative speedup, som funktion af antallet af processorer for TSP-tilfældet rat575, kørt med synkroniseret `updateSlave`-metode.

cirka 0,6. Dette er illustreret på figur 9.1. Den tilsvarende figur for rat575, uden synkronisering af `updateSlave`-metoden, findes i appendiks D (figur D.11).

Med hensyn til betydningen af denne fejl mener vi ikke, at de af denne fejl introducerede problemer i resultaterne, har en betydning for rapportens overordnede konklusion, som vi diskuterer senere.

For at følge op på hypotesen om at køretiden skulle være omvendt proportional med antallet af processorer, ændrede vi programmet til ikke at anvende kandidatlister. Dette svarer til Randall og Lewis' metode. Vores resultater med dette forsøg viser at vi opnår et speedup der er cirka dobbelt så stort som det vi opnår på samme problem med brug af kandidatlister. Dette forsøg er dog også foretaget på baggrund af kode hvor `updateSlave`-metoden ikke er synkroniseret. Dette betyder at vi må antage at det opnåede speedup er for stort, samt at det udsving der ses i værdien for speedup'et ved 4 processorer er en fejl der er opstået på grund af manglende synkronisering. Den stigning i speedup vi opnår ved at ændre programmet til ikke at anvende kandidatlister peger på at det er netværket der sætter grænsen for hvor hurtigt systemet afvikles. Vi fandt endvidere, efter at have ændret programmet til at anvende floats istedet for doubles, at vi opnåede en stigning i hastighed ved tæt på 100% for det distribuerede system og kun på cirka 10% for det sekventielle system. Denne undersøgelse af, hvad konsekvensen er af at halvere datamængden, der sendes imellem mester og slave, peger også på at flaskehalsen i systemet er netværket.

Årsagen til at det er netværket der er systemets flaskehals, mener vi skal findes i den type af distribution vi har foretaget. I et cluster af pc'er, hvor der ikke er fælles lager, er alle pheromonopdateringer nødt til at foregå over et almindeligt ethernet-netværk, som vi også nævner i afsnit 5.2. Hvis man skal opnå et anvendeligt speedup for ACS-TSP med denne distribueringsstrategi, bliver man altså nødt til at holde sig til stramt koblede systemer, eller som minimum anvende et netværk med meget højere båndbredde, hvor størrelsen af pheromonmatricen har mindre betydning for hastigheden af programmet. I en maskine, hvor alle cpu'er har fælles lager, må det antages, at pheromonmatricens størrelse

har omtrent samme indflydelse på hastigheden, som den har i den sekventielle system.

Da alt pegede på at det var netværkets båndbredde der satte grænsen for hvor hurtigt vores program kørte, lavede vi en lille grov undersøgelse af hvor meget båndbredde vores program rent faktisk anvendte. Vi fandt ud af at det distribuerede system allerede med kun én slave anvendte tæt ved al den tilgængelige båndbredde på et 100MBit ethernet. Ved to eller flere slaver va båndbreddeforbruget en anelse højere, altså fuld udnyttelse af den tilgængelige båndbredde.

På baggrund af vores resultater samt ovennævnte overvejelser må vi, på trods af fejl i programmet, konkludere at vores system ikke egner sig til distribuering af ACS-TSP efter paralleliseringsstrategien *Parallelle myrer* over et 100MBit netværk. Derfor vil det ikke være til nogen nytte at foretage en sammenligning af vores resultater med Randall og Lewis' resultater. Konklusionen må være at parallelisering af ACS-TSP ved hjælp af strategien *Parallelle myrer* egner sig bedst til systemer med fælles lager.

9.1.5 Yderligere kommentarer

Løsningerne for problemtilfældet d657 har vist sig at være relativt ringe i forhold til løsningerne for de resterende problemtilfælde. Det kan muligvis skyldes at en nærmeste nabo kandidatliste med 15 byer bliver for lille, når antallet byer vokser. Endvidere undrer vi os over, hvorfor kanterne fra den initielle nærmeste nabo tur ikke indgår i ACS-systemet som hidtil bedste tur, det vil sige, hvorfor der ikke fra start foretages global opdatering på kanterne tilhørende denne tur.

Randall og Lewis nævner i deres artikel, at det muligvis kan være af interesse at lade den processor der betjener mesteren betjene en arbejder såvel. Dette skyldes at mesterprocessoren ikke udnytter sin CPU-tid fuldt ud. Dette må antages at være korrekt, men hvis vi gjorde dette, mener vi ikke at vi ville måle på de rigtige størrelser, idet vi var interesserede i at undersøge om ACS kunne parallelliseres ved hjælp af strategien *Parallelle myrer* på et løst koblet system.

Til sidst skal det nævnes, at vores system kan arbejde med TSP'er generelt. Vi arbejder dog kun med symmetriske TSP'er, hvorfor vi kunne have valgt at repræsentere samtlige matricer som nedre trekantsmatricer og dermed have reduceret mængden af kommunikation til det halve (helt præcist fra n^2 til $(n^2-n)/2$). Vi har grundet vores øvrige resultater fundet, at dette ekstra tiltag, ikke ville føje noget nyt til vores konklusion om, at løst koblede systemer med individuelt lager ikke er videre gode til denne form for parallellisering.

9.2 Perspektiver

En mulighed for at forbedre på resultaterne af de af ACS producerede ture er at anvende en egentlig forbedringsheuristik på turene efterfølgende. Dette gør blandt andet Dorigo og Gambardella. Nærmere bestemt er det λ-opt, med $\lambda = 3$ de benytter. En mulighed er dog at man kunne have anvendt en mere simpel konstruktionsheuristik end ACS, som for eksempel nærmeste nabo eller savings-heuristikken, og måske dermed have opnået lige så gode resultater.

Hvis vi skulle komme med et bud på hvor myreintelligens ville være særligt egnet, skulle det være indenfor tynde dynamiske netværk. I et sådant netværk kommer der hele tiden nye byer til imens andre forsvinder. En myrealgoritme kunne her anvendes til hurtigt at finde en eller flere gode ruter imellem to knuder A og B, det vil sige til at løse korteste vej problemet. For eksempel vises det i [Bundgaard et al., 2002], hvordan myreintelligens kan bruges til at rute pakker i et simuleret computernetværk over flere ruter. Endvidere viser Bundgaard et al., at deres myreintelligensbaserede algoritme, kaldet *Ant Routing System* (ARS) i gennemsnit ruter pakker hurtigere igennem netværket end Dijkstras *Single-Source Shortest Paths* algoritme er i stand til.

Der findes også andre eksempler på at myrealgoritmer egner sig bedre til andre problemer end TSP. For eksempel beskriver Bonabeau et al. et system kaldet *Hybrid Ant System* (HAS) til løsning af det kvadratiske tildelings problem (QAP) [Bonabeau et al., 1999]. AS-algoritmen er her ændret fra at være en avanceret konstruktionsheuristik til at være en forbedringsheuristik. HAS-QAP er i stand til at konkurrere med selv de bedste QAP-algoritmer, men dog kun hvis de gode ture ligger i naboområdet til andre gode ture, altså må der kræves en god initialtur. Det lader således ikke til, at HAS-QAP er ikke istand til at bevæge sig ud af et lokalt minimum, som for eksempel *Tabu Search* (TS) heuristikker er det.

Helt afslutningsvist i projektforløbet syntes vi, at det kunne være sjovt at se, hvordan vores sekventielle program ville klare sig, hvis vi lavede turkonstruktionen i programmet flertrådet. Derfor findes i appendiks B.1.5 og B.1.6 kildekode for et sådant program. Vi har kun foretaget enkelte testkørsler af dette program på problemtilfældene gr24 og d657. Testkørslerne blev foretaget på en IMB pc med to Intel Pentium 2 processorer og Windows2000. På denne maskine ville et perfekt speedup være på 2,0. Testkørslerne viser, at programmet opnår et reelt speedup på cirka 1,7 for begge problemtilfælde. Holder man disse resultater op imod dem Randall og Lewis angiver, ser det særdeles fint ud. Speedup'et for gr24 er cirka ti gange større end det Randall og Lewis opnår for samme problemtilfælde. For d657 ligger speedup'et på omtrent samme niveau som det speedup Randall og Lewis finder.

10 Konklusion

Vi har implementeret en sekventiel udgave af ACS-TSP, som må antages at være korrekt, da resultaterne fundet med denne er sammenlignelige med dem fundet af Bonabeau et al.

Ligeledes har vi fundet en metode til at parallelisere ACS-TSP efter strategien *Parallelle myrer* på et løst koblet system.

Vi har efterfølgende testet løsningskvaliteten af vores distribuerede system og fundet, at denne fuldt ud kan måle sig med løsningskvaliteten af det sekventielle system.

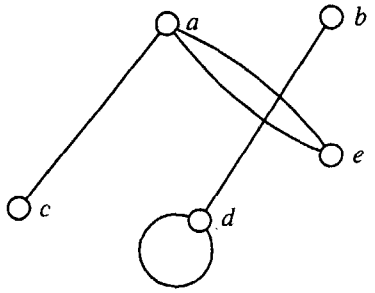
Undersøgelser af speedup'et for det distribuerede system er foretaget. Disse viser at paralleliseringsstrategien *Parallelle myrer* ikke egner sig til distribuering på et løst koblet system.

A Udvalgt grafteori

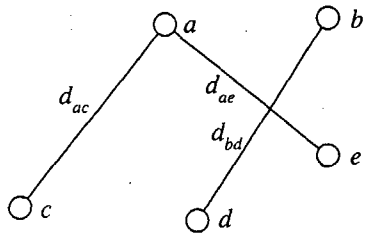
I det nedenstående er der taget udgangspunkt i [Dolan and Aldous, 1995], [Biggs, 1989] samt [Backchi et al., 2001], hvorfra centrale dele stammer.

En *graf* $G = (V, E)$ er et par bestående af en endelig mængde $V = \{1, 2, \ldots, n\}$, hvis elementer kaldes knuder, og en mængde E bestående af delmængder af V med to elementer, altså $V \supseteq \{i, j\} \in E$, herefter skrives de (i, j). Elementerne i E kaldes kanter (jævnfør figur A.1).

For en *simpel graf* gælder, at der kun findes én kant imellem to givne knuder i og j, samt at der ikke findes nogen kanter fra i til i (figur A.2).



Figur A.1 En graf $G = (V, E)$ med fem knuder og fem kanter, hvor to er ens [Backchi et al., 2001].



Figur A.2 En simpel vægtet graf med fem knuder og tre kanter [Backchi et al., 2001].

En graf kan vægtes ved at tildele enhver kant (i, j) en vægt d_{ij} (jævnfør figur A.2). En *vægtet graf* skrives $G = (V, E, D)$, hvor $D \subseteq \mathbf{R}$.

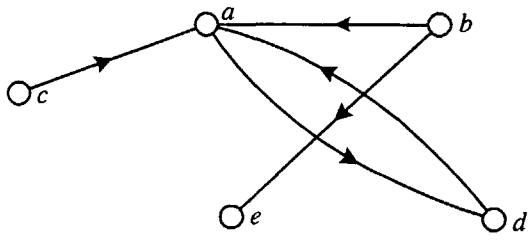
En *komplet graf* er en graf, hvor ethvert par af knuder (i, j) med $i \neq j$ er forbundet med netop én kant[1] (jævnfør figur A.4).

I nogle tilfælde kan det være nødvendigt at definere en retning på kanten, for eksempel hvis det kun er muligt at gå fra knude i til j, men ikke fra j til i. Her benyttes en digraf (directed graph) i stedet for en graf.

En *digraf*, også kaldet en *orienteret graf*, $H = (V, A)$ er en graf, hvor kanterne er retningsorienterede (jævnfør figur A.3). Det vil sige $A = \{(i, j) \mid i, j \in V\}$ er mængden af ordnede par af knuderne tilhørende V.

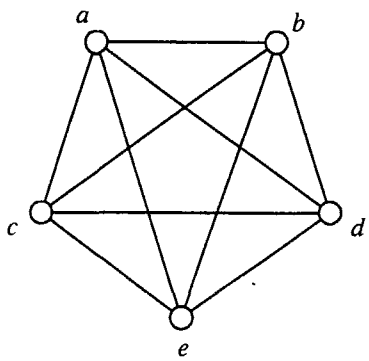
Når knuder og/eller kanter i en graf (eller en digraf) har tilknyttet talværdier kalder man også det samlede system for et *netværk*.

[1] Antallet af kanter i en komplet graf med n knuder er $\binom{n}{2} = \frac{n(n-1)}{2}$.

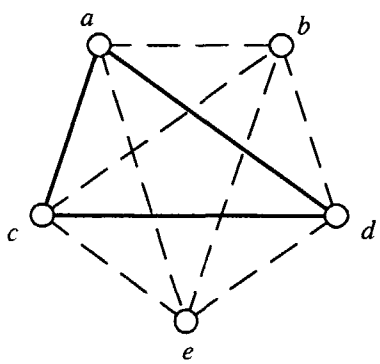


Figur A.3 En digraf $H = (V, A)$ med 5 knuder og 5 retningsorienterede kanter [Backchi et al., 2001].

En *cykel* i en graf $G = (V, E)$ er en delmængde $V_k \subseteq V$ med $|V_k| = k$. Elementerne i V_k opskrives som en endelig følge $i_1, i_2, \ldots, i_{k+1}$, med $i_1 = i_{k+1}$, og de resterende elementer i V_k kun optrædende én gang, det vil sige $i_j \neq i_l$ for alle $j, l \in V_k \setminus \{1, k+1\}$. To på hinanden følgende knuder i følgen er forbundet med en kant, altså $E_k = \{(i_j, i_{j+1}) \mid 1 \leq j \leq k\}$. Cyklen består altså af k knuder og k kanter, hvor der til enhver knude knyttes to kanter (jævnfør figur A.5). En cykel med k kanter kaldes også en *k-cykel*.



Figur A.4 En komplet graf $G = (V, E)$ med fem knuder og 10 kanter [Backchi et al., 2001].



Figur A.5 En 3-cykel i G [Backchi et al., 2001].

En *Hamiltonisk kreds* er en cykel af længde k i en graf med k knuder, det vil sige $V_k = V$.

B Java-kode

B.1 acs

B.1.1 ACS (side 77)

Denne klasse indeholder al den funktionalitet der relaterer til selve ACS-algoritmen, som for eksempel konstruktion af ture, lokal samt global pheromonopdatering.

B.1.2 SequentialACS (side 83)

Denne klasse kontrollerer forløbet af det sekventielle program. For eksempel antallet af iterationer, konstanter vedrørende ACS'et, samt hvilken tur der er den hidtil bedste til et givet tidspunkt. Kalder både turkonstruktion og pheromonopdateringsmetoder i `ACS`-klassen.

B.1.3 Master (side 86)

Denne klasse kontrollerer forløbet af det distribuerede program: Antallet af iterationer, konstanter vedrørende ACS'et (herunder antallet af slaver), samt hvilken tur der er den hidtil bedste til et givet tidspunkt. Kalder kun pheromonopdateringsmetoder i `ACS`-klassen.

B.1.4 Slave (side 92)

Denne klasse er en passiv klasse som foretager konstruktion af ture på foranledning af mesteren. Kalder kun turkonstruktionsmetoden i `ACS`-klassen.

B.1.5 ThreadedMaster (side 95)

Denne klasse kontrollerer forløbet af det trådede program. For eksempel antallet af iterationer, konstanter vedrørende ACS'et, samt hvilken tur der er den hidtil bedste til et givet tidspunkt. Kalder kun pheromonopdateringsmetoderne i `ACS`-klassen.

B.1.6 ThreadedSlave (side 99)

Denne klasse foretager aktiv konstruktion af ture. Kalder turkonstruktionsmetoden i `ACS`-klassen samt turtilføjningsmetoden i `TreadedMaster`-klassen.

B.2 acs.net

B.2.1 MasterMessageHandler (side 100)

Denne klasse modtager og reagerer på baggrund af de beskeder mesteren modtager fra slaverne.

B.2.2 SlaveMessageHandler (side 106)

Denne klasse modtager og reagerer på baggrund af de beskeder slaven modtager fra mesteren.

B.2.3 BeginMsg (side 109)

Denne og de resterende `Msg`-klasser i `acs.net` er de beskeder, der anvendes imellem mester og slave.

B.2.4 EndMsg (side 110)

B.2.5 InitValsMsg (side 111)

B.2.6 NewTourMsg (side 112)

B.2.7 UpdateValsMsg (side 113)

B.3 acs.util

B.3.1 ProblemLoader (side 114)

Denne klasse står for indlæsningen af problemtilfælde fra fil.

B.1 acs

B.1.1 ACS

```
1  /**
2   *Fælles klasse for de tre programmer, SequentialACS, Master og Slave, der
3   *udgør vores ACS system. I ACS klassen findes de metoder, de tre programmer
4   *har til fælles.
5   *
6   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
7   */
8
9  package acs;
10
11 import java.util.*;
12 import java.text.NumberFormat;
13 import java.io.IOException;
14
15 public class ACS {
16     /**
17      *Definition af konstanter til at vælge global opdatering fra
18      *henholdsvis Randall & Lewis's artikel eller fra Bonabeau et al.
19      */
20     public static final int RANDALL = 1, BONABEAU = 0;
21
22     public float tau0, GAMMA, RHO, q0;
23     public int cl, seed, n;
24     public float eta [][];
25     public int d [][], candidate[][];
26     Random random;
27
28     /**
29      * Initialiserer en ny ACS instans til brug af en slave. Kan ikke opdatere
30      *pheromon, kun lave nye ture.
31      *@param eta Synlighedsmatrix. eta[i][j] angiver synlighedsparameteren for
32      *kanten (i,j), imellem by i og by j.
33      *@param d Afstandsmatrix. d[i][j] angiver længden (omkostningen) af kanten,
34      *(i,j), imellem by i og by j.
35      *@param candidate Kandidatliste. candidate[i][j] angiver kandidat j for
36      *by i.
37      *@param q0 Tærskelværdi for deterministisk vs. nondeterministisk valg
38      *af næste by.
39      *@param seed Sæd for tilfældighedsgenerator.
40      */
41     public ACS(float eta[][], int d [][], int candidate[][],
42             float q0, int seed) {
43         this.eta = eta;
44         this.d = d;
45         this.candidate = candidate;
46         this.q0 = q0;
47         this.seed = seed;
48         this.cl = candidate[0].length;
49         random = new Random(seed);
50     }
51     /**
52      * Initialiserer en ny ACS instans til brug af mesteren. Kan både lave ture
53      *og opdatere pheromon. Bruges af Master klassen og SequentialACS.
54      *@param eta Synlighedsmatrix. eta[i][j] angiver synlighedsparameteren for
55      *kanten (i,j), imellem by i og by j.
56      *@param d Afstandsmatrix. d[i][j] angiver længden (omkostningen) af kanten,
57      *(i,j), imellem by i og by j.
```

```
58      *@param tau0 Det initielle pheromonniveau.
59      *@param GAMMA Parameter til kontrol af hvor stor vægt det nuværende
60      *pheromonniveau på den bedste tur skal have ved global opdatering.
61      *@param RHO Parameter til kontrol af hvor stor vægt det nuværende
62      *pheromonniveau på myrens tur skal have ved lokal opdatering.
63      *@param q0 Tærskelværdi for deterministisk vs. nondeterministisk valg
64      *af næste by.
65      *@param cl Antallet af kandidater.
66      *@param seed Sæd for tilfældighedsgenerator.
67      */
68     public ACS(float eta[][], int d [][], float tau0, float GAMMA, float RHO,
69             float q0, int cl, int seed) {
70         this.eta = eta;
71         this. d = d;
72         this.tau0 = tau0;
73         this.GAMMA = GAMMA;
74         this.RHO = RHO;
75         this.q0 = q0;
76         this.n = d.length;
77         this.cl = cl;
78         random = new Random(seed);
79         candidate = makeCandidates(n, cl);
80
81     }
82
83     /**
84      *Beregner procentvis afvigelse af a fra b.
85      *@param a Afviger.
86      *@param b Enhed (optimum i vores tilfælde).
87      *@return Afvigelse i procent.
88      */
89     public float percentDeviation(int a, int b) {
90         return ((float)a / (float)(b)-1)*100.0f;
91     }
92
93     /**
94      *Udskriver en tur til konsollen.
95      *@param tour En tur angivet ved rækkefølgen af byernes numre i arrayet.
96      */
97     public void printTour(int[] tour) {
98         for(int i = 0; i < tour.length; i++) {
99             System.out.print(tour[i] + "␣");
100        }
101        System.out.print("\n");
102    }
103
104    /**
105     *Udskriver pheromonmatricen. Debugging metode til at undersøge programmets
106     *korrekthed.
107     *@param tau En pheromonmatrix hvor tau[i][j] angiver pheromonniveau for
108     *kanten (i,j), imellem by i og by j.
109     */
110    public void printTau(float[][] tau) {
111        NumberFormat nf = NumberFormat.getInstance();
112        nf.setMaximumFractionDigits(1);
113        for(int i = 0; i < tau.length; i++) {
114            for(int j = 0; j < tau.length; j++) {
115                System.out.print(nf.format(100000*tau[i][j]) + "␣");
116            }
117            System.out.print("\n");
118        }
119    }
```

```
120
121     /**
122      *Finder en tur for et TSP ved hjælp af nærmeste nabo heuristikken.
123      *@param d Afstandsmatrix. d[i][j] angiver længden (omkostningen) af kanten
124      *(i,j), imellem by i og by j.
125      *@return Turen angivet ved byernes orden i det returnerede int array.
126      */
127     public static int[] nnTour(int d[][]) {
128         //Finder tur med nærmeste nabo heuristikken.
129         int n = d.length;
130         int nnTour[] = new int[n+1];
131         boolean tabu[] = new boolean[n];
132         for(int i = 1; i < n; i++)
133             tabu[i] = false;
134         nnTour[0] = nnTour[n] = 0;
135         tabu[0] = true;
136         for(int i = 1; i < n; i++) {
137             int nearest = 0;
138             for(int j = 0; j < n; j++) {
139                 if(!tabu[j]) {
140                     if(nearest == 0 || d[nnTour[i-1]][j] < d[i-1][nearest])
141                         nearest = j;
142                 }
143             }
144             nnTour[i] = nearest;
145             tabu[nearest] = true;
146         }
147         return nnTour;
148     }
149
150     /**
151      *Laver kandidatlisten.
152      *@param n Antallet af byer i TSP'et.
153      *@param cl Antallet af kandidater til hver by.
154      *@return Kandidatliste. candidate[i][j] angiver kandidat j for
155      *by i.
156      */
157     private int [][] makeCandidates(int n, int cl) {
158         int candidate[][] = new int[n][cl];
159         for(int i = 0; i < n; i++)
160             for(int j = 0; j < cl; j++)
161                 candidate[i][j] = Integer.MAX_VALUE;
162         for(int i = 0; i < n; i++) {
163             for(int j = 0; j < n; j++) {
164                 if(i != j) {
165                     for(int c = 0; c < cl; c++) {
166                         if(candidate[i][c] == Integer.MAX_VALUE) {
167                             candidate[i][c] = j;
168                             break;
169                         }
170                         else if(d[i][j] < d[i][candidate[i][c]]) {
171                             for(int a = cl-1; a > c; a--) {
172                                 candidate[i][a] = candidate[i][a-1];
173                             }
174                             candidate[i][c] = j;
175                             break;
176                         }
177                     }
178                 }
179             }
180         }
181         return candidate;
```

```
182     }
183
184     /**
185      *Returner dette problems kandidatliste.
186      */
187     public int [][] getCandidates() {
188         return candidate;
189     }
190
191     /**
192      *Opdaterer pheromonmatricen tau, jævnfør den lokale opdateringsregel.
193      *@param tour Turen der skal bruges i beregningen af pheromonopdateringen.
194      *@param tau Nuværende pheromonværdier.
195      *@return Matrix med nye pheromonværdier.
196      */
197     public float [][] localUpdate(int tour[], float tau[][]) {
198         int n = tour.length;
199         for(int i = 1; i < n; i++) {
200             tau[tour[i-1]][tour[i]] = tau[tour[i]][tour[i-1]]
201             = (1.0f - RHO) * tau[tour[i-1]][tour[i]] + RHO * tau0;
202         }
203         return tau;
204     }
205
206     /*
207      *Global opdatering med hensyn til den angivne tur og pheromonmatrix.
208      *@param type ACS.RANDALL eller ACS.BONABEAU.
209      *@param bestTour Den tur der skal opdateres pheromon over.
210      *@param bestLength Længden af turen der skal opdateres over.
211      *@param tau Pheromonmatrix der skal odateres over.
212      *@return Den opdaterede pheromonmatrix.
213      */
214     public float [][] globalUpdate(int type, int bestTour[],
215                                    int bestLength, float tau[][]) {
216         int n = bestTour.length-1;
217         if(type == 1)
218         {
219             float Q = 100;
220             for(int i = 0; i < n; i++)
221                 for(int j = 0; j < n; j++)
222                     tau[i][j] = (1 - GAMMA) * tau[i][j];
223
224             for(int i = 0; i < n; i++)
225                 tau[bestTour[i]][bestTour[i + 1]]
226                  = tau[bestTour[i + 1]][bestTour[i]]
227                 += GAMMA*(Q/bestLength);
228         }
229         else if(type == 0) {
230             float Q = 1;
231             for(int i = 0; i < n; i++)
232                 tau[bestTour[i]][bestTour[i + 1]]
233                 = tau[bestTour[i + 1]][bestTour[i]]
234                 = (1 - GAMMA) * tau[bestTour[i]][bestTour[i + 1]]
235                 + GAMMA*(Q/bestLength);
236
237         }
238         return tau;
239     }
240
241     /**
242      *Genererer et array med m byer i tilfældig orden, til brug ved udvælgelse
243      *af startposition for myre k.
```

```
244      *@param m Antallet af myrer.
245      *@param n Antallet af byer i TSP'et.
246      *@param seed Sæd til tilfældighedsgenerator. Lokal for denne metode.
247      *@return Returnerer en serie af længde m, med tilfædige tal mellem 0 og n.
248      */
249     public int[] getStartLocs(int m, int n, int seed) {
250         Random random = new Random(seed);
251         int loc[] = new int[m];
252         int tempLocs[] = new int[n];
253         for(int i = 0; i < n; i++)
254             tempLocs[i] = i;
255         for(int i = 0; i < m; i++) {
256             int rand = random.nextInt(n-i);
257             loc[i] = tempLocs[rand];
258             tempLocs[rand] = tempLocs[(n-1)-i];
259         }
260         return loc;
261     }
262
263     /*
264      *Beregner længden af turen, tour.
265      *@param tour Den tur, hvis længde skal findes.
266      *@param d Afstandsmatrix. d[i][j] angiver længden (omkostningen) af kanten
267      *(i,j), imellem by i og by j.
268      *@return Længden af turen.
269      */
270     public static int computeLength(int tour[], int d[][]) {
271         int length = 0;
272         for(int i = 0; i < tour.length-1; i++) {
273             length += d[tour[i]][tour[i+1]];
274         }
275         return length;
276     }
277
278     /**
279      *Bygger en ny tur udfra pheromonmatricen tau og en given startposition.
280      *@param tau Den nuværende pheromonmatrix.
281      *@param startLoc Nummeret på den by turen skal starte i.
282      *@return En ny tur angivet ved ordenen på byerne i det returnerede array.
283      */
284     public int[] buildTour(float tau[][], int startLoc) {
285         int n = tau.length;
286         boolean candidateExists;
287         boolean tabu[] = new boolean[n];
288         int tempTour[] = new int[n+1];
289         int tempLength, last, next = 0;
290         float weights[] = new float[c1];
291         float sigmaWeights;
292         float q, tempWeight, target;
293
294         // Nulstiller tabulisten.
295         for(int i = 0; i < n; i++)
296             tabu[i] = false;
297
298         //Placerer myren.
299         tempTour[0] = tempTour[n] = startLoc;
300         tabu[tempTour[0]] = true;
301
302         //Vælger næste by i ruten.
303         for( int i = 1; i < n; i++) {
304             last = tempTour[i-1];
305
```

```
306             q = random.nextFloat();
307             sigmaWeights = 0.0f;
308
309             //Udregner sandsynligheden for at gå til hver af de mulige næste
310             //byer, hvis der er tabu på en by gås ikke til den.
311             for(int j = 0; j < cl; j++) {
312                 weights[j] = tabu[candidate[last][j]] ?
313                                 0.0f
314                                 :
315                                 tau[last][candidate[last][j]]
316                                 * eta[last][candidate[last][j]];
317             }
318             candidateExists = false;
319             for(int j = 0; j < cl; j++)
320                 if(!tabu[candidate[last][j]])
321                     candidateExists = true;
322
323             if(candidateExists) {
324                 if(q <= q0) {
325                     tempWeight = 0.0f;
326                     for(int j = 0; j < cl; j++) {
327                         if(weights[j] > tempWeight) {
328                             tempWeight = weights[j];
329                             next = candidate[last][j];
330                         }
331                     }
332                 }
333                 else {
334                     tempWeight = 0.0f;
335                     for(int j = 0; j < cl; j++)
336                         sigmaWeights += weights[j];
337                     target = random.nextFloat() * sigmaWeights;
338                     for(int j = 0; j < cl; j++) {
339                         tempWeight += weights[j];
340                         if(tempWeight >= target) {
341                             next = candidate[last][j];
342                             break;
343                         }
344                     }
345                 }
346             }
347             else {
348                 int temp = Integer.MAX_VALUE;
349                 for(int j = 0; j < n; j++) {
350                     if(!tabu[j] && d[last][j] < temp) {
351                         next = j;
352                         temp = d[last][j];
353                     }
354                 }
355             }
356             tabu[next] = true;
357             tempTour[i] = next;
358         }
359         return tempTour;
360     }
361 }
```

B.1.2 SequentialACS

```
1  /**
2   *Det sekventielle program. I denne klasse findes de metoder der
3   *kontrollerer det sekventielle programs forløb samt de for ACS-systemet
4   *konstante værdier.
5   *
6   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
7   */
8
9  package acs;
10
11 import java.io.*;
12 import java.util.*;
13 import java.text.*;
14 import acs.util.ProblemLoader;
15
16 public class SequentialACS{
17
18     /**
19      *Størrelsen af BETA parameteren i overgangsregelen.
20      */
21     public static final float BETA = 2;
22     /**
23      *Størrelsen af RHO parameteren i den lokale opdateringsregel.
24      */
25     public static final float RHO = 0.1f;
26     /**
27      *Størrelsen af GAMMA parameteren i den globale opdateringsregel.
28      */
29     public static final float GAMMA = 0.1f;
30     /**
31      *Størrelsen af q0 parameteren i overgangsregelen.
32      */
33     public static final float q0 = 0.9f;
34     /**
35      *Størrelsen af Q parameteren i den globale opdateringsregel.
36      */
37     public static final float Q = 1.0f;
38
39     /**
40      *Antallet af myrer.
41      */
42     public static final int m = 20;
43     /**
44      *Størrelsen af kandidatlisten.
45      */
46     public static final int CL = 15;
47     /**
48      *Det maksimale antal diskrete tidskridt system gennemløber.
49      */
50     public static final int T_MAX = 5000;
51
52     Random random;
53     ACS acs;
54     int bestLength, n, iteration, optimal, seed;
55     long runTime, tempTime;
56     float deviation;
57     int bestTour[];
58     int d [][];
59     float tau [][];
60
```

```
61      /**
62       *Starter det TSP brugeren angiver som parameter til programmet på
63       *konsollen. Kører SequentialACS 10 gange med sæd fra 1 til 10.
64       */
65      public static void main(String args[]) {
66          int trials = 10;
67          NumberFormat nf = NumberFormat.getInstance();
68          nf.setMaximumFractionDigits(3);
69          float sumDeviation = 0.0f;
70          int sumLength = 0;
71          long sumRuntime = 0;
72          int sumIteration = 0;
73          int allTimeBest = Integer.MAX_VALUE;
74          for(int i = 1; i < trials+1; i++){
75
76              SequentialACS main = new SequentialACS(
77                  ProblemLoader.loadData("resources\\" + args[0] + ".tsp"),
78                                          Integer.parseInt(args[1]), CL, i);
79              if(main.bestLength < allTimeBest)
80                  allTimeBest = main.bestLength;
81              sumDeviation += main.deviation;
82              sumLength += main.bestLength;
83              sumRuntime += main.runTime;
84              sumIteration += main.iteration;
85          }
86          System.out.println("Number of trials: " + trials + ".");
87          System.out.println("Average deviation from optimal: " +
88                             nf.format(100*(sumDeviation / 10.0f)) + "%");
89          System.out.println("Average best length: " +
90                             ((float)sumLength / (float)trials));
91          System.out.println("Average runtime: " +
92                             ((float)sumRuntime / (float)trials) / 1000.0f);
93          System.out.println("All time best: " + allTimeBest);
94      }
95
96      /**
97       *Starter en ny kørsel af SequentialACS med sæd = nummeret på kørslen.
98       *@param d Afstandsmatrix. d[i][j] angiver længden (omkostningen) af kanten
99       *(i,j), imellem by i og by j.
100      *@param optimal Længden af den optimale løsning på det TSP der ønskes kørt.
101      *@param cl Længden af antallet af kandidater per by.
102      *@param seed Sæd for denne kørsel.
103      */
104     SequentialACS(int d[][], int optimal, int cl, int seed) {
105         this.seed = seed;
106         tempTime = System.currentTimeMillis();
107         initialize(cl, d, seed, optimal);
108         run();
109         System.out.println("Seed: " + seed);
110         System.out.println("Iterations: " + iteration);
111         System.out.println("Run time " + (runTime/1000.0f) + " seconds.");
112     }
113
114     /**
115      * Initialiserer  systemet med de i klassen konstruktørangivne parametre.
116      */
117     public void initialize(int cl, int d[][], int seed, int optimal) {
118         this.optimal = optimal;
119         this.seed = seed;
120         random = new Random(seed);
121         this.d = d;
122         n = d.length;
```

```
123         tau = new float[n][n];
124         float eta[][] = new float[n][n];
125         bestTour = ACS.nnTour(d);
126         bestLength = ACS.computeLength(bestTour, d);
127         float tau0 = 1.0f / (n * bestLength);
128         for(int i = 0; i < n; i++)
129             for(int j = 0; j < n; j++)
130                 tau[i][j] = tau0;
131         for(int i = 0; i < n; i++)
132             for(int j = 0; j < n; j++)
133                 eta[i][j] = (float)Math.pow(d[i][j],-BETA);
134         acs = new ACS(eta, d, tau0, GAMMA, RHO, q0, c1, seed);
135     }
136
137     /**
138      *Kører indtil T_MAX iterationer er gået eller den optimale løsning
139      *er fundet.
140      */
141     public void run() {
142         deviation = 100.0f;
143         while(iteration < T_MAX) {
144             iteration++;
145             if(bestLength == optimal) {
146                 runTime = System.currentTimeMillis() - tempTime;
147                 return;
148             }
149             int startLoc[] = acs.getStartLocs(m, n, random.nextInt(100000));
150             for(int k = 0; k < m; k++) {
151                 int tempTour[] = null;
152                 //Laver en tur og opdaterer pheromonen lokalt m gange.
153                 tempTour = acs.buildTour(tau, startLoc[k]);
154                 int tempLength = ACS.computeLength(tempTour, d);
155                 tau = acs.localUpdate(tempTour, tau);
156                 if( tempLength < bestLength) {
157                     bestTour = tempTour;
158                     bestLength = tempLength;
159                 }
160             }
161             //Opdaterer pheromonen globalt.
162             tau = acs.globalUpdate(ACS.BONABEAU, bestTour, bestLength, tau);
163             deviation = ((float)bestLength/(float)optimal - 1.0f);
164             runTime = System.currentTimeMillis() - tempTime;
165         }
166     }
167 }
```

B.1.3 Master

```
1  /**
2   *Mesterprogrammet til den distribuerede version af ACS. I Master klassen
3   *findes de metoder der kontrollerer det distribuerede programs forløb med
4   *hensyn til mesteren, samt de for ACS systemet konstante værdier.
5   *
6   *Magnus Kass Meinild og Uffe Thomas Volmer Jankvist.
7   */
8
9  package acs;
10
11 import java.util.*;
12 import java.text.NumberFormat;
13 import java.io.IOException;
14 import acs.util.*;
15 import acs.net.*;
16
17 public class Master {
18
19     /**
20      *Størrelsen af BETA parameteren i overgangsregelen.
21      */
22     public static final float BETA = 2;
23     /**
24      *Størrelsen af RHO parameteren i den lokale opdateringsregel.
25      */
26     public static final float RHO = 0.1f;
27     /**
28      *Størrelsen af GAMMA parameteren i den globale opdateringsregel.
29      */
30     public static final float GAMMA = 0.1f;
31     /**
32      *Størrelsen af q0 parameteren i overgangsregelen.
33      */
34     public static final float q0 = 0.9f;
35     /**
36      *Størrelsen af Q parameteren i den globale opdateringsregel.
37      */
38     public static final float Q = 1.0f;
39     /**
40      *Antallet af myrer.
41      */
42     public static final int m = 20;
43     /**
44      *Størrelsen af kandidatlisten.
45      */
46     public static final int CL = 15;
47     /**
48      *Det maksimale antal diskrete tidskridt system gennemløber.
49      */
50     public static final int T_MAX = 5000;
51     /**
52      *Den TCP port systemet bruger.
53      *Er i praksis hardkodet og skal stemme overens med den hardkodede
54      *PORT størrelse i Slave klassen.
55      */
56     public static final int PORT = 31337;
57     /**
58      *Det antal slaver systemet forventer vil forbinde. Når dette antal er nået
59      *startes systemet automatisk.
60      */
```

```
61      public static final int NUMBER_OF_SLAVES = 8;
62
63      int tempLength, bestLength, cl, n,
64          tours, optimal, seed, iteration, trial;
65      float sumLength, sumDeviation, sumRuntime;
66      long startTime, minRuntime = Long.MAX_VALUE,
67           maxRuntime = Long.MIN_VALUE;
68      int bestTour[], startLoc[];
69      int d [][], candidate[][];
70      float tau [][], eta [][];
71      String problem;
72      boolean slaveReady[] = new boolean[NUMBER_OF_SLAVES];
73      Random random;
74      ACS acs;
75      MasterMessageHandler MMH;
76
77      /**
78       *Starter mesteren.
79       *@param args Skal have navnet på det problem der ønsket kørt som første
80       *parameter og længden af den optimale tur for det givne problem som
81       *anden parameter. For eksempel: "java acs/Master kroa100 21282".
82       */
83      public static void main(String[] args) {
84          Master m = new Master(args[0], Integer.parseInt(args[1]));
85      }
86
87      /**
88       *Opretter nyt Master objekt og starter systemet.
89       *Sætter slavernes status til ikke-klar og laver derefter nyt
90       *MasterMessageHandler objekt og sætter det til at modtage
91       *op til NUMBER_OF_SLAVES forbindelser. Starter derefter
92       *MasterMessageHandler objektet i en seperat tråd og sætter seed til et
93       *tilfædigt tal (sædet er i praksis ikke tilfældigt da vi kører 10 kørsler
94       *med sædene fra 1 til 10). Initialiserer så systemet og starter det.
95       *@param problem Navnet på det TSP der ønskes kørt.
96       *@param optimal Længden af den optimale løsning på det TSP der ønskes kørt.
97       */
98      public Master(String problem, int optimal) {
99          trial = 1;
100         for(int i = 0; i < NUMBER_OF_SLAVES; i++)
101             slaveReady[i] = false;
102         MMH = new MasterMessageHandler(this, NUMBER_OF_SLAVES, PORT);
103         MMH.listen();
104         MMH.start();
105         d = ProblemLoader.loadData("Resources/" + problem + ".tsp");
106         initialize(CL, trial, optimal);
107         startSlaves();
108     }
109
110     /**
111      * Initialiserer alle slaver, venter på klarbesked fra alle slaver og
112      *sætter dem i gang med at lave ture.
113      *@see acs.net.MasterMessageHandler#sendMsg(java.lang.Object data)
114      *@see acs.net.BeginMsg
115      */
116     public void startSlaves() {
117         //System.out.print(" Initializing slaves ...") ;
118         try {
119             for(int i = 0; i < MMH.currentNumberOfSlaves(); i++)
120                 MMH.sendMsg(getInitVals(), i);
121             while(true) {
122                 if(slavesReady()) {
```

```
123                 //System.out.println("Done.");
124                 //System.out.println("Starting calculations ...") ;
125                 System.out.println();
126                 MMH.sendMsg(new BeginMsg());
127                 startTime = System.currentTimeMillis();
128                 return;
129             }
130         }
131     }
132     catch(IOException e) {
133         System.out.println("Passing_of_initial_data_to_slaves_failed.");
134         e.printStackTrace();
135     }
136 }
137
138 /**
139  *Sætter en given slaves status til klar.
140  *@param slaveID Nummeret på den slave, hvis status skal sættes til klar.
141  */
142 public void setSlaveReady(int slaveID) {
143     slaveReady[slaveID] = true;
144 }
145
146 /**
147  *Undersøger om alle slaver er klar til at lave ture.
148  *@return Sand hvis alle slaver er klar, falsk ellers.
149  */
150 public boolean slavesReady() {
151     for(int i = 0; i < NUMBER_OF_SLAVES; i++)
152         if(!slaveReady[i])
153             return false;
154     return true;
155 }
156
157 /**
158  *Sender besked til slaver med opdaterede systemværdier.
159  *@param id Nummeret på den slave der skal opdateres med værdierne
160  *fra getValues(). Kaldes normalt fra addTour(int[] tour) efter en ny tur
161  *er modtaget.
162  *@see #getValues()
163  *@see #addTour(int[] tour)
164  */
165 public synchronized void updateSlave(int id) {
166     try {
167         MMH.sendMsg(getValues(), id);
168     }
169     catch(IOException e) {
170         e.printStackTrace();
171     }
172 }
173
174 /**
175  *Afleverer initielle værdier.
176  *@return Returnerer nyt InitValsMsg objekt med initielle værdier
177  *for systemet.
178  *@see #getKthStartPos()
179  */
180 public InitValsMsg getInitVals() {
181     InitValsMsg initVals = new InitValsMsg(tau, eta, d, candidate,
182             random.nextInt(100000), getKthStartPos());
183     return initVals;
184 }
```

```
185
186     /**
187      *Afleverer den opdaterede pheromonmatrix og en ny startlokation.
188      *@return Returnerer nyt UpdateValsMsg objekt med de opdaterede værdier.
189      *@see #getKthStartPos()
190      */
191     public UpdateValsMsg getValues() {
192        UpdateValsMsg vals = new UpdateValsMsg(tau, getKthStartPos());
193        return vals;
194     }
195
196     /**
197      *Accepterer en ny tur fra en slave og opdaterer pheromonmatricen
198      *i henhold til den lokale og den globale opdateringsregel.
199      *Kontrollerer desuden at det maksimale antal diskrete tidsskridt og
200      *den maksimale størrelse af det kontinuerte interval ikke overskrides.
201      *Tilsidst gemmes den nye tur som den hidtil bedste tur hvis den er
202      *kortere end den nuværende hidtil bedste tur.
203      *@param tour Den nye tur der skal tilføjes til systemet.
204      *@return Sand returneres hvis alt går godt, ellers falsk.
205      *@see acs.ACS#localUpdate(int[] tour, float [][] tau)
206      *@see acs.ACS#globalUpdate(int type, int[] bestTour, int bestLength,
207      *float [][] tau)
208      */
209     public synchronized boolean addTour(int[] tour){
210        if(bestLength > optimal &&
211              iteration < T_MAX) {
212           tempLength = ACS.computeLength(tour, d);
213           //Gemmer ny tur som hidtil bedste, hvis den er bedre end den
214           //nuværende bedste.
215           if(tempLength < bestLength) {
216              bestLength = tempLength;
217              bestTour = tour;
218           }
219           //Opdaterer lokalt.
220           tau = acs.localUpdate(tour, tau);
221           if(tours % m == 0) {
222              //System.out.println("Iteration " + iteration);
223              //Hvis alle myrer har lavet en tur, opdateres globalt.
224              tau = acs.globalUpdate(ACS.BONABEAU, bestTour,
225                    bestLength, tau);
226              iteration++;
227           }
228           if(iteration % 500 == 0 && tours % m == 0)
229              System.out.println("Iteration " + iteration);
230           tours++;
231           return true;
232        }
233        else if(trial < 15 ){
234           //Kører 10 trials.
235           NumberFormat nf = NumberFormat.getInstance();
236           nf.setMaximumFractionDigits(1);
237           System.out.println("Best " + bestLength + ".");
238           System.out.println("Seed " + seed +".");
239           System.out.println("Iterations " + iteration);
240           System.out.println("Deviation: " + nf.format(
241                          acs.percentDeviation(bestLength, optimal))
242                           + "%.");
243           long CPUTime = System.currentTimeMillis() - startTime;
244           System.out.println("Duration: "
245                          + (CPUTime
246                          / 1000.0f) + "seconds.");
```

```
247             sumDeviation += acs.percentDeviation(bestLength, optimal);
248             sumLength += bestLength;
249             if(trial > 5) {
250                 sumRuntime += CPUTime;
251                 if(CPUTime < minRuntime)
252                     minRuntime = CPUTime;
253                 if(CPUTime > maxRuntime)
254                     maxRuntime = CPUTime;
255             }
256             trial++;
257             if(trial<6)
258                 initialize(c1, 1, optimal);
259             else
260                 initialize(c1, trial-5, optimal);
261             startSlaves();
262             return true;
263         }
264         else {
265             //Stopper og udskriver gennemsnittet af længden, afvigelsen
266             //fra optimum og gennemsnittet af køretiden.
267             MMH.end();
268             NumberFormat nf = NumberFormat.getInstance();
269             nf.setMaximumFractionDigits(1);
270             System.out.println("Best " + bestLength + ".");
271             System.out.println("Seed " + seed +".");
272             System.out.println("Iterations " + iteration);
273             System.out.println("Deviation: " + nf.format(
274                                 acs.percentDeviation(bestLength, optimal))
275                                  + "%.");
276             System.out.println("Duration: "
277                                 + ((System.currentTimeMillis() - startTime)
278                                 / 1000.0f) + "seconds.");
279             System.out.println();
280             System.out.println("Avg length: "
281                 + (float)sumLength / (float)(trial-1));
282             System.out.println("Avg deviation: "
283                 + sumDeviation / (float)(trial-1));
284             System.out.println("Min runtime: "
285                 + ((float)minRuntime) / 1000);
286             System.out.println("Max runtime: "
287                 + ((float)maxRuntime) / 1000);
288             long CPUTime = System.currentTimeMillis() - startTime;
289             if(CPUTime < minRuntime)
290                 minRuntime = CPUTime;
291             if(CPUTime > maxRuntime)
292                 maxRuntime = CPUTime;
293             System.out.println("Avg runtime: "
294                 + (float)((sumRuntime += CPUTime) / 1000) / (float)(trial-5));
295             return true;
296         }
297     }
298
299     /**
300      * Initialiserer  systemet på bagrund af de hardkodede konstanter, samt det
301      *indtastede problem og dets optimale længde.
302      *Følgende størrelser  beregnes: tau0, tau, eta og candidate.
303      *Følgende konstanter anvendes under initialiseringen : BETA, GAMMA, m,
304      *NUMBER_OF_SLAVES, PORT, Q, q0, RHO, T_MAX.
305      *@see acs.util .ProblemLoader#loadData(String problemName)
306      */
307     public void initialize(int c1, int seed, int optimal) {
308         tours = 1;
```

```
309         iteration = 1;
310         random = new Random(seed);
311         this.optimal = optimal;
312         this.cl = cl;
313         this.seed = seed;
314         n = d.length;
315         tau = new float[n][n];
316         eta = new float[n][n];
317
318         //En initiel tur beregnes ved hjælp af nærmeste nabo heuristikken.
319         bestTour = ACS.nnTour(d);
320         bestLength = ACS.computeLength(bestTour, d);
321
322         //tau0 beregnes.
323         float tau0 = 1.0f/(n * bestLength);
324         for(int i = 0; i < n; i++)
325             for(int j = 0; j < n; j++)
326                 tau[i][j] = tau0;
327         //eta beregnes.
328         for(int i = 0; i < n; i++)
329             for(int j = 0; j < n; j++)
330                 eta[i][j] = (float)Math.pow(d[i][j],-BETA);
331         //Nyt acs objekt med parametre der passer til at lave
332         //pheromonopdatering.
333         acs = new ACS(eta, d, tau0, GAMMA, RHO, q0, cl, seed);
334
335         //Startpositioner for iterationen 1 beregnes.
336         startLoc = acs.getStartLocs(m, n, random.nextInt(100000));
337
338         //Kandidatlisten beregnes.
339         candidate = acs.getCandidates();
340     }
341
342     /**
343      *Finder startpositionen for myre k af de m myrer.
344      *Når en iteration er gået beregnes m nye startpositioner.
345      *@return Startpositionen for myre k af de m myrer.
346      */
347     private int getKthStartPos() {
348         if(tours % m == 0)
349             startLoc = acs.getStartLocs(m, n, random.nextInt(100000));
350         return startLoc[tours % m];
351     }
352 }
```

B.1.4 Slave

```
1  /**
2   *Slaveprogrammet til den distribuerede version af ACS. I denne klasse findes
3   *de metoder der kontrollerer programmets forløb med hensyn til slaven samt
4   *få relevante konstanter for ACS systemet.
5   *
6   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
7   */
8
9  package acs;
10
11 import java.util.*;
12 import java.io.IOException;
13 import acs.net.*;
14 import acs.util.*;
15
16 public class Slave{
17
18     /**
19      *Den TCP port systemet bruger.
20      *Er i praksis hardkodet og skal stemme overens med den hardkodede
21      *PORT størrelse i Master klassen.
22      */
23     public static final int PORT = 31337;
24
25     /**
26      *Størrelsen af q0 parameteren i overgangsregelen.
27      */
28     public static final float   q0 = 0.9f;
29
30     int seed, startLoc;
31     float tau [][];
32     boolean ready = false;
33     Random random;
34
35     /**
36      *Håndterer kommunikation.
37      */
38     SlaveMessageHandler handler;
39
40     /**
41      *Håndterer beregninger af nye ture for slaven.
42      */
43     ACS acs;
44
45     /**
46      *Starter en slave. Slaveprogrammet tager én parameter, DNS navnet på
47      *den maskine mesterprogrammet kører på.
48      *@param args Første argument skal være DNS navnet på den maskine
49      *mesteren kører på.
50      */
51     public static void main(String[] args) {
52         Slave s = new Slave(args[0]);
53     }
54
55     /**
56      *Slaven laver ved konstruktion en ny SlaveMessageHandler og
57      *giver sig selv og navnet på den maskine der skal forbindes til , som
58      *parameter til det nye SlaveMessageHandler objekt.
59      *@param host DNS navnet på den maskine mesteren kører på.
60      */
```

```
61      public Slave(String host) {
62          handler = new SlaveMessageHandler(this, host, PORT);
63          handler.start();
64      }
65
66      /**
67       *Her indlæses systemets initielle værdier fra InitValsMsg beskeden.
68       *Når de initielle værdier er indlæst, sendes en klarbesked, ReadyMsg,
69       *til mesteren.
70       *@param initVals Objekt med initielle værdier fra mesteren. Modtages
71       *fra mesteren gennem SlaveMessageHandlerens metode readMsg().
72       *@see SlaveMessageHandler#readMsg()
73       */
74      public void init(InitValsMsg initVals) {
75          System.out.print("Got init message, initializing... ");
76          tau = initVals.tau;
77          seed = initVals.seed;
78          startLoc = initVals.startLoc;
79          random = new Random(seed);
80          //Nyt acs objekt med parametre der passer til at lave ture.
81          acs = new ACS(initVals.visibility, initVals.d, initVals.candidate,
82              q0, random.nextInt(100000));
83          ready = true;
84          System.out.println("Done.");
85          try {
86              //Sender klarbesked til mesteren.
87              handler.sendMsg(new ReadyMsg());
88          }
89          catch(IOException e) {
90              e.printStackTrace();
91          }
92      }
93
94      /**
95       *Opdateret pheromonmatrix og startby indlæses fra UpdateValsMsg beskeden
96       *og slaven sættes igang med at lave en ny tur.
97       *@param vals Objekt med opdaterede værdier for pheromonmatricen tau
98       *samt en ny startby startLoc.
99       */
100     public boolean update(UpdateValsMsg vals) {
101         try {
102             tau = vals.tau;
103             startLoc = vals.startLoc;
104             ready = true;
105             begin();
106             return true;
107         }
108         catch(ClassCastException cce) {
109             cce.printStackTrace();
110             return false;
111         }
112     }
113
114     /**
115      *Sætter turudregningen igang. Kalder buildTour() i ACS objektet.
116      *@return Sandt hvis alt gik godt, falsk hvis noget gik galt.
117      *@see #buildTour()
118      */
119     public boolean begin() {
120         if(ready) {
121             try {
122                 handler.sendMsg(buildTour(startLoc));
```

```
123             }
124             catch(IOException e) {
125                 e.printStackTrace();
126                 System.out.println("Passing new tour to master failed: " + e);
127             }
128             ready = false;
129             return true;
130         }
131         else {
132             System.out.println("I wasnt ready");
133             return false;
134         }
135     }
136     /**
137      *Stopper MasterHandler tråden i SlaveMessageHandler objektet og lukker
138      *programmet ned.
139      */
140     public void end() {
141         System.out.println("Got end message, exiting.");
142         handler = null;
143         System.exit(0);
144     }
145     /**
146      *Lav et NewTourMsg objekt på baggrund af en ny tur og returnerer dette.
147      *@return Et NewTourMsgobjekt med en ny tur i, lavet ved hjælp af
148      *buildTour() i ACS klassen.
149      *@see ACS#buildTour()
150      */
151     private NewTourMsg buildTour(int startLoc) {
152         return new NewTourMsg(acs.buildTour(tau, startLoc));
153     }
154 }
```

B.1.5 ThreadedMaster

```
1  /**
2   *Det trådede ACS program. I denne klasse findes de metoder der
3   *kontrollerer programmets forløb samt de for ACS-systemet
4   *konstante værdier.
5   *
6   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
7   */
8
9  package acs;
10
11 import java.io.*;
12 import java.util.*;
13 import java.text.*;
14 import acs.util.ProblemLoader;
15
16 public class ThreadedMaster {
17
18     /**
19      *Størrelsen af BETA parameteren i overgangsregelen.
20      */
21     public static final float BETA = 2;
22     /**
23      *Størrelsen af RHO parameteren i den lokale opdateringsregel.
24      */
25     public static final float RHO = 0.1f;
26     /**
27      *Størrelsen af GAMMA parameteren i den globale opdateringsregel.
28      */
29     public static final float GAMMA = 0.1f;
30     /**
31      *Størrelsen af q0 parameteren i overgangsregelen.
32      */
33     public static final float q0 = 0.9f;
34     /**
35      *Størrelsen af Q parameteren i den globale opdateringsregel.
36      */
37     public static final float Q = 1.0f;
38
39     /**
40      *Antallet af myrer.
41      */
42     public static final int m = 20;
43     /**
44      *Størrelsen af kandidatlisten.
45      */
46     public static final int CL = 15;
47     /**
48      *Det maksimale antal diskrete tidskridt system gennemløber.
49      */
50     public static final int T_MAX = 5000;
51
52     /**
53      *Det antal af slaver der ønskes. Programmet vil anvende dette
54      *antal ekstra tråde til at udføre programmet i ud over main-tråden.
55      */
56     public static final int NUMBER_OF_SLAVES = 2;
57
58     Random random;
59     ACS acs;
60     int tempLength, bestLength, n, iteration, optimal, seed, tours;
```

```
61      long runTime, tempTime;
62      float deviation;
63      int bestTour[], startLoc[];
64      int d [][];
65      float tau [][];
66      ThreadedSlave[] ts;
67
68      /**
69       *Starter det ACS-TSP brugeren angiver som parameter til programmet på
70       *konsollen.
71       */
72      public static void main(String args[]) {
73          NumberFormat nf = NumberFormat.getInstance();
74          nf.setMaximumFractionDigits(3);
75          ThreadedMaster main = new ThreadedMaster(
76              ProblemLoader.loadData("resources\\"␣+␣args[0]␣+␣".tsp"),
77                                      Integer.parseInt(args[1]), CL, 1);
78      }
79
80      /**
81       *Starter en ny kørsel af ThreadedMaster med sæd = nummeret på kørslen.
82       *@param d Afstandsmatrix. d[i][j] angiver længden (omkostningen) af kanten
83       *(i,j), imellem by i og by j.
84       *@param optimal Længden af den optimale løsning på det TSP der ønskes kørt.
85       *@param cl Længden af antallet af kandidater per by.
86       *@param seed Sæd for denne kørsel.
87       */
88      ThreadedMaster(int d[][], int optimal, int cl, int seed) {
89          tempTime = System.currentTimeMillis();
90          initialize(cl, d, seed, optimal);
91          start();
92          System.out.println("Seed:␣" + seed);
93      }
94
95      /**
96       * Initialiserer systemet med de i klassen konstruktørangivne parametre.
97       */
98      public void initialize(int cl, int d[][], int seed, int optimal) {
99          this.optimal = optimal;
100         this.seed = seed;
101         random = new Random(seed);
102         this.d = d;
103         n = d.length;
104         tau = new float[n][n];
105         float eta[][] = new float[n][n];
106         bestTour = ACS.nnTour(d);
107         bestLength = ACS.computeLength(bestTour, d);
108         float tau0 = 1.0f / (n * bestLength);
109         for(int i = 0; i < n; i++)
110             for(int j = 0; j < n; j++)
111                 tau[i][j] = tau0;
112         for(int i = 0; i < n; i++)
113             for(int j = 0; j < n; j++)
114                 eta[i][j] = (float)Math.pow(d[i][j],-BETA);
115         acs = new ACS(eta, d, tau0, GAMMA, RHO, q0, cl, seed);
116         startLoc = acs.getStartLocs(m, n, random.nextInt(100000));
117         ts = new ThreadedSlave[NUMBER_OF_SLAVES];
118         for(int i = 0; i < NUMBER_OF_SLAVES; i++)
119             ts[i] = new ThreadedSlave(i, this);
120     }
121
122     /**
```

```
123      *Kører indtil T_MAX iterationer er gået eller den optimale løsning
124      *er fundet.
125      */
126     public synchronized void addTour(int[] tour, int slaveID) {
127         deviation = 100.0f;
128         if(bestLength == optimal) {
129             //Afisutter programmet hvis en optimal tur er fundet.
130             runTime = System.currentTimeMillis() - tempTime;
131             System.out.println("Iterations: " + iteration);
132             System.out.println("Best: " + bestLength);
133             System.out.println("Run time " + (runTime/1000.0f) + " seconds.");
134             stop();
135             System.exit(0);
136         }
137         else if(iteration < T_MAX) {
138             //Tilføjer den nye tur til systemet.
139             tempLength = ACS.computeLength(tour, d);
140             if(tempLength < bestLength) {
141                 bestLength = tempLength;
142                 bestTour = tour;
143             }
144             tau = acs.localUpdate(tour, tau);
145             if(tours % m == 0) {
146                 //Hvis alle myrer har lavet en tur, opdateres globalt.
147                 tau = acs.globalUpdate(ACS.BONABEAU, bestTour,
148                         bestLength, tau);
149                 iteration++;
150             }
151             tours++;
152             deviation = ((float)bestLength/(float)optimal - 1.0f);
153             runTime = System.currentTimeMillis() - tempTime;
154         }
155         else {
156             //Afisutter programmet når det ønskede antal iterationer er nået.
157             runTime = System.currentTimeMillis() - tempTime;
158             System.out.println("Iterations: " + iteration);
159             System.out.println("Run time " + (runTime/1000.0f) + " seconds.");
160             stop();
161             System.exit(0);
162         }
163     }
164
165     /**
166      *Finder den næste startposition i systemet og returnerer den.
167      *Kaldes af slaverne.
168      *@return Den næste startposition.
169      */
170     public synchronized int getNextStartPos() {
171         if(tours % m == 0)
172             startLoc = acs.getStartLocs(m, n, random.nextInt(100000));
173         return startLoc[tours % m];
174     }
175
176     /**
177      *Henter pheromonmatricen tau synkroniseret.
178      *@return tau.
179      */
180     public synchronized float[][] getTau() {
181         return tau;
182     }
183
184     /**
```

```
185         *Starter systemets slaver.
186         */
187        public void start(){
188            for(int i = 0; i < NUMBER_OF_SLAVES; i++)
189                ts[i].start();
190        }
191
192        /**
193         *Stopper systemets slaver.
194         */
195        public void stop() {
196            for(int i = 0; i < NUMBER_OF_SLAVES; i++)
197                ts[i].end();
198        }
199    }
```

B.1.6 ThreadedSlave

```
1  /**
2   *Denne klasse varetager udelukkende turkonstruktion i det trådede
3   *ACS-program.
4   *
5   *
6   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
7   */
8
9  package acs;
10
11 public class ThreadedSlave extends Thread{
12     int id;
13     ThreadedMaster tm;
14     boolean done = false;
15
16     /**
17      *Laver en ny trådet slave-klasse.
18      *@param id Slavens ID.
19      *@param tm Reference til denne slaves mester.
20      */
21     public ThreadedSlave(int id, ThreadedMaster tm) {
22         this.id = id;
23         this.tm = tm;
24     }
25
26     /**
27      *Sætter denne slaves tråd igang med at generere ture. Tråden afsluttes
28      *når variablen done sættes til sand.
29      */
30     public void run() {
31         while(!done) {
32             tm.addTour(tm.acs.buildTour(tm.getTau(),
33                 tm.getNextStartPos()), id);
34         }
35         return;
36     }
37
38     /**
39      *Standser denne tråd ved at sætte variablen done til sand.
40      */
41     public void end() {
42         done = true;
43     }
44 }
```

B.2 acs.net

B.2.1 MasterMessageHandler

```
1  /**
2   *Klasser til håndtering af kommunikation imellem mester og slave(r).
3   *Specifik for vores ACS system.
4   *
5   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
6   */
7
8  package acs.net;
9
10 import java.io.*;
11 import java.net.*;
12 import java.util.*;
13 import acs.Master;
14
15 class SlaveConnection {
16     InputStream in;
17     OutputStream out;
18
19     /**
20      *Laver en ny SlaveConnection med in- og outputstrømme til den givne slave.
21      *@param i Inputstrømmen til den givne slave.
22      *@param o Outputstrømmen til den givne slave.
23      */
24     public SlaveConnection(InputStream i, OutputStream o) {
25         in = i;
26         out = o;
27     }
28 }
29
30 class SlaveHandler extends Thread{
31     boolean die = false;
32     MasterMessageHandler handler;
33     int slaveID;
34     boolean ready = false;
35
36     /**
37      *Laver en ny SlaveHandler til at afkode beskeder fra den slave den er
38      *tilknyttet gennem SlaveConnection via sit ID.
39      *@param slaveID ID'et på den slave denne SlaveHandler er tilknyttet.
40      *@param handler Reference til mesterens beskedhåndteringsklasse.
41      */
42     public SlaveHandler(int slaveID, MasterMessageHandler handler) {
43         this.slaveID = slaveID;
44         this.handler = handler;
45     }
46     /**
47      *Starter den løkke der afventer beskeder fra en given slave.
48      */
49     public void run() {
50         while(true) {
51             try {
52                 if(die)
53                     return;
54                 Object m = handler.readMsg(slaveID);
55
56                 //Afkoder og behandler indgående bedkeder afhængig af type.
57                 if(m != null) {
```

```
58                  if(m.getClass() == (new ReadyMsg()).getClass())
59                  {
60                      handler.m.setSlaveReady(slaveID);
61                  }
62                  if(m.getClass() == (new NewTourMsg()).getClass())
63                  {
64                      handler.m.addTour(((NewTourMsg)m).getTour());
65                      handler.m.updateSlave(slaveID);
66                  }
67              }
68          }
69          catch(IOException e) {
70              e.printStackTrace();
71          }
72      }
73  }
74
75  /**
76   *Afslutter løkken i run metoden og får tråden til at dø.
77   */
78  public void die() {
79      die = true;
80  }
81
82  }
83
84  public class MasterMessageHandler extends Thread {
85      TreeMap connections = new TreeMap();
86      TreeMap handlers = new TreeMap();
87      int port, numberOfSlaves;
88      Master m;
89
90      /**
91       *Laver nyt overordnet beskedhåndteringsobjekt til mesteren. Kan lytte
92       *efter nye forbindelser og håndtere dem og deres tilknyttede underordnede
93       *beskedhåndteringsobjekter - SlaveHandler.
94       *@param m Reference til det mesterobjekt der er tilknyttet dette
95       *beskedhåndteringsobjekt.
96       *@param numberOfSlaves Det ønskede antal slaver for en given kørsel.
97       *@param port Den port mesteren skal lytte efter forbindelser på.
98       */
99      public MasterMessageHandler(Master m, int numberOfSlaves, int port) {
100         this.m = m;
101         this.numberOfSlaves = numberOfSlaves;
102         this.port = port;
103     }
104
105     /**
106      *Stopper denne kørsel og sender stopbeskeder til alle slaver.
107      */
108     public void end() {
109         int id = 0;
110         try {
111             //Til alle slaver.
112             sendMsg(new EndMsg());
113         }
114         catch(IOException e) {
115             e.printStackTrace();
116         }
117         while(connections.size() > 0 )
118             removeSlave(id++);
119     }
```

```
120     /**
121      *Returner det næste ledige ID til en slavefobindelse.
122      */
123     synchronized public int nextSlaveId() {
124         return connections.size();
125     }
126
127     /**
128      *Returner en Vector med alle de ID'er der eksisterer på nuværende
129      *tidspunkt.
130      */
131     synchronized public Vector getSlaveIds() {
132         if(connections.size() > 0) {
133             Vector ids = new Vector();
134             Iterator itr = connections.keySet().iterator();
135             while(itr.hasNext())
136                 ids.add((Integer)itr.next());
137             return ids;
138         }
139         else
140             return new Vector();
141     }
142
143     /**
144      *Returner det nuværende antal af slaver.
145      */
146     synchronized public int currentNumberOfSlaves() {
147         return connections.size();
148     }
149
150     /**
151      *Tilføjer ny slaveforbindelse og slavehåndteringsobjekt til samlingen af
152      *forbindelser og slavehåndteringsobjekter, og tildeler den vilkårligt ID.
153      *@param i Inputstrøm til den nye slave.
154      *@param o Outputstrøm til den nye slave.
155      *@return Nummeret den nye slave fik.
156      */
157     synchronized public int addSlave(InputStream i, OutputStream o) {
158         int id = nextSlaveId();
159         addSlave(id, i, o);
160         return id;
161     }
162
163     /**
164      *Tilføjer ny slaveforbindelse og slavehåndteringsobjekt til samlingen af
165      *forbindelser og slavehåndteringsobjekter, og tildeler den vilkårligt ID.
166      *Erstatter eventuelt eksisterende forbindelse/håndteringsobjekt.
167      *@param id Det ønskede ID på forbindelsen.
168      *@param i Inputstrøm til den nye slave.
169      *@param o Outputstrøm til den nye slave.
170      */
171     synchronized public void addSlave(int id, InputStream i,
172             OutputStream o) {
173         connections.put(new Integer(id), new SlaveConnection(i,o));
174         SlaveHandler sh = new SlaveHandler(id, this);
175         sh.start();
176         handlers.put(new Integer(id), sh);
177     }
178
179     /**
180      *Fjerner og stopper slavehåndteringsobjekt med givet ID fra samlingen af
181      *eksisterende slavehåndteringsobjekter.
```

```
182      *@param id ID på den slave der ønskes fjernet og stoppet.
183      *@return Sandt hvis der var en slave med det givne ID, flask ellers .
184      */
185     synchronized public boolean removeSlave(int id) {
186         SlaveHandler h = (SlaveHandler)handlers.remove(new Integer(id));
187         if(h != null && connections.remove(new Integer(id)) != null ){
188             h.die();
189             return true;
190         }
191         return false;
192     }
193
194     /**
195      *Returnerer forbindelsesobjektet der er knyttet til et givet slave ID.
196      *@param id ID på den slave hvis forbindelsesobjekt ønskes.
197      *@return Forbindelsesobjekt til slave med nummer = id.
198      */
199     synchronized protected SlaveConnection getSlave(int id) {
200         return (SlaveConnection)connections.get(new Integer(id));
201     }
202
203     /**
204      *Lytter efter nye forbindelser indtil numberOfSlaves forbindelser er opnået.
205      *Når en ny slave forbinder, tildeles den et håndteringsobjekt og et
206      *forbindelsesobjekt.
207      *@return Det endelige antal slaver.
208      */
209     public int listen() {
210         ServerSocket socket = null;
211         try {
212             socket = new ServerSocket(port);
213         }
214         catch (IOException e) {
215             System.out.println("Failed to bind to port " + port);
216             e.printStackTrace();
217             System.exit(1);
218         }
219         System.out.println("Master running on port " + port);
220         while (currentNumberOfSlaves() < numberOfSlaves) {
221             try {
222                 if(socket != null) {
223                     Socket clientConn = socket.accept();
224                     System.out.println("Slave id "
225                         + addSlave(clientConn.getInputStream(),
226                         clientConn.getOutputStream()) + " added.");
227                 }
228             }
229             catch (Exception e) {
230                 e.printStackTrace();
231             }
232         }
233         return currentNumberOfSlaves();
234     }
235
236     /**
237      *Indlæser en besked fra en given slave og returnerer denne besked til
238      *slavens håndteringsobjekt.
239      *@param id Nummeret på den slave det ønskes at læse en besked fra.
240      *@return Beskeden som et Java Object.
241      */
242     public Object readMsg(int id) throws IOException {
243         Object msg = null;
```

```
244         SlaveConnection conn = getSlave(id);
245         if(conn != null) {
246             try {
247                 synchronized (conn.in) {
248                     ObjectInputStream oin = new ObjectInputStream(conn.in);
249                     msg = oin.readObject();
250                 }
251             }
252             catch(SocketException s) {
253                 System.out.println("Lost connection to slave " + id);
254                 s.printStackTrace();
255                 removeSlave(id);
256                 msg = null;
257             }
258             catch(ClassNotFoundException e) {
259                 e.printStackTrace();
260                 msg = null;
261             }
262         }
263         return msg;
264     }
265
266     /**
267      *Sender en besked til en given slave og returnerer sandt hvis det lykkedes
268      *og falsk ellers .
269      *@param msg Den besked der ønskes sendt til slaven.
270      *@param id Nummeret på den slave det ønskes at sende en besked til.
271      *@return Sandt hvis beskeden blev sendt, falsk ellers .
272      */
273     public boolean sendMsg(Object msg, int id) throws IOException {
274         SlaveConnection conn = getSlave(id);
275         if(conn != null) {
276             try {
277                 synchronized (conn.out) {
278                     ObjectOutputStream oout =
279                             new ObjectOutputStream(conn.out);
280                     oout.writeObject(msg);
281                 }
282             }
283             catch(SocketException s) {
284                 System.out.println("Lost connection to slave " + id);
285                 s.printStackTrace();
286                 removeSlave(id);
287                 return false;
288             }
289         }
290         return true;
291     }
292
293     /**
294      *Sender en besked til alle slaver og returnerer sandt hvis det lykkedes
295      *og falsk ellers .
296      *@param msg Den besked der ønskes sendt.
297      *@return Sandt hvis besked blev sendt, falsk ellers .
298      */
299     public boolean sendMsg(Object msg) throws IOException {
300         boolean success = true;
301         Iterator ids = connections.keySet().iterator();
302         while(ids.hasNext()) {
303             Integer id = (Integer)ids.next();
304             if(!sendMsg(msg, id.intValue()))
305                 success = false;
```

```
306             }
307             return success;
308         }
309     }
```

B.2.2 SlaveMessageHandler

```
1  /**
2   *Klasse til håndtering af kommunikation imellem slave og mester.
3   *Specifik for vores ACS system.
4   *
5   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
6   */
7
8  package acs.net;
9
10 import java.io.*;
11 import java.net.*;
12 import java.util.*;
13 import acs.Slave;
14
15 class MasterHandler extends Thread{
16     boolean die = false;
17     SlaveMessageHandler handler;
18
19     /**
20      *Laver et nyt beskedafkodningsobjekt til en slave.
21      *@param handler Reference til beskedhåndteringsobjektet i slaven.
22      */
23     public MasterHandler(SlaveMessageHandler handler) {
24         this.handler = handler;
25     }
26
27     /**
28      *Når denne tråd startes lytter MasterHandleren efter beskeder fra mesteren
29      *indtil en stopbesked modtages.
30      */
31     public void run() {
32         while(true) {
33             try {
34                 if(die)
35                     return;
36                 Object m = handler.readMsg();
37
38                 //Afkoder og behandler indgående bedkeder afhængig af type.
39                 if(m != null) {
40                     if(m.getClass() == (new InitValsMsg()).getClass())
41                     {
42                         handler.slave.init((InitValsMsg)m);
43                     }
44                     if(m.getClass() == (new BeginMsg()).getClass())
45                     {
46                         handler.slave.begin();
47                     }
48                     if(m.getClass() == (new UpdateValsMsg()).getClass())
49                     {
50                         handler.slave.update((UpdateValsMsg)m);
51                     }
52                     if(m.getClass() == (new EndMsg()).getClass())
53                     {
54                         handler.in.close();
55                         handler.out.close();
56                         handler.slave.end();
57                     }
58
59                 }
60             }
```

```
61              catch(IOException e) {
62                  e.printStackTrace();
63              }
64          }
65      }
66
67      /**
68       *Afslutter løkken i run metoden og får tråden til at dø.
69       */
70      public void die() {
71          die = true;
72      }
73
74  }
75
76  public class SlaveMessageHandler extends Thread {
77
78      InputStream in;
79      OutputStream out;
80      Slave slave;
81
82
83      /**
84       *Laver nyt beskedhåndteringsobjekt for slaven. Opretter en forbindelse til
85       *mesteren og håndterer beskeder fra mesteren.
86       *@param slave Reference til det slaveobjekt der er tilknyttet dette
87       *beskedhåndteringsobjekt.
88       *@param host DNS navn på den maskine mesteren kører på.
89       *@param port Den port mesteren skal kontaktes på.
90       */
91      public SlaveMessageHandler(Slave slave, String host, int port) {
92          this.slave = slave;
93          try {
94              Socket s = new Socket(host, port);
95              in = s.getInputStream();
96              out = s.getOutputStream();
97          }
98          catch(SocketException s) {
99              s.printStackTrace();
100         }
101         catch(IOException e) {
102             e.printStackTrace();
103         }
104         MasterHandler h = new MasterHandler(this);
105         h.start();
106     }
107
108     /**
109      *Læser besked ind fra mesteren og returnerer den som et objekt.
110      *@return Den modtagne besked som Java Object.
111      */
112     public Object readMsg() throws IOException {
113         Object msg = null;
114         try {
115             ObjectInputStream oin = new ObjectInputStream(in);
116             msg = oin.readObject();
117         }
118         catch(SocketException s) {
119             System.out.println("Lost connection to master.");
120             s.printStackTrace();
121             System.exit(1);
122         }
```

```
123         catch(ClassNotFoundException e) {
124             e.printStackTrace();
125             msg = null;
126         }
127         return msg;
128     }
129
130     /**
131      *Sender besked til mesteren og returnerer sandt hvis det lykkedes og
132      *falsk ellers .
133      *@param msg Den besked der ønskes afsendt.
134      */
135     public boolean sendMsg(Object msg) throws IOException {
136         try {
137             ObjectOutputStream oout = new ObjectOutputStream(out);
138             oout.writeObject(msg);
139         }
140         catch(SocketException s) {
141             System.out.println("dav");
142             System.out.println("Lost_connection_to_master.");
143             s.printStackTrace();
144             System.exit(1);
145         }
146         catch(IOException e) {
147             e.printStackTrace();
148             return false;
149         }
150         return true;
151     }
152
153 }
```

B.2.3 BeginMsg

```
1  /**
2   *Startbesked for ASC systemet.
3   *
4   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
5   */
6
7  package acs.net;
8
9  import java.io.Serializable;
10
11 public class BeginMsg implements Serializable {
12
13     public BeginMsg() {}
14
15 }
```

B.2.4 EndMsg

```
1  /**
2   *Slutbesked for ACS –systemet.
3   *
4   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
5   */
6
7  package acs.net;
8
9  import java.io.Serializable;
10
11 public class EndMsg implements Serializable {
12     public EndMsg() {
13     }
14 }
```

B.2.5 InitValsMsg

```
1  /**
2   *Besked der indholder de værdier der skal bruges af slaverne for at
3   * initialisere systemet.
4   *
5   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
6   */
7
8  package acs.net;
9
10 import java.io.Serializable;
11
12 public class InitValsMsg implements Serializable {
13     public float [][] tau, visibility;
14     public int [][] d, candidate;
15     public int seed, startLoc;
16
17     public InitValsMsg() {}
18
19     public InitValsMsg(float[][] tau, float[][] visibility, int[][] d,
20                        int [][] candidate, int seed, int startLoc) {
21         this.tau = tau;
22         this.visibility = visibility;
23         this.d = d;
24         this.candidate = candidate;
25         this.seed = seed;
26         this.startLoc = startLoc;
27     }
28 }
```

B.2.6 NewTourMsg

```
1  /**
2   *Besked der indehodler en ny tur fra en slave.
3   *
4   *Magnus Kass Meinild og Uffe Thomas Volmer Jankvist.
5   */
6
7  package acs.net;
8
9  import java.io.Serializable;
10
11 public class NewTourMsg implements Serializable {
12     int [] tour;
13
14     public NewTourMsg() {}
15
16     public NewTourMsg(int[] tour) {
17         this.tour = tour;
18     }
19     public int[] getTour() {
20         return tour;
21     }
22 }
```

B.2.7 UpdateValsMsg

```
1  /**
2   *Besked med opdaterede pheromonværdier som en slave modtager efter at
3   *have lavet en ny tur.
4   *
5   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
6   */
7
8  package acs.net;
9
10 import java.io.Serializable;
11
12 public class UpdateValsMsg implements Serializable {
13     public float [][] tau;
14     public int startLoc;
15
16     public UpdateValsMsg() {}
17
18     public UpdateValsMsg(float[][] tau, int startLoc) {
19         this.tau = tau;
20         this.startLoc = startLoc;
21     }
22 }
```

B.3 acs.util

B.3.1 ProblemLoader

```
1  /**
2   *Problem loader til distribueret ACS.
3   *
4   *Magnus Kaas Meinild og Uffe Thomas Volmer Jankvist.
5   */
6  package acs.util;
7
8  import java.util.*;
9  import java.io.*;
10
11 public class ProblemLoader {
12
13     static int n = 0;
14     static String workStr = "";
15
16     /**
17      *Indlæser problemdata som en int matrix.
18      *@param problemName Problemets navn, uden ".tsp" i enden.
19      *@return Omkostningsmatricen for det valgte problem.
20      */
21     public static int [][] loadData(String problemName) {
22         StringTokenizer strTok;
23         if(problemName != null) {
24             BufferedReader reader;
25             try
26             {
27                 reader = new BufferedReader(new FileReader(problemName));
28                 while(reader.ready())
29                 workStr += reader.readLine()+"\n";
30             }
31             catch(FileNotFoundException e) {
32                 System.err.println(e +
33                                    "\nFilen blev ikke fundet. " +
34                                    "Indtast nyt filnavn og prøv igen.");
35             }
36             catch(IOException e) {
37                 System.err.println(e +
38                                    "\nHardwarefejl under læsning. " +
39                                    "Indtast nyt filnavn og prøv igen.");
40             }
41         }
42
43         n = getNumberOfCities(new StringTokenizer(workStr, " \n\t\r\f"));
44
45         return buildDistMatrix(new StringTokenizer(workStr, " \n\t\r\f"));
46     }
47
48     /**
49      *Finder antallet af byer i et problem.
50      *@param strTok En StringTokenizer over indholdet i problemfilen.
51      *@return Antallet af byer i problemet.
52      */
53     private static int getNumberOfCities(StringTokenizer strTok) {
54         String tempStr = "";
55         while(true) {
56             tempStr = strTok.nextToken();
57             if( tempStr.equals("DIMENSION")) {
```

```
58              strTok.nextToken();
59              tempStr = strTok.nextToken();
60              return Integer.parseInt(tempStr);
61          }
62          else if(tempStr.equals("DIMENSION:")) {
63              tempStr = strTok.nextToken();
64              return Integer.parseInt(tempStr);
65          }
66      }
67  }
68
69  /**
70   *Afhængigt af hvilken form data er på, indlæses de enten direkte som en
71   *matrix eller som euklidiske koordinater der efterfølgende konverteres
72   *til en matrix.
73   *@param strTok En StringTokenizer over indholdet i problemfilen.
74   *@return Omkostningsmatricen for det valgte problem.
75   */
76  private static int [][] buildDistMatrix(StringTokenizer strTok) {
77      String tempStr = "";
78      String edgeWeightType = "UNKNOWN";
79
80      while(true) {
81          tempStr = strTok.nextToken();
82          if( tempStr.equals("EDGE_WEIGHT_TYPE")) {
83              strTok.nextToken();
84              tempStr = strTok.nextToken();
85              edgeWeightType = tempStr;
86              if(edgeWeightType.equals("EXPLICIT")) {
87                  return buildDistMatrixEXPLICIT(
88                      new StringTokenizer(workStr, "␣\n\t\r\f"));
89              }
90              else
91              {
92                  return buildDistMatrixEUC_2D(
93                      new StringTokenizer(workStr, "␣\n\t\r\f"));
94              }
95          }
96          else if(tempStr.equals("EDGE_WEIGHT_TYPE:")) {
97              tempStr = strTok.nextToken();
98              edgeWeightType = tempStr;
99              if(edgeWeightType.equals("EXPLICIT")) {
100                 return buildDistMatrixEXPLICIT(
101                     new StringTokenizer(workStr, "␣\n\t\r\f"));
102             }
103             else
104             {
105                 return buildDistMatrixEUC_2D(
106                     new StringTokenizer(workStr, "␣\n\t\r\f"));
107             }
108         }
109
110     }
111 }
112
113 /**
114  *Konstruerer afstandsmatricen udfra euklidiske koordinater.
115  *@param strTok En StringTokenizer over indholdet i problemfilen.
116  *@return Omkostningsmatricen for det valgte problem.
117  */
118 private static int [][] buildDistMatrixEUC_2D(StringTokenizer strTok) {
119     final int X = 0;
```

```
120         final int Y = 1;
121         int tempMatrix[][] = new int[n][n];
122         String tempStr = "";
123         float x,y = 0.0f;
124         int counter = 0;
125         float coords[][] = new float[n][2];
126         int dist = 0;
127         while(true) {
128             tempStr = strTok.nextToken();
129             if( tempStr.equals("NODE_COORD_SECTION")) {
130 
131                 while(!strTok.nextToken().equals("EOF")) {
132                     coords[counter][X] = Float.parseFloat(
133                         strTok.nextToken());
134                     coords[counter][Y] = Float.parseFloat(
135                         strTok.nextToken());
136                     counter++;
137                 }
138                 break;
139             }
140         }
141         for(int j = 0; j < coords.length; j++) {
142             for(int i = j; i < coords.length; i++) {
143                 float xd, yd;
144                 xd = coords[i][X] - coords[j][X];
145                 yd = coords[i][Y] - coords[j][Y];
146 
147                 dist = (int)(0.5+Math.sqrt( xd * xd + yd * yd ));
148 
149                 tempMatrix[i][j] = dist;
150                 tempMatrix[j][i] = dist;
151             }
152         }
153         return tempMatrix;
154     }
155 
156     /**
157      *Konstruerer afstandsmatricen udfra explicit givne koordinater.
158      *@param strTok En StringTokenizer over indholdet i problemfilen.
159      *@return Omkostningsmatricen for det valgte problem.
160      */
161     private static int [][] buildDistMatrixEXPLICIT(StringTokenizer strTok) {
162         int tempMatrix[][] = new int[n][n];
163         int countI = 0;
164         int countJ = 0;
165         String tempStr = "";
166 
167         while(true) {
168             tempStr = strTok.nextToken();
169             if( tempStr.equals("EDGE_WEIGHT_SECTION")) {
170                 while(true) {
171                     tempStr = strTok.nextToken();
172                     if(tempStr.equals("EOF")) {
173                         return tempMatrix;
174                     }
175                     if(tempStr.equals("0")) {
176                         tempMatrix[countI][countJ] =
177                             Integer.parseInt(tempStr);
178                         tempMatrix[countJ][countI] =
179                             Integer.parseInt(tempStr);
180                         countI++;
181                         countJ=0;
```

```
182                     }
183                     else
184                     {
185                         tempMatrix[countI][countJ] =
186                             Integer.parseInt(tempStr);
187                         tempMatrix[countJ][countI] =
188                             Integer.parseInt(tempStr);
189                         countJ++;
190                     }
191                 }
192             }
193         }
194     }
195
196 }
```

C Køretider

Nedenfor findes en tabel (table C.1) indeholdende de gennemsnitlige køretider samt standard afvigelserne for hvert af de i rapporten behandlede otte TSP-tilfælde. Standard afvigelsen er givet ved

$$s = \sqrt{\frac{\sum(x - \bar{x})^2}{n - 1}}, \tag{C.1}$$

hvor x er den målte værdi, $\bar{x}$ er middelværdien (gennemsnittet) og n er antallet af observationer.

Tabellen er arrangeret på følgende vis: 1. søjle er navnet på problemet. 2. søjle er antallet af iterationer kørt på problemet 3. søjle er den gennemsnitlige køretid for det sekventielle program. De næste søjler er de gennemsnitlige køretider for det distribuerede program på p processorer. For hvert problemtilfælde er først angivet den gennemsnitlige køretid over 10 kørsler, dernæst standard afvigelsen.

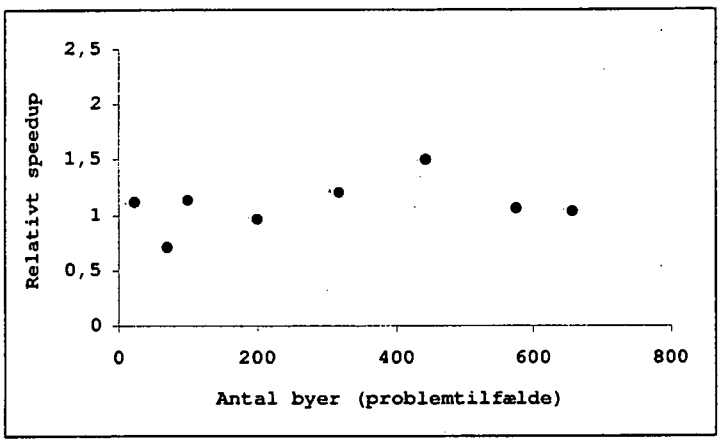
Alle vores køretider er målt ved at måle tiden på systemets hardware når programmet starter og når programmet stopper, og derefter beregne det tidsinterval programmet har kørt over. Samtlige tider er opgivet i sekunder og rundet af til to decimaler. Begrundelsen for det varierende antal af iterationer findes i afsnit 8.2.2.

problem	# iter.	sekv.	$p = 1$	$p = 2$	$p = 4$	$p = 6$	$p = 8$
gr24	1000	0,33	14,59	13,03	12,94	14,61	16,44
		0,02	3,03	1,03	0,23	0,62	0,98
st70	500	0,56	26,13	36,85	17,74	18,12	30,03
		0,17	1,11	2,30	0,04	0,09	1,79
kroA100	500	0,73	40,75	35,66	35,39	35,35	62,88
		0,01	0,32	0,02	0,03	0,01	2,30
kroA200	250	0,83	138,36	142,92	69,53	69,51	132,01
		0,01	8,48	5,71	0,02	0,02	6,71
lin318	100	0,65	84,91	70,00	69,60	129,24	136,54
		0,02	1,12	0,06	0,11	6,65	5,74
pcb442	50	0,83	100,55	66,64	66,28	120,70	130,40
		0,02	10,95	0,03	0,05	4,49	2,78
rat575	30	0,90	145,66	136,53	66,38	128,85	66,38
		0,03	7,81	3,07	0,06	4,73	0,17
d657	20	0,92	59,59	57,15	56,79	105,34	56,14
		0,03	0,23	0,26	0,05	3,52	0,26

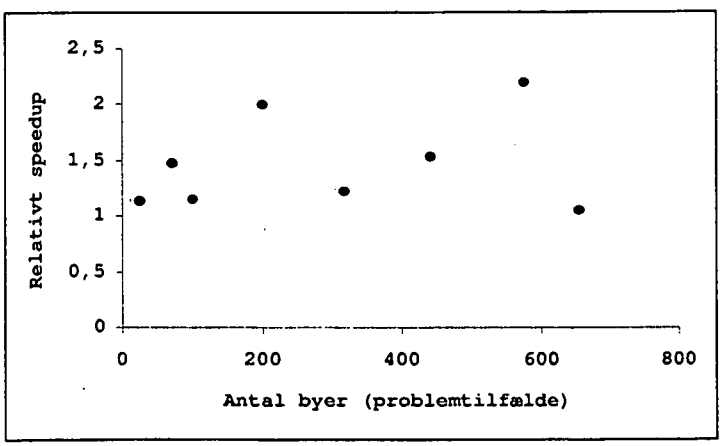
Tabel C.1 Gennemsnitlige køretider og standard afvigelse for de otte TSP-tilfælde kørt med deres respektive antal af iterationer.

D Grafer

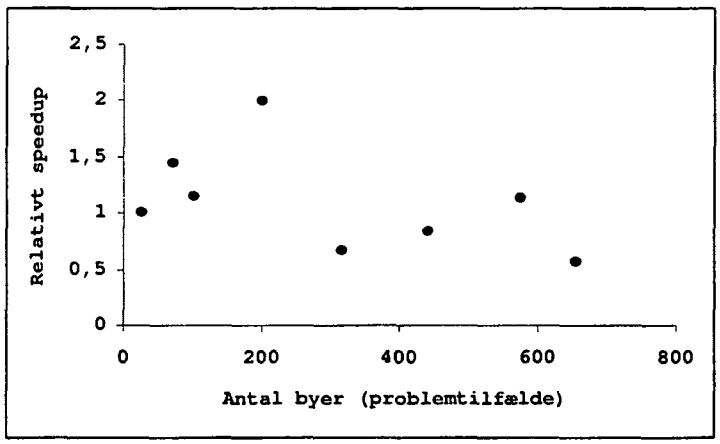
D.1 Relativt speedup som funktion af n



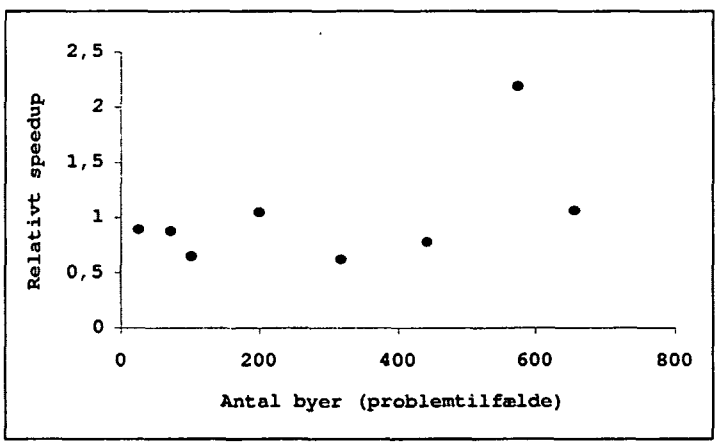
Figur D.1 Det gennemsnitlige relative speedup som funktion af antallet af byer i problemtilfældet for $p = 2$.



Figur D.2 Det gennemsnitlige relative speedup som funktion af antallet af byer i problemtilfældet for $p = 4$.

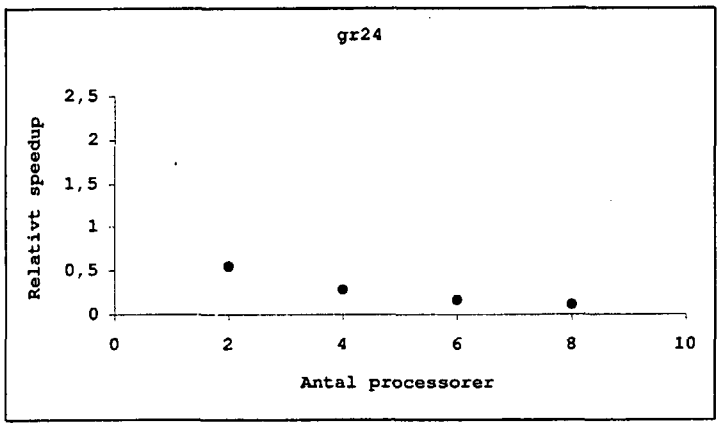


Figur D.3 Det gennemsnitlige relative speedup som funktion af antallet af byer i problemtilfældet for $p = 6$.

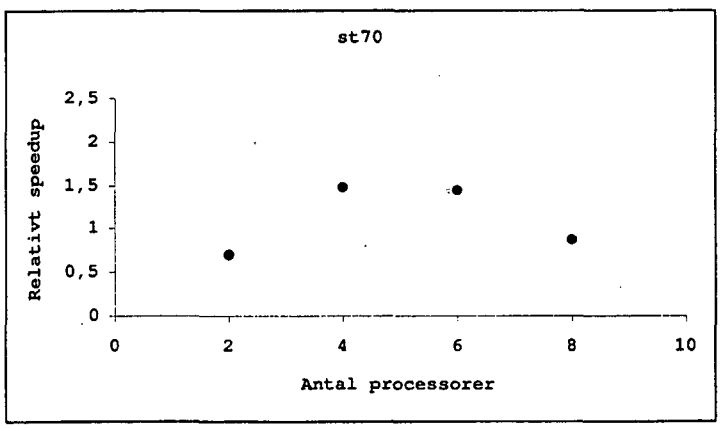


Figur D.4 Det gennemsnitlige relative speedup som funktion af antallet af byer i problemtilfældet for $p = 8$.

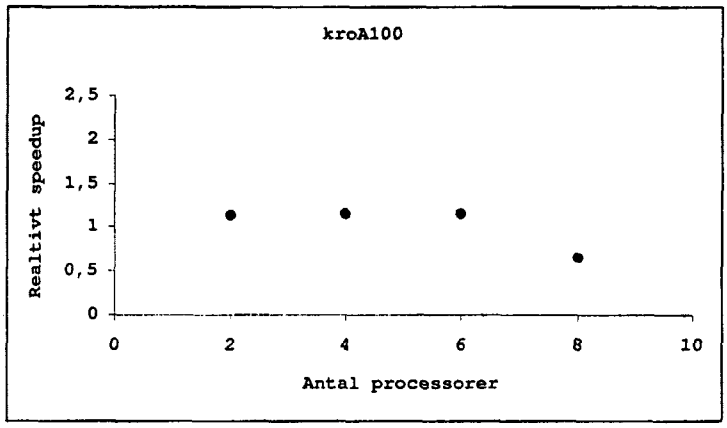
D.2 Relativt speedup som funktion af p



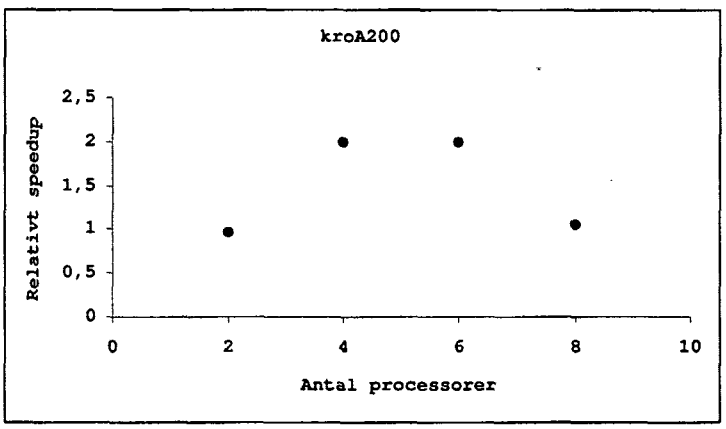
Figur D.5 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet gr24.



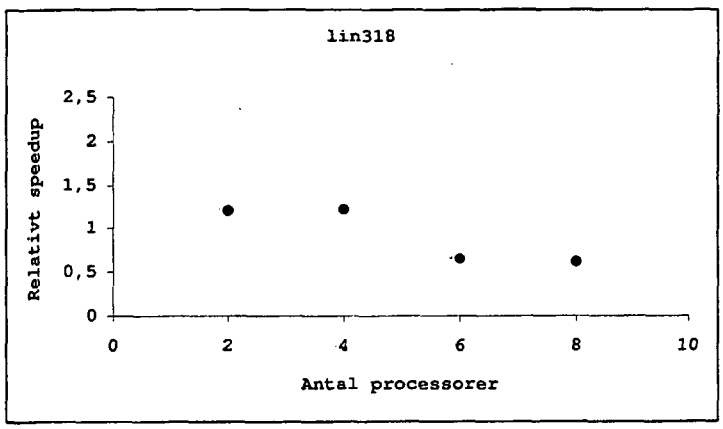
Figur D.6 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet st70.



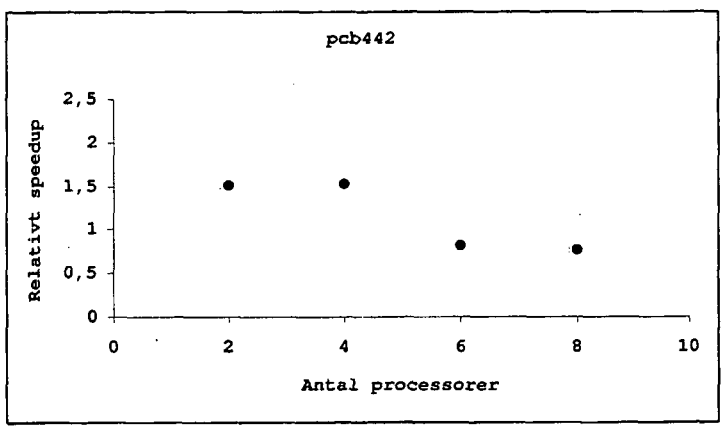
Figur D.7 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet kroA100.



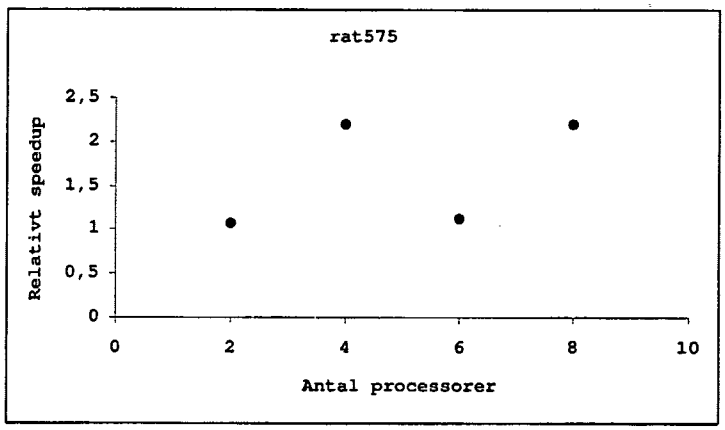
Figur D.8 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet kroA200.



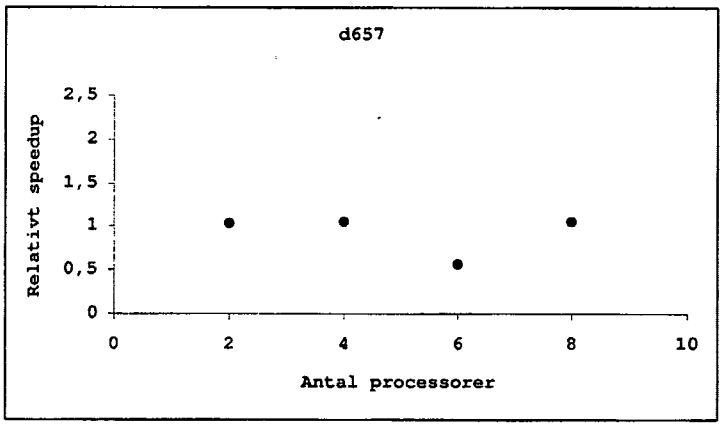
Figur D.9 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet lin318.



Figur D.10 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet pcb442.



Figur D.11 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet rat575.



Figur D.12 Det gennemsnitlige relative speedup som funktion af antallet af processorer. Her for TSP-tilfældet d657.

Litteratur

S. G. Aki. *The Design and Analysis of Parallel Algorithms.* Prentice-Hall, Inc., 1989.

D. H. Backchi, R. B. Godiksen, U. T. V. Jankvist, J. M. Poulsen, and N. Saglanmak. real life routing – en strategi for et virkeligt vrp. *IMFUFA Tekster*, (405), 2001.

R. S. Barr and B. L. Hickman. Reporting Computaional Experiments with Parallel Algorithms: Issues, Messures, and Experts' Opinions. *ORSA Journal of Computing*, 2(18), 1993.

N. L. Biggs. *Discrete Mathematics.* Oxford Science Publications, 1989.

E. Bonabeau, M. Dorigo, and G. Theraulaz. *Swarm Intelligence – From Natural to Artificial Systems.* Oxford University Press, 1999.

M. Bundgaard, T. C. Damgaard, F. Decara, and J. W. Winther. Ant Routing System – a routing algorithm based on ant algorithms applied to a simulated network. IT University of Copenhagen – Internet Technology. 2002.

A. Dolan and J. Aldous. *Networks and Algorithms.* John Wiley & Sons Ltd., 1995.

M. Dorigo and L. M. Gambadella. Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. *IEEE Trans. Evolut. Comput.*, 1(1), 1997.

J. Farley. *Java – Distributed Computing.* O'Reilly & Associates, Inc., 1998.

K. Helsgaun. An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operational Reasearch*, 126, 2000.

K. Helsgaun. Den rejsende sælgers problem. *Forelæsningsslides*, 2003a.

K. Helsgaun. Paralelle algoritmer. *Forelæsningsslides*, 2003b.

A. H. Karp and H. P. Flatt. Measuring Parallel Processor Performance. *Communications of the ACM*, 33(5), May 1990.

E. Lawler, J. Lenstra, A. R. Kan, and D. Shmoys. *The Traveling Salesman Problem.* Wiley-interscience Series in Discrete mathematics. John Wiley & Sons, New York, 1985.

P. T. Metaxas. Thinking and Programming in Parallel – CS331 Class Notes. *Forelæsningsnoter*, 1997.

E. Minieka. *Optimization Algorithms for Networks and Graphs.* Marcel Dekker, Inc., New York, 1978.

E. D. Nielsen, J. Danielsen, and N. B. Johansen. Parallelle alogritmer. *IMFUFA Tekster*, (282), 1994.

M. Randall and A. Lewis. A Parallel Implementation of Ant Colony Optimization. *Journal of Parallel and Distributed Computing*, 62, 2002.

G. Reinelt. *The Traveling Salesman.* Lectures in Computer Notes. Springer-Verlag, Heildelberg, 1994.

A. S. Tanenbaum. *Structured Computer Organization, Third ed.* Prentice-Hall, Inc., 1990.

Liste over tidligere udsendte tekster kan ses på IMFUFA's hjemmeside: http://mmf.ruc.dk eller rekvireres på sekretariatet, tlf. 46 74 22 63 eller e-mail: imfufa@ruc.dk.

332/97 ANOMAL SWELLING AF LIPIDE DOBBELTLAG
Specialerappoprt af: Stine Korremann
Vejleder: Dorthe Posselt

333/97 Biodiversity Matters
an extension of methods found in the literature on monetisation of biodiversity
by: Bernd Kuemmel

334/97 LIFE-CYCLE ANALYSIS OF THE TOTAL DANISH ENERGY SYSTEM
by: Bernd Kuemmel and Bent Sørensen

335/97 Dynamics of Amorphous Solids and Viscous Liquids
by: Jeppe C. Dyre

336/97 Problem-orientated Group Project Work at Roskilde University
by: Kathrine Legge

337/97 Verdensbankens globale befolkningsprognose
- et projekt om matematisk modellering
af: Jørn Chr. Bendtsen, Kurt Jensen, Per Pauli Petersen

338/97 Kvantisering af nanolederes elektriske ledningsevne
Første modul fysikprojekt
af: Søren Dam, Esben Danielsen, Martin Niss,
Esben Friis Pedersen, Frederik Resen Steenstrup
Vejleder: Tage Christensen

339/97 Defining Discipline
by: Wolfgang Coy

340/97 Prime ends revisited - a geometric point of view -
by: Carsten Lunde Petersen

341/97 Two chapters on the teaching, learning and assessment of geometry
by: Mogens Niss

342/97 A global clean fossil scenario DISCUSSION PAPER prepared by Bernd Kuemmel for the project LONG-TERM SCENARIOS FOR GLOBAL ENERGY DEMAND AND SUPPLY

343/97 IMPORT/EKSPORT-POLITIK SOM REDSKAB TIL OPTIMERET UDNYTTELSE AF EL PRODUCERET PÅ VE-ANLÆG
af: Peter Meibom, Torben Svendsen, Bent Sørensen

344/97 Puzzles and Siegel disks
by: Carsten Lunde-Petersen

345/98 Modeling the Arterial System with Reference to an Anesthesia Simulator
Ph.D. Thesis
by: Mette Sofie Olufsen

346/98 Klyngedannelse i en hulkatode-forstøvningsproces
af: Sebastian Horst
Vejledere: Jørn Borggren, NBI, Niels Boye Olsen

347/98 Verificering af Matematiske Modeller
- en analyse af Den Danske Eulerske Model
af: Jonas Blomqvist, Tom Pedersen, Karen Timmermann, Lisbet Øhlenschlæger
Vejleder: Bernhelm Booss-Bavnbek

348/98 Case study of the environmental permission procedure and the environmental impact assessment for power plants in Denmark
by: Stefan Krüger Nielsen
project leader: Bent Sørensen

349/98 Tre rapporter fra FAGMAT - et projekt om tal og faglig matematik i arbejdsmarkedsuddannelserne
af: Lena Lindenskov og Tine Wedege

350/98 OPGAVESAMLING - Bredde-Kursus i Fysik 1976 - 1998
Erstatter teksterne 3/78, 261/93 og 322/96

351/98 Aspects of the Nature and State of Research in Mathematics Education
by: Mogens Niss

352/98 The Herman-Swiatec Theorem with applications
by: Carsten Lunde Petersen

353/98 Problemløsning og modellering i en almendannende matematikundervisning
Specialerapport af: Per Gregersen og Tomas Højgaard Jensen

354/98 A Global Renewable Energy Scenario
by: Bent Sørensen and Peter Meibom

355/98 Convergence of rational rays in parameter spaces
by: Carsten Lunde Petersen and Gustav Ryd

356/98 Terrænmodellering
Analyse af en matematisk model til konstruktion af digitale terrænmodeller
Modelprojekt af: Thomas Frommelt, Hans Ravnkjær Larsen og Arnold Skimminge
Vejleder: Johnny Ottesen

357/98 Cayleys Problem
En historisk analyse af arbejdet med Cayleys problem fra 1870 til 1918
Et matematisk videnskabsfagsprojekt af: Rikke Degn, Bo Jakobsen, Bjarke K.W. Hansen, Jesper S. Hansen, Jesper Udesen, Peter C. Wulff
Vejleder: Jesper Larsen

358/98 Modeling of Feedback Mechanisms which Control the Heart Function in a View to an Implementation in Cardiovascular Models
Ph.D. Thesis by: Michael Danielsen

359/99 Long-Term Scenarios for Global Energy Demand and Supply
Four Global Greenhouse Mitigation Scenarios
by: Bent Sørensen (with contribution from Bernd Kuemmel and Peter Meibom)

360/99 SYMMETRI I FYSIK
En Meta-projektrapport af: Martin Niss, Bo Jakobsen & Tune Bjarke Bonné
Vejleder: Peder Voetmann Christiansen

361/99 Symplectic Functional Analysis and Spectral Invariants
by: Bernhelm Booß-Bavnbek, Kenro Furutani

362/99 Er matematik en naturvidenskab? - en udspænding af diskussionen
En videnskabsfagsprojekt-rapport af: Martin Niss
Vejleder: Mogens Nørgaard Olesen

363/99 EMERGENCE AND DOWNWARD CAUSATION
by: Donald T. Campell, Mark H. Bickhard, and Peder V. Christiansen

364/99 Illustrationens kraft - Visuel formidling af fysik
Integreret speciale i fysik og kommunikation
af Sebastian Horst
Vejledere: Karin Beyer, Søren Kjørup

365/99 To know - or not to know - mathematics, that is a question of context
by: Tine Wedege

366/99 LATEX FOR FORFATTERE - En introduktion til LATEX
og IMFUFA-LATEX
af: Jørgen Larsen

367/99 Boundary Reduction of Spectral Invariants and Unique Continuation Property
by: Bernhelm Booss-Bavnbek

368/99 Kvartvejsraport for projektet SCENARIER FOR SAMLET UDNYTTELSE AF BRINT SOM ENERGIBÆRER I DANMARKS FREMTIDIGE ENERGISYSTEM
Projektleder: Bent Sørensen

369/99 Dynamics of Complex Quadratic Correspondences
by: Jacob S. Jalving
Supervisor: Carsten Lunde Petersen

370/99 OPGAVESAMLING - Bredde-Kursus i Fysik 1976 - 1999
Eksamensopgaver fra perioden 1976 - 1999. Denne tekst erstatter tekst nr. 350/98

371/99 Bevisets stilling - beviser og bevisførelse i en gymnasial matematik undervisning
Et matematikspeciale af: Maria Hermannsson
Vejleder: Mogens Niss

372/99 En kontekstualiseret matematikhistorisk analyse af ikke-lineær programmering: Udviklingshistorie og multipel opdagelse
Ph.d.-afhandling af Tinne Hoff Kjeldsen

373/99 Criss-Cross Reduction of the Maslov Index and a Proof of the Yoshida-Nicolaescu Theorem
by: Bernhelm Booss-Bavnbek, Kenro Furutani and Nobukazu Otsuki

374/99 Det hydrauliske spring - Et eksperimentelt studie af polygoner og hastighedsprofiler
Specialeafhandling af: Anders Marcussen
Vejledere: Tomas Bohr, Clive Ellegaard, Bent C. Jørgensen

375/99 Begrundelser for Matematikundervisningen i den lærde skole hhv. gymnasiet 1884-1914
Historiespeciale af Henrik Andreassen, cand.mag. i Historie og Matematik

376/99 Universality of AC conduction in disordered solids
by: Jeppe C. Dyre, Thomas B. Schrøder

377/99 The Kuhn-Tucker Theorem in Nonlinear Programming: A Multiple Discovery?
by: Tinne Hoff Kjeldsen

378/00 Solar energy preprints:
1. Renewable energy sources and termal energy storage
2. Integration of photovoltaic cells into the global energy system
by: Bent Sørensen

379/00 EULERS DIFFERENTIALREGNING
Eulers indførelse af differentialregningen stillet over for den moderne
En tredjesemesters projektrapport på den naturvidenskabelige basisuddannelse
af: Uffe Thomas Volmer Jankvist, Rie Rose Møller Pedersen, Maja Bagge Pedersen
Vejleder: Jørgen Larsen

380/00 MATEMATISK MODELLERING AF HJERTEFUNKTIONEN
Isovolumetrisk ventrikulær kontraktion og udpumpning til det cardiovaskulære system
af: Gitte Andersen (3.moduls-rapport), Jakob Hilmer og Stine Weisbjerg (speciale)
Vejleder: Johnny Ottesen

381/00 Matematikviden og teknologiske kompetencer hos kortuddannede voksne
- Rekognosceringer og konstruktioner i grænselandet mellem matematikkens didaktik og forskning i voksenuddannelse
Ph. d.-afhandling af Tine Wedege

382/00 Den selvundvigende vandring
Et matematisk professionsprojekt
af: Martin Niss, Arnold Skimminge
Vejledere: Viggo Andreasen, John Villumsen

383/00 Beviser i matematik
af: Anne K.S.Jensen, Gitte M. Jensen, Jesper Thrane, Karen L.A.W. Wille, Peter Wulff
Vejleder: Mogens Niss

384/00 Hopping in Disordered Media: A Model Glass Former and A Hopping Model
Ph.D. thesis by: Thomas B. Schrøder
Supervisor: Jeppe C. Dyre

385/00 The Geometry of Cauchy Data Spaces
This report is dedicated to the memory of Jean Leray (1906-1998)
by: B. Booss-Bavnbek, K. Furutani, K. P. Wojciechowski

386/00 Neutrale mandatfordelingsmetoder – en illusion?
af: Hans Henrik Brok-Kristensen, Knud Dyrberg, Tove Oxager, Jens Sveistrup
Vejleder: Bernhelm Booss-Bavnbek

387/00 A History of the Minimax Theorem: von Neumann's Conception of the Minimax Theorem - - a Journey Through Different Mathematical Contexts
by: Tinne Hoff Kjeldsen

388/00 Behandling af impuls ved kilder og dræn i C. S. Peskins 2D-hjertemodel
et 2. moduls matematik modelprojekt
af: Bo Jakobsen, Kristine Niss
Vejleder: Jesper Larsen

389/00 University mathematics based on problemoriented student projects: 25 years of experience with the Roskilde model
By: Mogens Niss
Do not ask what mathematics can do for modelling. Ask what modelling can do for mathematics!
by: Johnny Ottesen

390/01 SCENARIER FOR SAMLET UDNYTTELSE AF BRINT SOM ENERGIBÆRER I DANMARKS FREMTIDIGE ENERGISYSTEM Slutrapport, april 2001
Projektleder: Bent Sørensen
Projektdeltagere: DONG: Aksel Hauge Petersen, Celia Juhl, Elkraft System[#]: Thomas Engberg Pedersen[##], Hans Ravn, Charlotte Søndergren, Energi 2[#]: Peter Simonsen, RISØ Systemanalyseafd.: Kaj Jørgensen[*], Lars Henrik Nielsen, Helge V. Larsen, Poul Erik Morthorst, Lotte Schleisner, RUC: Finn Sørensen[**], Bent Sørensen
[#]Indtil 1/1-2000 Elkraft, [##] fra 1/5-2000 Cowi Consult
[*] Indtil 15/6-1999 DTU Bygninger & Energi, [**] fra 1/1-2001 Polypeptide Labs.
Projekt 1763/99-0001 under Energistyrelsens Brintprogram

391/01 Matematisk modelleringskompetence – et undervisningsforløb i gymnasiet
3. semesters Nat.Bas. projekt af: Jess Tolstrup Boye, Morten Bjørn-Mortensen, Sofie Inari Castella, Jan Lauridsen, Maria Gøtzsche, Ditte Mandøe Andreasen
Vejleder: Johnny Ottesen

392/01 "PHYSICS REVEALED" THE METHODS AND SUBJECT MATTER OF PHYSICS
an introduction to pedestrians (but not excluding cyclists)
PART III: PHYSICS IN PHILOSOPHICAL CONTEXT
by: Bent Sørensen.

393/01 Hilberts matematikfilosofi
Specialerapport af: Jesper Hasmark Andersen
Vejleder: Stig Andur Pedersen

394/01 "PHYSICS REVEALED" THE METHODS AND SUBJECT MATTER OF PHYSICS
an introduction to pedestrians (but not excluding cyclists)
PART II: PHYSICS PROPER
by: Bent Sørensen.

395/01 Menneskers forhold til matematik. Det har sine årsager!
Specialeafhandling af: Anita Stark, Agnete K. Ravnborg
Vejleder: Tine Wedege

396/01 2 bilag til tekst nr. 395: Menneskers forhold til matematik. Det har sine årsager!
Specialeafhandling af: Anita Stark, Agnete K. Ravnborg
Vejleder: Tine Wedege

397/01 En undersøgelse af solvents og kædelængdes betydning for anomal swelling i phospholipiddobbeltlag
2. modul fysikrapport af: Kristine Niss, Arnold Skimminge, Esben Thormann, Stine Timmermann
Vejleder: Dorthe Posselt

398/01 Kursusmateriale til "Lineære strukturer fra algebra og analyse" (E1)
Af: Mogens Brun Heefelt

399/01 Undergraduate Learning Difficulties and Mathematical Reasoning
Ph.D Thesis by: Johan Lithner
Supervisor: Mogens Niss

400/01 On Holomorphic Critical quasi circle maps
By: Carsten Lunde Petersen

401/01 Finite Type Arithmetic
Computable Existence Analysed by Modified Realisability and Functional Interpretation
Master's Thesis by: Klaus Frovin Jørgensen
Supervisors: Ulrich Kohlenbach, Stig Andur Pedersen and Anders Madsen

402/01 Matematisk modellering ved den naturvidenskabelige basisuddannelse
- udvikling af et kursus
Af: Morten Blomhøj, Tomas Højgaard Jensen, Tinne Hoff Kjeldsen og Johnny Ottesen

403/01 Generaliseringer i integralteorien
- En undersøgelse af Lebesgue-integralet, Radon-integralet og Perron-integralet
Et 2. modul matematikprojekt udarbejdet af: Stine Timmermann og Eva Uhre
Vejledere: Bernhelm Booss-Bavnbek og Tinne Hoff Kjeldsen

404/01 "Mere spredt fægtning"
Af: Jens Højgaard Jensen

405/01 Real life routing
- en strategi for et virkeligt vrp
Et matematisk modelprojekt af: David Heiberg Backchi, Rasmus Brauner Godiksen, Uffe Thomas Volmer Jankvist, Jogvan Martin Poulsen og Neslihan Saglanmak
Vejleder: Jørgen Larsen

406/01 Opgavesamling til dybdekursus i fysik
Eksamensopgaver stillet i perioden juni 1976 til juni 2001
Denne tekst erstatter tekst nr. 25/1980 + efterfølgende tillæg

407/01 Unbounded Fredholm Operators and Spectral Flow
By: Bernhelm Booss-Bavnbek, Matthias Lesch, John Phillips

408/02 Weak UCP and Perturbed Monopole Equations
By: Bernhelm Booss-Bavnbek, Matilde Marcolli, Bai-Ling Wang

409/02 Algebraisk ligningsløsning fra Cardano til Cauchy
- et studie af kombinationers, permutationers samt invariansbegrebets betydning for den algebraiske ligningsløsning *før* Gauss, Abel og Galois
Videnskabsfagsprojekt af: David Heiberg Backchi, Uffe thomas Volmer Jankvist, Neslihan Saglanmak
Vejleder: Bernhelm Booss-Bavnbek

410/02 2 projekter om modellering af influenzaepidemier
Influenzaepidemier- et matematisk modelleringsprojekt
Af: Claus Jørgensen, Christina Lohfert, Martin Mikkelsen, Anne-Louise H. Nielsen
Vejleder: Morten Blomhøj
Influenza A: Den tilbagevendende plage – et modelleringsprojekt
Af: Beth Paludan Carlsen, Christian Dahmcke, Lena Petersen, Michael Wagner
Vejleder: Morten Blomhøj

411/02 Polygonformede hydrauliske spring
Et modelleringsprojekt af: Kåre Stokvad Hansen, Ditte Jørgensen, Johan Rønby Pedersen, Bjørn Toldbod
Vejleder: Jesper Larsen

412/02 Hopfbifurkation og topologi i væskestrømning – en generel analyse samt en behandling af strømningen bag en cylinder
Et matematisk modul III professionsprojekt af. Kristine Niss, Bo Jakobsen
Vejledere: Morten Brøns, Johnny Ottesen

413/03 "Elevernes stemmer" Fysikfaget, undervisningen og lærerroller, som eleverne opfatter det i det almene gymnasium i Danmark
Af: Carl Angell, Albert Chr. Paulsen

414/03 Feltliniediagrammer En vej til forståelse?
Et 1. modul fysikprojekt af: Ditte Gundermann, Kåre Stokvad Hansen, Ulf Rørbæk Pedersen
Vejleder: Tage Emil Christensen

415/03 FYSIKFAGET I FORANDRING Læring og undervisning i fysik i gymnasiet med fokus på dialogiske processer, autenticitet og kompetenceudvikling
Ph.d.-afhandling i fysikdidaktik af: Jens Dolin

416/03 Fourier og Funktionsbegrebet
- Overgangen fra Eulers til Dirichlets funktionsbegreb
Projektrapport af: Rasmus Brauner Godiksen, Claus Jørgensen, Tony Moyer Hanberg, Bjørn Toldbod
Vejleder: Erik von Essen

417/03 The Semiotic Flora of Elementary particles
By: Peder Voetmann Christiansen

418/03 Militærmatematik set med kompetencebriller
3. modul projektrapport af: Gitte Jensen og specialerapport af: Jesper Thrane
Vejleder: Tine Wedege

419/03 Energy Bond Graphs – a semiotic formalization of modern physics
By: Peder Voetmann Christiansen

420/03 Stemning og Musikalsk Konsonans
Et matematisk modelleringsprojekt af: Claus Jørgensen
Vejleder: Johnny Ottesen

421/03 OPGAVESAMLING
Bredde-kursus i fysik 1976 – 2003.
Denne tekst erstatter tekst nr. 370/99

422/03 Vurdering af dynamisk blodstrømningsmodel
- ved numerisk simulering med FEMLAB
Et 2. modul matematikprojekt af: Sofie Inari Castella, Ingunn Gunnarsdóttir og Jacob Kirkensgaard Hansen
Vejleder: Johnny Ottesen

423/03 Fysikkens historie i en almendannende fysikundervisning
- Eksemplificeret med Millikan Ehrenhaft kontroversen
Specialerapport af: Marianne Wilcken Bjerregaard
Vejleder: Albert Chr. Paulsen

424/03 Dielectric and Shear Mechanical Relaxation in Glass Forming Liquids
- A thorough analysis and experimental test of the DiMarzio-Bishop model
Master thesis in physics by: Kristine Niss and Bo Jakobsen
Supervised by: Niels Boye Olsen

425/03 Fysiske forklaringer i undervisning
Specialerapport af: Kirsten Ringgard Jensen
Vejleder: Jens Højgaard Jensen