Sikring af kommunikationsnetværk ved brug af Single Backup Path Protection

02125 - Bachelorprojekt i Softwareteknologi

Udarbejdet af

(s042143) Christopher Følsgaard
(s022015) Anders Spælling

Vejleder

Thomas Stidsen

Kongens Lyngby, 6. Juli 2007
IMM, Danmarks Tekniske Universitet

Forord

Dette bachelor projekt er udarbejdet ved Institut for Informatik og Matematisk Modellering ved Danmarks tekniske Universitet DTU i forbindelse med erhvervelsen af bachelor graden på civilingeniøruddannelsens Softwareteknologi retning.

Opgaven omhandler simulering af kommunikationsnetværk samt optimering af trafikfordelingen for at gøre den samlede belastning så lille så mulig. Der er hovedsageligt lagt vægt på udvikling af hurtige algoritmer til at route trafikken effektivt gennem forskellige netværk.

Det forudsættes at læseren er bekendt med grund begreber indenfor algoritmer og datastrukturer samt software udvikling.

Projektet består af denne rapport, et bilag med kildekode samt en eksekverbar version af programmet vedlagt på CD-ROM. Kildekoden og tilhørende Javadoc er desuden vedlagt på samme CD-ROM.

Projektet er udført i perioden 1. April 2007 til 6. juli 2007.

Kongens Lyngby, 6. Juli 2007

Christopher Følsgaard & Anders Spælling

Indhold

Kapitel 1

Introduktion

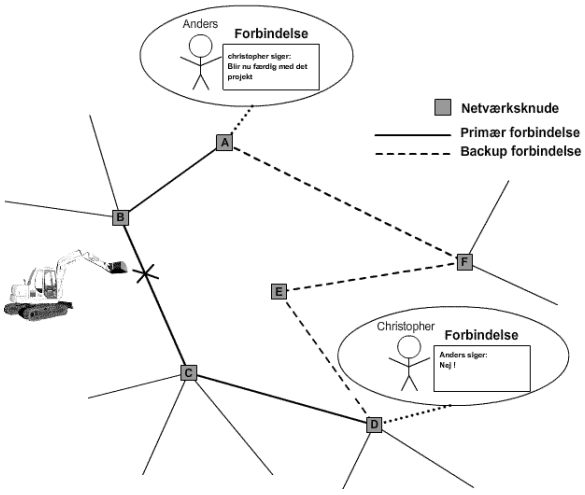
Dette bachelorprojekt er opstået på baggrund af forskning indenfor Operationsanalyse udført ved Institut for Informatik og Matematisk Modellering vedrørende minimering af den samlede belastning på et kommunikationsnetværk.
Formålet med dette bachelorprojekt er at udvikle en applikation der for et givent kommunikationsnetværk kan beregne, hvordan trafikken skal routes sikkert således at den samlede belastning på netværket bliver så lille som mulig.

1.1 Indledning

I dag ses en stadig stigende afhængighed af kommunikationsnetværk. Dette betyder et øget forbrug og dermed en større belastning. Tænker man på omkostningerne for nedgravning af nye kabler samt køb af netværksudstyr der kan håndtere de trafik mængder vi i dag sender på tværs af hele verden, er behovet for at kunne udnytte de eksisterende netværk optimalt i stigende grad vigtig. Optimeringen sker ved at route trafikken gennem netværket således at belastningen spredes over flere links. I sidste ende giver dette en bedre udnyttelse af netværket og der frigøres endvidere plads til yderligere trafik.

Ligesom udnyttelsen er pålideligheden af forbindelserne mindst lige så vigtig. De fleste med en internetforbindelse har oplevet at forbindelsen til deres internet svigter for eksempel på grund af kabelfejl. Det sker at en gravemaskine ved en fejl gnasker sig igennem et af udbyderens kabler. Dette bemærker de fleste dog ikke. Dette skyldes, at når man i dag opretter en forbindelse sendes data

via 2 forbindelser (en primær og en backup) på samme tid. Sagt med andre ord, man går med livrem og seler så data sendes dobbelt for at sikre at de kommer frem. Dette vil belaste netværket unødigt, da det trods alt ikke er altid at forbindelserne svigter.
For at minimere belastningen er det smart kun at koble en backup forbindelse på hvis den primære svigter. Denne strategi kaldes i fagsprog for "Single Backup Path Protection". Et eksempel på dette ses nedenfor.



Hver gang der oprettes forbindelse på et kommunikationsnetværk sendes i princippet en forespørgsel (herefter kaldet demand) til udbyderen om at få etableret en forbindelse til den ønskede server. Udbyderen tilføjer nu denne demand og der reserveres plads på kommunikationsnetværket.

Lad os forestille os følgende scenarie. Et firma med en afdeling i København og en i Bruxelles ønsker at udveksle en stor bunke data. Afdelingen sender en demand til udbyderen og der reserveres plads.
I resten af landet foretages der samtidig massere af demands, som udbyderen også tilføjer. Det kan være alt fra folk ønsker at downloade film, til folk der blot ønsker at chatte. Alle disse operationer foregår automatisk.
I et samfund som vores hvor stort set alle husstande har en internetforbindelse, og en stor del af handel og kommunikation foregår via netværk, kan antallet af demands hurtigt eksplodere. Dette kan vi selvfølgelig ikke gøre noget ved.
Istedet kan vi, når en demand modtages, forsøge at vælge en rute gennem netværket (herefter path), som mindsker belastningen mest muligt.

1.2 Problemformulering

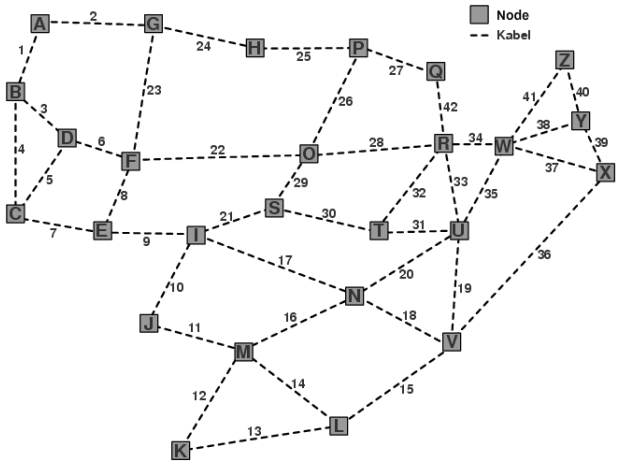
I denne rapport er vores opgave at undersøge, hvordan trafikken i et kommunikationsnetværk routes bedst muligt. Målet er at udvikle en applikation med en grafisk brugergrænseflade, der ved indlæsning af ekstern data kan simulere dette. I applikationen vil vi ved brug af heuristisk optimering finde løsninger til, hvordan trafikken routes i et statisk netværk. Med statisk netværk menes at alle demands er givet på forhånd og skal fordeles i et afgrænset netværk.

Da dette problem tilhører klassen NP-komplette problemer (problemer der ikke kan løses i polynomiel tid), er det nødvendigt at benytte en anden tilgang til problemet. Vi antager at brugen af meta heuristikker er en måde at tilnærme sig løsningen til en givet teoretisk lowerbound for routingen af trafikken i et kommunikationsnetværk.
På baggrund af erfaringer gjort med meta heuristikker i kurset 02719 Optimization using metaheuristics, har vi valgt at bruge Simulated Annealing da den er let at implementere og yderligere returnerer bedre løsninger en en random generator. Derudover formoder vi at implementeringen af meta heuristikken Greedy Randomized Adaptive Search Procedure (herefter GRASP) vil give endnu bedre løsninger.

Vi vil, ved hjælp af simulering af trafikkens mulige fordelinger i et netværk, forsøge at påvise en hurtig og effektiv metode til at route trafikken, som derefter kan anvendes i praksis. For at kunne foretage denne simulering må man først kende strukturen på kommunikationsnetværket. For at effektivisere routingsalgoritmen til simulering er det hensigtsmæssigt at algoritmen er i stand til at kunne vælge mellem en række foretrukne sæt hvor hvert sæt består af en primær og en backup path (herefter kaldet demand paths). Disse findes mellem samtlige netværksknuder (herefter kaldet nodes) i netværket. Derudover skal man have adgang til samtlige demands.

1.2.1 Netværksstruktur

I det følgende gives et eksempel fra start til slut på hvordan demands tilføjes som forbindelser i et kommunikationsnetværk. Først gives strukturen på netværket, vist nedenfor.



1.2.2 Demands & Demand paths

En demand er en forespørgsel om at få oprettet en forbindelse mellem 2 nodes med en ønsket båndbredde. I tabellen nedenfor ses eksempler på demands.

Forbindelse	Båndbredde
A ↔ O	5 MBit
A ↔ O	8 MBit
K ↔ U	5 MBit
K ↔ U	3 MBit
K ↔ U	10 MBit

Til dette netværk følger en række demand paths mellem samtlige nodes. En demand path består af både en primær og en backup path. I netværk af denne størrelse kan antallet af disse ligge omkring 2500, men vi har her kun udvalgt et par stykker. Disse ses i tabellen nedenfor.

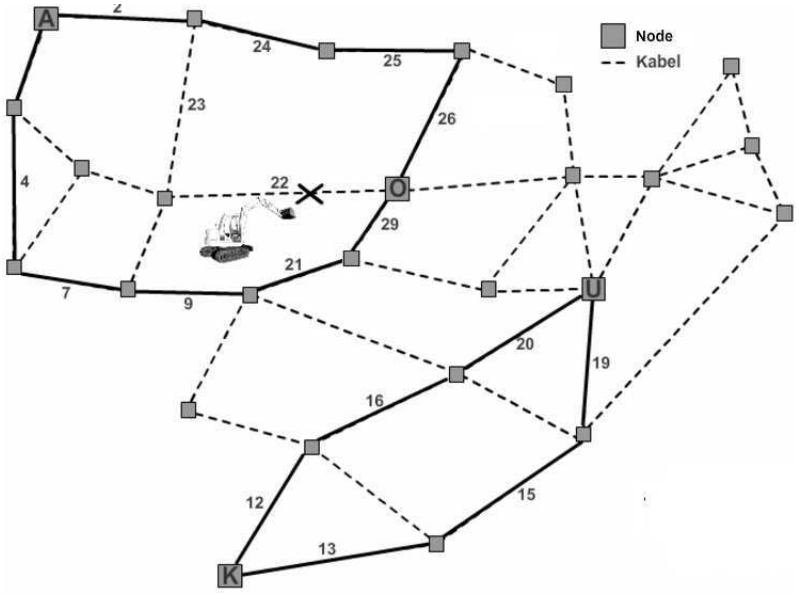
#	Start node	Slut node	Primær path	Backup path
1	A	O	P(2, 24, 25, 26)	B(1, 4, 7, 9, 21, 29)
2	A	O	P(2, 23, 22)	B(1, 4, 7, 9, 21, 29)
3	A	O	P(1, 3, 6, 22)	B(2, 24, 25, 26)
4	K	U	P(13, 15, 19)	B(12, 16, 20)
5	K	U	P(12, 16, 20)	B(13, 15, 19)

Som det ses kan primære paths også fremgå som backup paths og visa versa.

Når vi kender netværksstrukturen, forbindelsespunkterne for vores demands samt deres båndbredde, og vi har informationer om demand paths, kan vi fodre vores routingsalgoritme med alle disse oplysninger. I nedenstående tabel har routingsalgoritmen fundet den bedste fordeling for de givne demands.

Forbindelse	Valgt demand path	Båndbredde
A ↔ O	P(#1)	5 MBit
A ↔ O	B(#2)	8 MBit
K ↔ U	P(#4)	5 MBit
K ↔ U	P(#4)	3 MBit
K ↔ U	P(#5)	10 MBit

Denne fordeling giver følgende billede på netværket (de valgte forbindelser er markeret med fed).



For forbindelse mellem A og O på 8 mbit var der oprindeligt valgt den primære rute P(2, 23, 22), men fordi en gravemaskine afbrød forbindelsen et sted på link 22 anvendtes istedet backuppenbackup pathen.
Valget af demand paths vil i simuleringen blive evalueret som et udtryk for den samlede belastning på netværket når alle demands er tilføjet. Routingsalgoritmen vil herefter udskifte de valgte demand paths med andre demand paths og derefter reevaluere løsningen.

1.3 Kravspecifikation

I dette projekt skal der udvikles en prototype af et værktøj, der kan benyttes til simulering af kommunikationsnetværk samt beregne hvor godt trafikken kan routes gennem disse.

Problemstillingen til hvilken værktøjet skal bruges, giver anledning til 3 punkter som behandles i denne rapport.

- Simulering af kommunikationsnetværk
- Optimering af trafik routing gennem netværket
- Præsentationskomponent til visualisering af netværk og trafik fordeling.

For at opfylde disse punkter har vi opstillet en række krav til værktøjet.

- Programmet skal kunne indlæse et kommunikationsnetværk.
- Det skal være muligt at indlæse demands og demand paths til et eksisterende kommunikationsnetværk.
- Det skal være muligt at vælge mellem 2 forskellige algoritmer til at route trafikken.
- Programmet skal kunne afvikles på operativsystemerne Windows, Unix og Linux.
- Programmet skal kunne give en løsning på hvorledes trafikken routes bedst muligt.

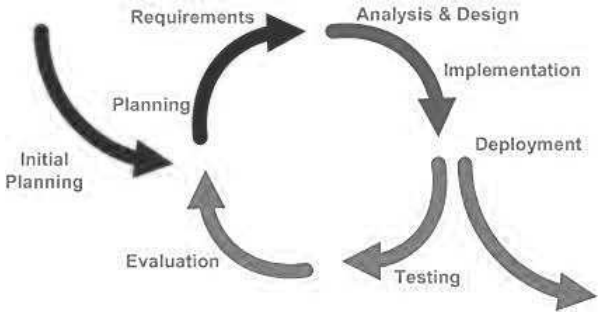
Kravene underbygges af en afgrænsning i forhold til lagring af data.

- Lagring af data i form af kommunikationsnetværk skal forekomme i filer placeret på harddisken i et forudbestemt fil format.
- Lagring af data i form af demand paths tilknyttet et indlæst netværk skal forekomme i filer placeret på harddisken i et forudbestemt fil format.
- Lagring af data i form af demands der ønskes routet gennem et netværk skal forekomme i filer placeret på harddisken i et forudbestemt fil format.

Da en del af problemstillingen omhandler et optimeringsproblem, stilles der store krav til valg af datastrukturer samt hastigheden af algoritmerne. Derfor er high-level programmeringssprog med dårlig performance indenfor basale aritmetiske operationer ikke at foretrække. Ydermere afgrænses valget af udviklingssprog i-det værktøjet ønskes lanceret til både Windows Unix og Linux. Her vil der kunne spares tid og kræfter ved at vælge et standard framework der understøttes af de krævede operativ systemer. Vores valg er faldet på udviklingssproget Java fra Sun Microsystems idet store mængder dokumentation og udviklingsværktøjer er tilgængeligt da hele konceptet bygger på open source.

1.4 Udviklingsstrategi

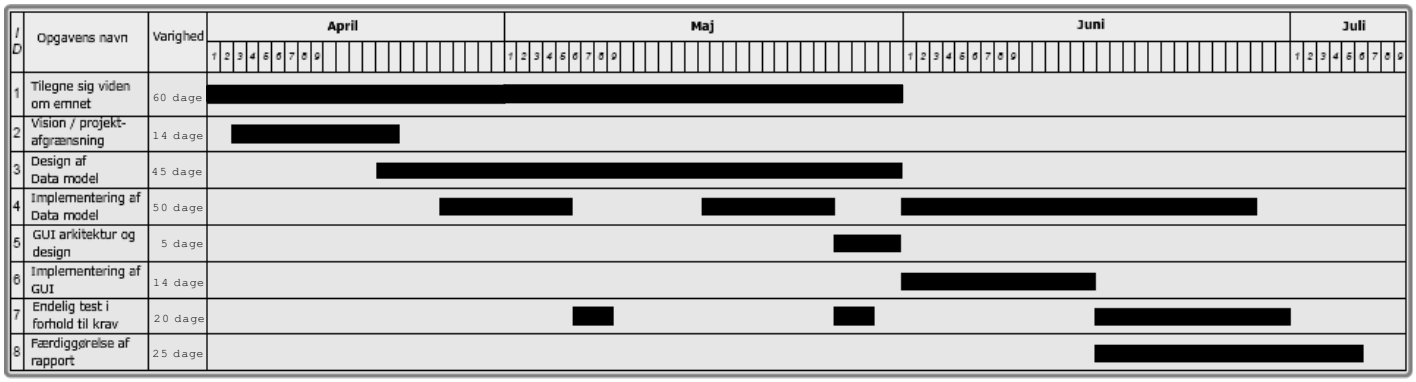
Idet vores projekt indeholder en række moduler, der kan udvikles uafhængigt af hinanden, er fremgangsmåden for opbygning af vores værktøj baseret på den iterative objektorienterede udviklingsmodel inden for software design kaldet (Incremental software development model)[1] eller på dansk, iterativ udviklingsproces, vist nedenfor.



1.5 Tidsplan

Projektet påbegyndes d. 1. April 2007 og skal afleveres pr. 6. Juli 2007.

I nedenstående Gantt diagram er estimeret en tidsplan for projektet.



Grundet udvikling af flere moduler, der skal kobles op mod samme data model er design og implementering af denne den absolut vigtigste del.
Der vil gennem implementeringsfasen løbende blive foretaget tests op imod de

[1] http://en.wikipedia.org/wiki/Iterative_development

stillede krav. En egentlig iterationsplan er ikke lavet, istedet vil enkelte udviklingstrin løbende blive gennemgået med vores vejleder. På denne måde vil det være muligt, tidligt at finde frem til fejl og misforståelser, der derved hurtigt kan rettes.

1.6 Riscici

Som i alle andre udviklingsprojekter er der forskellige risici at tage højde for. Der fokuseres på tre områder. Her beskrives hvad der kan gå galt, samt hvorledes problemer undgås eller tackles i opløbet.

Person risici omhandler hvad der kan gå galt på det personlige niveau.
Skulle en af os under udviklingsforløbet eksempelvis blive ramt af alvorlig sygdom, overdrages projektet til den anden, og vi vil stadig være istand til at kunne færdiggøre opgaven inden for afleveringstidspunktet.

Data risici omhandler risici og reduktion af selv samme i forbindelse med håndtering af dokumenter og sourcecode.
Dokumenter, sourcecode, rapportudkast og alle andre dokumenter lagres på G-barens cvs repository således, at vi begge har adgang til de nyeste versioner af alt data. Efter endt arbejdsdag synkroniseres alt arbejde med cvs repositoriet. Herved haves kopi af samtlige filer på hver af vores computere samt i cvs repositoriet. Det antages at være usandsynligt, at både lokal harddiske og cvs repository går tabt under normale forhold.

Program risici omhandler risici i forbindelse med kompleksiteten af produktet.
Skulle kompleksiteten af omfanget af implementeringen føre til risiko for at deadline ikke overholdes, kan kravene til softwaren skæres ned i form af afgrænsning. Den tilnærmede iterative udviklingsproces betyder, at afgrænsning af krav ikke vil føre til et program, der ikke kan køre.

Kapitel 2

Teknologi og værktøjer

2.1 Java Sun Microsystems Framework

Som teknologisk platform er løsningen bygget på Sun Microsystems Java framework. Denne platform må betragtes som værende blandt de mest moderne på markedet, og kan benyttes til såvel webapplikationer, som traditionelle applikationer. Desuden kan Java applikationer køres på stort set alle operativsystemer hvis package standarden overholdes.
Java er et såkaldt "managed environment" ligesom for eksempel Microsoft .Net. Det er derfor ikke er muligt at tilgå hukommelsesområder vilkårligt. Derudover har Java også automatisk garbage collection, hvilket betyder at applikationen selv styrer sit hukommelsesforbrug, hvilket kan være både godt og skidt.

2.2 Eclipse SDK

Eclipse editoren bliver benyttet som udviklingsværktøj under hele projektet. Eclipse tilbyder udover en god editor med on-the-fly compiler også diverse plugins til udvikling af for eksempel UML og sekvens diagrammer samt Javadoc. Desuden er der indbygget et profiling værktøj, så man kan holde styr på de enkelte funktioners cpu forbrug.

2.3 Omondo Eclipse UML

Omondo Eclipse UML er et plugin i Eclipse der bruges til udvikling af diagrammer. Alternativt kan det bruges som reverse engineering for at se, hvorledes klasserne kommunikerer.

2.4 GAMS script

Til kvalitetskontrol af vores resultater, benyttes et GAMS script, udviklet af vores vejleder. I dette script indlæses et kommunikationsnetværk og en række demand paths, hvorefter en række demands genereres og en teoretisk lowerbound (TLB) for den bedste fordeling af disse findes. Bounden sammenholdes med resultaterne fra routingsalgoritmerne og den relative forskel i fht. bounden bruges som kvalitetscheck af algoritmerne.

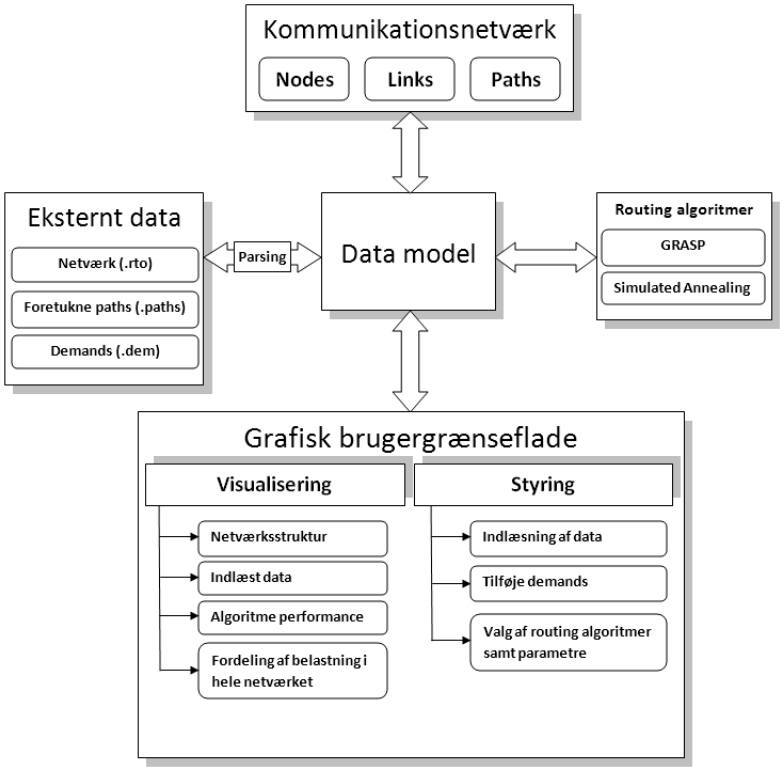
Kapitel 3

Design og Arkitektur

For at opfylde de 3 overordnede punkter i kravspecifikationen skal programmet som minimum indeholde følgende moduler som designes uafhængigt af hinanden.

- Design af parser til eksterne data
- Opbygning af et kommunikationsnetværk
- Design af en Data model til lagring af netværkstruktur, demands, demand paths samt algoritme resultater
- Design af routingsalgoritmer
- Design af grafisk brugergrænseflade til visualisering og styring

Det tilstræbes at overholde Model View Control princippet, således at den grafiske brugergrænseflade opdateres ved selv at hente informationer fra data modellen. Den overordnede program struktur tilstræbes efter nedenstående figur.



I det følgende gives en mere teknisk tilgang til områderne vi finder nødvendige at uddybe. Idet vores valg af datastrukturer i implementeringsfasen underbygges af forståelsen for de enkelte begreber er det vigtigt at disse forklares her.

3.1 Kommunikationsnetværk

Et kommunikationsnetværk består af en række nodes og links. 2 nodes kan være forbundet via et link. Ydermere kan der mellem 2 nodes være en forbindelse som består af en række links.

3.2 Data model

Data modellen indeholder en række komponenter der er essentielle for optimeringsdelen, men også for udveksling af informationer med den grafiske brugergrænseflade.
I første omgang vil vi dog koncentrere os om de komponenter, der er nødvendige

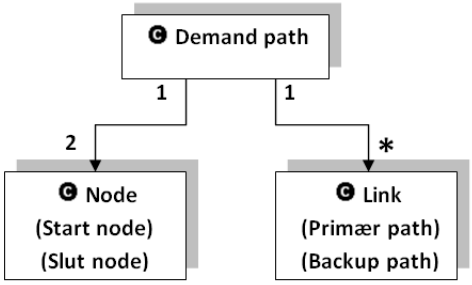
for at kunne køre routingsalgoritmerne. Det drejer sig om:

1. En datastruktur til repræsentation af demands og demand paths
2. En funktion til at indeksere alle demands og demand paths når de indlæses
3. En komponent til at beregne den samlede belastning i et netværk

I de følgende afsnit gennemgås slavisk de ovenstående komponenter.

3.2.1 Demand paths

Når der indlæses en række demand paths til at route demands igennem, indlæses for hver både en primær og en backup path. Det væsentlige er her at alle links i den primære path er forskellige fra dem i backup path'en. Derved kan man altid opnå forbindelse via backup path'en, uanset hvilke link der måtte fejle i den primære path. Datastrukturen for en demand path er vist på figuren nedenfor.

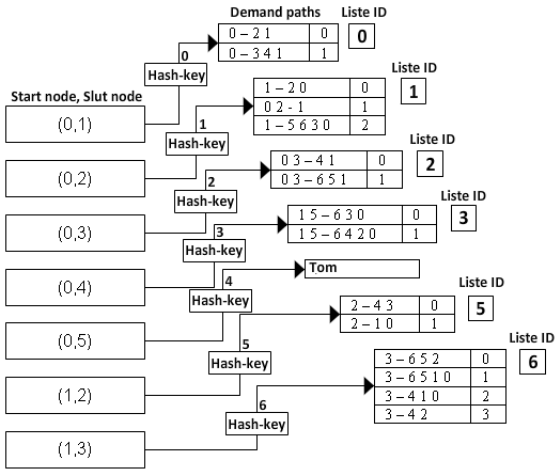


3.2.2 Indeksering af demand paths (Hashing)

Når demand paths indlæses fra ekstern data i vores program, ønskes det at indeksere dem således at alle med start node **1** og slut node **2** kommer i samme liste, tilsvarende for alle med start node **1** og slut node **3** og så videre. Istedet for at sortere dem med en algoritme, hvilket afhængig af antallet af demand paths kan være tidskrævende, har vi valgt at benytte en hash funktion af eget design.
En hash funktion er en matematisk metode til at konvertere noget data til et unikt id (en hash-key). Således kan man sikre sig at hver demand path med samme start og slut node bliver indekseret ens. Hash funktionen ser ud som følger:

$$\text{Hash-key} = i \cdot n - \sum_{h=0}^{i}(h) + j - 1$$

Hvor i er start node, j er slut node og n er det samlede antal nodes i netværket. Baggrunden for funktionens virkemåde vil ikke blive forklaret yderligere. Istedet ses i nedenstående figur et eksempel på indeksering (hashing) af en række demand paths indlæst fra ekstern data.



Køretiden for at beregne en hash-key er $O(1)$ i modsætning til et balanceret søgetræ der for indsættelse af et element er $O(\log n)$.

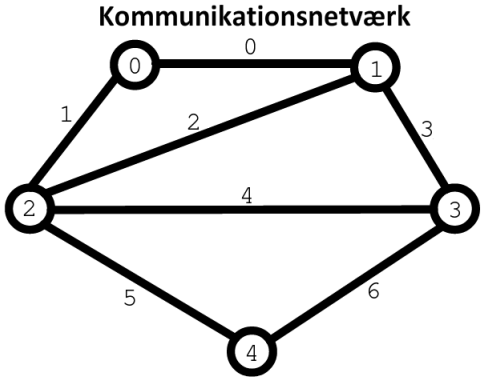
3.2.3 Demands

En demand er en forespørgsel på at få oprettet en forbindelse mellem 2 nodes i et netværk med en given båndbredde. I denne rapport omtales båndbredden som "volume". Ligesom med demand paths bliver demands indekseret ved brug af samme hash funktion.

3.2.4 Komponent til beregning af belastning i et netværk

Belastningen for hvert link i et netværk er forholdsvis kompliceret at holde styr på og kræver en speciel datastruktur som benævnes **"Failure Matrix"**. Formålet med denne datastruktur er at kunne repræsentere alle links i et netværk udfra en matrix med en dimension svarende til antallet af links L, altså $L \times L$. Når belastningen på et netværk omtales, ses der udelukkende på et "worst case scenario", hvor links ofte fejler. Derfor reserveres der både plads til primære og backup paths, så der i tilfælde af svigt på en primær path er taget højde for den belastning der vil fremkomme ved brug af den tilhørende backup path. Her

kommer formålet med matrix-strukturen til sin ret, idet man her kan repræsentere både belastningen ved brug af primære paths, samt backup paths i tilfælde af svigt på den primære.
Nedenfor er vist et kommunikationsnetværk med tilhørende Failure matrix. Det skal bemærkes at der endnu ikke er tilføjet demands til netværket, hvorfor belastningen på hvert enkel link er 0.



Tilhørende Failure Matrix

Hvis link *n* fejler i den primære path, udvælg links der benyttes i backup-pathen lodret

X indikerer at hvis link *n* fejler, så fejler link *n*

Link *n* bruges i den primære path

	0	1	2	3	4	5	6
0	X	0	0	0	0	0	0
1	0	X	0	0	0	0	0
2	0	0	X	0	0	0	0
3	0	0	0	X	0	0	0
4	0	0	0	0	X	0	0
5	0	0	0	0	0	X	0
6	0	0	0	0	0	0	X

Tilføjelse af en demand til failure matricen foregår i to trin.

1. trin består i at tilføje links der indgår i den primære path. Hvis link 1 for eksempel indgår i den primære path, lægges volumen for den ønskede demand til i samtlige felter for række 1.
2. trin består i at reservere plads til links der indgår i backup path'en for hvert link i den primær path. Hvis link 1 for eksempel indgår i den primære path, reserveres for link 1, volumen for hvert link i backup path'en. I tilfælde af at link 1 i den primære path skulle fejle er der således taget højde for dette.

Bemærk at hvis et link fejler i den primære path betragtes resten der indgår som at fejle hvorfor backup path'en benyttes istedet.

Operationerne for at tilføje primær og backup path kan også beskrives i pseudokode hvilket er gjort nedenfor.

Algorithm 1 Tilføj primære path

```
1: initialize Failurematrix F; Demandpaths p;
2: for each link l in primary path of p do
3:    F.Row[l]+=volume;
4: end for
```

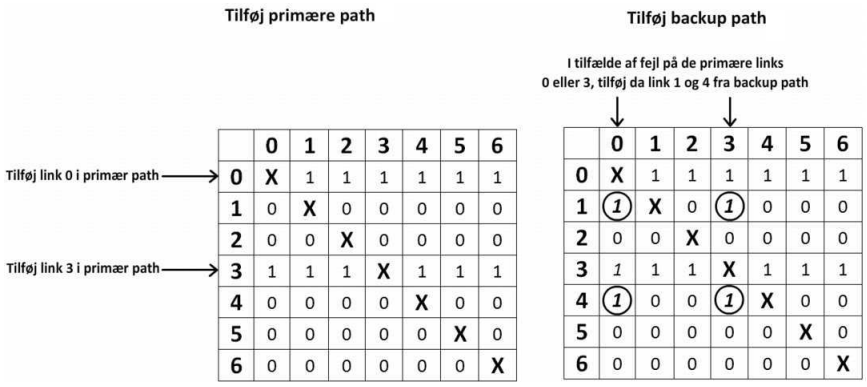
Algorithm 2 Tilføj Backup path i tilfælde af svigt på primære

```
1: initialize Failurematrix F; Demandpaths p;
2: for each link l in primary path of p do
3:    for each link l' in backup path of p do
4:       F[l][l']+=volume;
5:    end for
6: end for
```

Hvis man ønsker at tilføje nedenstående demand til et netværk, ændres failure matricen som følge deraf.

Start node	Slut node	Primær path	Backup path	Volume
0	1	P(0, 3)	B(4, 1)	1

Failure matricen kommer efter tilføjelse af ovenstående demand til at se ud som følger:



	0	1	2	3	4	5	6
0	X	1	1	1	1	1	1
1	0	X	0	0	0	0	0
2	0	0	X	0	0	0	0
3	1	1	1	X	1	1	1
4	0	0	0	0	X	0	0
5	0	0	0	0	0	X	0
6	0	0	0	0	0	0	X

	0	1	2	3	4	5	6
0	X	1	1	1	1	1	1
1	(1)	X	0	(1)	0	0	0
2	0	0	X	0	0	0	0
3	1	1	1	X	1	1	1
4	(1)	0	0	(1)	X	0	0
5	0	0	0	0	0	X	0
6	0	0	0	0	0	0	X

I det følgende afsnit beskrives hvordan ovenstående bruges til at bestemme den samlede belastning på et netværk.

3.2.5 Evalueringsfunktion

Hvis man forestiller sig at følgende demands indlæst i tidligere nævnte netværk, kommer failure matricen til at se ud som følger:

Demands

Start node	Slut node	Primær path	Backup path	Volume
0	1	P(0)	B(2, 1)	1
0	2	P(1)	B(2, 0)	1
0	3	P(0, 3)	B(4, 1)	1
0	4	P(1, 5)	B(6, 3, 0)	1
1	2	P(2)	B(4, 3)	1
1	3	P(3)	B(6, 5, 2)	1
1	4	P(2, 5)	B(6, 4, 1, 0)	1
2	3	P(4)	B(3, 2)	1
2	4	P(5)	B(6, 3, 2)	1
3	4	P(6)	B(5, 2, 3)	1

Failure matrix efter indsættelse

	0	1	2	3	4	5	6
0	X	4	2	2	2	3	2
1	4	X	2	3	2	2	2
2	3	3	X	3	2	2	2
3	2	3	4	X	2	4	2
4	2	1	2	3	X	2	2
5	3	3	3	3	4	X	4
6	1	2	2	1	2	4	X

Den samlede belastning på hvert enkelt link vil kunne ses som den største værdi i hver række. For eksempel vil belastningen for link 1 (4) kunne ses i række 1. Summen af disse værdier udgør den samlede belastning på netværket. Matematisk kan dette repræsenteres ved følgende udtryk

$$\text{Samlet belastning} = \sum_{i=0}^{n} \text{Max}(\text{række}(i))$$

hvor n er antallet af links i netværket. Den samlede belastning i netværket fra før vil i dette tilfælde være 26 som vist nedenfor.

	0	1	2	3	4	5	6	
0	X	(4)	2	2	2	3	2	4
1	(4)	X	2	3	2	2	2	4
2	(3)	3	X	3	2	2	2	3
3	2	3	(4)	X	2	4	2	4
4	2	1	2	(3)	X	2	2	3
5	3	3	3	3	(4)	X	4	4
6	1	2	2	1	2	(4)	X	4

Total: 26

Failure matricen er fundamentet til vores routing algoritmer, idet strategien for disse, som vi også kommer ind på i næste afsnit, er at evaluere så mange løsninger så muligt. I alle tilfælde vil algoritmerne forsøge at route de indlæste demands via alternative demand paths i håb om at få en bedre løsning.

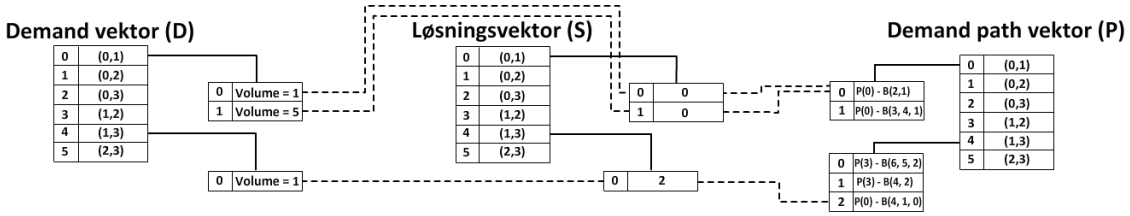
3.2.6 Repræsentation af en løsning

Med udgangspunkt i evalueringsfunktionen ønskes yderligere en måde at repræsentere en løsning på. En løsning er en serie af par bestående af én demand og en tilknyttet demand path. Det er allerede nævnt, at både demands og demand paths hashes ind i 2 forskellige lister. Udseendet af disse 2 lister er ens i den forstand at hashfunktionen returnerer en henvisning til samme node par for begge

lister.
Hvis man forestiller sig at hashnøglen 2 henviser til node parret (0,3) vil der i demand listen refereres til samtlige demands mellem (0,3) og i demand path listen vil der refereres til samtlige demand paths mellem (0,3).
Man kan nu kæde de 2 lister sammen ved at oprette en ny liste (løsningsvektoren) der indeholder relationen mellem disse, fuldstændig ligesom en database relation.
Der oprettes nu en liste identisk med demand listen men uden indhold. Her er det vigtigt at holde sig for øje, at indekserne stemmer overens med indekserne i demand listen. Derefter tilføjes indholdet i form af en henvisning til de valgte demand paths. Nedenfor ses et eksempel på en løsning for routing af 3 demands.
Demands:

#	Start node	Slut node	Volume
1	0	1	1
2	0	1	5
3	1	3	1

Serien af relationer mellem demands og de tilknyttede demand paths er repræsenteret i løsningsvektoren:



Løsningen ses som relationen mellem demands og demand pathes som vist i nedenstående tabel.

#	Demand	Tilhørende Demand path
1	(0,1) volume=1	P(0) - B(2, 1)
2	(0,1) volume=1	P(0) - B(2, 1)
3	(0,1) volume=1	P(0) - B(4, 1, 0)

3.3 Beskrivelse af anvendte routing algoritmer

Strategien for hvordan demand paths udvælges til at indgå i en løsning, foregår ved brugen af meta heuristikker. I dette projekt er der udvalgt 2 forskellige meta heuristikker, Simulated Annealing (Simuleret udglødning) og GRASP (Greedy Randomized Adaptive Search Procedure). Desuden er anvendt en optimeringsfeature kendt som Delta evaluering. I det følgende gennemgås algoritmerne i deres traditionelle form.

3.3.1 Simulated Annealing

Simulated annealing (herefter SA) er en meta heuristik baseret på sandsynlighedsteori for et optimeringsproblem. Målet er at finde en god tilnærmelse til problemets optimale løsning i et stort søgerum.

Oprindelig blev SA brugt i termodynamiske problemer. I et termodynamisk system blev en indledende tilstand valgt med en energi E og en temperatur T. Ved at holde T konstant påvirkedes den indledende tilstand og ændringen af energien dE kunne beregnes. Hvis dE var negativ, accepteredes den nye konfiguration. Hvis dE var positiv accepteredes tilstanden med en sandsynlighed givet af Boltzmann faktoren $exp(-(dE/T))$. Denne proces blev gentaget nok gange for at give gode statistiske valg for den nåede temperatur. Derefter sænkedes temperaturen og processen blev gentaget indtil et fast stadie blev fundet for $T = 0$.
Overstående kan bruges analogt i tilgangen til optimeringsproblemer. Den aktuelle tilstand i det termodynamiske system svarer til den aktuelle løsning af optimeringsproblemet. Energi-ligningen for systemet svarer til evalueringsfunktionen, og den lavest mulige energi i systemet svarer til den teoretiske lowerbound. Den store udfordring i implementeringen af algoritmen er, at der ikke er nogen oplagt sammenhæng mellem temperaturen T og en fri parameter i optimeringsproblemet. Endvidere, for at undgå at blive fanget i et lokalt minimum, er SA afhængig af valget af start temperaturen, antallet af iterationer der foretages for hver temperatur, og hvor meget temperaturen falder for hver iteration i optimeringsprocessen.

Naboerne til en løsning

I vores problem har vi valgt at en nabo $s \in N(s)$, for enhver løsning s kan findes ved at udskifte en tilfældig værdi i løsningsvektoren med en anden mulighed. Løsningen skal stadig være mulig.

En simpel iteration

I hver iteration udvælger algoritmen en række naboer s' fra løsningen s og vælger statistisk mellem at ændre løsningen ved at bruge de nye naboer eller blive i den nuværende løsning. Typisk er dette skridt gentaget indtil systemet når en tilstand der er god nok for applikationen eller indtil en given stop parameter er

nået.

Sandsynligheden for overgangen mellem 2 løsninger

Sandsynligheden for en overgang mellem den nuværende løsning s og en kandidat s' kan ses som en funktion $P(e, e', T)$ for energierne $e = E(s)$ og $e' = E(s')$ og temperaturen T.
Et væsentligt krav for overgangssandsynligheden P er, at den skal være forskellig fra 0 når $e' > e$, hvilket betyder at algoritmen kan vælge at gå til den nye tilstand selvom denne er dårligere (har højere energi). Det er denne egenskab der forhindrer algoritmen i at blive fanget i et lokalt minimum. Omvendt, når T bliver 0 skal sandsynligheden $P(e, e', T)$ være 0 for $e' > e$ og 1 for $e' < e$ og algoritmen (for lave værdier af T) vil foretrække at gå nedad mod bedre løsninger. Når dette sker er der tale om en grådig algoritme der kun foretager overgang mellem 2 løsninger, hvis den nye løsning er bedre.

Reducering af T

En anden essentiel egenskab for SA er, at temperaturen gradvist reduceres som algoritmen arbejder sig frem. Som udgangspunkt er T sat til en høj værdi som udfra en funktion reduceres i hvert skridt. Det mest almindelige er at bruge en simpel statisk reduceringsfunktion for T som vist nedenfor.

$$T = \alpha \cdot T$$

α er en værdi lavere men tæt på 1.

Konvergens til optimal løsning

Sandsynligheden for at SA afslutter med at have fundet den optimale løsning vil nærme sig som reduceringsfunktionen udvides. Dette teoretiske resultat er imidlertid ikke særlig nyttigt, da beregningstiden krævet for at sikre sig tilstrækkelig stor sandsynlighed for succes, ofte overskrider tiden krævet for at afprøve alle løsninger.

Genstart

Nogle gange kan det være hensigtsmæssigt at gå tilbage til en tidligere løsning og udfra denne afsøge andre dele af søgerummet. Dette kaldes genstart. For

at gøre dette sættes s og e til sb og eb og reduceringsfunktionen vil måske genstartes. Beslutningen om at genstarte kan være baseret på et bestemt antal iterationer, eller på om den nuværende løsning er for dårlig i forhold til den bedst fundne hidtil. I vores problem er vi dog i den situation at mange løsninger kan have samme værdi, og der er derfor ikke nogen indikation for, om det bedre kan svare sig at genstarte fra den hidtil bedste løsning. Det er kun genstart af reduceringsfunktionen der tillader algoritmen i at bevæge sig til løsninger dårligere og måske væk fra lokale minimum. Algoritmen for SA er forklaret i pseudokode nedenfor.

Algorithm 3 Simulated Annealing

```
1:  initialize: s := s0; e := E(s); k := 0; T := T0;
2:  while k < kmax do
3:     sn := N(s);
4:     en := E(sn);
5:     if en < eb then
6:        sb := sn; eb := en;
7:     end if
8:     if en < e then
9:        s := sn; e := en;
10:    else
11:       if random[0;1] < P(en, e, T) then
12:          s := sn; e := en;
13:       end if
14:    end if
15:    T := T*α;
16:    k := k+1;
17: end while
18: return s;
```

Linje 1 initialiserer s til en tilfældig genereret løsning.
Energien af løsningen beregnes og temperaturen T sættes til en værdi specificeret af brugeren.
E er evalueringsfunktionen, N genererer en løsning i nabolaget for s og P beregner en værdi baseret på eN, e og T til $\exp((e - eN)/T)$, der afgører om den nye løsning skal benyttes selvom den er dårligere end hidtil bedste.

3.3.2 GRASP

Greedy Randomized Adaptive Search Procedure eller GRASP er en simpel meta heuristik opdelt i 2 faser. I første fase findes en tilfældig grådig (Adaptive Greedy Randomized) løsning ved at udvælge elementer tilfældigt fra en Restricted Candidate List der indeholder lovende kandidater til en god løsning. I anden fase iværksættes en lokal søgning som forsøger at forbedre løsningen ved at erstatte dele af løsningen med elementer fra nabolaget.

Greedy Randomized funktionen

I konstruktionsfasen af en GRASP, bliver en løsning opbygget ét element af gangen, hvert element tilfældigt valgt fra Restricted Candidate listen bestemt af Adaptive Greedy Randomized funktionen. Algoritmen for denne er beskrevet i pseudokode nedenfor.

Algorithm 4 ConstrucGreedyRandomizedSolution(Solution)

```
1: initialize Solution = ;
2: for Solution construction NOT done do
3:    MakeRCL(RCL);
4:    s SelectElementAtRandom(RCL);
5:    Solution = Solution ∪ s;
6:    AdaptGreedyFunction(s);
7: end for
```

Local Search funktionen

Fordelen ved en Local Search algoritme er, at man foretager adskillige lokal søgninger med vidt forskellige udgangspunkter. Desuden kan man afhængig af problemet selv vælge,s hvilken type lokal søgning man vil foretage. Om dette er en simpel hillclimber, en TABU search eller for den sags skyld en Simulated Annealing algoritme er op til udvikleren at bestemme og skal ses i forhold til, hvad der vil virke bedst for det specifikke problem. Lokal søgningen minimerer yderligere risikoen for at havne i et lokalt minimum. Algoritmen for Local Search er beskrevet i pseudokode på næste side.

Algorithm 5 LocalSearch(P,N(P),s)

```
1: for for s not locally optimal do
2:    Find a better solution t ∈ N(s);
3:    Let s = t;
4: end for
5: Return(s as local optimal for P);
```

Restricted Candidate List

En Restricted Candidate List er en liste der indeholder elementer, som udvikleren fra start af har specificeret som gode i en løsning. Man kunne for eksempel sige at alle demand paths kunne være elementer i en Restricted Candidate List hvis vores problem blev udvidet så routingsalgoritmen selv skulle finde vej mellem 2 noder og generere en demand path for at etablere en forbindelse.

Den samlede algoritme for meta heuristikken GRASP er beskrevet i pseudokode nedenfor.

Algorithm 6 GRASP()

```
1: InputInstance();
2: for GRASP stopping criterion not satisfied do
3:    ConstructGreedyRandomizedSolution(Solution);
4:    LocalSearch(Solution);
5:    Updatesolution(Solution,BestSolutionFound);
6: end for
7: Return(BestSolutionFound);
```

3.4 Delta evaluering

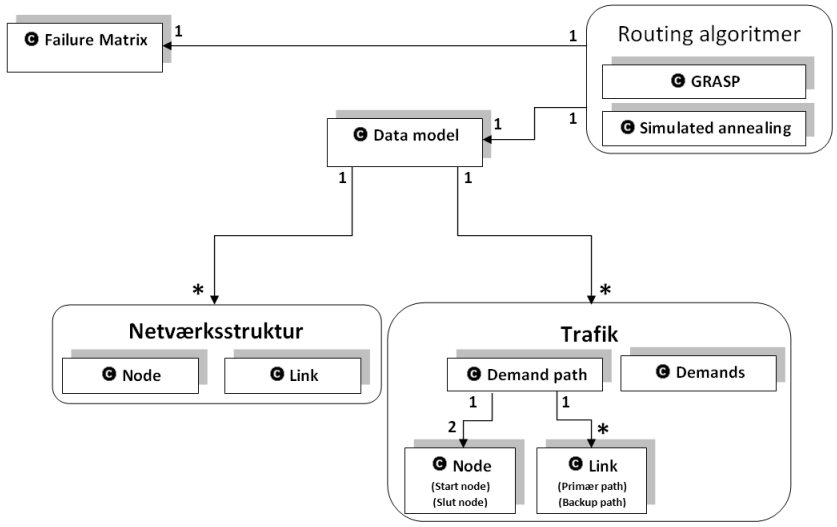
Evaluering af løsninger med evalueringsfunktionen kan være en tidskrævende affære. Derfor udnytter vi fordelen af, at der for enhver løsning kun sker små ændringer i hver iteration. Dette betyder at vi kan foretage små justeringer i failure matricen ved brug af såkaldt delta evaluering. Delta evaluering er en metode til at ændre på en given løsning for at få bedre resultater istedet for at generere en ny løsning og derefter evaluere. Da evalueringsfunktionen afhængig af probleme og meta heuristikken, bliver eksekveret mange gange i løbet af en optimering, kan man ved brug af delta evaluering opnå langt flere iterationer og derved bedre performance for algoritmen.
I vores problem ville algoritmen til delta evaluering skulle tage 2 løsninger som input. Den gamle løsning (oS) som er den der ændrer failure matricen, og en ny løsning (nS) som vil ændre failure matricen efter delta evaluering. Delta evalueringen trækker først en demand path fra den gamle løsning og tilføjer denne til den nye. Herefter evalueres den nye løsning. Dette vil tilfælde hvor en ny løsning tager lang tid at generere, være langt hurtigere. Algoritmen for delta evaluering er beskrevet i pseudo kode nedenfor.

Algorithm 7 Delta evaluering

```
1: initialize oS; nS;
2: for i := 1...length(oS) do
3:    if oS[i] != nS[i] then
4:       oDP := getDemandPath(oS[i]);
5:       nDP := getDemandPath(nS[i]);
6:       SubstractDemandPath(oDP);
7:       AddDemandPath(nDP);
8:    end if
9: end for
```

3.5 UML diagram for data model

Samspillet mellem de nævnte komponenter der indgår i data modellen indtil nu kan ses i et simplificeret UML diagram vist nedenfor.



3.6 Parsing af data

Til repræsentation af kommunikationsnetværk, demands og demand paths følger 3 forskellige formater. Desuden benytter det anvendte Gams script omtalt i sektion 2 yderligere en række gams formater for demand paths og kommunikationsnetværk. Data modellen kan kun benytte én repræsentation for hver af disse, hvorfor det er nødvendigt at kunne foretage følgende konverteringer.

- Parsing af kommunikationsnetværk til Data model
- Parsing af demand paths til Data model
- Parsing af demands til Data model
- Konvertering af kommunikationsnetværk fra Gams format til almindeligt format.
- Konvertering af demand paths fra Gams format til almindeligt format

Filformatet for de enkelte repræsentationer kan ses i appendix A.

3.7 Grafisk brugergrænseflade

I følgende afsnit vil vi gennemgå opbygningen af den grafiske brugergrænseflade. Kravspecifikationen i kapitel 1 udtrykker en række punkter der i forbindelse med visualisering og styring kræves implementeret i programmet. Disse er nedenfor udtrykt i form af use cases.

Use Case # 1 - Indlæs netværk fra fil

USE CASE 1	Indlæs netværk fra fil	
Goal in Context	Målet er at indlæse et kommunikationsnetværk til programmet fra en lokal fil	
Scope & Level	Primær opgave	
Preconditions	Systemet skal være klar	
Success End Condition	Et netværk indlæses til programmet	
Failed End Condition	Et netværk indlæses ikke, f.eks. pga. fejl i netværksfilen	
Primary Actors	En hvilken som helst bruger	
Secondary Actors	Ikke nødvendig	
Trigger #1	En bruger ønsker at indlæse et netværk til programmet	
DESCRIPTION	Step	Action
	1	*Brugeren vælger i menuen File → Load Network*
	2	*Brugeren vælger filen der ønskes indlæst og trykker Open*
EXTENSION	Step	Branching Action
	2a	*Ved indlæsning af ugyldigt netværk indlæses data ikke til programmet*

RELATED INFORMATION	Indlæs netværk fra fil
Priority	Høj
Performance	Udførelsen forventes at tage under 2 minutter.
Frequency	Ukendt på nuværende tidspunkt.
Channels to actors	Interaktiv
OPEN ISSUES	Ingen
Due Date	6/7-2007
... Any other management information	Ingen
Superordinates	Ingen
Subordinates	Use case 2, Use case 3

Use Case # 2 - Indlæs Demands fra fil

USE CASE 1	Indlæs Demands fra fil	
Goal in Context	Målet er at indlæse et antal Demands fra en lokal fil til programmet	
Scope & Level	Primær opgave	
Preconditions	Systemet skal være klar og der skal være indlæst et netværk, samt en række demand paths til programmet	
Success End Condition	En række Demands indlæses i programmet	
Failed End Condition	Demands indlæses ikke, f.eks. pga. fejl i Demands-filen	
Primary Actors	En hvilken som helst bruger	
Secondary Actors	Ikke nødvendig	
Trigger #1	En bruger ønsker at indlæse et antal Demands til et indlæst netværk i programmet	
DESCRIPTION	Step	Action
	1	*Brugeren vælger i menuen File → Load Demands*
	2	*Brugeren vælger filen der ønskes indlæst og trykker Open*
EXTENSION	Step	Branching Action
	2a	*Ved indlæsning af ugyldige demands indlæses demands ikke til programmet*

RELATED INFORMATION	Indlæs Demands fra fil
Priority	Høj
Performance	Udførelsen forventes at tage under 2 minutter.
Frequency	Ukendt på nuværende tidspunkt.
Channels to actors	Interaktiv
OPEN ISSUES	Ingen
Due Date	6/7-2007
... Any other management information	Ingen
Superordinates	Use case 1, Use case 5
Subordinates	Use case 4

Use Case # 3 - Tilføj enkelt demand til netværk

USE CASE 1	Tilføj Demand til netværk	
Goal in Context	Målet er at indsætte en enkelt Demand i et indlæst netværk	
Scope & Level	Primær opgave	
Preconditions	Systemet skal være klar og der skal være indlæst et netværk og en række demand paths	
Success End Condition	En Demand indsættes i netværket	
Failed End Condition	Demand'en indsættes ikke, f.eks. pga. fejl i formatet	
Primary Actors	En hvilken som helst bruger	
Secondary Actors	Ikke nødvendig	
Trigger #1	En bruger ønsker at indsætte en Demand i et indlæst netværk	
DESCRIPTION	Step	Action
	1	*Brugeren vælger i menuen Traffic → Add Demand*
	2	*I popup vinduet indtaster brugeren Demand'ens data i textboksen*
EXTENSION	Step	Branching Action
	2a	*Ved ugyldig indtastning informeres brugeren om at Demand'en ikke kunne indlæses*

RELATED INFORMATION	Tilføj Demand til netværk
Priority	Høj
Performance	Udførelsen forventes at tage under 5 minutter.
Frequency	Ukendt på nuværende tidspunkt.
Channels to actors	Interaktiv
OPEN ISSUES	Ingen
Due Date	6/7-2007
... Any other management information	Ingen
Superordinates	Use case 1, Use case 5
Subordinates	Use case 4

Use Case # 4 - Kør Routing-algoritme

USE CASE 1	Kør Routing-algoritme	
Goal in Context	Målet er at køre en Routing-algoritme på et indlæst kommunikationsnetværk	
Scope & Level	Primær opgave	
Preconditions	Systemet skal være klar, der skal være indlæst et netværk, et antal demand paths og et antal Demands	
Success End Condition	Routing-algoritmen returnerer et resultat om belastningen på netværket	
Failed End Condition	Routing-algoritmen køres ikke, f.eks. pga. fejl i netværk eller demands	
Primary Actors	En hvilken som helst bruger	
Secondary Actors	Ikke nødvendig	
Trigger #1	En bruger ønsker at køre Routing-algoritmen på et indlæst netværk	
DESCRIPTION	Step	Action
	1	*Brugeren vælger i menuen Run... → Routing-algorithm*
	2	*Brugeren vælger den ønskede algoritme der ønskes kørt på det indlæste netværk*
	3	*Brugeren vælger parametre for algoritmen og trykker RUN*
EXTENSION	Step	Branching Action
	3a	*Ved indtastning af ugyldige parametre køres algoritmen ikke og brugeren informeres herom*

RELATED INFORMATION	Kør Routing-algoritme
Priority	Høj
Performance	Udførelsen forventes at tage under 5 minutter.
Frequency	Ukendt på nuværende tidspunkt.
Channels to actors	Interaktiv
OPEN ISSUES	Ingen
Due Date	6/7-2007
... Any other management information	Ingen
Superordinates	Use case 1, Use case 2, Use case 5
Subordinates	Ingen

Use Case # 5 - Indlæs Demand Paths fra fil

USE CASE 1	Indlæs Demand Paths fra fil	
Goal in Context	Målet er at indlæse et antal Demand Paths fra en lokal fil til programmet	
Scope & Level	Primær opgave	
Preconditions	Systemet skal være klar og der skal være indlæst et netværk til programmet	
Success End Condition	En række Demand Paths indlæses i programmet	
Failed End Condition	Demand Paths indlæses ikke, f.eks. pga. fejl i filen	
Primary Actors	En hvilken som helst bruger	
Secondary Actors	Ikke nødvendig	
Trigger #1	En bruger ønsker at indlæse et antal Demand Paths til et indlæst netværk i programmet	
DESCRIPTION	Step	Action
	1	*Brugeren vælger i menuen File → Load Paths...*
	2	*Brugeren vælger filen der ønskes indlæst og trykker Open*
EXTENSION	Step	Branching Action
	2a	*Ved forsøg på indlæsning af ugyldige demand paths informeres brugeren om dette.*

RELATED INFORMATION	Indlæs Demands fra fil
Priority	Høj
Performance	Udførelsen forventes at tage under 2 minutter.
Frequency	Ukendt på nuværende tidspunkt.
Channels to actors	Interaktiv
OPEN ISSUES	Ingen
Due Date	6/7-2007
... Any other management information	Ingen
Superordinates	Use case 1

Kapitel 4

Implementering

I de følgende afsnit beskrives detaljer om implementeringen af begreberne omtalt i kapitel 3 ved gennemgang af data modellen. Begreberne er implementeret som klasser på traditionel vis i Java. Desuden beskrives hvilke biblioteker der er anvendt til opbygning til den grafiske brugergrænseflade. Enkelte specielt interessante metoders implementering vil blive beskrevet ved hjælp af kodeeksempler. Der er yderligere til hver klasse vedhæftet et tilhørende UML diagram. Samspillet mellem data modellen og den grafiske brugergrænseflade vil være beskrevet med sekvensdiagrammer.

4.1 Model aspekter

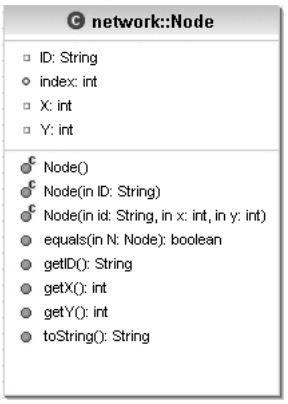
Klasserne der indgår i data modellen præsenteres i følgende format:

- **Klassens navn**, er navnet på klassen og Java filen.
- **Variabler**, her nævnes de vigtigste variabler brugt i klassen.
- **Beskrivelse**, her gives en kort gennemgang af klassen samt vigtige metoder.
- **Tilhørende UML diagram**

4.1.1 Node

Variabler: String `ID`, **int** `x`, `y`, `index`, **Vector<Node>** `neighbours`.

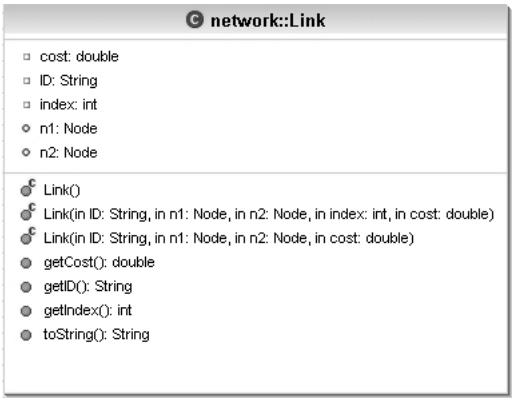
Beskrivelse: Node-klassen beskriver en node i et netværk. Den består af et unikt ID som er navnet på noden, to integers der beskriver dens koordinater i netværket, samt et index der beskriver hvor i rækkefølgen den blev indlæst. Klassediagrammet ses i figuren nedenfor.



4.1.2 Link

Variabler: String *ID*, **Node** *n1*, *n2*, **double** *cost*

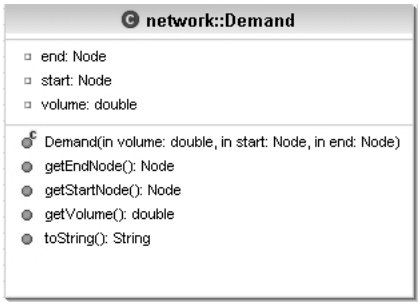
Beskrivelse: Klassen består af et unikt ID som er navnet på linket, 2 nodes som linket forbinder og en variabel der indikerer, hvad det koster at benytte linket. Klassediagrammet ses i figuren nedenfor.



4.1.3 Demand

Variabler: double *volume*, **Node** *start*, *end*

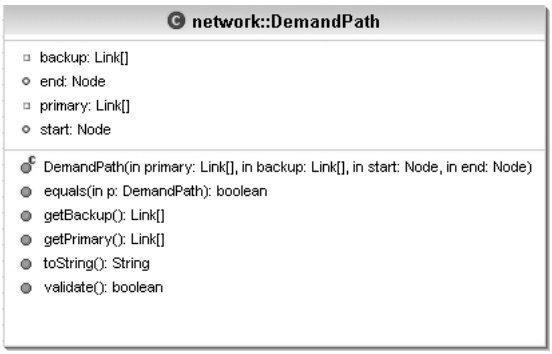
Beskrivelse: Hver demand er tilknyttet en start og en slut node samt en volume til repræsentation af den brugte båndbredde for hvert link for både den primære og backup path'en. Klassediagrammet ses i figuren nedenfor.



4.1.4 DemandPath

Variabler: Link[] `primary`, `backup` **Node** `start`, `end`

Beskrivelse: En demand path består af 2 `Link` arrays som beskriver primær og backup path. Derudover beskrives hvilke nodes demand path'en starter og slutter ved.
Det er desuden muligt at validere en given demand med metoden `validate()`. Metoden undersøger først om start og slut nodes stemmer overens med hvad der er angivet i begge arrays og undersøger dernæst om link i er forbundet med link $i+1$ for begge arrays.
`equals()` metoden returnerer om 2 demand paths, *p1* og *p2*, er ens hvilket gøres ved at undersøge om `primary` array i *p1* stemmer overens med `primary` array'et i *p2* og ligeledes for `backup`. Til dette benyttes Java's indbyggede `Arrays.equals()` metode. Klassediagrammet ses i figuren nedenfor.



4.1.5 DataObject

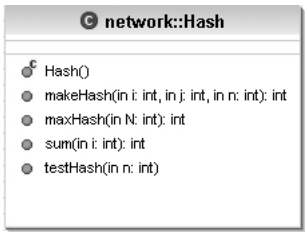
Variabler: int `N`, `L`, **Link[]** `Nodes`, `Links` **ArrayList<ArrayList<DemandPath>>** `P`,
ArrayList<ArrayList<Demand>> `D`

Beskrivelse: Klassen bruges til at parse kommunikationsnetværk, demands og demand paths fra eksterne filer ved hjælp af Java's StringTokenizer fra biblioteket `java.util.StringTokenizer`. Det indlæste kommunikationsnetværk gemmes i `N`, `L`, `Nodes` og `Links`. De indlæste demand paths gemmes i `P` og de indlæste demands gemmes i `D`.
Et kommunikationsnetværk bliver indlæst ved brug af metoden `readAndParseNetwork()`. Indlæsning af demand paths er delt op i en række metoder der muliggør at man kan indlæse dem enkeltvis. Ideen med dette er, at man senere kan indlæse ekstra ved direkte indtastning. Det samme gør sig gældende for indlæsning af demands. Klassediagrammet ses i figuren nedenfor.

4.1.6 Hash

Variabler: Ingen.

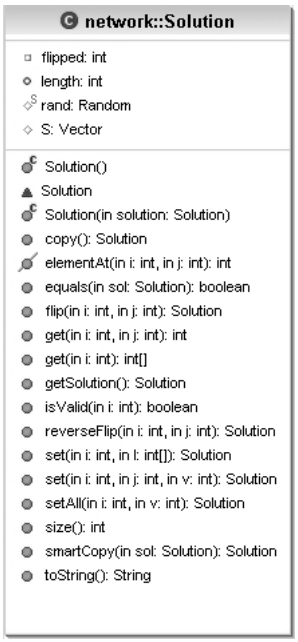
Beskrivelse: Klassen bruges til at beregne en nøgle (indekset) for en given demand / demand path baseret på start og slut node. Nøglen angiver placeringen i ArrayListerne `D` og `P`.



4.1.7 Solution

Variabler: Vector<int[]> `S`

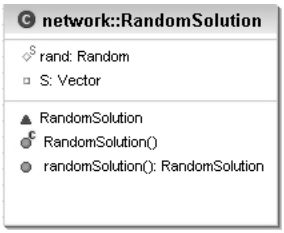
Beskrivelse: Solution består af en **Vector** hvis index svarer direkte overens med både `D` og `P` i DataObject klassen. På enhver gyldig plads, det vil sige en plads hvor der er et tilsvarende antal demand paths i `P`, er der et integer array der angiver relationen mellem hver `demand` og dens tilknyttede `demand path`. `isValid()` metoden undersøger om et givet indeks i `S` er gyldigt, forstået på den måde at det er ugyldigt hvis det tilsvarende indeks i `P` er tomt. I så fald vil der på pladsen i `S` stå -1.
Metoderne `flip()` og `reverseFlip()` "flipper" et givet indeks, hvilket betyder at man ændrer værdierne ved hjælp af Random klassen fra biblioteket `java.util.Random`. Værdien huskes i `flipped` så det senere er muligt at fortryde flippet med metoden `reverseFlip()`.
`equals()` metoden undersøger om 2 Solutions er ens ved at sammenligne deres arrays. Klassediagrammet ses på næste side.



4.1.8 RandomSolution

Variabler: Ingen.

Beskrivelse: RandomSolution er en nedarving af Solution klassen og har en enkelt metode, som genererer en tilfældig løsning. Klassediagrammet ses i figuren nedenfor.



4.1.9 FailureMatrix

Variabler: double[][] `F`, **double** `fit`, **double[]** `max`.

Beskrivelse: FailureMatrix bruges i forbindelse med validering af en Solution. Størstedelen af alle beregninger, der foretages i data modellen, sker i denne klas-

se.
Klassen har 2 constructors. Første constructor tager en FailureMatrix som argument og bruges til at kopiere instanser af klassen. Anden constructor er en standard constructor.
Metoden *initFailMatrix()* tager en *Solution* som argument og initialiserer variablerne i *FailureMatrix* klassen. Dette foregår primært ved gentagne kald af *addDemandPath()* og *substractDemandPath()*.
Til delta evaluering benyttes der 2 metoder, hvor den ene kalder den anden. Disse hedder begge *DeltaEval()*. Begge tager 2 (forskellige) løsninger som argument. Forskellen ligger i, at den ene metode slavisk gennemgår løsningerne og undersøger hvor forskellene ligger, og den anden metode foretager den reelle delta-evaluering. I selve delta evalueringen gøres der brug af metoderne *substractDemandPath()* og *addDemandPath()*.

addDemandPath() opererer ved hjælp af 3 for-løkker, hvoraf de 2 af dem er nestede. I den første løkke køres den primære path igennem. For hvert link i den primære path loop'es over backup path'en, og den nødvendige kapacitet reserveres i *F*. Et relevant stykke kode fra denne metode ses nedenfor.

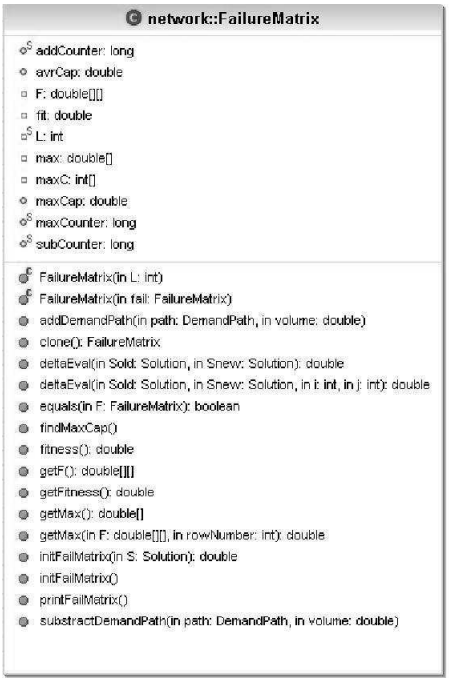
```
if (F[bi][pi] + cap > max[bi]) {

    fit += F[bi][pi] + cap - max[bi];
    max[bi] = F[bi][pi] + cap;
    maxC[bi] = 1;

} else if (F[bi][pi] + cap == max[bi])
    maxC[bi]++;

F[bi][pi] += cap;
```

bi og *pi* er indeks for de givne links for *i's* henholdsvis backup- og primær path. *cap* er kapaciteten der skal reserveres, og *max* indeholder den maksimale kapacitet for en given række i *F*. Der undersøges herefter om den reserverede kapacitet kommer til at overstige den største værdi i den nåede række. Er dette er tilfældet, skal 2 variabler opdateres: *max* og *maxC*. *maxC* tæller hvor mange værdier i en given række der indeholder den største værdi. Denne information bruges til at minimere antallet af tunge funktions-kald til metoden *getMax()*. *max* opdateres med den nye værdi og ligeledes *maxC*. Er den reserverede kapacitet derimod kun lig den største værdi for rækken, opdateres blot *maxC*. Til sidst opdateres *F* uafhængigt af udfaldet for de 2 if-sætninger.

`substractDemandPath()` opererer på det samme princip som *addDemandPath*. Forskellen ligger i at kun den maksimale værdi for en given række er interessant. Derfor bliver det en smule mere problematisk hvis der trækkes kapacitet fra en max værdi. Et relevant stykke kode fra denne metode ses nedenfor.

```
if(max[bi] == F[bi][pi]) {
     if(maxC[bi] == 1) {
         changed = true;
     }
     else
         maxC[bi]--;
 }
 F[bi][pi] -= cap;
```

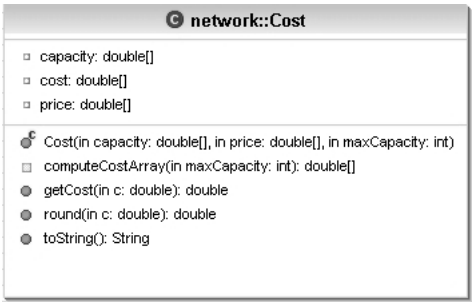
Først undersøges det om det forsøges at reducere en max værdi. Da der ikke kan være nogen værdi for den givne række i `F` (værdien er større end hvad der står i `max` for rækken), kan man nøjes med at undersøge om den er lig med. Klassediagrammet ses i figuren nedenfor.



4.1.10 Cost

Variabler: double[] *cost*, *capacity*, *price*

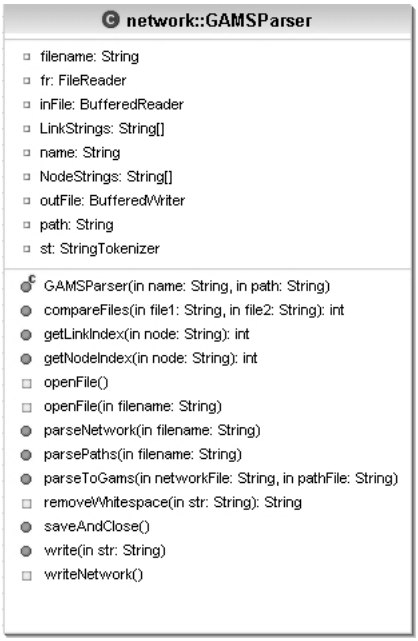
Beskrivelse: Cost klassen indeholder et array med modulære kapaciteter, for eksempel 2.5, 4, 10, 40 og et array med en tilsvarende pris. Der beregnes priser for stigende kapaciteter. Det er gjort således, at priserne beregnes forud for kørslen og gemmes i en **Vector** man kan bruge som opslag.
Klassen indeholder endvidere en metode der runder en given kapacitet op til nærmeste mindste kapacitet. Hvis den mindste modulære kapacitet for eksempel er 2.5 vil en ønsket volume på 3 skulle rundes op til 4. Denne klasse er ikke taget med i rapporten da vi ikke beskæftiger os med omkostninger. Man kan dog senere implementere denne klasse ved eventuel udvidelse af programmet. Klassediagrammet ses i figuren nedenfor.



4.1.11 GAMSParser

Variabler: String[] *NodeStrings*, *LinkStrings*

Beskrivelse: Klassen bruges til at konvertere data fra et Gams format til det anvendte format der bruges til indlæsning af kommunikationsnetværk og demand paths i DataObject klassen. Fil formaterne for samtlige filtyper kan ses i appendix A. Klassediagrammet ses på næste side.



4.1.12 StopWatch

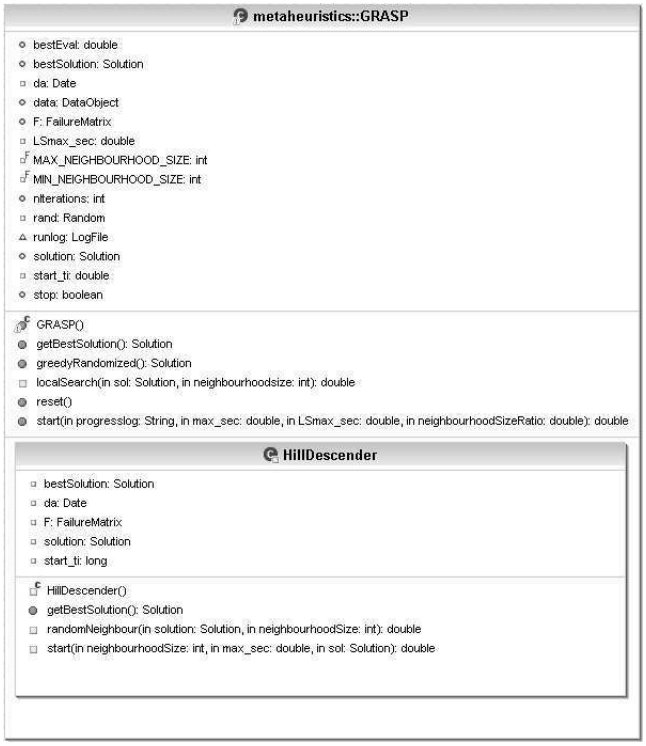
Variabler: Ingen.

Beskrivelse: Stopwatch er præcis som det lyder, et stopur, som benytter Javas indbyggede funktion *System.currentTimeMillis()* til at tage tid på diverse metoder. Klassen indeholder metoderne `start()`, `stop()` og `resume()`. Klassediagrammet ses i figuren nedenfor.

4.1.13 GRASP

Variabler: Ingen.

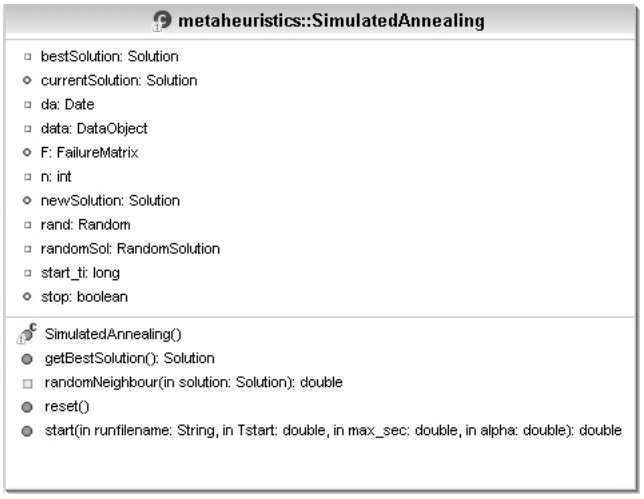
Beskrivelse: Vi har, set i lyset af problemets kompleksitet, udviklet en lidt anderledes `greedyRandomized()` metode. Normal ville man bygge en Restricted Candidate List (RCL) og udvælge en vis procentdel af løsningen fra denne. Da stort set alle demand paths i dette specifikke problem påvirker hinanden er det ikke umiddelbart muligt at udvælge gode kandidater, hvorfor vi har valgt ikke at benytte os af denne metode.
Istedet har vi i local search delen af GRASP valgt, at man kan angive hvor stor en procentdel af det gyldige neighbourhood man ønsker at gennemsøge. Et gyldigt neighbourhood findes, hvor der er mulighed for at vælge mere end en demand path, og hvor der findes en demand.
Det skal også nævnes at der er brugt en slags lokal "tabu-liste"så man ikke besøger den samme nabo mere end en gang. Ovenstående er implementeret ved hjælp af et HashMap med det givne indeks som nøgle, og en boolean der angiver om indekset allerede er forsøgt besøgt. Klassediagrammet ses i figuren nedenfor.



4.1.14 SimulatedAnnealing

Variabler: Ingen.

Beskrivelse: Simulated annealing er opbygget meget traditionelt og er beskrevet i kapitel 3. Klassen bruges kun til sammenligning af resultater med GRASP og der er derfor ikke implemteret særlige metoder til at forbedre algoritmen. Klassen består af en `start()` metode der eksekverer algortimen, en metode til at generere en tilfældig løsning fra neighbourhood'et (`randomNeighbour()`) samt en get() metode til at hente den bedste løsning fundet hidtil (`getBestSolution()`). Klassediagrammet ses i figuren nedenfor.



4.2 Visningsaspekter

For at overholde kravet om at kunne køre programmet på flere operativsystemer har vi begrænset os til at anvende Javas indbyggede grafiske komponenter. Vi har brugt biblioteket `Javax.Swing` der indeholder komponenterne `JPanel`, `JFrame`, `JMenu` og så videre. Til selve optegningen af netværket og grafen for belastning på hvert enkelt link, har vi anvendt biblioteket `java.awt`.

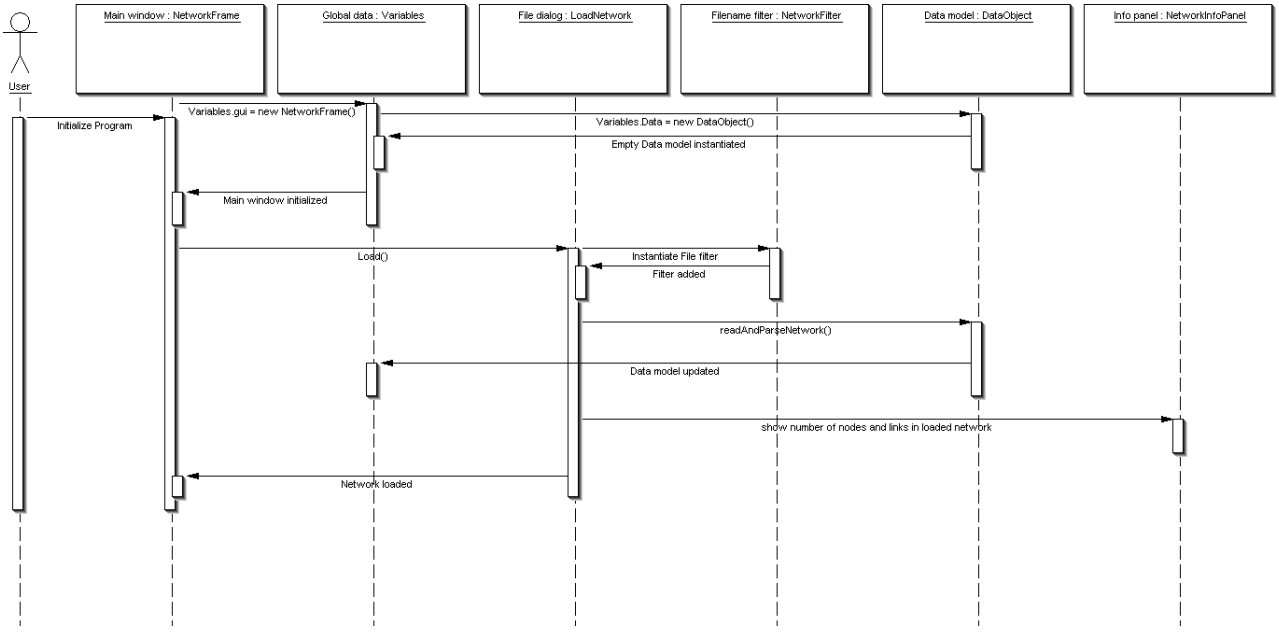
4.2.1 Design valg

Da udvikling af en grafisk brugergrænseflade i Java kan være meget omfattende har vi for brugervenlighedens skyld, holdt antallet af menuer og knapper så lavt så muligt. Den grafiske brugergrænseflade er designet på en sådan måde, at brugeren kan overskue alle programmets dele i samme vindue. Alle funktioner er lagt i en menu placeret i toppen af programmet for genkendelighedens skyld, da dette er meget almindeligt i andre programmet. En brugervejledning til alle programmets funktioner findes i appendix B.
Programmet består af et Panel til at visualisere netværket (DisplayPanel.java), to paneler til at vise demands og demand paths (NetworkInfoPanel.java og DemandInfoPanel.java), et panel til at visualisere belastningen på hvert enkelt link (CapacityDiagramPanel.java) og et panel til at vise den totale belastning på netværket (TotalCapacityPanel.java).

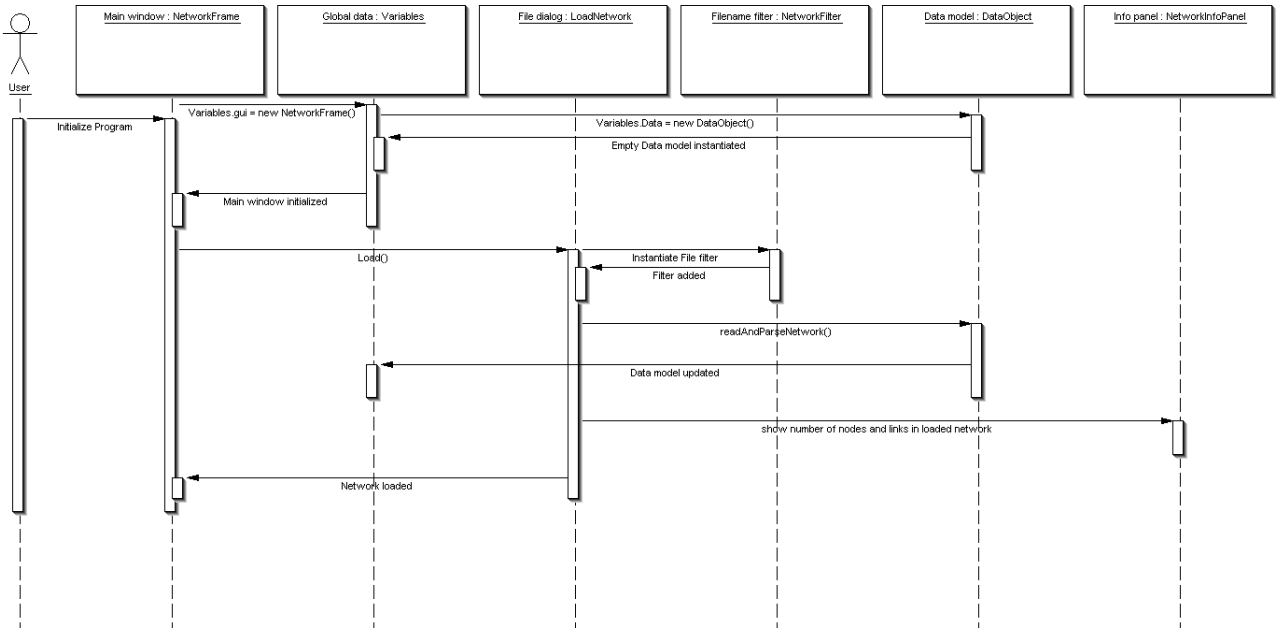
4.3 Styringsaspekter

Alle komponenter til interaktion med brugeren er hentet fra `java.swing.event` biblioteket. Herfra bruges f.eks. `ActionListener` og `WindowListener`. Med disse kan man kontrollere knapper, menuer og nedlukningsfunktioner for vinduer.
Selve interaktionen med brugeren er beskrevet i form af use cases i kapitel 3. Disse er nedenfor konverteret til sekvens diagrammer så kaldene til de forskellige klasser fremgår.

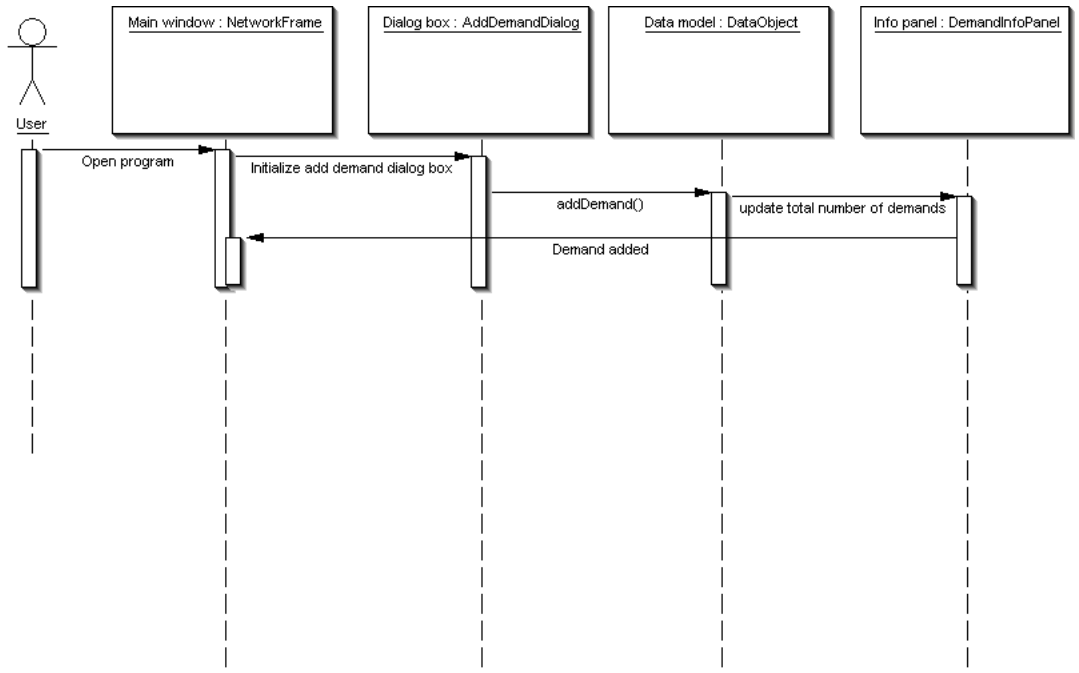
Sekvens diagram 1 - Indlæs netværk



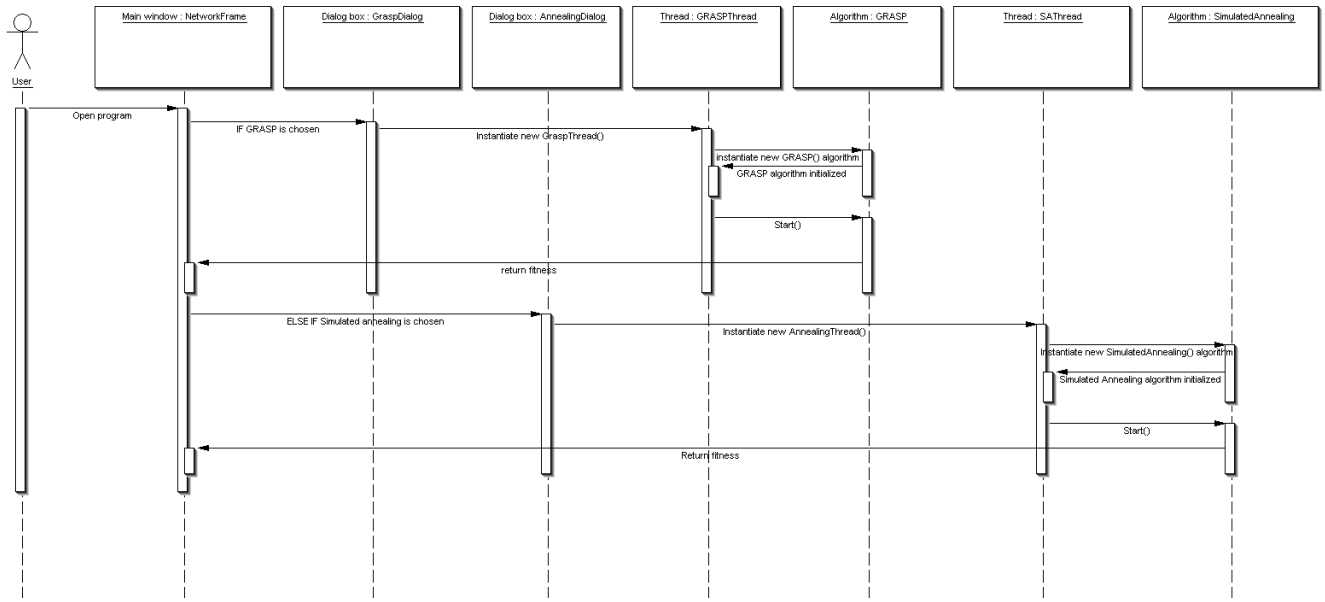
Sekvens diagram 2 - Indlæs demands



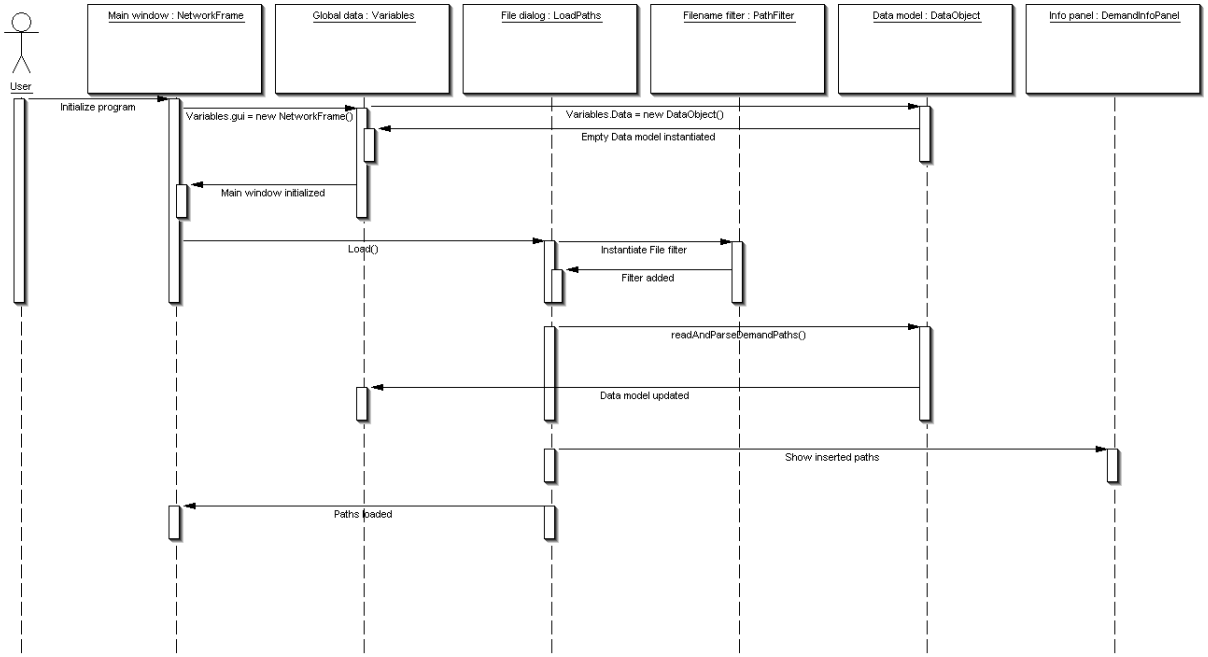
Sekvens diagram 3 - Tilføj enkelt demand



Sekvens diagram 4 - Execute Routing algoritme



Sekvens diagram 5 - Indlæs demand paths



4.3.1 Fejlhåndtering

Fejlhåndteringen i vores program tager udgangspunkt i de fem use cases vist i kapitel 3. Fejl fanges i programmet af Javas indbyggede try catch funktion.

```
try {
    ...
} catch (Exception ex) {
    ...
}
```

Til at informere brugeren om at vedkommende har begået en fejl, benyttes Javas indbyggede dialog *`JOptionPane.showMessageDialog()`* fra *`javax.swing`* biblioteket. Et eksempel på sådan en dialog ses nedenfor.

```
JOptionPane.showMessageDialog(null, "No network", "Error",JOptionPane.ERROR_MESSAGE);
```

Der er i stor udstrækning forsøgt at fjerne valgmuligheder fra brugeren, der vil kunne udløse fejl. Dette er gjort ved at lægge et fil filter (fra biblioteket *`javax.swing.filechooser.FileFilter`*) ind i dialogboksene, når der indlæses forskellig data. Derved ses kun filer med rigtige endelser. For kommunikationsnetværk er fil endelsen eksempelvis .rto.

Kapitel 5

Optimering

Optimeringen er udført ud fra 90/10 princippet, hvor 90% af kørselstiden bliver brugt i 10% af koden. Derfor har vi forsøgt at optimere den kode som er dyrest at eksekvere. Det har vist sig at disse 90% omfatter `addDemandPath()`, `substractDemandPath()` og `getMax()` som alle findes i FailureMatrix.java. Bemærk at når der i det følgende nævnes flere eller færre iterationer er der tale om GRASP eller SA. Yderligere kan for eksempel 4% flere iterationer for en given optimering, måske kun være en 2% flere iterationer på et senere tidspunkt på grund af andre optimeringer.
De følgende sektioner er i kronologisk rækkefølge efter, hvornår den givne optimering blev implementeret i koden.

5.1 Optimering ved delta evaluering

Det første skridt i optimeringen bestod i at gøre det muligt at foretage en delta evaluering. Uden denne ville man for hver løsning skulle genberegne Failure Matricen hvilket efter "mange bække små, gør en stor å"- princippet bliver til en dyr affære.
Delta evaluering er beskrevet i Design-afsnittet. Vi har dog forbedret konceptet yderligere ved at angive for hvilket indeks løsningerne er forskellige. Derved undgår man at gennemsøge begge løsninger hver gang.

5.2 Omvendt delta evaluering

Omvendt delta evaluering betyder, at man bytter om på to løsninger så man på den måde vender tilbage til den "oprindelige"Failure Matrice. Dette fungerer som en slags undo-funktion, hvis man har ændret i en løsning og ønsker at gå tilbage til den tidligere løsning.

5.3 Kopiering af et array

Kopiering af arrays spillede en overgang en større rolle i implementeringen. Vi erfarede i processen, at af de tre muligheder der er for at kopiere arrays i Java, er `System.arraycopy` hurtigst. De to andre muligheder er `clone` og en god gammeldags løkke. Nedenfor tests kopiering af 2000x2000 array over fem runs

```
Using for-loops: 1523.0 milliseconds
Using arrayCopy: 1433.0 milliseconds
Using cloning: 9824.0 milliseconds
```

Cloning er altså en klar taber. Det skal desuden nævnes, at det er en overfladisk kloning, så man kan ikke umiddelbart slippe afsted med at klone et helt flerdimensionelt array. Hvert enkelt array skal derfor klones ved hjælp af en eller flere for-løkker.

5.4 Genberegning af max-værdier

Hvis man har en række af værdier: $\{1, 4, 3, 6, 4, 6\}$ og trækker for eksempel 2 fra det 4. element (6), og man kun har information om at 6 er det største element, så bliver man nødt til gå hele rækken igennem for at bestemme, hvilket element der er størst.
Samme problem bliver vi ofte konfronteret med, når metoden *`substractDemandPath()`* kaldes. Rækken gennemløbes kun hvis det er nødvendigt. Dette gøres med følgende (simplificerede) stykke kode:

```
if(max[bi] == F[bi][pi]) {
    changed = true;
    F[bi][pi] -= cap;
    if(changed) {
        max[bi] = getMax(F, bi);
        changed = false;
    }
}
```

5.5 Løbende opdatering af fitness værdien

En lille detalje i koden er en løbende opdatering af fitness værdien.
Denne ændrer sig givetvis kun når en af værdierne i max arrayet ændrer sig. f.eks.:

```
if(F[bi][pi] + cap > max[bi]) {
    fit += F[bi][pi] + cap - max[bi];
```

Her ved vi at max vil ændre sig og øger derfor fitness værdien tilsvarende.

5.6 Hurtige løkker

En anden lille men betydelig detalje er, at er de "travle"for-løkker baglæns. Med dette menes for-løkker i *addDemandPath()* og *substractDemandPath()* samt *getMax()*. En for-løkke vil altså ændre sig fra:

```
for(int i = 0; i < L; i++)
```

til

```
for(int i = L-1; i >= 0; i--)
```

En hurtig test med et relativt kort array, som bliver gennemløbet 25 millioner gange, tager 20.5 sekund med et baglæns loop, imens det tager 20.92 sekunder med et konventionelt loop. Det er altså ca. 2% langsommere. Dette skyldes at det er hurtigere at sammenligne med 0 i forhold til alle andre tal.
Alt ialt gav dette ca 5% flere iterationer.

Vi har dog valgt ikke at køre samtlige for-løkker baglæns, da de er svære at læse. Til at bestemme hvor de travle for-løkker findes i koden har vi benyttet en Java profiler. Profileren muliggør yderligere optimering af for-løkkerne ved brug af nedenstående.

```
for(int i = L; --i >= 0;)
```

Dette giver ca. 3,7% flere iterationer. Det er faktisk muligt at optimere yderligere ved brug af:

```
    try {
        for (i=A.length; ; --i)
            A[i] = someVar;
    } catch (ArrayIndexOutOfBoundsException e) {}
```

***quote**: This can be considerably faster, but if you have short arrays, it's less optimal since throwing and catching a new Exception takes 9,500ns and doing a >= int compare takes about 250ns (on a 200MHz UltraSPARC), so the breakpoint is about 40 elements.*
Med andre ord burde dette kun være hurtigere på USANetwork og FranceNetwork, da de begge har over 40 Links. Vi har derfor ikke benyttet os af dette. Koden bliver også hurtigt meget uoverskuelig, hvis man bruger variablen `i` til at tilgå flere forskellige arrays, da hver enkelt skal have en `try-catch` omkring sig.

5.7 Globale og lokale referencer

En anden detalje er at man kan vinde ganske få iterationer ved at undgå ikke-lokale referencer. For eksempel er `F` en global variabel i FailureMatrix.java. Man kan vinde knap 1.5% flere iterationer ved at tilføje ganske lidt i koden for *`addDemandPath()`* og *`substractDemandPath()`*, som vist her. I starten:

```
double[][] F = this.F;
```

I slutningen:

```
this.F = F;
```

Vi har valgt blot at lave lokale referencer til F i de to omtalte metoder.

5.8 Max counter

Indførslen af *`maxC`* arrayet kom til sent i projektet, og dets virkemåde er dækket i kapitel 4. En test viser at vi vandt ca. 41% flere iterationer ved at benytte en forsvindende lille mængde ekstra hukommelse.

5.9 Manuel delta evaluering

Med indførsel af flere demands pr. node par gik vi bort fra en "automatisk"delta evaluering og over til en manuel delta evaluering da delta evaluringsmetoden krævede to Solutions som input. Ved indførsel af flere demands pr. node par blev Solution en mere kompleks størrelse, og derfor noget, vi ville undgå at lave kopier af.
Typisk vil man hver gang man kaldte *`randomNeighbour()`* (i GRASP eller

SA) lave en kopi af den nuværende Solution, ændre i denne, og så foretage delta evaluering, eventuelt efterfulgt af en omvendt delta evaluering.

5.10 SmartCopy

Kopiering af Solutions kan dog ikke helt undgås. Hvergang en ny bedste løsning findes, skal denne kopieres. SA laver desuden en kopi af den aktuelle løsning ved hvert kald af `randomNeighbour()`.
Til dette blev metoden `smartCopy` opfundet, som kun kopiere forskelle mellem den gamle bedste løsning og den nye bedste løsning.

5.11 Yderligere optimering

Det følgende er ideer til, hvad der yderligere kan filføjes for at øge effektiviteten af vores data model.

`F` (variablen `F` i FailureMatrix.java) er et 2-dimensionelt *double* array. Alt afhængig af data kunne vi klare os med 2-dimensionelt *short* array som ville fylde $4^2 = 16$ gange mindre, og derfor også være ca. 16 gange hurtigere. Et 2-dimensionelt *float* array ville tilsvarende være 4 gange hurtigere.
En anden optimering man kunne foretage på `F` ville være at have det i 1 dimension, hvilket ville give hurtigere tilgang i men en mere kompleks kode.

Yderligere kunne koden optimeres ved at lave vores egen ikke-synkroniserede Vector klasse, da synkroniserede metoder kan være op til 10 gange langsommere end tilsvarende ikke synkroniserede metoder.
***quote**: synchronized methods: These are by far the slowest, since an object lock has to be obtained, and take around 1,500ns.*[1]

Den grafiske brugergrænseflade benytter et såkaldt event loop som med jævne intervaller spørger forskellige dele af den grafiske brugergrænseflade om der er foretaget klik på knapper. Selvom en profiling viser denne er suverænt i toppen med 50%, så er det næppe problematisk at den bruger halvdelen af cpu kraften.
En JIT (Just-In-Time), eller en AOT (Ahead-Of-Time) compiler som har fokus

[1] http://www.cs.cmu.edu/∼jch/java/speed.html

på de enkelte operationer som bliver udført oftest i koden kunne have en markant effekt.
Vi kunne selvfølgelig have taget skridtet fuldt ud og benytte et hurtigere sprog som c++.

Multi-threading kan man sagtens forestille sig vil forbedre GRASP. For eksempel vil samtlige tråde kunne starte samtidigt og på "aftalte"tidspunkter konfererer og derefter genstarte tråde som har opnået dårlige(re) løsninger.

5.12 Profiling

Vi har benyttet en profileren som er indbygget i Eclipse og er skrevet af Sun Microsystem. For en omfattende beskrivelse af hvordan den benyttes henvises til[2].

Profiling er foretaget med tilføjelse af nedenstående linje ved eksekvering af programmet.

```
Java main.java -Xrunhprof:cpu=samples,depth=4,interval=5,thread=y
```

som poller med 5 millisekunders interval på hvor ofte et givent kald ligger øverst i call-stacken. depth betyder at man tracer op igennem 4 metoder, f.eks:

```
network.FailureMatrix.substractDemandPath(FailureMatrix.java:215)
metaheuristics.GRASP$HillDescender.randomNeighbour(GRASP.java:271)
metaheuristics.GRASP$HillDescender.start(GRASP.java:226)
metaheuristics.GRASP$HillDescender.access$1(GRASP.java:211)
```

hvor `substractDemandPath` er blevet kaldt af `randomNeighbour` som er blevet kaldt af `start` metoden i `HillDescender` klassen.

En udskrift af de 10 dyreste operationer ses i nedenstående.

```
rank   self  accum   count trace method
   1 43.23% 43.23%     821 300297 sun.awt.windows.WToolkit.eventLoop
   2  5.69% 48.92%     108 300443 network.FailureMatrix.getMax
   3  3.90% 52.82%      74 300452 network.FailureMatrix.addDemandPath
   4  3.69% 56.50%      70 300444 network.FailureMatrix.substractDemandPath
   5  3.21% 59.72%      61 300109 java.lang.ClassLoader$NativeLibrary.load
   6  3.05% 62.77%      58 300456 network.FailureMatrix.substractDemandPath
   7  2.74% 65.51%      52 300448 network.FailureMatrix.addDemandPath
   8  2.58% 68.09%      49 300454 network.FailureMatrix.addDemandPath
   9  2.37% 70.46%      45 300250 sun.awt.Win32GraphicsEnvironment.initDisplay
  10  2.32% 72.78%      44 300453 network.FailureMatrix.addDemandPath
```

[2]http://java.sun.com/developer/technicalArticles/Programming/HPROF.html

Kapitel 6

Test

I dette kapitel gennemgås strategien for opbygning af test cases der skal eliminere fejl og dermed sikre at værktøjet gøres så brugbart som muligt. Udførelsen af disse er også beskrevet. Enkelte er af hensyn til læsbarhed og overskuelighed, vedlagt i appendix D.

6.1 Test strategi

Da omfanget af en komplet test for et projekt som dette langt overstiger den afsatte tid, har vi istedet udvalgt enkelte væsentlige metoder og klasser. Der vil blive foretaget udelukkende funktionelle tests af data modellen, da omfanget af simple strukturelle tests er meget stort grundet programmets kompleksitet. En egentlig test af den grafiske brugergrænseflade udføres ikke, da der istedet er vedlagt en brugervejledning til værktøjet i appendix B. Til test af de implementerede meta heuristikker som routingsalgoritmer vil der blive foretaget performance test i form af parameter tuning for forskellige netværk. Til kvalitetskontrol benyttes de tilhørende teoretiske lowerbounds genereret af GAMS scriptet.
I den funktionelle test af data modellen er der kun foretaget test af metoder fra klasserne DemandPath.java, FailureMatrix.java og Hash.java. Metoderne er valgt på baggrund af antallet af kald, hvorfor metoder anvendt i for eksempel evalueringsfunktionen der eksekveres et stort antal gange er valgt. Yderligere er metoder til opdatering af failure matricen valgt, idet vigtigheden i at denne virker korrekt overskygger andre dele af programmet.

6.2 Udvalgte områder til funktionel test

- Metoden *validate()* fra klassen DemandPath.java der bruges til at validere en demand path.
- Metoden *getMax()* fra klassen FailureMatrix.java der bruges til at finde max værdien for en given række i failure matricen.
- Metoderne *addDemandPath()* og *substractDemandPath()* fra klassen FailureMatrix.java der bruges til at tilføje og fjerne demand paths til failure matricen.
- Metoden *initFailMatrix(Solution)* fra klassen FailureMatrix.java der bruges til at initialisere en failure matrix med en løsning S.
- Metoderne *testHash()* og *makeHash()* fra klassen Hash.java. *testHash()* udskriver hashnøgler for alle node par i et netværk af n nodes. Ved test af denne kan man sikre sig at alle demands og demand paths indekseres korrekt. *makeHash()* benyttes i *testHash()* og genererer et indeks på baggrund af en start og slut node.
- Der skal yderligere foretages en mere omfattende test af en Solutions gyldighed (feasibility).

De egentlige tests af disse områder findes i appendix D.

6.3 Data sets

Til funktionelle tests af data modellen benyttes nedenstående data.

For alle områder benyttes følgende kommunikationsnetværk:

```
3
3
A   2   3   -1
B   4   3   -1
C   1   2   -1
SO1     A   B   1
SO2     A   C   1
SO3     B   C   1
```

For alle områder på nær *initFailMatrix()* og den omfattende test en Solutions gyldighed benyttes følgende demand paths:

```
0 2
//0 1 2 2 0
0 2 -1 1

0 2
//0 2 2 1 0
1 -1 2 0

1 2
//1 2 2 0 1
2 -1 1 0
```

For *initFailMatrix()* og den omfattende test en Solutions gyldighed benyttes ovenstående demand paths, derudover benyttes en ekstra demand path som kan ses nedenfor.

```
1 2
//1 0 2 2 1
0 1 -1 2
```

For alle områder på nær *initFailMatrix()* og den omfattende test af Solutions gyldighed benyttes følgende demands:

```
   36
3

1   3    4
1   3    6
1   3    2
2   3    1
2   3    3
2   3    5
```

For *initFailMatrix()* og den omfattende test en Solutions gyldighed benyttes der for overskuelighedens skyld kun 2 demands per node par. Disse demands ses nedenfor.

```
   30
2

1   3    4
1   3    2
2   3    1
2   3    5
```

6.4 Test af meta heuristikker

Testen af de implementerede meta heuristikker er delt op i 2 dele. Første del går ud på at finde de parametre, der giver de bedste resultater for algoritmerne. Dette kaldes for parameter tuning.
Anden del er den endelig test, hvor problemer løses på de tilgængelige netværk og algoritmernes performance sammenlignes. Yderligere holdes begge op imod den teoretiske lowerbound som er genereret for det specifikke problem ved hjælp af GAMS scriptet omtalt i kapitel 2 og appendix C. I tabellen nedenfor ses de netværk vi har arbejdet med. Det skal nævnes at ét af de anvendte netværk er

Netværk	Paths	Nodes	Links	|S|
USANetwork	2653	28	45	406
COST239	1370	11	26	66
FranceNetwork	6836	43	71	946
PanEuropean	371	13	21	91
Norway_sln2	3302	27	51	378
ta1_sln2	3686	24	51	300

Tabel 6.1: Netværks data

af en størrelse der ikke gør GAMS scriptet istand til at finde en lowerbound, især for flere demands pr. node par. Derfor er dette netværk udeladt fra testen. Det drejer sig om FranceNetwork.

6.4.1 Parameter tuning

Til parameter tuning er valgt netværket "PanEuropean" og netværket "USA-Network", begge med 5 demands per node par og en fast maksimal volumen på 250.
De tilgængelige netværk kan anskues på 2 forskellige måder, enten udfra deres størrelse angivet ved antallet af links og nodes, eller udfra deres kompleksitet angivet ved antallet af demand paths samt antallet af demands per node par.
Dette giver med et fast antal demands per node par 4 forskellige slags netværk. Et lille og simpelt netværk (PanEuropean), et lille men komplekst netværk (COST239), et stort men simpelt netværk (USANetwork) samt et stort og komplekst netværk (FranceNetwork).
Baseret på erfaring om at komplekse netværk generelt ikke lader sig løse tæt på lowerbound, har vi valgt at fokusere på et stort og et lille netværk med samme lave kompleksitet.

Vi har valgt at tage den samlede køretid med i parameter tuningen, dels for at se hvordan det påvirker de andre parametre, men også for at se om der er noget at hente i forhold til størrelsen af netværket. I den statistiske test vil GRASP og Simulated Annealing blive givet samme køretid.

Parameter tuning for GRASP

GRASP har teknisk set kun 2 parametre. Begge er tilknyttet lokal søgningsdelen. Den ene parameter angiver, hvor mange lokalsøgninger der ønskes foretaget, imens den anden angiver hvor stort nabolaget skal være (0.50 svarer til 50%). I nedenstående tabel ses de valgte parametre for parameter tuningen af GRASP.

Parameter	Mulige Værdier	Test Værdier
GRASP runtime, t	t > 0	$\{3, 5, 7, 10\}$
Lokal søgninger, l	$l > 0$	$\{3, 5, 10\}$
Størrelse af nabolag, N	$1 \geq N > 0$	$\{0.10, 0.25, 0.50, 0.75, 1.00\}$

Tabel 6.2: Parameter tuning for GRASP

Resultaterne ses i appendix D. På baggrund af disse vælges 3 lokal søgninger og et nabolag på 10% til den statistiske sammenligning. Det er aldrig godt at ende med at skulle vælge et eller flere extremum fra en liste af givne parametre i en parameter tuning.
Det vides ikke om færre lokale søgninger end 3, eller et nabolag mindre end 10% vil forbedre parametrene. Vores resultater viser dog at det er brøkdele af en promille der adskiller kvaliteten for de forskellige parametre, hvorfor det valgt ikke at undersøge yderligere parametre.

Parameter tuning for Simulated Annealing

De to parametre T_0 og α afgører tilsammen sandsynligheden for at Simulated Annealing accepterer en given dårlig løsning. I nedenstående tabel ses de valgte parametre for parameter tuningen af SA.

Parameter	Mulige Værdier	Test Værdier
SA runtime, t	t > 0	$\{3, 5, 7, 10\}$
Start temperatur, T_0	$T_0 > 0$	$\{100, 500, 1000, 2500\}$
alpha α	$\alpha < 1$	$\{0.8, 0.90, 0.95, 0.99\}$

Tabel 6.3: Parameter tuning for SA

Resultaterne kan ses i appendix D. Ingen parameter sæt var markant afgørende for resultatet af Simulated Annealing. Derfor sættes $T_0 = 400$ og $\alpha = 0.875$ i den statistiske sammenligning med GRASP.

6.4.2 Statistisk sammenligning af algoritme resultater

Ligesom i parameter tuningen er demands valgt med en max volumen på 250. Parametrene for GRASP sættes til (0.10, 3). Parametrene for SA sættes til (400, 0.875). De endelige resultater ses i tabellerne nedenfor.

	GRASP				Simulated Annealing			
Demands	Best	Mean	Gap%	σ%	Best	Mean	Gap%	σ%
1	13203	13468	19,23%	0,67%	12977	13217	17,00%	1,25%
2	24194	24297	10,69%	0,21%	23590	23957	9,14%	0,70%
5	62635	62793	5,85%	0,13%	61332	61734	4,07%	0,33%
10	128745	128960	3,94%	0,09%	127149	127712	2,93%	0,26%
25	322005	322293	2,47%	0,05%	320393	321974	2,37%	0,17%
50	650415	650786	2,08%	0,03%	650406	651889	2,25%	0,11%

Tabel 6.4: COST239

	GRASP				Simulated Annealing			
Demands	Best	Mean	Gap%	σ%	Best	Mean	Gap%	σ%
1	33054	33094	0,29%	0,04%	33179	33449	1,38%	0,28%
2	59339	59387	0,13%	0,06%	59549	59833	0,89%	0,23%
5	157277	157422	0,35%	0,07%	157589	158228	0,86%	0,20%
10	315666	315927	0,61%	0,06%	315721	316403	0,76%	0,13%
25	816412	817093	0,46%	0,04%	817992	818935	0,69%	0,07%
50	1622235	1625222	0,71%	0,06%	1629966	1631608	1,11%	0,03%
100	3255990	3265260	0,76%	0,13%	3291816	3297753	1,77%	0,06%
250	8269456	8304383	1,61%	0,22%	8575521	8599084	5,22%	0,13%

Tabel 6.5: PanEuropean

	GRASP				Simulated Annealing			
Demands	Best	Mean	Gap%	σ%	Best	Mean	Gap%	σ%
1	74596	74786	1,69%	0,12%	74778	75099	2,12%	0,22%
2	478326	479085	1,57%	0,08%	478769	479808	1,73%	0,15%
4	979203	980755	1,38%	0,09%	980904	982474	1,55%	0,09%
6	1475765	1478755	1,38%	0,08%	1478478	1480997	1,53%	0,08%
10	2478322	2482821	1,47%	0,09%	2484547	2487346	1,66%	0,07%
25	6275830	6284743	1,69%	0,08%	6306258	6311120	2,11%	0,05%

Tabel 6.6: USANetwork

	GRASP				Simulated Annealing			
Demands	Best	Mean	Gap%	σ%	Best	Mean	Gap%	σ%
1	121667	122189	3,09%	0,18%	122272	123073	4,04%	0,44%
2	258783	259269	2,21%	0,11%	259747	260850	2,83%	0,18%
5	637216	638355	2,21%	0,10%	639059	641009	2,64%	0,13%
10	1263191	1266128	2,21%	0,14%	1273251	1276330	3,03%	0,13%
25	3226767	3232019	2,49%	0,10%	3303683	3303683	4,76%	0,11%
50	6440880	6451699	3,00%	0,12%	6604151	6617195	5,64%	0,07%

Tabel 6.7: ta1_sln2

	GRASP				Simulated Annealing			
Demands	Best	Mean	Gap%	σ%	Best	Mean	Gap%	σ%
1	195362	196069	3,65%	0,20%	197317	198709	5,04%	0,40%
2	377869	379003	3,00%	0,12%	379974	383214	4,41%	0,26%
4	777031	779203	3,03%	0,14%	783869	786688	4,02%	0,25%
10	1973595	1978686	2,85%	0,13%	1998272	2002803	4,10%	0,25%
25	5011890	5024231	3,40%	0,14%	5201644	5220901	7,44%	0,21%
100	20599544	20753697	7,14%	0,34%	22934055	23012444	18,80%	0,19%

Tabel 6.8: norway_sln2

Gap angiver hvor langt resultaterne er fra lower bound og σ% angiver den relative standard afvigelse i procent.

6.5 Diskussion af resultater

I **COST239:** netværket viser SA sig faktisk at være bedst, hvorimod GRASP er mere stabil. Dette kunne skyldes at COST239 er et netværk med høj kompleksitet, og man kunne måske med fordel have brugt mere tid i local search. Gennemgående er resultaterne acceptable for begge algoritmer som bliver relativt bedre når antallet af demands øges.

I **PanEuropean** netværket er GRASP markant bedre og væsentligt mere stabil end SA. Det generelt dårligere resultat for 250 demands i begge algoritmer kan forklares ved at der ikke har været nok tid til løse et problem af denne størrelse.

I **USANetwork** er GRASP en smule bedre end SA. Man kunne forestille sig at GRASP ville få væsentligt overtaget hvis begge algoritmer fik lov til at køre lidt længere tid.

I **ta1_sln2** netværket ses samme mønster som i USA netværket. GRASP er dog en smule bedre her.

I **norway_sln2:** netværket er GRASP tydeligt bedre og mere stabil end SA. Som i PanEuropean netværket er 100 demands for stort et problem at løse inden for den begrænsede mængde tid.

6.5.1 Always a bug

I performance testen har det for GRASP i særlige tilfælde være muligt at nå under den teoretiske lowerbound beregnet af GAMS scriptet. Dette eftervises med et af de mindste netværk, PanEuropean, da det er tilstrækkeligt lille til at vi med stor sandsynlighed finder næsten optimale løsninger.
Nedenstående er resultatet af 4 kald af GRASP på 4 forskellige sæt af demands. Første kald er med max volumen 1 og 1 demand per node par. Andet kald er med max volumen 2 og 1 demand per node par. Tredje kald er med 2 demands per node par og max volumen 1 og sidste kald er med 2 demands pr. node par og max volumen 2. Ved sidst nævnte tilfælde opstår problemet.

```
GRASP: 263.0 1,15%
GRASP: 402.0 2,74%
GRASP: 525.0 ,96%
```

```
GRASP: 750.0 -,78%
```

Da det ikke burde være muligt at nå ned under en teoretisk lowerbound har vi måtte bruge alternative metoder til at debugge problemet. Da det ikke vides om fejlen findes i GAMS scriptet eller vores eget program, har vi valgt at undersøge om løsningen rent faktisk eksisterer (er feasible). Til dette er foretaget en meget omfattende test på et lille netværk bestående af 5 nodes. Til dette netværk genereres et antal demands af GAMS scriptet (flere demands pr. node par og volume <> 1) og GRASP sættes til at beregne en løsning. En funden løsning under lowerbound (i dette tilfælde 68 som skulle ligge 0.96% under lowerbound) valideres herefter manuelt ved at tilføje hver demand til løsningsvektoren og validere den tilhørende failure matrix. Dette er gjort i appendix D.2.3. Det viser sig at løsningen 68 rent faktisk eksisterer hvilket giver anledning til at tro at der er fejl i GAMS scriptet. Dette er dog ikke endeligt påvist, men eftersom at løsningen rent faktisk eksisterer må det formodes at fejlen ikke er at finde i vores program.

Kapitel 7

Konklusion

Vi har med dette bachelor projekt udviklet en applikation, der effektivt og sikkert kan route trafikken i et kommunikationsnetværk. Både den visuelle såvel som den underliggende model opfylder vores krav. Vi har fra start af valgt ikke at lægge særlig vægt på den visuelle del, hvilket vi er tilfredse med da udvikling af en grafisk brugergrænseflade i Java er meget tidskrævende.

GRASP var som forventet bedre i næsten alle henseender end Simulated Annealing. Begge algoritmer var i stand til at producere tilfredsstillende resultater i løbet af kort tid, hvilket er imponerende med antagelsen om at problemet er NP-komplet.

Vores resultater er overraskende gode, men viser det sig at den teoretiske lowerbound beregnes for højt af GAMS scriptet, påvirkes samtlige performance resultater for de valgte meta heuristikker negativt og er ikke helt lige så gode som angivet. Resultaterne viser dog at vi ikke er langt fra en optimal løsning - måske så tæt på man kan komme.

7.1 Fremtidige forbedringer

En udbyder kører typisk med modulærer kapaciteter på deres netværk, og med et princip om at prisen pr. mbit falder med større kapaciteter. At implementere dette ville være et skridt imod en mere realistisk simulering.
Vi har allerede banet vejen med Cost klassen, og formoder at det meste hvis ikke alt arbejdet vil foregå i FailureMatrix.java.

Ydermere kunne det være interessant at kunne beregne den sparede belastning ved brug af Single Backup Path Protection set i forhold til traditionel routing.

Litteratur

[1] Feo, Thomas A. and Resende, Mauricio G.C.
Greedy Randomized Adaptive Search Procedures
Journal of Global Optimization 6: 109-133, 1995.

[2] Springer, Edmund K. Burke, Graham Kendall
Search Methodologies ISBN: 0-387-23460-8

[3] Cormen, T. H., Leiserson, C. E. and Rivest, R. L.
Introduction to Algorithms. ISBN 0-262-53091-0

[4] Thomas Stidsen, Bjørn Petersen, Kasper Bonne Rasmussen, Simon Spoorendonk, Martin Zachariasen, Franz Rambach, Moritz Kiese
Optimal Routing with Single Backup Path Protection

[5] Stuart J. Russell and Peter Norvig
Artificial Intelligence - A Modern Approach ISBN 0-13-103805-2

[6] Randy L. Haupt, Sue Ellen Haupt
Practical Genetic Algorithms ISBN 0-471-45565-2

Bilag A

Ekstern data

A.1 Kommunikationsnetværk

Filtype: .rto
Fil formatet for et kommunikationsnetværk er

```
5
7
A   2   3   -1
B   4   3   -1
C   1   2   -1
D   5   2   -1
E   3   1   -1
S01     A   B   1
S02     A   C   1
S03     B   C   1
S04     B   D   1
S05     C   D   1
S06     C   E   1
S07     D   E   1
```

- 1. linje angiver antallet af nodes i et netværket (5)
- 2. linje angiver antallet af links i netværket (7)
- de næste 5 linjer angiver nodes i formatet (navn, x-koordinat, y-koordinat, -1 (ses bort fra)
- de næste 7 linjer angiver links i formatet (navn, start node, slut node, cost)

A.2 Demand paths

Filtype: .rto.paths
Fil formatet for en demand path er

```
1 2
//1 2 2 3 1
2 -1 4 3
```

- 1. linje angiver start node, slut node
- 2. linje angiver primær og backup path (adskilt af 2 ens tal i rækkefølge) som nodes
- 3. linje angiver primær og backup path (adskilt af -1) som links

A.3 Demands

Filtype: .rto.gms.dem
Fil formatet for en demand er

```
    102.27
1

1   2     1
1   3     1
1   4     1
1   5     1
1   6     1
1   7     1
```

- 1. linje angiver den teoretiske lowerbound hvis alle demands indsættes i det tilhørende kommunikationsnetværk
- 2. angiver antallet af demands pr. node par. Altså mellem (1 2), (1 3) osv.
- Resten af linjerne angiver demands separeret med linjeskift. Demands er på formen (start node, slut node, volume)

Bilag B

Brugervejledning til programmet

B.1 Installation og opstart af programmet

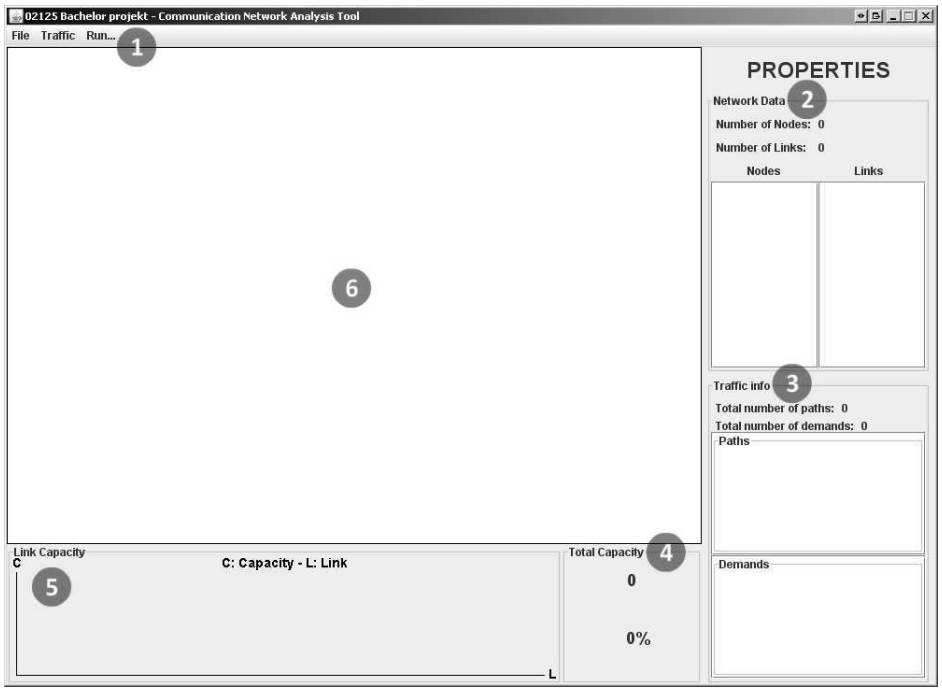
Applikationen er pakket i en `.jar` fil (`main.jar`). Inden programmet kan køres skal man sikre sig at Sun's Java virtual machine (version 1.5 eller nyere) er installeret på den lokale maskine. Er dette ikke tilfældet kan denne hentes her

`http://java.sun.com/javase/downloads/index.jsp.`

Når Java er installeret kan man køre programmet ved at stille sig i biblioteket hvor programfilen ligger og skrive følgende

```
java -jar main.jar
```

Når programmet åbner dukker hovedvinduet op som vist på figuren nedenfor.



Programmet er opdelt i en række sektioner.

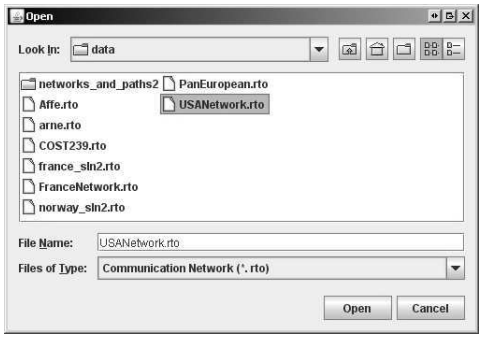
1. Menu med programmets funktioner
2. Information om indlæst netværk
3. Information om indlæste demands og demand paths
4. Information om den samlede belastning på netværket
5. Panel til grafisk visualisering af belastning på hvert enkelt link i et netværk
6. Panel til visualisering af netværk

B.2 Gennemgang af program funktioner

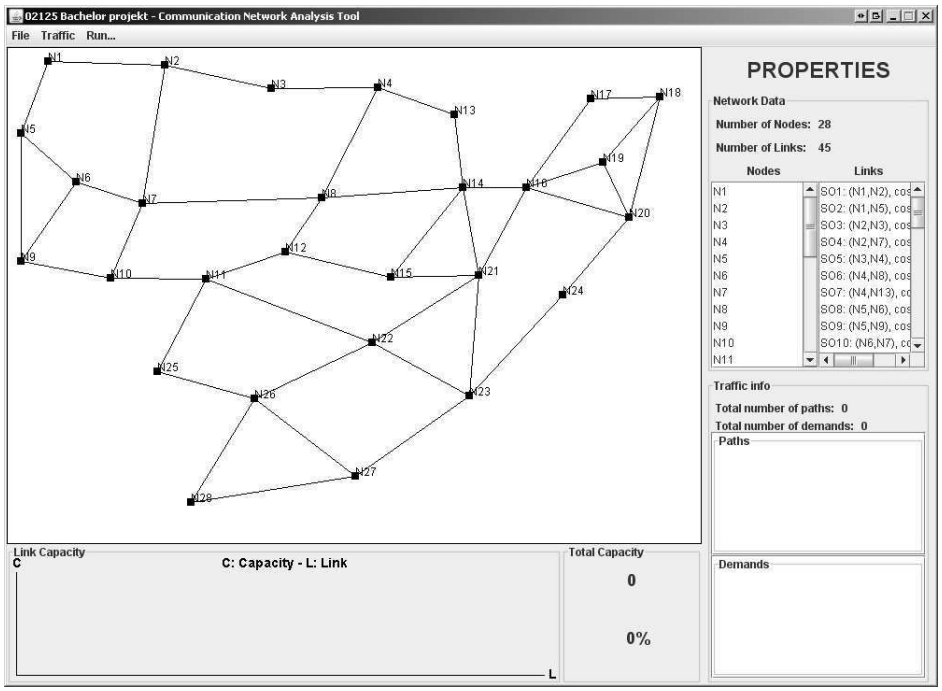
Alle programmets funktioner findes i menuen placeret i toppen af hovedvinduet. I det følgende gennemgås hver enkelt.

Load network

Denne funktion bruges til at indlæse et kommunikationsnetværk til programmet fra en ekstern fil. Når der vælges File -> Load Network... åbner en dialogboks som vist nedenfor og det ønskede netværk kan vælges.

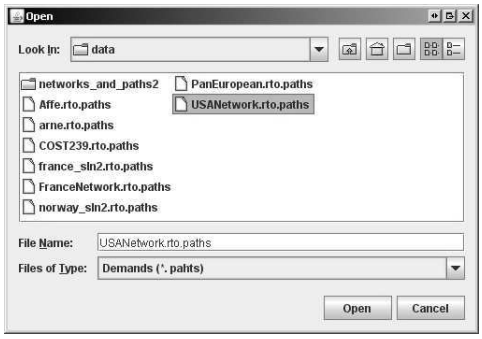


Marker det ønskede netværk, tryk open. Programmet vender nu tilbage til hovedvinduet med det indlæste netværk. Et eksempel på hvordan hovedvinduet kan se ud ses i figuren nedenfor.

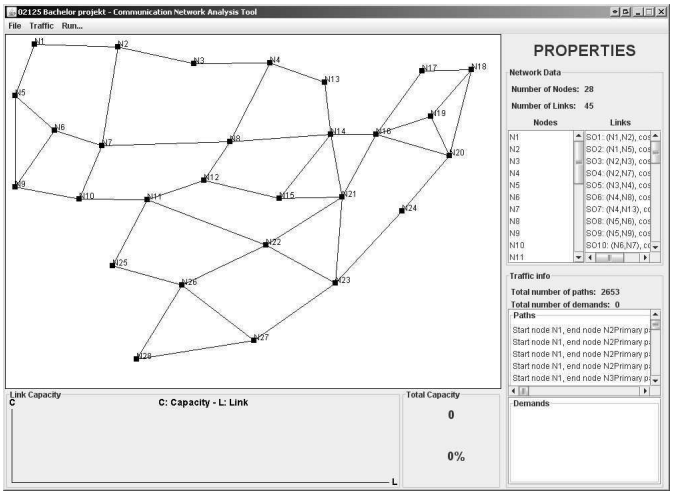


Load paths

Denne funktion bruges til at indlæse et demand paths til programmet fra en ekstern fil. For at bruge funktionen kræves det at der er indlæst et netværk. Når der vælges File -> Load Paths... åbner en dialogboks som vist nedenfor.

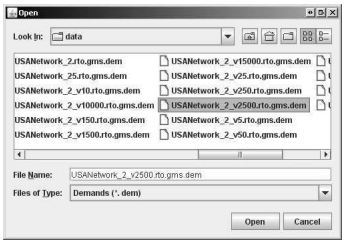


Marker den ønskede paths fil, tryk open. Programmet vender tilbage til hovedvinduet med de indlæste demand paths. Et eksempel på hvordan hovedvinduet nu kan se ud ses i figuren nedenfor.

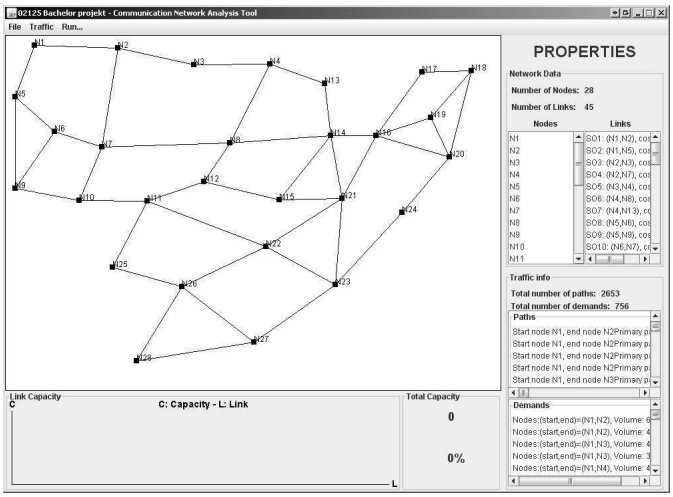


Indlæs demands

Denne funktion bruges til at indlæse demands til programmet fra en ekstern fil. For at bruge funktionen kræves det at der er indlæst et netværk og en række demand paths. Når der vælges File -> Load Demands åbner en dialogboks som vist nedenfor.

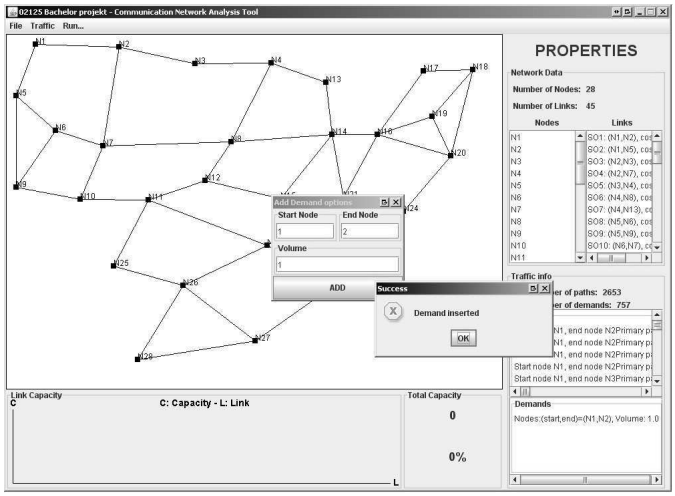


Marker valgte demands, tryk open. Programmet vender tilbage til hovedvinduet med de indlæste demands. Et eksempel på hvordan hovedvinduet nu kan se ud ses i figuren nedenfor.



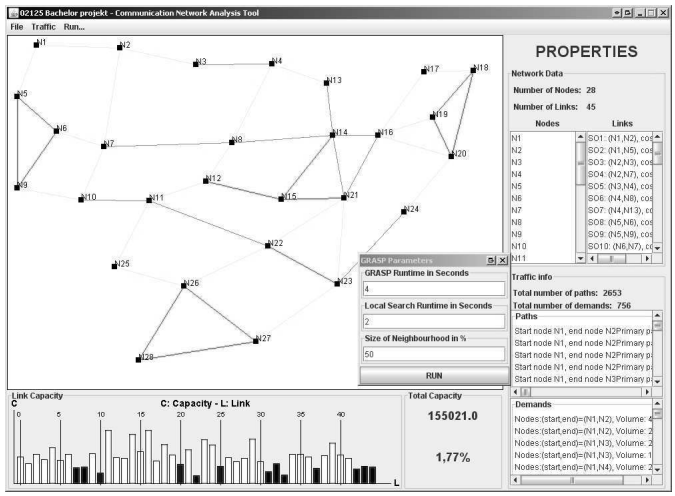
Add demand

Denne funktion bruges til at tilføje en enkelt demand til netværket. For at køre funktionen kræves det at der er indlæst et netværk. Når der vælges Traffic -> Add Demand vil en dialog åbne hvori oplysninger om demand'en kan indtastes. Når ADD vælges vil det i hovedvinduet fremgå at en demand er tilføjet og brugeren informeres herom via en ny dialogboks. Et eksempel på hvordan hovedvinduet nu kan se ud ses i figuren nedenfor.



GRASP og Simulated Annealing

Denne funktion bruges til at køre routing algoritmen GRASP eller Simulated Annealing på indlæste data. For at bruge funktionerne kræves det at der er indlæst et netværk, der er indlæst et antal demand paths og én eller flere demands. Når der vælges Run... -> Routing Algorithm og enten GRASP eller Simulated Annealing vælges, vil en dialogboks åbne hvori brugeren kan vælge parametre for algoritmen. Efter der vælges RUN i dialogboksen, vises algoritmens resultater i hovedvinduet. et eksempel på dette ses i figuren nedenfor.



B.3 Fejl meddelelser

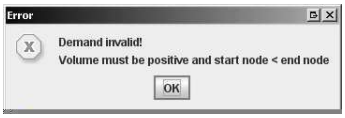
I det følgende beskrives en række fejl meddelelser man kan konfronteres med ved forkert brug af programmet.

Forsøges det at indlæse demand paths uden der er indlæst et netværk informeres brugeren om dette via nedenstående dialogboks.

Forsøges det at indlæse ugyldige paths informeres man om dette via nedenstående dialogboks.

Forsøges det at indtaste ulovlige parametre i de forskellige dialoger informeres man om dette via nedenstående dialogboks.

Forsøges det at tilføje en demand med forkerte start og slut nodes informeres man om dette via nedenstående dialogboks



Forsøges det at tilføje en demand uden at der er tilføjet demand paths informeres man om dette via nedenstående dialogboks

Bilag C

Brugervejledning til GAMS script

C.1 Kørsel af script

Scriptet kan kun køres fra g-baren på Danmarks tekniske universitet. Scriptet eksekveres med følgende kommando:

```
gams bound_sbpp.gms u1=network.rto.gms u2=network.rto.gms.paths
u3=network.rto.gms.dem u4=1 u5=1
```

Hvor bound_sbpp.gms er filnavnet for scriptet, u1 er et kommunikationsnetværk, u2 er tilhørende demand paths, u3 er placeringen på den fil demands bliver genereret i, u4 er antallet af demands pr. node par og u5 er den maksimale volumen demands bliver genereret med.

Bilag D

Test dokumentation

D.1 Funktionelle tests

D.1.1 validate()

Forudsætninger for at teste metoden:

- Der skal være indlæst et netværk

I tilfælde af at man forsøger at tilføje en ugyldig demand path kaldes metoden *`validate()`* i DemandPath.java. Demand Path'en printes i konsollen hvis den er ugyldig. I DataObject.java vil en ugyldig path ikke blive indlæst.
Følgende ugyldige demand forsøges nu indlæst.

```
1 2
//0 1 1 2
0 -1 2
```

Backup path'en forbinder B og C som den skal, derimode er der fejl i den primære path. Programmet returnerer i konsollen som forventet et svar om at demand'en er ugyldig. Et konsol output ses nedenfor.

```
invalid demandpath:
Nodes:(start,end)=(B,C)
Primary path:  (SO1, A,B)
Backup path:  (SO3, B, C)
```

D.1.2 getMax()

Forudsætninger for at teste metoden: Ingen.

getMax() testes ved at oprette en matrix i form af et todimensionelt array. Dette gøres nedenfor.

```
double[][] F = {{0, 1, 2},{2, 4, 7},{3, 4, 1}};
System.out.println(Variables.F.getMax(F, 0));
System.out.println(Variables.F.getMax(F, 1));
System.out.println(Variables.F.getMax(F, 2));
```

Det forventede output for programmet er (2.0 7.0 4.0). Et konsol output ses nedenfor.

```
2.0 7.0 4.0
```

Resultatet stemmer overens med vores forventning.

D.1.3 addDemandPath()

Forudsætninger for at teste metoden:

- Der skal være indlæst et netværk
- Der skal være indlæst et antal demand paths
- Der skal være indlæst et antal demands
- Der skal være initialiseret en failure matrix

En enkelt demand indlæses nu:

```
Variables.F.addDemandPath(P.get(1).get(0),
D.get(1).get(0).getVolume());
```

Demand'en er beskrevet ved

```
Nodes:(start,end)=(A,C), Volume: 4.0
```

Demand'ens tilknyttede demand path er

```
Start node A, end node C
Primary path:  (SO1, A, B) (SO3, B, C)
Backup path:  (SO2, A, C)
```

Efter indlæsning ses failure matricen som

```
 X 4.0 4.0 | 4.0 | 2
4.0  X 4.0 | 4.0 | 2
4.0 4.0  X | 4.0 | 2
           12.0
```

Failure matricen stemmer overens med den indlæste demand og tilknyttede demand path. Primær path benytter link 1 og link 3, hvilket stemmer med række 1 og 3 i F. Backup path'en benytter link 2 hvilket ses at stemme da række 2 har tilføjet 4 i søjle 1 og 3.

Endnu en demand indlæses nu:

```
Variables.F.addDemandPath(P.get(1).get(1),
D.get(1).get(2).getVolume());
```

Demand'en er beskrevet ved

```
Nodes:(start,end)=(A,C), Volume: 2.0
```

Demand'ens tilknyttede demand path er

```
Start node A, end node C
Primary path: (SO2, A, C)
Backup path: (SO3, B, C) (SO1, A, B)
```

Efter indlæsning ses failure matricen som

```
 X 6.0 4.0 | 6.0 | 1
6.0  X 6.0 | 6.0 | 2
4.0 6.0  X | 6.0 | 1
           18.0
```

Igen ses failure matricen at stemme overens med vores forventning. Forklaringen er analog til tidligere eksempel.

D.1.4 substractDemandPath()

Forudsætninger for testen:

- Der skal være indlæst et netværk

- Der skal være indlæst et antal demand paths
- Der skal være indlæst et antal demands
- Der skal være initialiseret en failure matrix

Fjerner første demand (og demand path) der blev tilføjet:

```
 X 2.0 0.0 | 2.0 | 1
2.0  X 2.0 | 2.0 | 2
0.0 2.0  X | 2.0 | 1
           6.0
```

Dette vil svare til at starte med en tom failure matrix og tilføje førnævnte demand og demand path. Fjernes næste demand igen bør man få en nulstillet failure matrix. Demand path fjernes:

```
 X 0.0 0.0 | 0.0 | 2
0.0  X 0.0 | 0.0 | 2
0.0 0.0  X | 0.0 | 2
           0.0
```

Som forventet returnerede programmet en tom failure matrix.

D.1.5 initFailMatrix(Solution)

Forudsætninger for testen:

- Der skal være indlæst et netværk

Det ønskes nu at indlæse et antal demands til en failure matrix der skal fungere som en løsning (Solution). Til kontrol af *`initFailmatrix()`* oprettes en Solution manuelt med de samme indlæste demands og resultaterne sammenlignes. Først initialiseres failure matricen og et antal demands tilføjes.
1. demand tilføjes, følgende demand path benyttes:

```
Start node A, end node C
Primary path: (SO1, A, B) (SO3, B, C)
Backup path: (SO2, A, C)
```

Failure matricen ses nedenfor.

```
 X 4.0 4.0 | 4.0 | 2
4.0  X 4.0 | 4.0 | 2
4.0 4.0  X | 4.0 | 2
           12.0
```

2. demand tilføjes, følgende demand path benyttes:

```
Start node A, end node C
Primary path: (SO2, A, C)
Backup path: (SO3, B, C) (SO1, A, B)
```

Failure matricen ses nedenfor.

```
 X 6.0 4.0 | 6.0 | 1
6.0  X 6.0 | 6.0 | 2
4.0 6.0  X | 6.0 | 1
           18.0
```

3. demand tilføjes, følgende demand path benyttes:

```
Start node B, end node C
Primary path: (SO3, B, C)
Backup path: (SO2, A, C) (SO1, A, B)
```

Failure matricen ses nedenfor.

```
 X 6.0 5.0 | 6.0 | 1
6.0  X 7.0 | 7.0 | 1
5.0 7.0  X | 7.0 | 1
           20.0
```

4. demand tilføjes, følgende demand path benyttes:

```
Start node B, end node C
Primary path: (SO1, A, B) (SO2, A, C)
Backup path: (SO3, B, C)
```

Failure matricen ses nedenfor.

```
 X 11.0 10.0 | 11.0 | 1
11.0  X 12.0 | 12.0 | 1
10.0 12.0  X | 12.0 | 1
             35.0
```

Failure matricen har en tilhørende løsningsvektor med følgende udseende:

```
null
[0 1]
null
[0 1]
null
null
```

I nedenstående stykke kode oprettes en Solution svarende til den ovenstående:

```
Vector<int[]> V = new Vector<int[]>(D.size());
V.add(0, null);
int[] I = {0,1};
V.add(1,I);
V.add(2, null);
V.add(3, I);
V.add(4, null);
V.add(5, null);
Solution S = new Solution(V);
System.out.println(S);
Variables.F.initFailMatrix(S);
Variables.F.printFailMatrix();
```

Løsningsvektoren og den tilhørende failure matrix ses nedenfor.

```
S[0]: empty
S[1]: [0, 1]
S[2]: empty
S[3]: [0, 1]
S[4]: empty
S[5]: empty

 X 11.0 10.0 | 11.0 | 1
11.0  X 12.0 | 12.0 | 1
10.0 12.0  X | 12.0 | 1
               35.0
```

Som det ses stemmer løsningsvektorerne og failure matricerne overens.

D.1.6 Omfattende test af Solution

Forudsætninger for testen:

- Der skal være indlæst et netværk
- Der skal være indlæst et antal demand paths
- Der skal være indlæst et antal demands
- Der skal være initialiseret en failure matrix

En Solution testes ved først at udtrække den bedste løsning for en given kørsel af GRASP og den tilsvarende evaluering. En ny failure matrix oprettes og initialiseres med den givne løsning og de 2 evalueringer sammenlignes. Dernæst undersøges om værdierne i de to failure matricer er forskellige. Der undersøges så om længden af S, P og D stemmer overens.
Tilsidst gennemgås hele S og det undersøges om de tomme indekser, det værende værdien null eller et tomt array, stemmer overens med tilsvarende indekser i P og D. Er S ikke er tom på et givent index undersøges det om arrayet på dette index har samme længde som antallet af demands.

```
Solution S = grasp.getBestSolution();
graspEval = grasp.bestEval;
FailureMatrix F = new FailureMatrix(Variables.Data.L);
F.initFailMatrix(S);
newEval = F.getFitness();
if(graspEval != newEval)
    System.out.println("fejl i evaluering");

if(!Variables.F.equals(F))
    System.out.println("fejl i failure matrix");

if(S.length != Variables.Data.P.size() || S.length != Variables
    .Data.D.size())
    System.out.println("Fejl i længden af S");

for(int i=0; i < S.length; i++) {

    if((!S.isValid(i) || S.get(i).length == 0) &&
        (!Variables.Data.P.get(i).isEmpty() || !Variables.Data.
            D.get(i).isEmpty())
       )
        System.out.println("Fejl på index: " + i);
```

```
            if(S.isValid(i) && (S.get(i).length != Variables.Data.D.get
                (i).size()))
                System.out.println("Fejl på index: " + i);
        }
```

Som forventet ingen fejlmedddelelser i konsollen.

D.1.7 testHash() og makeHash()

Forudsætninger for testen: Ingen.

Først køres *testHash()* metoden til generering af indekser for 3 nodes i et netværk

```
(0,1): 0
(0,2): 1
(0,3): 2
(1,2): 3
(1,3): 4
(2,3): 5
```

Som det ses udregner metoden forskellige indekser for hvert node par. Demand paths i data sættet (2 mellem (0,2) og 1 mellem (1,2)) burde i såfald blive indlæst på indekserne 1 og 3 i P med funktionen *makeHash()*. Indekseringen foretaget af programmet ses nedenfor.

```
index 0: null

index 1 Start node A, end node C Primary path:  (SO1, A, B) (SO3, B, C) Backup path:
(SO2, A, C)

Start node A, end node C Primary path:  (SO2, A, C) Backup path: (SO3, B, C) (SO1, A,
B)

index 2: null

index 3 Start node B, end node C Primary path:  (SO3, B, C) Backup path:  (SO2, A, C)
(SO1, A, B)

index 4: null

index 5: null
```

Som det ses stemmer resultatet overens med vores forventning.
De 6 demands fra data sættet (3 mellem nodes (0,2) og 3 mellem (1,2)) burde i så fald blive indlæst på indekserne 1 og 4 i D. Nedenfor ses demands fra data sættet indekseret med funktionen *makeHash()*

```
index 0: null

index 1: Nodes:(start,end)=(A,C), Volume: 4.0 Nodes:(start,end)=(A,C), Volume: 6.0
```

```
Nodes:(start,end)=(A,C), Volume: 2.0

index 2: null

index 3: Nodes:(start,end)=(B,C), Volume: 1.0 Nodes:(start,end)=(B,C), Volume: 3.0
Nodes:(start,end)=(B,C), Volume: 5.0

index 4: null

index 5: null
```

Som det ses stemmer resultatet også her overens med vores forventning.

D.2 Performance test

D.2.1 Resultater for parameter tuning af GRASP

N	3	5	10
0.10	(0,40%,0,08%)	**(0,40%,0,06%)**	(0,48%,0,08%)
0.25	(0,63%,0,20%)	(0,54%,0,15%)	(0,58%,0,10%)
0.50	(0,67%,0,24%)	(0,61%,0,19%)	(0,63%,0,14%)
0.75	(0,68%,0,16%)	(0,69%,0,17%)	(0,66%,0,16%)
1.00	(0,75%,0,18%)	(0,73%,0,14%)	(0,71%,0,16%)

0.10	**(0,34%,0,09%)**	(0,37%,0,05%)	(0,39%,0,04%)
0.25	(0,59%,0,20%)	(0,53%,0,17%)	(0,46%,0,13%)
0.50	(0,62%,0,13%)	(0,56%,0,16%)	(0,54%,0,16%)
0.75	(0,57%,0,24%)	(0,62%,0,17%)	(0,56%,0,14%)
1.00	(0,74%,0,18%)	(0,68%,0,20%)	(0,67%,0,15%)

0.10	**(0,29%,0,05%)**	(0,31%,0,04%)	(0,35%,0,04%)
0.25	(0,55%,0,18%)	(0,51%,0,18%)	(0,44%,0,16%)
0.50	(0,54%,0,18%)	(0,55%,0,15%)	(0,51%,0,11%)
0.75	(0,68%,0,22%)	(0,54%,0,19%)	(0,49%,0,13%)
1.00	(0,59%,0,17%)	(0,54%,0,15%)	(0,58%,0,13%)

0.10	**(0,26%,0,03%)**	(0,29%,0,04%)	(0,31%,0,05%)
0.25	(0,49%,0,15%)	(0,48%,0,19%)	(0,49%,0,09%)
0.50	(0,55%,0,24%)	(0,49%,0,16%)	(0,46%,0,12%)
0.75	(0,57%,0,20%)	(0,59%,0,14%)	(0,48%,0,16%)
1.00	(0,67%,0,18%)	(0,51%,0,13%)	(0,54%,0,14%)

Tabel D.1: Parameter tuning resultater (GRASP) på "PanEuropean"

N	3	5	10
0.10	**(1,47%,0,10%)**	(1,54%,0,11%)	(1,64%,0,07%)
0.25	(1,57%,0,08%)	(1,58%,0,10%)	(1,67%,0,11%)
0.50	(1,60%,0,09%)	(1,60%,0,06%)	(1,72%,0,10%)
0.75	(1,56%,0,10%)	(1,62%,0,10%)	(1,72%,0,11%)
1.00	(1,59%,0,16%)	(1,65%,0,07%)	(1,71%,0,08%)

0.10	**(1,40%,0,09%)**	(1,44%,0,08%)	(1,55%,0,08%)
0.25	(1,53%,0,08%)	(1,49%,0,09%)	(1,57%,0,11%)
0.50	(1,46%,0,12%)	(1,49%,0,11%)	(1,60%,0,07%)
0.75	(1,51%,0,10%)	(1,50%,0,12%)	(1,60%,0,06%)
1.00	(1,44%,0,10%)	(1,54%,0,10%)	(1,59%,0,07%)

0.10	**(1,39%,0,08%)**	(1,42%,0,10%)	(1,48%,0,08%)
0.25	(1,44%,0,12%)	(1,45%,0,08%)	(1,50%,0,08%)
0.50	(1,47%,0,09%)	(1,45%,0,10%)	(1,53%,0,09%)
0.75	(1,43%,0,11%)	(1,49%,0,10%)	(1,52%,0,08%)
1.00	(1,46%,0,10%)	(1,44%,0,09%)	(1,53%,0,06%)

0.10	(1,41%,0,06%)	**(1,32%,0,09%)**	(1,39%,0,07%)
0.25	(1,39%,0,09%)	(1,41%,0,06%)	(1,46%,0,07%)
0.50	(1,43%,0,11%)	(1,42%,0,08%)	(1,46%,0,06%)
0.75	(1,43%,0,09%)	(1,44%,0,09%)	(1,47%,0,08%)
1.00	(1,42%,0,09%)	(1,43%,0,07%)	(1,46%,0,07%)

Tabel D.2: Parameter tuning resultater (GRASP) på "USANetwork"

D.2.2 Resultater for parameter tuning af Simulated Annealing

	100	500	1000	2500
0,80	(0,87%,0,22%)	(0,91%,0,21%)	(0,76%,0,18%)	(0,88%,0,20%)
0,90	(0,89%,0,23%)	(0,75%,0,24%)	(0,80%,0,18%)	(0,85%,0,16%)
0,95	**(0,73%,0,23%)**	(0,88%,0,20%)	(0,87%,0,21%)	(0,90%,0,24%)
0,99	(0,82%,0,18%)	(0,90%,0,21%)	(0,86%,0,19%)	(0,88%,0,19%)

0,80	(0,79%,0,21%)	(0,83%,0,21%)	(0,81%,0,17%)	(0,76%,0,22%)
0,90	(0,76%,0,20%)	(0,79%,0,17%)	(0,77%,0,18%)	(0,74%,0,21%)
0,95	(0,87%,0,19%)	**(0,73%,0,19%)**	(0,74%,0,20%)	(0,82%,0,21%)
0,99	(0,80%,0,15%)	(0,84%,0,19%)	(0,85%,0,22%)	(0,79%,0,16%)

0,80	(0,83%,0,21%)	(0,78%,0,17%)	(0,76%,0,15%)	(0,68%,0,21%)
0,90	**(0,67%,0,18%)**	(0,75%,0,21%)	(0,72%,0,21%)	(0,77%,0,24%)
0,95	(0,70%,0,19%)	(0,77%,0,24%)	(0,81%,0,22%)	(0,80%,0,19%)
0,99	(0,68%,0,18%)	(0,71%,0,21%)	(0,74%,0,19%)	(0,76%,0,21%)

0,80	(0,73%,0,25%)	(0,82%,0,21%)	(0,74%,0,24%)	**(0,68%,0,25%)**
0,90	(0,69%,0,18%)	(0,71%,0,23%)	(0,79%,0,16%)	(0,83%,0,25%)
0,95	(0,78%,0,18%)	(0,71%,0,18%)	(0,71%,0,25%)	(0,83%,0,23%)
0,99	(0,77%,0,22%)	(0,78%,0,20%)	(0,76%,0,21%)	(0,75%,0,21%)

Tabel D.3: Parameter tuning resultater (SA) på "PanEuropean"

	100	500	1000	2500
0,80	(1,62%,0,07%)	**(1,57%,0,10%)**	(1,60%,0,09%)	(1,62%,0,08%)
0,90	(1,67%,0,09%)	(1,61%,0,10%)	(1,64%,0,08%)	(1,62%,0,08%)
0,95	(1,68%,0,08%)	(1,62%,0,09%)	(1,67%,0,11%)	(1,65%,0,07%)
0,99	(1,68%,0,08%)	(1,66%,0,10%)	(1,65%,0,08%)	(1,64%,0,08%)

0,80	(1,53%,0,08%)	(1,50%,0,10%)	(1,57%,0,09%)	(1,52%,0,10%)
0,90	(1,51%,0,08%)	**(1,50%,0,07%)**	(1,54%,0,08%)	(1,53%,0,11%)
0,95	(1,52%,0,08%)	(1,55%,0,08%)	(1,53%,0,10%)	(1,51%,0,10%)
0,99	(1,52%,0,09%)	(1,56%,0,11%)	(1,59%,0,07%)	(1,52%,0,11%)

0,80	(1,49%,0,06%)	(1,49%,0,08%)	(1,51%,0,09%)	(1,50%,0,09%)
0,90	(1,52%,0,06%)	**(1,44%,0,07%)**	(1,50%,0,10%)	(1,51%,0,09%)
0,95	(1,50%,0,08%)	(1,45%,0,09%)	(1,48%,0,10%)	(1,48%,0,09%)
0,99	(1,52%,0,08%)	(1,47%,0,09%)	(1,53%,0,07%)	(1,55%,0,07%)

0,80	**(1,42%,0,09%)**	(1,43%,0,10%)	(1,45%,0,09%)	(1,44%,0,09%)
0,90	(1,48%,0,10%)	(1,46%,0,09%)	(1,48%,0,08%)	(1,50%,0,10%)
0,95	(1,45%,0,09%)	(1,45%,0,08%)	(1,47%,0,09%)	(1,48%,0,09%)
0,99	(1,49%,0,09%)	(1,46%,0,10%)	(1,48%,0,11%)	(1,49%,0,09%)

Tabel D.4: Parameter tuning resultater (SA) på "USANetwork"

D.2.3 Test af løsning under teoretisk lowerbound

printing demand paths...

```
(0,1)    0 - 2 1            S01 - S03 S02
(0,1)    0 - 3 4 1          S01 - S04 S05 S02
(0,2)    1 - 2 0            S02 - S03 S01
(0,2)    0 2 - 1            S01 S03 - S02
(0,2)    1 - 5 6 3 0        S02 - S06 S07 S04 S01
(0,3)    0 3 - 4 1          S01 S04 - S05 S02
(0,3)    0 3 - 6 5 1        S01 S04 - S07 S06 S02
(0,4)    1 5 - 6 3 0        S02 S06 - S07 S04 S01
(0,4)    1 5 - 6 4 2 0      S02 S06 - S07 S05 S03 S01
(1,2)    2 - 4 3            S03 - S05 S04
(1,2)    2 - 1 0            S03 - S02 S01
(1,3)    3 - 4 2            S04 - S05 S03
(1,3)    3 - 4 1 0          S04 - S05 S02 S01
(1,3)    3 - 6 5 2          S04 - S07 S06 S03
(1,3)    3 - 6 5 1 0        S04 - S07 S06 S02 S01
(1,4)    2 5 - 6 3          S03 S06 - S07 S04
(1,4)    2 5 - 6 4 1 0      S03 S06 - S07 S05 S02 S01
(1,4)    3 6 - 5 2          S04 S07 - S06 S03
(1,4)    3 6 - 5 1 0        S04 S07 - S06 S02 S01
(2,3)    4 - 6 5            S05 - S07 S06
(2,3)    4 - 3 2            S05 - S04 S03
(2,4)    5 - 6 4            S06 - S07 S05
(2,4)    5 - 6 3 2          S06 - S07 S04 S03
(3,4)    6 - 5 4            S07 - S06 S05
(3,4)    6 - 5 2 3          S07 - S06 S03 S04
```

printing demands...

```
0   1     2
0   1     1
0   2     1
0   2     1
0   3     1
0   3     2
0   4     1
0   4     2
1   2     1
1   2     2
1   3     1
1   3     1
1   4     2
1   4     1
2   3     1
2   3     2
```

```
2   4      2
2   4      1
3   4      2
3   4      1
```

I det følgende eksekveres GRASP:

```
starting GRASP... adjusted neighbourhood size: 5 greedy value=72.0, found in 0.0
milliseconds number of iterations (Local Search)=30723, Eval = 68.0 number of
iterations (Local Search)=37075, Eval = 68.0 number of iterations on average: 0
GRASP: 68.0 -,96%

Printing failure matrix...

i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  0  0  0  0  0  0 |  0 |  6
  1|  0  x  0  0  0  0  0 |  0 |  6
  2|  0  0  x  0  0  0  0 |  0 |  6
  3|  0  0  0  x  0  0  0 |  0 |  6
  4|  0  0  0  0  x  0  0 |  0 |  6
  5|  0  0  0  0  0  x  0 |  0 |  6
  6|  0  0  0  0  0  0  x |  0 |  6
                           0
tilføjer P[0][1] og D[0][0]

Nodes:(start,end)=(A,B)
 Primary path:  (SO1, A, B)
 Backup path:  (SO4, B, D) (SO5, C, D) (SO2, A, C)

Nodes:(start,end)=(A,B), Volume: 2.0 Printing failure matrix...

i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  2  2  2  2  2  2 |  2 |  6
  1|  2  x  0  0  0  0  0 |  2 |  1
  2|  0  0  x  0  0  0  0 |  0 |  6
  3|  2  0  0  x  0  0  0 |  2 |  1
  4|  2  0  0  0  x  0  0 |  2 |  1
  5|  0  0  0  0  0  x  0 |  0 |  6
  6|  0  0  0  0  0  0  x |  0 |  6
                           8
tilføjer P[0][0] og D[0][1]

Nodes:(start,end)=(A,B)
 Primary path:  (SO1, A, B)
 Backup path:  (SO3, B, C) (SO2, A, C)

Nodes:(start,end)=(A,B), Volume: 1.0 Printing failure matrix...
```

```
i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  3  3  3  3  3  3 |  3 |  6
  1|  3  x  0  0  0  0  0 |  3 |  1
  2|  1  0  x  0  0  0  0 |  1 |  1
  3|  2  0  0  x  0  0  0 |  2 |  1
  4|  2  0  0  0  x  0  0 |  2 |  1
  5|  0  0  0  0  0  x  0 |  0 |  6
  6|  0  0  0  0  0  0  x |  0 |  6
                           11
tilføjer P[1][0] og D[1][0]

Nodes:(start,end)=(A,C)
 Primary path:  (SO2, A, C)
 Backup path:  (SO3, B, C) (SO1, A, B)

Nodes:(start,end)=(A,C), Volume: 1.0 Printing failure matrix...

i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  4  3  3  3  3  3 |  4 |  1
  1|  4  x  1  1  1  1  1 |  4 |  1
  2|  1  1  x  0  0  0  0 |  1 |  2
  3|  2  0  0  x  0  0  0 |  2 |  1
  4|  2  0  0  0  x  0  0 |  2 |  1
  5|  0  0  0  0  0  x  0 |  0 |  6
  6|  0  0  0  0  0  0  x |  0 |  6
                           13
tilføjer P[1][0] og D[1][1]

Nodes:(start,end)=(A,C)
 Primary path:  (SO2, A, C)
 Backup path:  (SO3, B, C) (SO1, A, B)

Nodes:(start,end)=(A,C), Volume: 1.0 Printing failure matrix...

i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  5  3  3  3  3  3 |  5 |  1
  1|  5  x  2  2  2  2  2 |  5 |  1
  2|  1  2  x  0  0  0  0 |  2 |  1
  3|  2  0  0  x  0  0  0 |  2 |  1
  4|  2  0  0  0  x  0  0 |  2 |  1
  5|  0  0  0  0  0  x  0 |  0 |  6
  6|  0  0  0  0  0  0  x |  0 |  6
                           16
tilføjer P[2][1] og D[2][0]
```

```
Nodes:(start,end)=(A,D)
 Primary path:  (SO1, A, B) (SO4, B, D)
 Backup path:  (SO7, D, E) (SO6, C, E) (SO2, A, C)

Nodes:(start,end)=(A,D), Volume: 1.0 Printing failure matrix...

i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  6  4  4  4  4  4 |  6 |  1
  1|  6  x  2  3  2  2  2 |  6 |  1
  2|  1  2  x  0  0  0  0 |  2 |  1
  3|  3  1  1  x  1  1  1 |  3 |  1
  4|  2  0  0  0  x  0  0 |  2 |  1
  5|  1  0  0  1  0  x  0 |  1 |  2
  6|  1  0  0  1  0  0  x |  1 |  2
                           21
tilføjer P[2][1] og D[2][1]

Nodes:(start,end)=(A,D)
 Primary path:  (SO1, A, B) (SO4, B, D)
 Backup path:  (SO7, D, E) (SO6, C, E) (SO2, A, C)

Nodes:(start,end)=(A,D), Volume: 2.0 Printing failure matrix...

i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  8  6  6  6  6  6 |  8 |  1
  1|  8  x  2  5  2  2  2 |  8 |  1
  2|  1  2  x  0  0  0  0 |  2 |  1
  3|  5  3  3  x  3  3  3 |  5 |  1
  4|  2  0  0  0  x  0  0 |  2 |  1
  5|  3  0  0  3  0  x  0 |  3 |  2
  6|  3  0  0  3  0  0  x |  3 |  2
                           31
tilføjer P[3][0] og D[3][0]

Nodes:(start,end)=(A,E)
 Primary path:  (SO2, A, C) (SO6, C, E)
 Backup path:  (SO7, D, E) (SO4, B, D) (SO1, A, B)

Nodes:(start,end)=(A,E), Volume: 1.0 Printing failure matrix...

i\j   0  1  2  3  4  5  6   max   maxC
  0|  x  9  6  6  6  7  6 |  9 |  1
  1|  9  x  3  6  3  3  3 |  9 |  1
  2|  1  2  x  0  0  0  0 |  2 |  1
  3|  5  4  3  x  3  4  3 |  5 |  1
  4|  2  0  0  0  x  0  0 |  2 |  1
```

```
  5|  4  1  1  4  1  x  1 |  4 |  2
  6|  3  1  0  3  0  1  x |  3 |  2
                           34
tilføjer P[3][0] og D[3][1]

Nodes:(start,end)=(A,E)
 Primary path:  (SO2, A, C) (SO6, C, E)
 Backup path:  (SO7, D, E) (SO4, B, D) (SO1, A, B)

Nodes:(start,end)=(A,E), Volume: 2.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   6   6   6   9   6 |  11 |   1
   1|  11   x   5   8   5   5   5 |  11 |   1
   2|   1   2   x   0   0   0   0 |   2 |   1
   3|   5   6   3   x   3   6   3 |   6 |   2
   4|   2   0   0   0   x   0   0 |   2 |   1
   5|   6   3   3   6   3   x   3 |   6 |   2
   6|   3   3   0   3   0   3   x |   3 |   4
                                    41
tilføjer P[5][1] og D[5][0]

Nodes:(start,end)=(B,C)
 Primary path:  (SO3, B, C)
 Backup path:  (SO2, A, C) (SO1, A, B)

Nodes:(start,end)=(B,C), Volume: 1.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   6   6   9   6 |  11 |   1
   1|  11   x   6   8   5   5   5 |  11 |   1
   2|   2   3   x   1   1   1   1 |   3 |   1
   3|   5   6   3   x   3   6   3 |   6 |   2
   4|   2   0   0   0   x   0   0 |   2 |   1
   5|   6   3   3   6   3   x   3 |   6 |   2
   6|   3   3   0   3   0   3   x |   3 |   4
                                    42
tilføjer P[5][0] og D[5][1]

Nodes:(start,end)=(B,C)
 Primary path:  (SO3, B, C)
 Backup path:  (SO5, C, D) (SO4, B, D)

Nodes:(start,end)=(B,C), Volume: 2.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
```

```
   0|   x  11   7   6   6   9   6 |  11 |   1
   1|  11   x   6   8   5   5   5 |  11 |   1
   2|   4   5   x   3   3   3   3 |   5 |   1
   3|   5   6   5   x   3   6   3 |   6 |   2
   4|   2   0   2   0   x   0   0 |   2 |   2
   5|   6   3   3   6   3   x   3 |   6 |   2
   6|   3   3   0   3   0   3   x |   3 |   4
                                     44
tilføjer P[6][0] og D[6][0]

Nodes:(start,end)=(B,D)
 Primary path:  (SO4, B, D)
 Backup path:  (SO5, C, D) (SO3, B, C)

Nodes:(start,end)=(B,D), Volume: 1.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   6   6   9   6 |  11 |   1
   1|  11   x   6   8   5   5   5 |  11 |   1
   2|   4   5   x   4   3   3   3 |   5 |   1
   3|   6   7   6   x   4   7   4 |   7 |   2
   4|   2   0   2   1   x   0   0 |   2 |   2
   5|   6   3   3   6   3   x   3 |   6 |   2
   6|   3   3   0   3   0   3   x |   3 |   4
                                     45
tilføjer P[6][0] og D[6][1]

Nodes:(start,end)=(B,D)
 Primary path:  (SO4, B, D)
 Backup path:  (SO5, C, D) (SO3, B, C)

Nodes:(start,end)=(B,D), Volume: 1.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   6   6   9   6 |  11 |   1
   1|  11   x   6   8   5   5   5 |  11 |   1
   2|   4   5   x   5   3   3   3 |   5 |   2
   3|   7   8   7   x   5   8   5 |   8 |   2
   4|   2   0   2   2   x   0   0 |   2 |   3
   5|   6   3   3   6   3   x   3 |   6 |   2
   6|   3   3   0   3   0   3   x |   3 |   4
                                     46
tilføjer P[7][3] og D[7][0]

Nodes:(start,end)=(B,E)
 Primary path:  (SO4, B, D) (SO7, D, E)
```

```
 Backup path:  (SO6, C, E) (SO2, A, C) (SO1, A, B)

Nodes:(start,end)=(B,E), Volume: 2.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   8   6   9   8 |  11 |   1
   1|  11   x   6  10   5   5   7 |  11 |   1
   2|   4   5   x   5   3   3   3 |   5 |   2
   3|   9  10   9   x   7  10   7 |  10 |   2
   4|   2   0   2   2   x   0   0 |   2 |   3
   5|   6   3   3   8   3   x   5 |   8 |   1
   6|   5   5   2   5   2   5   x |   5 |   4
                                    52
tilføjer P[7][3] og D[7][1]

Nodes:(start,end)=(B,E)
 Primary path:  (SO4, B, D) (SO7, D, E)
 Backup path:  (SO6, C, E) (SO2, A, C) (SO1, A, B)

Nodes:(start,end)=(B,E), Volume: 1.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   9   6   9   9 |  11 |   1
   1|  11   x   6  11   5   5   8 |  11 |   2
   2|   4   5   x   5   3   3   3 |   5 |   2
   3|  10  11  10   x   8  11   8 |  11 |   2
   4|   2   0   2   2   x   0   0 |   2 |   3
   5|   6   3   3   9   3   x   6 |   9 |   1
   6|   6   6   3   6   3   6   x |   6 |   4
                                    55
tilføjer P[9][1] og D[9][0]

Nodes:(start,end)=(C,D)
 Primary path:  (SO5, C, D)
 Backup path:  (SO4, B, D) (SO3, B, C)

Nodes:(start,end)=(C,D), Volume: 1.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   9   6   9   9 |  11 |   1
   1|  11   x   6  11   5   5   8 |  11 |   2
   2|   4   5   x   5   4   3   3 |   5 |   2
   3|  10  11  10   x   9  11   8 |  11 |   2
   4|   3   1   3   3   x   1   1 |   3 |   3
   5|   6   3   3   9   3   x   6 |   9 |   1
   6|   6   6   3   6   3   6   x |   6 |   4
```

```
                                    56
tilføjer P[9][0] og D[9][1]

Nodes:(start,end)=(C,D)
 Primary path:  (SO5, C, D)
 Backup path:  (SO7, D, E) (SO6, C, E)

Nodes:(start,end)=(C,D), Volume: 2.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   9   6   9   9 |  11 |   1
   1|  11   x   6  11   5   5   8 |  11 |   2
   2|   4   5   x   5   4   3   3 |   5 |   2
   3|  10  11  10   x   9  11   8 |  11 |   2
   4|   5   3   5   5   x   3   3 |   5 |   3
   5|   6   3   3   9   5   x   6 |   9 |   1
   6|   6   6   3   6   5   6   x |   6 |   4
                                    58
tilføjer P[10][0] og D[10][0]

Nodes:(start,end)=(C,E)
 Primary path:  (SO6, C, E)
 Backup path:  (SO7, D, E) (SO5, C, D)

Nodes:(start,end)=(C,E), Volume: 2.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   9   6   9   9 |  11 |   1
   1|  11   x   6  11   5   5   8 |  11 |   2
   2|   4   5   x   5   4   3   3 |   5 |   2
   3|  10  11  10   x   9  11   8 |  11 |   2
   4|   5   3   5   5   x   5   3 |   5 |   4
   5|   8   5   5  11   7   x   8 |  11 |   1
   6|   6   6   3   6   5   8   x |   8 |   1
                                    62
tilføjer P[10][0] og D[10][1]

Nodes:(start,end)=(C,E)
 Primary path:  (SO6, C, E)
 Backup path:  (SO7, D, E) (SO5, C, D)

Nodes:(start,end)=(C,E), Volume: 1.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   9   6   9   9 |  11 |   1
   1|  11   x   6  11   5   5   8 |  11 |   2
```

```
   2|   4   5   x   5   4   3   3 |   5 |   2
   3|  10  11  10   x   9  11   8 |  11 |   2
   4|   5   3   5   5   x   6   3 |   6 |   1
   5|   9   6   6  12   8   x   9 |  12 |   1
   6|   6   6   3   6   5   9   x |   9 |   1
                                    65
tilføjer P[12][1] og D[12][0]

Nodes:(start,end)=(D,E)
 Primary path:  (SO7, D, E)
 Backup path:  (SO6, C, E) (SO3, B, C) (SO4, B, D)

Nodes:(start,end)=(D,E), Volume: 2.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   9   6   9   9 |  11 |   1
   1|  11   x   6  11   5   5   8 |  11 |   2
   2|   4   5   x   5   4   3   5 |   5 |   3
   3|  10  11  10   x   9  11  10 |  11 |   2
   4|   5   3   5   5   x   6   3 |   6 |   1
   5|   9   6   6  12   8   x  11 |  12 |   1
   6|   8   8   5   8   7  11   x |  11 |   1
                                    67
tilføjer P[12][0] og D[12][1]

Nodes:(start,end)=(D,E)
 Primary path:  (SO7, D, E)
 Backup path:  (SO6, C, E) (SO5, C, D)

Nodes:(start,end)=(D,E), Volume: 1.0 Printing failure matrix...

 i\j    0   1   2   3   4   5   6   max   maxC
   0|   x  11   7   9   6   9   9 |  11 |   1
   1|  11   x   6  11   5   5   8 |  11 |   2
   2|   4   5   x   5   4   3   5 |   5 |   3
   3|  10  11  10   x   9  11  10 |  11 |   2
   4|   5   3   5   5   x   6   4 |   6 |   1
   5|   9   6   6  12   8   x  12 |  12 |   2
   6|   9   9   6   9   8  12   x |  12 |   1
                                    68
```