Universitetet i Oslo
Institutt for informatikk

Irregulær kommunikasjonstruktur for enbrikkesystem med statisk TDMA svitsjing

Jørgen Schøyen Nicolaysen

2. februar 2005

Sammendrag

I et enbrikkesystem kan funksjonsblokkene ha individuelle kommunikasjonsbehov. For at funksjonsblokkene skal kunne kommunisere sammen finnes det svitsjer tilfeldig plassert i arealet. Gitt at de forskjellige kommunikasjonsbehovene har en fast overføringshastighet må man finne en topologi som sørger for at funksjonsblokkene kan kommunisere sammen. Det foreslås her en metode som kan generere en effektiv topologi innen rimelig tid.

For nettverket generert vil rutingen være statisk. Det gjøres også et forsøk på å implementere rutingen med tidsdividert multipleksing (TDMA).

Takk til

Først og fremst vil jeg takke mine foreldre som har gitt meg enorm støtte under arbeidet med hovedfaget.

Deretter vil jeg takke sosionomtjenesten ved universitet i Oslo, som er i stand til å sette fokus på hva en hovedoppgave egentlig dreier seg om, og for å gi gode skrivetips.

Geir Åge Noven har foreslått oppgaven og fungert som eksternveileder. Oddvar Søråsen har stått som internveileder. Takk til dere begge for tilbakemeldinger under arbeidet med oppgaven.

Innhold

1 Introduksjon **10**

2 Kommunikasjonstrukturer **12**
2.1 Elektrisk Buss 12
2.2 Tverrkobling 13
2.3 Samkjørte nettverk 13
2.3.1 Topologi 13
2.3.2 Ruting 14
2.3.3 Flytkontroll 15
2.4 Parallelle og irregulære nettverk 15
2.4.1 Parallelle systemer og nettverksdesign 15
2.4.2 Irregulære nettverk og nettverksdesign 16
2.5 Tidsdelt mulitpleksing 16
2.6 Kommunikasjonstukturer for enbrikkesystemer 17
2.6.1 Noen topologier for enbrikkesystemer 18
2.6.2 VLSI utlegg og manhattan avstand 19

3 Oppgaven **21**
3.1 Presentasjon av nettverket 21
3.1.1 Svitsj 21
3.1.2 Linker 21
3.1.3 Kommunikasjonsbehov 22
3.1.4 Hvordan vil dette virke 23
3.2 Mål med oppgaven 24

4 Algoritmer **25**
4.1 Kompleksitetsanalyse 26
4.2 Grafer 26
4.3 Noen kjente algoritmer 27
4.3.1 Geometriske Algoritmer 28

5 Generering av topologi **31**
5.1 Nettverksdesign ved generering av alle kombinasjoner av veier. 31
5.1.1 Finne veier mellom to svitsjer 35
5.1.2 Eksempel på valg av områder 37
5.1.3 Kjøretid 38
5.2 Grådig metode 40
5.2.1 Detaljer for grådig metode 41
5.2.2 Kjøretid 45

6 Generering av svitsjetabeller **46**
6.1 Krav til svitsjetabellen 46
6.2 En svitsj 47
6.2.1 Eksempel 47
6.3 For kjent trafikk 49
6.4 Løse for hele nettverket 52

7 Diskusjon **54**
7.1 Oppsummering 54
7.2 Nettverksgenerering 54
7.2.1 Valg av område 54
7.2.2 Alle veier 55
7.2.3 Grådig metode 55
7.2.4 Generelt 56
7.3 Svitsje tabeller 56
7.4 Forslag til videre arbeid 56
7.4.1 Vurderinger av andre kommunikasjonstrukturer 56
7.4.2 Effektivitet av svitsjer 57
7.4.3 Deling av kommunikasjonsbehov 57
7.4.4 Ruting for irregulære nett 57
7.4.5 Trafikk av eller på 57
7.5 Annet relatert arbeid 58

8 Konklusjon **59**

A Forslag til analyse / optimalisering **63**
A.1 Ventetid og diameter 63
A.2 Antall linker 63
A.3 Ubrukt båndbredde 64

B Eksempler på kjøringer **65**
B.0.1 All veier 65
B.1 Kjøringer grådig metode 66

C Kombinasjoner **70**

D Implementasjon **73**
D.1 Perl 73
D.2 Kommentarer til kode 74
D.2.1 Graf 74

E Program **75**

Kapittel 1

Introduksjon

Et *enbrikkesystem* (System-On-Chip (SoC)) er en enkelt integrert krets som er sammenslått av flere forskjellig funksjonsblokker. Funksjonsblokkene er elektroniske komponenter som tidligere krevde plass på egne kretskort eller i egne pakker. Eksempler på dette kan være mikroprosessor, minne eller spesielt tilpassede kretser.

Utviklingstrenden for enbrikkesystemer er at de skal håndtere flere og flere funksjoner, samtidig som det er krav om at utviklingstiden skal reduseres. Et godt eksempel er mobiltelefoner i dag, når nye telefoner ankommer markedet skal de gjerne ha det siste innen ekstrafunksjoner som kamera, mp3-spiller, radio og være mindre i størrelse. En viktig årsak til at dette er mulig er at transistorstørrelsen stadig reduseres, dermed øker antall transistorer som er mulig å pakke inn i en og samme krets. I tillegg er det en økning i klokkehastigheten som transistorene kan operere på. Ved slutten av dette årtiet er det beregnet at antall transistorer som kan plasseres i en integrert krets vil passere en milliard og operere på 10 GHz.

Til nå har felles buss vært en foretrukket kommunikasjonsløsningen mellom forskjellige funksjonsblokker i et enbrikkesystem. Med den utviklingstrenden som er i dag viser det seg at felles buss ikke lengre vil være en tilstrekkelig løsning for kommunikasjon mellom funksjonsblokkene. Valget av kommunikasjonsløsning har betydning for selve ytelsen av kretsen. For å bygge mer effektive systemer har forskere og ingeniører begynt å ta i bruk prinsipper fra samkjørte nettverk for å oppnå en mer effektiv kommunikasjon mellom funksjonsblokkene.

I denne hovedoppgaven er det foreslått å bruke en type TDMA basert irregulært nettverk som kommunikasjonstruktur for enbrikkesystemer. Hvis nettverket skal kunne realiseres i stor skala må det utvikles egne automatiserte metoder som sørger for at nettverket fungerer korrekt med hensyn til

de trafikkbehovene som er satt. Hovedproblemstillingen er å finne ut hvordan dette kan implementeres. Til tross for at målet med nettverket er rettet mot enbrikkesystemer er dette en hovedoppgave som ligger mer i retning av kommunikasjonsystemer enn mikroelektronikk. Det kan pekes ut to primære mål med hovedoppgaven:

- Utvikle en metode for å finne et nettverk som effektivt knytter sammen de forskjellige funksjonsblokkene ut fra gitte trafikkbehov.
- Utvikle en metode som finner svitsjetabeller for nettverket slik at trafikken raskest mulig blir sendt gjennom nettverket.

Kapittel 2

Kommunikasjonstrukturer

2.1 Elektrisk Buss

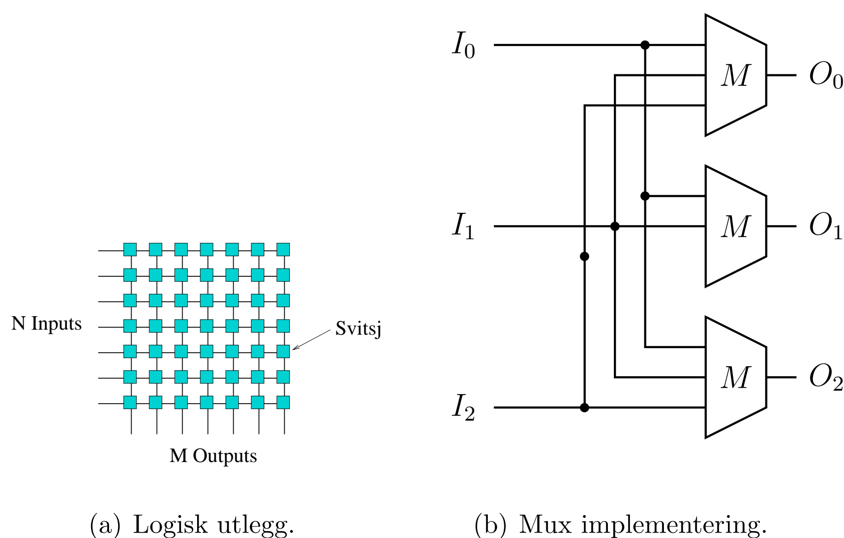
Elektrisk buss [1] benytter et felles sett med ledninger bestående av data-, adresse- og kontrolledninger som er koblet mellom de forskjellige elektroniske enhetene. For å kunne holde kontroll over hvilken enhet som skal benytte bussen er hver enhet tilkoblet en felles *fordelingsenhet* (arbiter). Fordelingsenheten sørger for at den som har ønske om å sende data får tilgang til bussen i korte perioder av gangen. Fordelingsenheten sørger også for at enkelte enheter kan ha prioritet over bussen.

En måte å effektivisere buss basert kommunikasjon er å dele bussen slik at de enhetene som kommuniserer ofte med hverandre er på samme buss. Enheten som kobler de delte bussene sammen kalles for en *bru*. Typisk inndeling er det at benyttes en høyhastighetsbuss (mellom cpu og minne), en systembuss, og en for ytre enheter (I/O kontrollere).

Begrensingene med en bussbasert kommunikasjonstruktur er både elektriske og kommunikasjonsmessige. De elektriske begrensningene skyldes at en buss gjerne har relativt lange ledninger som er med på å sette øvre grense for overføringshastigheten som kan benyttes. Det er også en begrensning på hvor mange enheter som effektivt kan kobles til samme ledningsett. Dette er av betydning nå som integrerte kretser begynner å operere i giga-hertz området. Den kommunikasjonsmessig begrensning skyldes at et felles sett med ledninger hindrer transaksjoner i å bli utført parallelt. Det er kun en enhet (master) som kan benytte ledningsettet av gangen, mens resterende enheter tilkoblet ledningsettet må vente til den gående transaksjonen er ferdig. Hvordan fordelingskjemaet gjør sine prioriteringer ved flere samtidige forespørsler er også en av begrensningene for felles buss.

2.2 Tverrkobling

Tverrkobling (crossbar) gir hvilken som helst funksjonsblokk mulighet til å koble seg direkte til en hvilken som helst annen funksjonsblokk i systemet. Et utlegg er vist i figur 2.1. Tverrkobling knytter samtlige enheter i systemet sammen ved å la bussene være sammenkoblet ved hjelp av svitsjer. På denne måten kan flere enn to funksjonsblokker kommunisere uavhengig av hverandre og operere parallelt. Tverrkobling og kan implementers ved hjelp av blant annet multipleksere, eller tristate drivere [2]. Den største ulempen er arealforbruket, for et stort tverrkoblet nettverk vil det være ineffektivt med hensyn på antall svitsjer. For eksempel vil det med $N \times M$ svitsjer ikke være mulig å koble flere enn $N + N$ enheter når N er mindre enn M. Tverrkoblinger er også et sentralt element ved bygging av svitsjer/rutere.



(a) Logisk utlegg.

(b) Mux implementering.

Figur 2.1: *Tverrkoblet nettverk.*

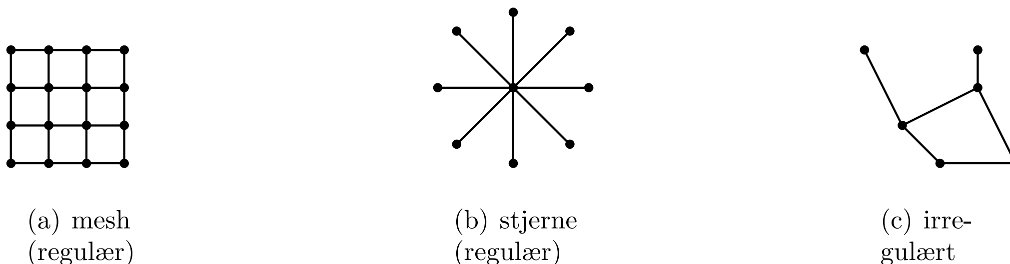
2.3 Samkjørte nettverk

2.3.1 Topologi

Topologi betyr i denne sammenhengen fordelingen av noder (svitsjer og rutere) og forbindelsene (linkene) som knytter disse sammen. En topologi kan

enten være *regulær* eller *irregulær*, eksempler er vist i figur 2.2. Et nettverk kan enten være *direkte*, hvor nodene har en prosesseringsfunksjon, eller *indirekte* hvor nodene ikke har mulighet til å prosessere data. Nodene kan enten være *rutere*, som er utfører mer avanserte funksjoner med trafikken som protokoll konverteringer, eller å kunne avgjøre videre ruting av pakkene, eller *svitsjer* er en enklere enhet, som ikke avhengig av å tolke hele protkollstakken for å sende trafikken videre. En *link* er en forbindelse mellom nodene. En *kanal* er en link og i tillegg deler av logiske enheter (buffere) i nodene. Lengden på linkene setter en begrensing på hvor stor hastigheten for overføringen kan være.

En *vei* (path) er hvilke noder en pakke går igjennom mellom en start og mål node. *Diameteren* for et nettverk er den lengste av de korteste veiene mellom to svitsjer. Diameteren i figur 2.2(a) regnes ut fra nodene som står diagonalt mot hverandre fra hvert sitt hjørne og er 6. *Halveringsbredde* (bisection width), beregnes ved å dele nettverket i to like deler over færrest mulig ledninger. Tilsvarende *halveringsbåndbredde* båndbredden over halveringsbredden.



(a) mesh (regulær)

(b) stjerne (regulær)

(c) irregulært

Figur 2.2: *Eksempler på forskjellige topologier*

2.3.2 Ruting

Ruting handler ikke bare om *hvor* en melding skal sendes men også om *hvordan* en melding skal gå fra en startnode til en målnode i nettverket. Rutingen kan enten være *deterministisk* eller *adaptiv*. Med deterministisk ruting benyttes det alltid samme vei gjennom nettverket og det benyttes egne tabeller som inneholder rutinginformasjonen. Adaptiv ruting vil benytte seg av statusen i nettverket for å bestemme hvor pakken skal sendes. Status av nettverket kan for eksempel være feil i nettverket, eller overbelastning (congestion) som gjør at en annen alternativ vei må begnyttes.

For å oppnå en pålitelig overføring bør rutingfunksjonen oppfylle følgende egenskaper: *Konnektivitet* (connectivity) mulighet for å rute pakker fra hvil-

ken som helst node i nettverket til hvilken som helst annen node. *Tilpasningsdyktighet* (Adaptivity) og *feiltoleranse* mulighet for å rute pakker gjennom alternative forbindelser i tilfelle sperre eller feil i nettverket. *Dødlås* (deadlock) frihet, at rutingalgoritmen ikke blokkerer i nettverket for evig tid. *Sendelås (livelock) frihet* betyr at rutingstrategien ikke sender pakker rundt i nettet i evig tid. Det kan også hende at pakker ikke kommer frem fordi at veien videre er blokkert av andre pakker som har prioritet for de samme resursene i nettverket, denne situasjonen kalles for *hunger* (starvation).

Pakkesvitsjing assosieres ofte med at hele pakken blir lagret i en ruter før den videresendes. Dette er ikke ønskelig av to grunner, for det første vil det gi høy ventetid for nettverket, for det andre vil det være nødvendig å ha stor lagringsplass i ruteren. En mer effektiv pakkesvitsjingsteknikk er å sørge for videresending av pakken straks adresseinformasjonen er tolket. Hvis videre overføring ikke er mulig lagres hele pakken i ruteren. Denne måten å håndtere pakkesvitsjing på kalles for *gjennomskjæring* (cut-through) og er svært gunstig når det gjelder å oppnå lav *ventetid* (latency) i nettverket. En annen variant av denne rutingstrategien er *slangehull* ruting (wormhole), hovedforskjellen her er at bufferne i ruterne holdes til et minimum gjerne til et par flits. Hvis videresending av pakken ikke er mulig, stanser trafikken i de berørte linkene.

2.3.3 Flytkontroll

Flytkontroll bestemmer hvordan nettverket skal håndtere overbelastning av nettverket. Følgende måter å bygge dette på er *dropping*; hvor dataene rett og slett kastes. Ulempen her er at hvis pakker ofte droppes blir den effektive båndbredden redusert på grunn av stadig re-sending av pakker. *Buffering* sørger for at pakkene blir mellomlagret i egne buffere. Denne teknikken gir relativt store rutere på grunn av stort buffer behov. *Blokkering*; all trafikk stanser hvis linkene/svitsjene videre er okkupert. *Misrouting*; leder pakkene slik at de tar en alternativ runde i nettverket, dette er også med på å redusere båndbredden i nettverket.

2.4 Parallelle og irregulære nettverk

2.4.1 Parallelle systemer og nettverksdesign

Ideen med parallelle systemer er at et problem kan løses raskere og billigere ved å benytte flere prosessorer samtidig. Utgangspunktet er at man har n antall prosesser som skal knyttes sammen, og målet er å finne den topologi-

en, rutingen og flytkontrollen som minimaliserer *ventetiden* og maksimerer *gjennomstrømningen*. Det finnes allerede en rekke kjente regulære topologier hvor ytelses parametrene allerede er kjent.

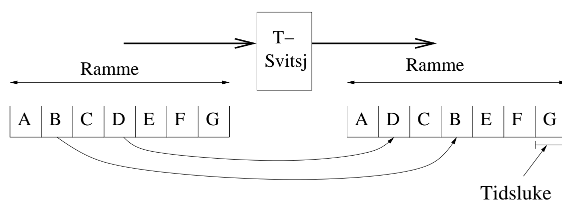
2.4.2 Irregulære nettverk og nettverksdesign

Hovedfokus for analyse irregulære systemer er rettet mot nettverk for arbeidstasjoner (også kalt NOW - Network of Workstations). De har ikke nødvendigvis lagt like stor vekt på lav ventetid og gjennomstrømningen i nettverket som i parallelle nettverk. Hovedfokus her er rettet mot pålitelig ruting, og siden et slikt nettverk ofte utsettes for stadig utskiftninger av kabler eller rutere er høy feiltoleranse et annet designmål for nettverket. Tradisjonelle parametere fra regulære nettverk (parallelle systemer) som halveringsbredde og diameter gir svært lite informasjon om ytelsen[3].

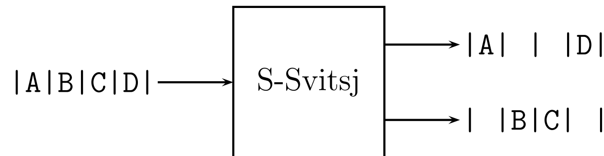
2.5 Tidsdelt mulitpleksing

Tidsdelt multipleksing, eller TDMA- (time-division-multiple-access), går ut på å la et medium, ledninger eller radiobølger, dele sin båndbredde med flere brukere i korte tidsintervaller. Båndbredden for mediet må derfor være lik eller høyere enn det som totalt kreves av brukerne. TDMA svitsjede nettverk benyttes også i kommunikasjonstruktur for enbrikkesystemer. Det benyttes også i busstrukturer og spesielt i telenettet [4, 5]. I denne oppgaven er tidsdelt multipleksing begrenset til å benyttes i ledninger.

En TDMA *Ramme* er delt inn i et fast antall *tidsluker*, hvor hver tidsluke bærer et dataord. Illustrasjon i figur 2.3. I TDMA svitsjede nettverk kan et dataord bytte retning, dette kalles for *romsvitsj* (S-svitsj) (figur 2.4). En ramme vil alltid operere med et fast antall tidsluker, det hender at det er ønsker å bytte plassering av en tidsluke i rammen. En slik rokkering gjøres med en *tidsvitsj* (T-svitsj) (figur 2.3. For å bygge TDMA nettverk benyttes S- og T- svitsjer benyttes gjerne i kombinasjon som STS eller TST.



Figur 2.3: *Tid svitsj, bytte av posisjon i en tidsluke*



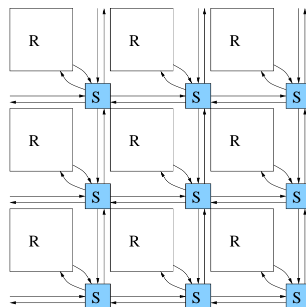
Figur 2.4: *Rom svitsj*

2.6 Kommunikasjonstukturer for enbrikkesystemer

Moores lov [6] bygger på en empirisk observasjon om at antall transistorer i integrerte kretser dobles for hver 18 måned. Denne loven ser ut til å fortsatt gjelde ut år 2010. For fremtiden ser man for seg at et enbrikkesystem kan inneholde opp til 100 [7] mikroprosessorer og operere med klokkefrekvenser i gigahertzsområdet [8]. Buss basert kommunikasjonstruktur som er svært vanlig i enbrikkesystemer møter selvfølgelig de samme kommunikasjonsmessige og fysiske begrensningene som pekt ut i tidligere avsnitt. Med bakgrunn i utviklingstrenden peker det i klar retning av at bussbasert design ikke lenger vil være tilstrekkelig som kommunikasjonstruktur for enbrikkesystemer.

Ved å ta å ta i bruk teknikker fra samkjørte nettverk er det mulig å bygge mer effektive kommunikasjonstrukturer for enbrikkesystemer. Nettverk for enbrikkesystemer har gjerne forkortelsen NoC, for Network On Chip [9]. I forskjell fra andre nettverk har NoC i integrerte kretser et par fordeler og ulemper fra hva som møtes i tradisjonelle nettverk og bussbasert design. Med pakkesvitsjing vil adresseinformasjonen ligge i hodet av pakkene, slik at adresseledninger ikke lenger er nødvendig. Nettverk for integrerte kretser vil skille seg ut fra samkjørte nettverk ved at blant annet arealet er en begrensende resurs og må deles mellom funksjonsblokker og det ledninger og rutere for nettverket. NoCs kan ha en datatrafikk som kan forutsees[1]. Et nettverk vil kunne ansees som en egen funksjonsblokk i enbrikkesystemet, og testing vil foregå ved simulering kun mot nettverket. Det vil også være mulig å kunne utføre transaksjoner parallelt. For tilgang til delt minne gir pakkesvitsjing en ytelsestap. Her er det forslag å utvikle en hybrid kommunikasjonstruktur for dette som gir en linjesvitsjet aksess til delt minne [9].

[1] For eksempel videosystemer



Figur 2.5: *Svitsjer (S) og funksjonsblokker (R) er ordnet i rutenett.*

En av de mest lovende arkitekturene tar sikte på er å benytte egne metallag i den integrerte kretsen for nettverket [10, 7], en illustrasjon er gitt i figur 2.5. Disse nettverkene er strukturerte med fast ledningslengde, noe som hjelper til med å forutse de elektriske karakterestikkene og kortere ledninger gir muligheter for å kunne operere på høyere klokkefrekvenser. Med rutere vil også problemer knyttet til at det benyttes forskjellige klokkedomener på kretsen bli løst. Hva disse nye nettverkene ikke tar hensyn til er forskjellig fasong på funksjonsblokkene som skal tilkobles.

2.6.1 Noen topologier for enbrikkesystemer

Buss systemer

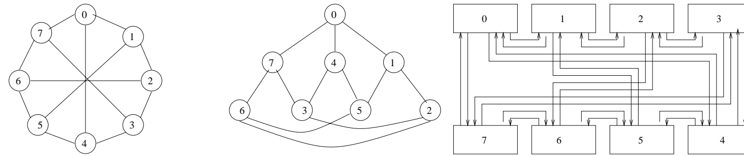
SONICS SMART Interconnect[11], SiliconBackplane og Sonics3220 busser benytter TDMA. Selve rutingen er delt slik at funksjonsblokkene har faste tidluker å forholde seg til, hvis mer båndbredde er nødvendig kan de reservere flere tidluker når de i korte perioder trenger høyere overføringshastighet [8]. Produktet kommer også med et sett med programvare som hjelper til med å korte ned utviklingstiden, slik at utviklerne kan konsentrere seg mer rundt utviklingen av funksjonsblokker.

Andre: AMBA[12] utviklet av ARM. Core Connect[13] utviklet av IBM. Palmchip[14] fra Palmchip Corporation, WhishBone[15].

OC-768

Kombinasjonsløsningene for OC768 [16] benytter seg av et enkelt samkjørt nett. Rutingen omtales som 'oktagon ruting', oppsettet mellom rutere og funksjonsblokker illustreres i figur 2.6. Fordelen med denne topologien er at det ikke vil være mer enn to hopp mellom hver funksjonsblokk. Artikkelen

[16] viser til et veldig viktig poeng, dette samkjørte nettet gir høyere ytelse enn felles buss og tverrkobling. I tillegg benytter den færre ledninger enn tverrkobling.



Figur 2.6: *OC768 Nettverk: oktagon utlegg, trestruktur og fysisk utlegg.*

Prophid

Med Prophid [17] er målet å lage en krets for miksing av videosignaler i realtid. Nettverket i denne har TDMA basert og det benyttes *todelt sammenslåing (bipartite matching)* som ruting strategi. Hovedmålet her er å kunne håndtere den kontinuerlige datastrømmen fra videosignalene. I tillegg skal kretsen kunne håndtere at et videosignal brått forsvinner, ved f.eks. frakobling av videoutstyr.

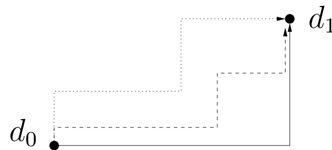
2.6.2 VLSI utlegg og manhattan avstand

For VLSI[2] utlegg kan ledninger kun legges i lengde- eller bredderetningen. Når en skal regne ut avstanden mellom to punkter må dette dette taes med i beregningen. Å regne ut avstanden på denne måten har fått navnet *manhattan-avstand*, ettersom det assosieres med hvor langt en må bevege seg for å komme seg fra et sted til et annet når en befinner seg i en storby med et veinett formet som et rutenett[3]. Det kan være litt uvant å forholde seg til avstander med denne geometrien siden vi normalt tenker vi oss at den korteste avstanden mellom to punkter er den rette linjen som går mellom disse to. For manhattan avstand blir dette annerledes; for alle veier innenfor området avgrenset av de horisontale og vertikale segmentene mellom to punkter, *vil totalavstanden alltid være lik så lenge en beveger seg fra det første punktet til det andre uten å gå tilbake*[18]. Når en kun er begrenset til lengde- og breddeavstand gir dette flere muligheter for å bevege seg mellom to punkter, og allikevel tilbakelegge samme totalavstand, som vist i figur 2.7. Avstanden

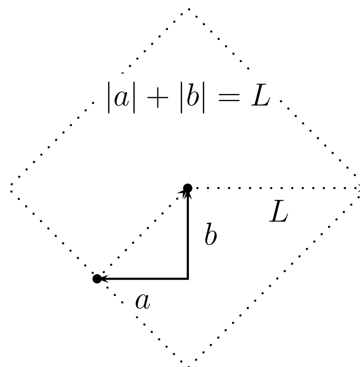
[2] Very Large Scale Integration
[3] andre kaller dette for *taxicab metric*

mellom to punkter beregnes ved å summere absoluttverdien av avstanden mellom punktene. En av egenskapene med dette er at hvis vi skal forholde oss til en fast avstand fra et punkt vil den geometriske figuren dette utgjør være en firkant som vist i figur 2.8, til forskjell fra den euklidske geometrien hvor dette ville ha utgjort en sirkel.

$$D((x_0, y_0), (x_1, y_1)) = |x_1 - x_0| + |y_1 - y_0|$$



Figur 2.7: *Flere måter å beveges seg fra et punkt til et annet, alle veier har samme lengde.*



Figur 2.8: *Omkretsen av manhattanavstanden vil utgjøre et kvadrat.*

Kapittel 3

Oppgaven

3.1 Presentasjon av nettverket

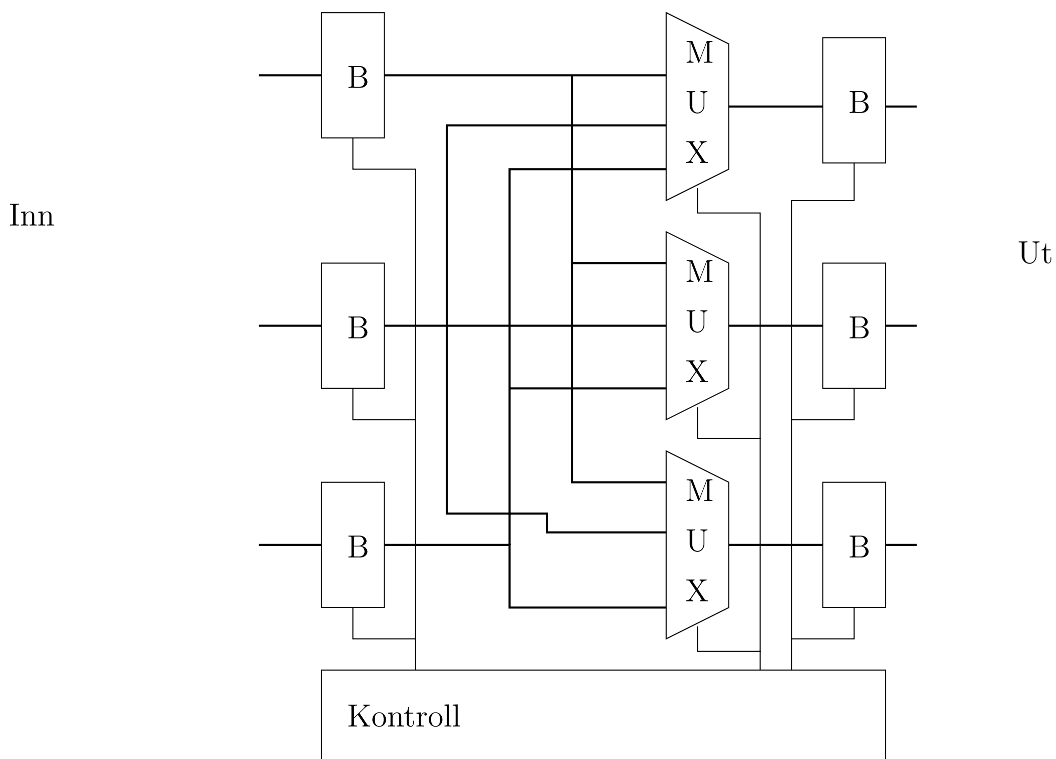
3.1.1 Svitsj

Det vil ikke bli tatt mye hensyn til den interne arkitekturen av svitsjen gjennom hovedoppgaven, men i utgangspunktet blir hver svitsj ansett som en enkel blokk hvor det leses et dataord fra inngangen og sendes videre i løpet av en klokkesykel, såfremt neste tidluke er ledig. Hvis neste tidluke ikke er ledig vil dataordet mellomlagres i svitsjen. Nettverket er derfor ikke pakkeorientert, men bit orientert. Klokkesykelen her er klokkehastigheten som nettverket operer på.

En mulig skisse av svitsjen vises i figur 3.1. Svitsjen bygges opp ved hjelp av multipleksere (MUX), det er buffere (f.eks. d-lås/flip-flop) på inngangen og utgangen. Bufferne er blant annet nødvendig for å stabilisere spenningen på ledningene. Hvis neste tidsluke i rammen ikke er ledig etter en ekstra klokkepuls etter at dataordet er mottatt, vil dette mellomlagres med ekstra buffere på inngangen (ikke vist). Under konstruksjon av nettverket vil det være mulig å legge til nye porter på svitsjen etter behov.

3.1.2 Linker

Ledningene mellom funksjonsblokker og svitsjer består av et sett med parallelle datalinjer og er unidireksjonale. Hvordan dataene skal tolkes, hvordan en hel melding håndteres er det opp til sender og mottaker å finne en passende protokoll for. I oppgaven vil det ikke bli lagt vekt på å finne en slik protokoll. I tillegg taes det heller ikke hensyn til hvordan ledningene skal legges, ledninger kan gjerne også krysse hverandre.



Figur 3.1: *Foreslått skisse av svitsj.*

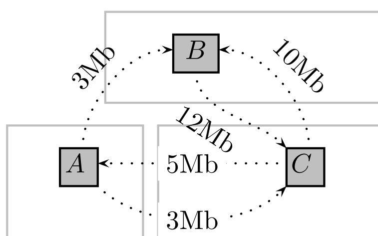
3.1.3 Kommunikasjonsbehov

Vi tenker oss at funksjonsblokkene til enbrikkesystemet har forskjellig størrelse og fasong. Som et resultat av dette vil funksjonsblokkenes porter og svitsjende enheter være tilfeldig plassert i arealet som enbrikkesystemet dekker. Kommunikasjonsbehovet mellom portene vil være kjent og være av konstant karakter, men siden det kan være varierende oppgaver som funksjonsblokkene skal løse mellom seg trenger ikke trafikkbehovet mellom to porter ha lik størrelse.

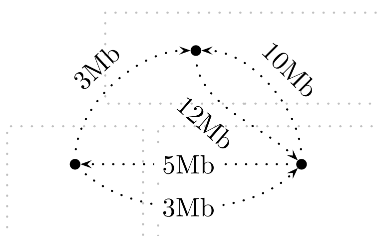
Figur 3.2 er et eksempel på plassering av funksjonsblokker (F) med tilhørende porter (A, B, C) som i dette tilfelle også skal fungere som svitsjende enheter. De prikkete pilene forteller hvilke porter som skal kommunisere sammen og viser også trafikkbehovet som må opprettholdes. Kommunikasjonen mellom C og B er en punkt til punkt forbindelse og har en forespørsel som sendes på 10 Mbit og får en tilsvarende respons på 12 Mbit fra B tilbake til C. For port A er det et punkt til multipunkt forbindelse på 3 Mbit til portene B og C. Porten C har også andre oppgaver som skal løses mot port

A, med behov for overføringshastighet på 5 Mbit.

Siden en svitsj utgjør en plassering i arealet vil denne kun utgjøre et punkt i planet. Som en forenkling trengs det ikke å tas hensyn til kantene til funksjonsblokkene. Representasjonen av svitsjene kan derfor kun bestå av et punkt (x- og y-koordinater) og et krav til kommunikasjonen mellom disse. Nettverket i figur 3.2 kan reduseres til representasjonen i figur 3.3.



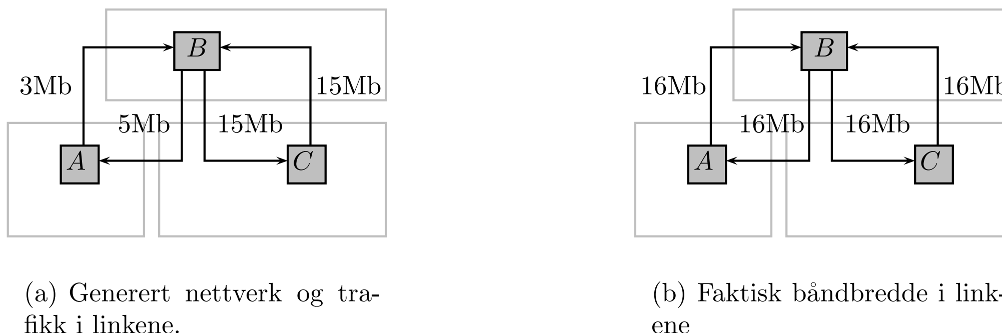
Figur 3.2: *Grunnleggende utlegg.*



Figur 3.3: *Representasjon av kommunikasjonsbehovene.*

3.1.4 Hvordan vil dette virke

For å gi et bilde av hvordan det hele kommer til å virke har vi et enkelt utlegg i figur 3.4(a) med tilhørende kommunikasjonsbehov. Et generert nettverk kan være som i figur 3.4(b). Et eksempel på en ramme ut fra dette er vist i tabell 3.1. Her er det regnet ut med 3 Mbit/tidluke, som er den minste overføringsbehovet som er nødvendig. Her vil det bli tomme tidluker, som betyr ubrukt kapasitet i nettverket. Dette bør helst unngås.



(a) Generert nettverk og trafikk i linkene.

(b) Faktisk båndbredde i linkene

Figur 3.4: *Et utlegg.*

Link	0	1	2	3	4	5
A→B	AB_0					
B→C	BC_0	AB_0	BC_1	BC_2	BC_3	
C→B	CB_0	CB_1	CB_2	CB_3	CA_0	CA_1
B→A	CA_1					CA_0

Tabell 3.1: *Ramme for nettverket i figur 3.4(b)*

3.2 Mål med oppgaven

Målet med oppgaven er å bygge et optimalt nettverk ut fra et gitt trafikkbehov og plasseringen av svitsjende enheter. Rutingen vil være statisk svitsjende TDMA, det skal også genereres svitsjetabeller for svitsjene.

Målet er å finne metoder for å automatisere genereringen av et nettverk med de komponentene som er gitt her. For å generere nettverket er utgangspunktet at en har allerede oppgitt plasseringer av svitsjer i planet som VLSI kretsen dekker. Som en forenkling blir svitjene kun ansett som et punkt i planet. Et *kommunikasjonsbehov* forteller hvor stort overføringsbehovet skal være og hvilken svitsj den starter i og hvilken eller hvilke svitsjer som er målet. Det taes ikke spesielt hensyn til de elektriske karaterestikkene som er typiske for integrerte kretser. Det legges vekt på å finne en effektiv metode slik at kriterier for nettverksgenerering senere[1] kan endres.

[1]Som kan være f.eks. de elektriske karakterestikkene som vi ser bort fra nå.

Kapittel 4

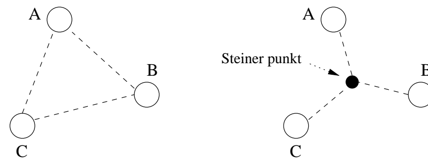
Algoritmer

Ordet *algoritme* stammer fra en persisk forfatter, *Abu Ja'far Mohammed ibn Musa al Khowarizmi* som skrev en bok om matematikk i år 825 e.kr. Innen informatikk blir ordet brukt i betydningen; *en presis metode brukbar for en datamaskin for å løse et problem*[19]. Det finnes mange algoritmer som går igjen og vil fungerer som en slags mal for andre problemer, som for eksempel "den reisende handelsmann" problemet, maks/min flyt problemer, Dijkstra etc. Slike algoritmer går igjen i mange lærebøker [20, 21] og forhåpentligvis finnes det en allerede kjent algoritme for å løse et eksisterende (optimalisering) problem. På denne måten kan mye arbeid kan være spart hvis problemet kan løses med en allerede dokumentert algoritme. Dessverre blir det i noen tilfeller å lete etter en passende algoritme bli som å lete etter nåla i høystakken. Hvilken "mal" av algoritme som passer kan være uklart, og det vil derfor være nødvendig å designe en egen algoritme.

For å designe en algoritme finnes det noen grunnleggende metoder for å gjøre dette. Her gjengis kun en kort oversikt over strategien til disse metodene. For dypere beskrivelse og eksempler på anvendelser av disse algoritmene henvises det til annen litteratur[20, 21]. *Dynamisk programmering* går i korthet ut på å regne ut samtlige kombinasjoner i en tabell, og deretter søke seg bakover igjen i tabellen etter den beste løsningen. Dette benyttes blant annet i tekstsøk eller når en skal finne en minimumstid som et arbeid skal ha. *Grådig metode* går ut på å løse et mindre problem av gangen og forhåpentligvis fører dette til en optimal løsning for hele problemet. *Splitt og hersk* tar først og deler hele problemet i mindre deler og for hver deling blir problemet lettere å løse.

I "The shortest network problem"[22] blir det gitt et innblikk i hvordan utviklingen av en algoritme for steiner-punkt problemet løses, noe som er en god introduksjon til algoritmedesign. Steiner punkt problemet handler om hvordan en kan legge til et ekstra punkt for å finne korteste totale forbindelse

mellom 3 (eller flere) punkter, som for eksempel vist i figuren 4.1. Dette er viktig i forbindelse med planlegging for å bruke minst mulig bruk av rør, eller trekking av ledninger mellom flere punkter. Utgangspunktet for algoritmen er en observasjon om at et slikt punkt finnes. I dag finnes ingen algoritmer som løser dette for alle instanser av dette problemet, selv om det gradvis skjer en forbedring av algoritmene for dette problemet.



Figur 4.1: *Utgangspunktet til venstre. Til høyre vises hvordan punktene A, B og C kan kobles sammen ved hjelp av et steiner punkt.*

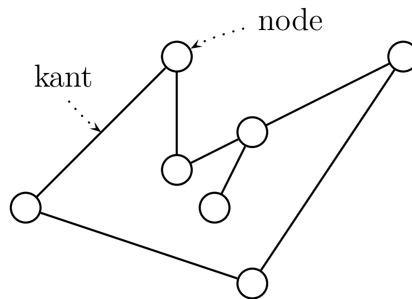
4.1 Kompleksitetsanalyse

Kompleksitetsanalyse for algoritmer går ut på å gi et mål for kjøretiden til en algoritme. Med kjøretid i dette tilfelle er det ikke snakk om tiden det tar, men heller antall steg en algoritme må utføre. Generelt sett er det slik at vi bør unngå å ha algoritmer som har en kjøretid som vokser eksponensielt i forhold til størrelsen av inngangsparametere. Den mest vanlige måten å si noe om kompleksitet er $O(f(n))$ (Stor O), som gjerne har en direkte sammenheng med iterasjoner som algoritmen tar. For eksempel hvis en algoritme benytter en løkke[1] for å gå igjennom en liste har den en $O(n)$ kompleksitet hvor n er antallet av inngangsparametere. Hvis algoritmen benytter enda en løkke innenfor en løkke for hele listen den skal undersøke, vil kompleksiteten være $O(n^2)$. $O(f(n))$ analyse gir et øvre grense for kjøretiden. For noen algoritmer er det hensiktsmessig å notere den nedre grensen av kjøretiden, dette noteres med $\Omega(f(n))$ (Stor omega). For algoritmer som både har en øvre og nedre grense for kjøretiden angis dette som $\Theta(f(n))$.

4.2 Grafer

Grafer er en formell "pekerstruktur" bestående av et sett med *noder* og et sett med forbindelser mellom nodene som kalles for *kanter*, slik som vist i

[1] For eksempel `for`, `while` eller `repeat-until`



Figur 4.2: *Eksempel på graf og tilhørende kanter og noder.*

figur 4.2. Siden en graf kan representere svært tilfeldige forbindelser egner de seg til å representere nettverk, hvor nodene representerer svitsjer/rutere og kantene representerer kablingen mellom disse. Både nodene og kantene kan ha tilordnede verdier mellom seg, disse kalles for *attributter*. Eksempler på attributter kan være alt som skal til for å fortelle om et system, i dette tilfelle med nettverk kan for eksempel attributtene være lengde og overføringskapasitet på kabler, plassering av svitsjer også videre.

4.3 Noen kjente algoritmer

Noen algoritmer går igjen og kan brukes for å løse enkelte oppgaver. For ordens skyld henvises det til litteraturen for en dypere innsikt til disse algoritmene, det er nå ønskelig å kun gi en kort oversikt over hvordan de virker. Disse vil gå igjen i senere kapitler.

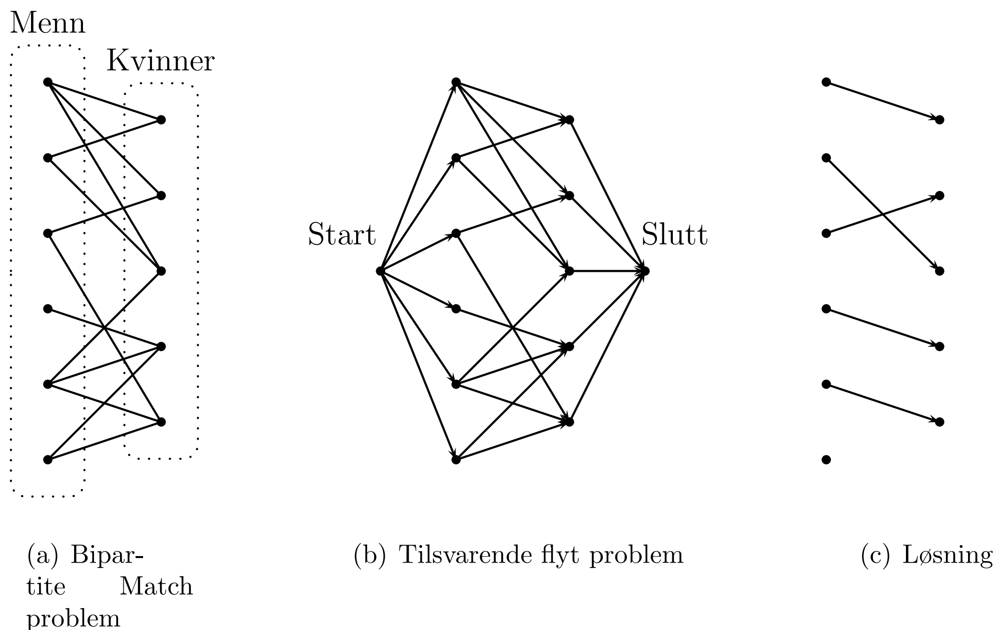
Todelt sammenslåing - Bipartite Matching

Todelt sammenslåing[21] går ut på å finne maks antall parvise treff. En analogi er hvis vi har en gruppe ungkarer og en gruppe ugifte kvinner, hvor hver mann kjenner en eller flere kvinner av gruppen med kvinner. Det vi ønsker å finne ut er om det er mulig for alle mennene å gifte seg med en kvinne de allerede kjenner, eller eventuelt hvor mange som har mulighet for å gifte seg med en de allerede kjenner[2]. Situasjonen kan illustreres med en graf som har to sett med noder, det ene settet representerer mennene og det andre settet representerer kvinnene. Hver node er en person. For å representere ønskene

[2]Dette er kjent som "giftemålsproblemet", at dette ble formulert i 1935 kan forklare noe av den gammelmodige analogien.

er det et sett med kanter mellom disse og forbinder disse to sammen som i figur 4.3(a).

Som algoritme kan problemet løses ved å sette problemet opp som et maksimums gjennomstrømnings problem som i figur 4.3(b). Et maksimum gjennomstrømningsproblem går ut på å finne, som navnet tilsier, den høyeste gjennomstrømningeni et nettverk. Hver kant har lik gjennomstrømnings kapasitet. Ved å finne mest mulig trafikk som er mulig fra start node til slutt node igjennom dette nettverket, finner vi også samtidig flest mulige parvise treff. Dette kan løses ved hjelp av allerede kjente algoritmer som finner maksimum gjennomstrømning gjennom et nettverk[3].



(a) Bipartite Match problem

(b) Tilsvarende flyt problem

(c) Løsning

Figur 4.3: *Todelt sammenslåing*

4.3.1 Geometriske Algoritmer

Graham Skann

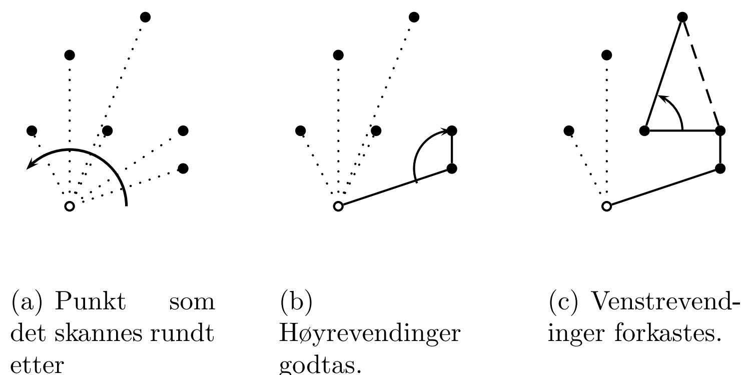
Graham skanning [20] er den mest kjente algoritmen for å finne "ytterkanten" av et sett med punkter i planet. Til å begynne med må man finne et punkt som allerede ligger i ytterkanten av settet med punkter. Dette kan kan ganske enkelt være den med minst x eller y verdi. Rundt dette punktet vil

[3]Som for eksempel Ford-Fulkerson[21]

vi skanne mot klokka (figur 4.4(a)). Hvis tre etterfølgende punkter utgjør en høyrevending vil den være et punkt som ligger ytterst (figur 4.4(b)). Hvis det utgjør en venstrevending (figur 4.4(c) vil dette være et punkt som ligger innenfor yttergrensa.

Finne punkter i et polygon

For å finne et punkt i et polygon[20] tas det utgangspunkt i en algoritme brukt for å finne ut om en linje krysser hverandre eller ikke[4]. Algoritmen "later som" at punktet i planet er startpunktet på en linje og trekker seg utover en retning uendelig langt. Ved å telle antall ganger denne linja krysser polygonet kan man finne ut om det ligger på innsiden eller utsiden. Hvis antallet kryss er 0 eller partall ligger det på utsiden, hvis det er et oddetall ligger det på innsiden. Hvis et punkt ligger langs en kant eller streifer borti et hjørne er det ikke sikkert at den vil være i stand til å bestemme om punktet ligger innenfor eller utenfor, og en av tilstandene for dette vil bli returnert. En illustrasjon er gitt i figur 4.5.



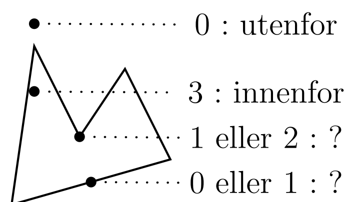
(a) Punkt som det skannes rundt etter

(b) Høyrevendinger godtas.

(c) Venstrevendinger forkastes.

Figur 4.4: *Graham skanning.*

[4]For nærmere forklaring her se for eksempel [20, kapittel 10]



Figur 4.5: *Finn punkter i et polygon.*

Kapittel 5

Generering av topologi

5.1 Nettverksdesign ved generering av alle kombinasjoner av veier.

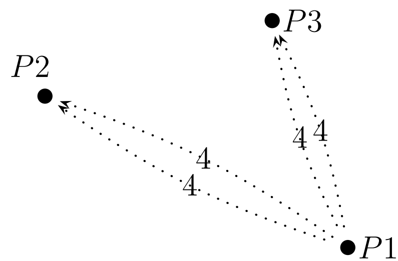
En måte å finne et mulig nettverk er å først generere alle mulige veier gjennom nettverket for hvert eneste kommunikasjonsbehov som finnes for systemet. Deretter sammenlignes de ulike veiene for å finne det nettverket som er best. Ved generering av veier skal algoritmen helst prøve å finne mange nok kombinasjoner slik at et optimalt nettverk blir funnet, finner algoritmen for få veier er det ikke garantert at en er like heldig med utfallet av nettverket.

For å gi en innføring av hvordan denne metoden fungerer begynner vi med et lite eksempel. Detaljene for hvordan metoden skal finne veier kommer senere. Vi tar utgangspunkt i figur 5.1, med følgende kommunikasjonsbehov:

P1⇒P2 ,4 Mbit
P1⇒P3,P2,4 Mbit (Punkt til multipunkt)
P1⇒P3 ,4 Mbit

Metoden søker etter mulige veier og figur 5.2 viser alternative veier hver av kommunikasjonsbehovene kan ta. Legg merke til det siste alternativet i figur 5.2(b) hvor veiene til slutt går mot hverandre. Dette vil ikke være et uvanlig alternativ, hvorfor det blir slik blir klarere når metoden for å søke veier blir forklart i avsnitt 5.1.1.

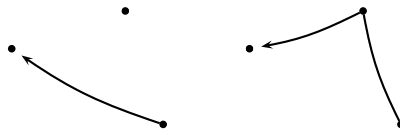
Det som nå gjøres er å velge et alternativ fra hver av kombinasjonene og sjekke om dette utgjør et godt nettverk. Hvis vi velger de første veiene for hver av alternativene får vi et nettverket som vist i figur 5.3(a). Den neste kombinasjonen blir som i figur 5.3(b) og som er et bedre alternativ. Alternativet fra figur 5.3(a) kan da forkastes. Metoden må fortsette å søke igjennom resterende kombinasjoner, slik at ingen bedre løsninger blir utelatt.



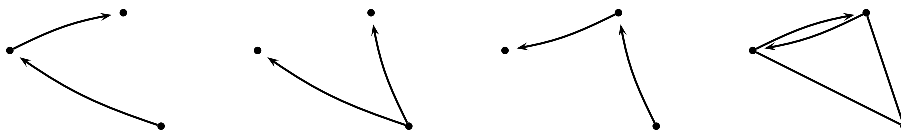
Figur 5.1: *Punktene representerer svitsjenes plasseringer og de prikkede linjene kommunikasjonsbehovene mellom disse.*

Vi har to steg i denne metoden, en for å finne veier, og en for å sammenligne disse. Som pseudokode blir dette:

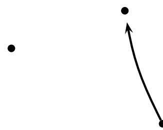
1. Generer samtlige kombinasjoner av veier:
   1.1. Finn et sett med mulige veier for et kommunikasjonsbehov
   1.2. Repeter punkt 1.1, for alle kommunikasjonsbehov
2. Søk alle kombinasjoner for å finne beste nettverk



(a) Alternativer for P1⇒P2

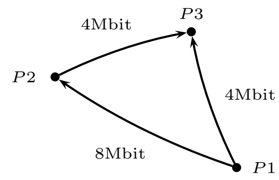


(b) Alternativer for P1⇒P2,P3, (punkt-til-multipunkt).

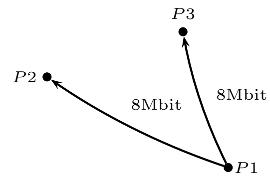


(c) Alternativ for P1⇒P3

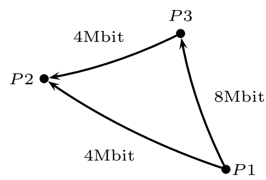
Figur 5.2: *Mulige veier for hver av kommunikasjonsbehovene.*



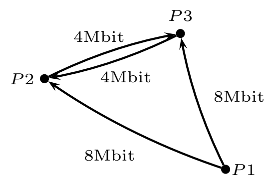
(a)



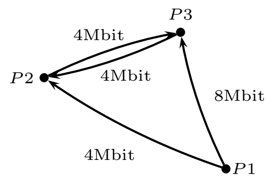
(b)



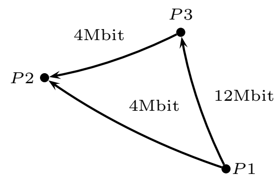
(c)



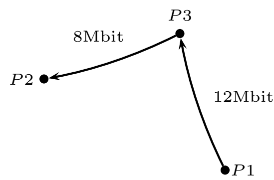
(d)



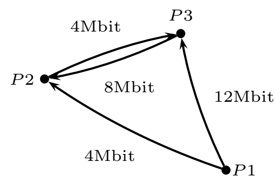
(e)



(f)



(g)



(h)

Figur 5.3: *Sammenligning av kombinasjoner.*

5.1.1 Finne veier mellom to svitsjer

For å finne flest mulig kombinasjoner bør en se om det er mulig å finne veier som ikke nødvendigvis fører til en vei direkte mot målsvitsjen, men som allikevel tar oss nærmere målsvitsjen. Kriteriet for dette er ganske enkelt at avstanden til målsvitsjen ved neste valg av svitsj skal være mindre enn den forrige avstanden. I stedet for å teste avstanden mellom en valgt svitsj og målsvitsj, viser det seg at samtlige svitsjer som ligger nærmere målsvitsjen vil utgjøre et særegent område. Som et implementasjonsvalg kan en da godt benytte seg av algoritmer som kan finne punkter innenfor et polygon (se avsnitt 4.3.1). Metoden er rekursiv og basisen er som følger; *fra startsvitsjen sjekkes samtlige svitsjer i et gitt område, alle svitsjer som ikke fører oss nærmere målsvitsjen forkastes, og alle svitsjer som fører oss nærmere taes med til neste søk*. Husk at avstanden som skal regnes ut er *manhattan-avstanden*, og at området som manhattan avstanden utgjør er en særegen firkant som vist tidligere i figur 2.8.

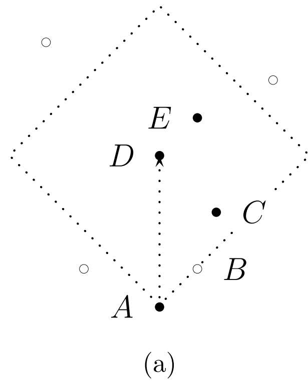
Eksempel

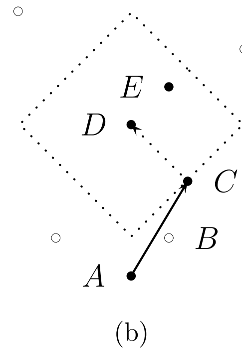
I figur 5.4 ser vi et eksempel på et slikt søk. Her skal vi finne alternative veier fra A til D. Først undersøkes området rundt målsvitsjen D, disse vil inkludere C, E og D, se figur 5.4(a). Svitsjer (punkter) som er aktuelle er merket med en prikk ($\bullet$), og de uaktuelle med en sirkel ($\circ$). Merk at B ikke er inkludert som et alternativ, grunnen er at den havner på grensa av området i figur 5.4(a), det vil være uklart om slike punkter vil bli inkludert eller ikke[1]. La oss nå si at svitsjen C vil være først i køen av nye svitsjer som funksjonen rekursivt søker videre på. Fra C har vi området i figur 5.4(b), dette inkluderer E og D. Velger ut E og kaller funksjonen rekursivt en gang til. I dette kallet har vi kommet til målsvitsjen D se figur 5.4(d). Sekvensen $A - C - E - D$ lagres og funksjonen returnerer til situasjonen i figur 5.4(c), her er det ikke noe mer å søke etter, funksjonen returnerer enda en gang og vi er tilbake i situasjonen i figur 5.4(e), som er lik situasjonen i 5.4(b) men uten svitsjen E. Da er det bare en mulighet igjen og det er direkte til målsvitsjen. Sekvensen $A - C - D$ lagres. Andre sekvenser vil bli $A - D$ og $A - E - D$.

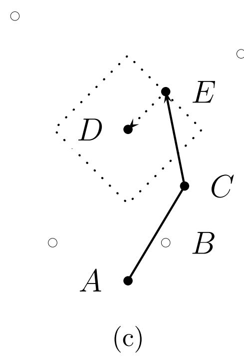
Finne veier for punkt til multipunkt

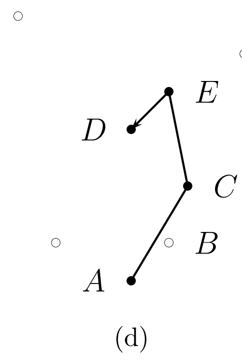
I figur 5.2(b) er det en punkt til multipunkt forbindelse. Denne bygges opp ved først å finne veien mellom to forbindelser av gangen og deretter forene

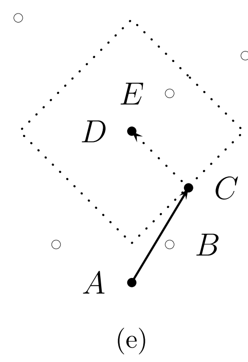
[1] Dette har med implementering av metoden for å velge punkter ut av et areal, se [20]











Figur 5.4: *Eksempel på søk etter vei*

disse. Det vil si først finner vi kombinasjonen av veier for P1⇒P2 og deretter P1⇒P3.

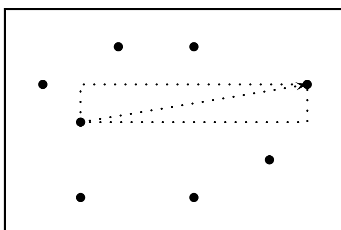
5.1.2 Eksempel på valg av områder

Hvordan en skal velge et begrenset areal kan variere etter behov. Noen eksempler på utregning av areal er vist i 5.6. Et kriterium for å velge område er å sørge for at en kommer nærmere målsvitsjen for hver iterasjon. Det er en fordel at det neste området som velges er mindre enn det forrige. Hvis ikke kan resultatet bli at en valgt vei "krysser" seg selv, ved at det først velges en svitsj som ligger nærmere og deretter en svitsj som ligger lenger unna igjen. Uansett vil en svitsj kun bli valgt en gang og det er ingen fare for at algoritmen går i en iterasjon mellom to svitsjer. Forhåpentligvis blir veier som kanskje ikke er så aktuelle bli fjernet under sammenligningen.

Merk at ved praktisk implementering skal punktene utgjøre et polygon og må derfor bli representert i rekkefølge.

Firkant - horisontal og vertikal avgrensning

En firkant som utgjør diagonalen av start- og målsvitsj kan benyttes hvis en ønsker at totallengden aldri skal bli lengre enn manhattan avstanden mellom start- og målsvitsj, se fig 5.6(a). Ulempen her er at hvis målsvitsjen ligger rett vertikalt eller horisontalt fra startsvitsjen vil det ikke være mulig å finne alternative svitsjer mellom disse to som vist i figur 5.5. Noe som også typisk vil skje hvis en kun ser etter korteste vei mellom to svitsjer.



Figur 5.5: *Ingen alternativer kan bli funnet*

Omkrets av manhattan avstand

Et bedre alternativ er vist i figur 5.6(b), som tar for seg "omkretsen" av manhattan avstanden. På den måten er en sikret å få med seg alternative svitsjer. Lengden L kan bestemmes ut fra avstanden mellom start- og målsvitsj, eller den kan være under en viss maksimumslengde.

Rektangel

Rektangel vil ikke ta med seg området bak målsvitsjen.

5.1.3 Kjøretid

Det vil bare bli tatt for seg kjøretiden for sammenligningen av veiene, og ikke for å finne alternative veiene. Det er under sammenligningen at mesteparten av tiden går med. Kjøretiden til denne algoritmen bestemmes ut fra hvor mange mulige alternative veier vi har per kommunikasjonsbehov. Teoretisk kjøretid regnes ut ved å multiplisere hver av disse veiene med hverandre, noe som gir en eksponensielt vekst.

$$O(N^x) \approx K_1 \times K_2 \times K_3 \times K_4 \dots$$

Som et eksempel på hvor ille det kan bli tar vi utgangspunkt i utlegget i figur 5.7. Sirklene merket med A,B ... også videre er svitsjer, plasseringen er relativ i forhold til plasseringen i arealet. Hver svitsj har flere individuelle kommunikasjonsbehov med samme mengde trafikk som skal sendes til hver av de andre svitsjene.

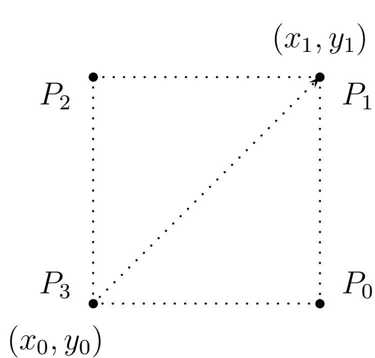
Antall kombinasjoner er som følger; for A til I er det 26 måter legge en vei mellom disse svitsjene, det er 4 slike muligheter til i nettverket mellom andre kommunikasjonsbehov (De andre er I til A, G til C og C til G). For A til F er det 8 forskjellige måter å lage forbindelse, det er 16 andre tilsvarende muligheter i nettverket. For A til E er det 3 måter å tilknytte, det er 16 slike muligheter til i nettverket. Mellom A til C er det 2 mulige kombinasjoner, det er 12 andre slike muligheter i nettverket. Mellom A til B er det 1 måte å koble disse to svitsjene sammen, det er 24 andre slike muligheter i nettverket. En oversikt over alternative veier fra svitsj A til resten av nettverket finnes i appendiks C. Merk at her har jeg brukt firkant som i figur 5.6(a) for å finne veiene[2].

Kjøretiden her vil bli :

$$1^{20} \times 2^{12} \times 3^{16} \times 8^{16} \times 26^4 = 2.27 \times 10^{31}$$

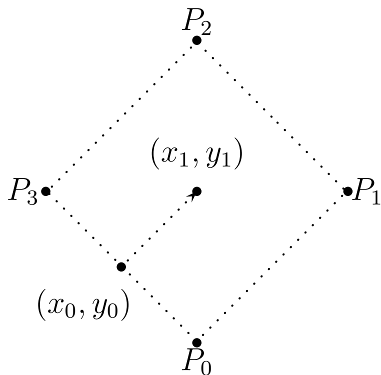
I denne situasjonen blir metoden satt i en litt ekstrem situasjon. Det kan selvfølgelig argumenteres for at vi ikke kommer til å ha et symmetrisk nettverk, eller at vi ikke kommer til å ha såpass mange kommunikasjonsbehov. Her er algoritmen blitt brukt i sin enkleste form. Det er mulig å omskrive slik at kjøretiden blir betydelig mindre, ved å f.eks. bare se på en liten del av

[2]Siden svitsjene ligger i rutenett kan dette være hensiktsmessig. Dette er et eksempel på å endre hvordan området endres etter behov.



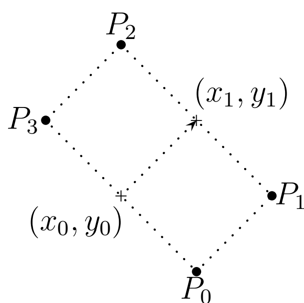
$C_x = \frac{x_1 - x_2}{2} + x_1$
$C_y = \frac{y_2 - y_1}{2} + y_1$
$\Delta_x = |\frac{x_2 - x_1}{2}| + avvik$
$\Delta_y = |\frac{y_2 - y_1}{2}| + avvik$
$P_0(C_x + \Delta_x, C_y - \Delta_y)$
$P_1(C_x + \Delta_x, C_y + \Delta_y)$
$P_2(C_x - \Delta_x, C_y + \Delta_y)$
$P_3(C_x - \Delta_x, C_y - \Delta_y)$

(a) Firkant



$\Delta = |x_1 - x_0| + |y_1 - y_0| + avvik$
$P_0(x_1, y_1 - \Delta)$
$P_1(x_1 + \Delta, y_1)$
$P_2(x_1, y_1 + \Delta)$
$P_3(x_1 - \Delta, y_1)$

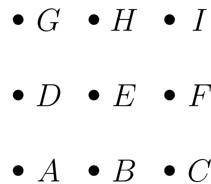
(b) "Manhattan" firkant



$\Delta x = x_1 - x_0$
$\Delta y = y_1 - y_0$
$P_0(x_0 + \Delta y, y_0 - \Delta x)$
$P_1(x_1 + \Delta y, y_1 - \Delta x)$
$P_2(x_1 - \Delta y, y_1 + \Delta x)$
$P_3(x_0 - \Delta y, y_0 + \Delta x)$

(c) Rektangel

Figur 5.6: *Eksempel på områder, og hvordan disse regnes ut.*



Figur 5.7:

nettverket som svitsjene A,B,D og E, resten av nettverket vil være en speiling av dette. Å lage en helt ny algoritme etter hvordan svitsjene er plassert er ikke ønskelig.

5.2 Grådig metode

I bunn og i grunn er ideen veldig enkel; *Det velges ut en node og deretter søkes det etter den optimale trafikkfordelingen til og fra denne*. Som et eksempel hvordan dette utvikler seg tar vi utgangspunkt i utlegget i figur 5.8(a), og viser stegene for å finne en link for innkommende trafikk for svitsj A. De prikkete pilene mellom svitsjene viser kommunikasjonsbehovet mellom disse svitsjene. Svitsjene som her har behov for å sende noe til A er B og E. Det første vi gjør er å se etter hvilke svitsjer hver av disse to kan benytte for å koble seg til A. Alternative svitsjer kalles for *kandidater*. Svitsj B har kandidatene C og seg selv, svitsj E har også C og seg selv som kandidat, dette er vist i figurene 5.8(b) og 5.8(c). Det neste steget er å søke etter den beste kombinasjonen av disse kandidatene. La oss nå for enkelhets skyld si at dette blir tilknytningen C→A. Når denne tilknytningen er valg, må kommunikasjonsbehovene flyttes fra svitsj A til C. Etter denne flyttingen har vi situasjonen i figur 5.8. Videre skritt nå er å finne en tilknyting for utgående trafikk for A og deretter finne en ny svitsj for å gjenta prosessen. Når man så er ferdig med å finne tilknytninger til og fra A skal det ikke tilknyttes nye linker til denne svitsjen. Som pseudokode blir dette:

Finn en svitsj for å optimalisere:
  Finn et sett med kandidat linker *til* denne svitsjen
    Sammenlign kandidater for beste løsning
    Oppdater veier
  Finn et sett med kandidat linker *fra* denne svitsjen
    Sammenlign kandidater for beste løsning
    Oppdater veier
  Flytt kommunikasjonsbehov til den nye tilkoblede svitsjen
  Marker svitsj som ferdig optimalisert

5.2.1 Detaljer for grådig metode

Valg av kandidater

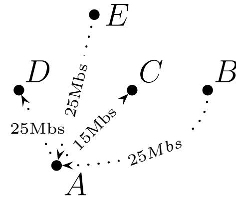
Valg av kandidater kan skje ved å velge ut svitsjer i et område rundt målsvitsjen på samme måte som vist i figurene på side 39. Svitsjer som ligger innenfor dette området regnes som kandidater.

Punkt-til-multipunkt konfigurasjon har en startsvitsj og flere målsvitsjer. Først bør man finne kandidater for hver av målsvitsjene og deretter må resultatene av disse kombineres for å utgjøre kandidatene for sammenligningen. Hvis vi tar for oss nettverket i figur 5.8 og lar A ha punkt til multipunkt konfigurasjon til E og B, betyr dette at vi skal kombinere delløsningene fra figur 5.8(b) og 5.8(c). Dette vil utgjøre fire kombinasjoner som vist i figur 5.9.

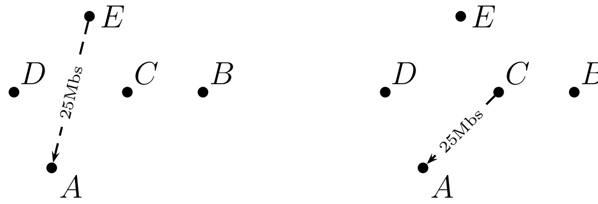
Valg av svitsj for å optimalisere til og fra

Som vi ser i skissen til metoden begynner algoritmen med å velge en svitsj som det skal optimalisere til og fra, deretter er det ikke mulig å koble til nye linker til denne svitsjen. Valg av akkurat denne svitsjen vil påvirke utfallet av nettverket og er kanskje det viktigste steget med denne metoden. Velger vi en tilfeldig svitsj kan dette føre til at en svitsj blir fjernet hvor det senere egentlig burde ha blitt tilkoblet en link.

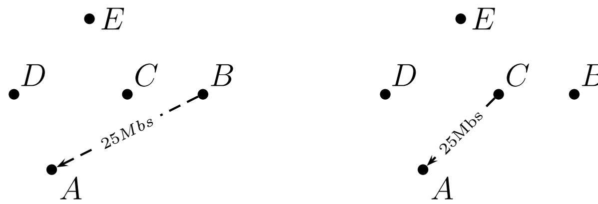
Svitsjen som skal velges ut må derfor ligge i ytterkanten av området av svitsjene som er igjen å undersøke. For svitsjer som ligger innenfor vil det være uklart om link senere vil bli tilkoblet denne eller ikke. For å finne de svitsjene som ligger ytterst benytter en "graham skanning" fra avsnitt 4.3.1. Av svitsjene som ligger ytterst må det også sjekkes hvor passende denne er. Dette gjøres ved å telle hvor ofte en svitsj i ytterkanten vil være en kandidat



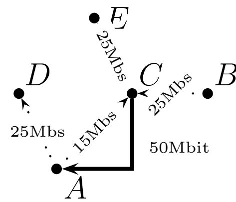
(a) Utgangspunkt kommunikasjonsbehov



(b) Kandidater for E

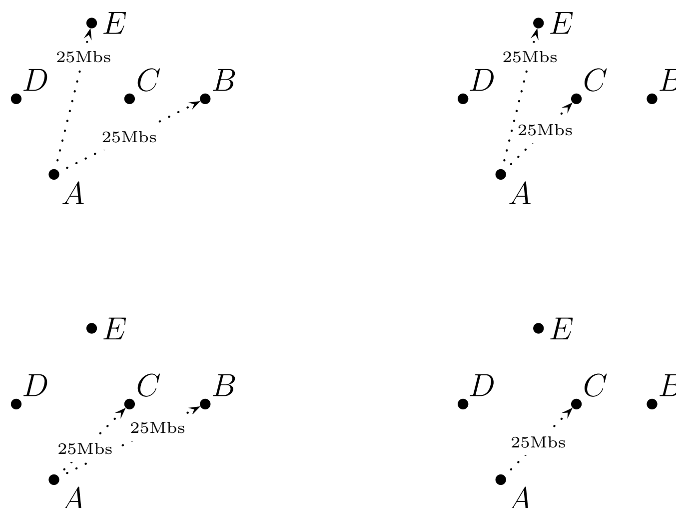


(c) kandidater for B



(d) Delløsning, merk at kommunikasjonsbehovene er flyttet.

Figur 5.8: *Optimalisering av trafikken til A.*



Figur 5.9: *Kombinasjoner hvis en har punkt til multipunkt fra A til E og B.*

for hver av de gjenværende kommunikasjonsbehovene. Det skal ikke telles med tilknytninger som går direkte mellom en startsvitsj og målsvitsj. Den svitsjen som er blitt telt færrest ganger skal velges, ideelt sett skal bli 0.

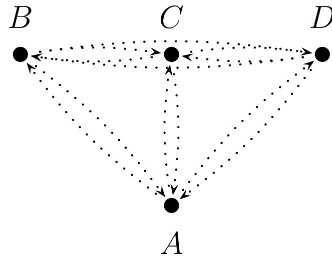
Et eksempel er av nettverket i figur 5.10. Velger vi rekkefølgen B, C, D, A for å finne et nettverk vil nettverket bli som i figur 5.11(a). Grunnen til dette er at når svitsj C skal optimaliseres[3] har den mulige løsninger som vil gå igjennom seg selv. Etter at C er optimalisert vil det kun være A og D som er gjenværende svitsjer og kan kun optimalisere mellom seg. Hvis rekkefølgen av optimaliseringen er B, A, D, C får vi nettverket som er vist i figur 5.11(b). Dette har klart færre linker.

```
liste_svitsjer_utkant = graham_skann( liste_svitsjer )
   regn_ut_sannsynlighet( liste_svitsjer_utkant )
returner svitsj med minst sannsynlighet
```

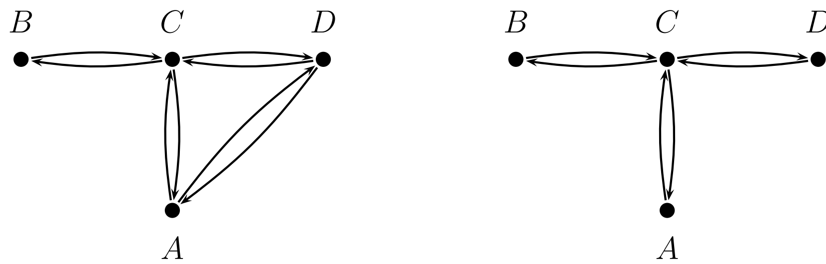
Delløsninger

Ut fra optimaliseringskriteriene er det fullt mulig å ende opp med kandidater som det ikke er mulig å finne en videre løsning på. La oss si at et kriterium

[3]Kriterier for optimaliseringen her er å velge færrest mulig kanter(linker) og de kantene som er kortest.



Figur 5.10: *Utgangspunkt, alle svitsjer overfører lik mengde trafikk mellom hverandre.*



(a) Løses i rekkefølgen B, C, D, A.

(b) Løses i rekkefølgen B, A, D, C.

Figur 5.11: *Forskjellig løsninger. Valg av rekkefølgen av svitsjene påvirker utfallet av metoden.*

for optimaliseringen er at avstanden på linkene ikke skal overstige en maksimumslengde, og ingen av kandidatene oppfyller dette kravet. Metoden fører ikke frem, og videre utvikling av nettet stopper opp.

Det er slik at for hver gang en svitsj er blitt optimalisert er det mulig at det finnes flere løsninger. Disse delløsningene kan lagres i en tre struktur, og hvis algoritmen skulle stoppe opp, er det mulig å søke seg tilbake i treet og finne en tidligere løsning som det er mulig å bygge videre på.

5.2.2 Kjøretid

Det meste av kjøretiden for grådig skjer når kandidatene skal sammenlignes, og har også $O(n^x)$ som kjøretid på samme måte som ved å prøve ut alle kombinasjoner av veier. Det som allikevel gjør at denne metoden kjører raskere er at størrelsen på n er betydelig lavere.

Kjøretiden har sammenheng med hvor mange kommunikasjonsbehov en skal regne ut fra en svitsj og påvirker hvordan n utvikler seg gjennom kjøringen. Etter de første søkene etter mulige tilknytninger flyttes kommunikasjonsbehovene over til nye svitsjer. Dette fører til at kompleksiteten først begynner å øke, for deretter å falle raskt (logaritmisk).

Kapittel 6

Generering av svitsjetabeller

En *svitsjetabell* er informasjon tilhørende en enkelt en svitsj som skal styre hvor trafikken skal sendes. Strategien jeg har benyttet for å finne svitsjetabeller for hele nettverket er å ta for seg en og en svitsj av gangen.

Til forskjell fra de fleste ruting/svitsje algoritmer som tar for seg en innkommende strøm av data og bestemmer hvordan ruteren/svitsjen skal håndtere dette på en best mulig måte. Under genereringen av svitsjetabeller vil det i dette tilfellet ende opp med å måtte bestemme svitsjingen utfra at rammene ved inngående og utgående linker både har kjent og ukjent plassering av tidluker.

6.1 Krav til svitsjetabellen

Utgangspunktet er at det for hvert kommunikasjonsbehov er beskrevet en vei gjennom nettverket og antall dataord den vil benytte. Siden dette vil være trafikk som følger faste mønstre bør det være mulig å lage en algoritme som vil fungere på en hvilken som helst topologi.

- *Det må holdes oversikt over rekkefølgen til hvert av datordene gjennom nettverket*

  Dette for å være sikker på at en hel melding bygges opp korrekt igjen ved mål porten. En kan sørge for at svitsjetabellene genereres slik at de sørger for at datordene ankommer i samme rekkefølge som de ble sendt. Hvis ikke en sørger for å opprettholde rekkefølgen gjennom nettverket er det nødvendig med ekstra logikk ved mottaksporten for å stokke ankommende dataord korrekt ved ankomst. (Det jeg har tatt sikte på er at dataordene ankommer målporten i den rekkefølgen den ble sendt.)

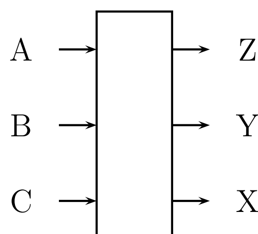
- *Ventetiden i hver svitsj skal holdes til et minimum.*

Normalt forvenes det at et dataord kun venter en klokkesykel før den videresendes på en ny link, men av og til vil det være nødvendig å la et dataord vente to eller flere klokkesykler, fordi den neste tidluken som dataordet skal sendes videre på er opptatt. Slike forsinkelser skal holdes til et minimum.

6.2 En svitsj

Det kan tenkes at det er ønskelig å bare benytte en svitsj for å bygge nettverket[1]. Det som da er ønskelig er å finne den kombinasjonen som gir best mulig gjennomstrømning i svitsjen. Dette kan løses ved å benytte *todelt sammenslåing* (bipartite matching).

Måten å gjøre dette på er at det ene settet av sammenslåingsgrafen er inngående linker og det andre settet er utgående linker. Av det settet som utgjør inngående linker vil det være flere ønsker om en tilknytning til en utgående port i sammenslåingsgrafen.



Figur 6.1: *Illustrasjon av svitsjen.*

6.2.1 Eksempel

La oss si at vi har følgende dataord som skal overføres gjennom svitsjen i figur 6.1. I tabellen 6.1 vises datordene som skal overføres. Metoden starter ved å lage sammenslåingsgrafen for samtlige forbindelser som er nødvendig mellom inngående og utgående linker, for deretter å løse denne. Figur 6.2(a) viser sammenslåingsgrafen med eksplisitt visning av ønskelige overføringer i svitsjen, figur 6.2(b) viser sammenslåingsgrafen i praksis.

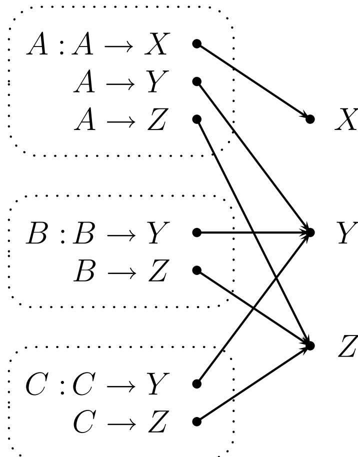
Grafen er vist i figur 6.3(a), her vises løsningen med kontinuerlige streker, de prikkete linjene er de forbindelsene som ikke "vant frem" ved kjøringen

[1]Dette vil tilsvare et tverrkoblet nettverk, se avsnitt 2.2 side 13

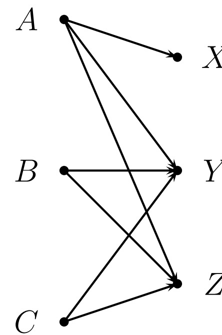
Inn	Ut	Antall dataord
A	X	2
A	Y	1
A	Z	1
B	X	1
B	Y	1
C	X	1
C	Y	1
C	Z	1

Tabell 6.1: *Overføring for svitsjen i figur 6.1.*

av sammenslåingsgrafen. Vi har nå løst for tidluke 0 til tidluke 1. Merk at for tidluke 1 til tidluke 2 må vi fortsatt ha en kant mellom A til X siden det fortsatt er et dataord som skal overføres mellom her. Metoden fortsetter på samme måte helt til alle overføringer er løst. Merk at for siste sammenslåingsgraf skal det ikke være nødvendig å løse. Det skal ikke være flere etter denne. TDMA rammen for denne er vist i tabell 6.2.

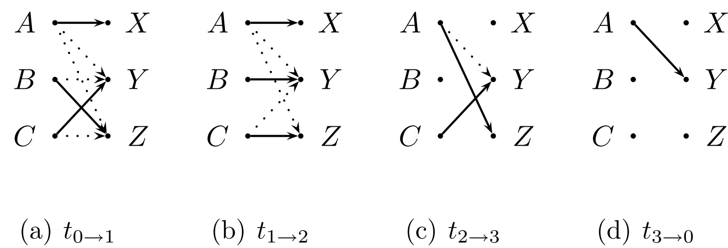


(a) Ønskelige overføringer



(b) Sammenslåing i praksis

Figur 6.2: *For å løse en svitsj*



(a) $t_{0\to1}$ (b) $t_{1\to2}$ (c) $t_{2\to3}$ (d) $t_{3\to0}$

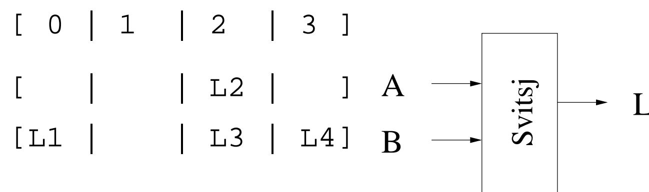
Figur 6.3: *Utførelse av svitsjetabell med sammenslåingsgraf.*

	Tidluke					
	0	1	2	3	0	...
A	A_0	A_1	A_2	A_3		
B	B_0	B_1				
C	C_0	C_1	C_2			
X		A_0	A_1			
Y	(A_3)	C_0	B_1	C_2	A_3	
Z		B_0	C_1	A_2		

Tabell 6.2: *Ramme for løsningen.*

6.3 For kjent trafikk

For kjent trafikkstrøm kan vi benytte prinsipp fra dynamisk programmering. Ved å lage en tabell hvor innkommende tidluker settes i kolonner i stigende rekkefølge etter ankomstid for tidlukene, og deretter regne ut forsinkelsen til tidlukene i radene. Dybden i radene viser da tidlukene på utgående ramme. Dermed fåes en tabell som viser forsinkelsen og samtidig er det mulig å kontrollerer rekkefølgen til trafikkstrømmen. Figur 6.5 vises et eksempel på tabellen med oversikt over mulige forsinkelser for utgang L fra figur 6.4. Det som nå gjenstår er å bruke tabellen til å finne fordelingen med minst forsinkelse.



Figur 6.4: *Svitsj med tilhørende rammer på inngangen.*

Tabellen brukes på følgende måte; Ved å ta utgangspunkt i figur 6.6(a)

Inn

		L_1	L_2	L_3	L_4
		0	2	2	3
Ut Tidluker	0	4	2	2	1
	1	1	3	3	2
	2	2	4	4	3
	3	3	1	1	4

Figur 6.5: *Forsinkelser*

starter vi med innkommende tidluke L_1 og del minste forsinkelsen den kan ha, dette er ved utgående tidluke 1. Vi kan nå prøve på tidluke L_2, den minste forsinkelsen er nå på utgående tidluke 3. Merk at for inngående tidluke L_3 er den beste utgående tidluken allerede opptatt, derfor må den ta neste utgående og forsinkelsen for denne er 2. For tidluke L_4 vil den eneste ledige utgående tiluken være 2, men her vil rekkefølgen av dataordene faktisk krysse hverandre. Dette søket feiler hvis man skal opprettholde rekkefølgen.

	0	1	2	3	0	1	2	3
A:			L_2			L_2		
B:	L_1		L_3	L_4	L_1		L_3	L_4
L:		L_1	L_4	L_2	L_3	L_1	L_4	L_2

Siden dette skal være svitsjing på fast basis, kan en prøve et nytt søk ved å starte fra en annen tidluke. I figur 6.6(b) starter vi i tidluke 2. Her lykkes man med å finne en fordeling som opprettholder rekkefølgen.

Implementering

Det er ikke nødvendig å generere en tabell siden det den er repeterende. Det er bare forsinkelsen som skal regnes ut. Søket etter utgående linker forgår ved å først sette tidlukene i rekkefølge etter ankomst i en liste, sammen med informasjon om ankomstid.

A[] = [L_1(0), L_2(2), L_3(2), L_4(3)]

- Den første tidluken får da tildelt utgående tidluke ved å legge til forsinkelsen den skal ta igjennom svitsjen (som er 1).

Inn

Ut

Tidluker	L_1	L_2	L_3	L_4
	0	2	2	3
1	1	3	3	2
2	2	4	4	3
3	3	1	1	4
0	4	2	2	1

(a) Start tidsluke 0

Inn

Ut

Tidluker	L_2	L_3	L_4	L_1
	2	2	3	0
3	1	1	4	3
0	2	2	1	4
1	3	3	2	1
2	4	4	3	2

(b) Start tidsluke 2

Figur 6.6: *Forsinkelser*

- Det samme som foregående punkt gjøres med de neste tidlukene, men nå er det mulig at tidluken havner på en utgående tidluke som allerede er okkupert. Da skal neste utgående tidluke havne rett etter den forrige. Forsinkelsen finner man ved å finne differansen mellom neste tidluke og trekke fra ankomstiden. Denne situasjonen skjer for eksempel når en skal regne ut forsinkelsen for L_3 i figur 6.6(a).
- Tidluker vil krysse hverandre dersom den tidluken vi undersøker har en differanse mellom første tidluke som er større enn rammestørrelsen.

pseudokode:

$A[0]_{forsinkelse} = 1$
fra $t = 1$ til $Rammestorrelse$: (
  $A[t]_{forsinkelse} = 1$
    hvis $A[t]_{avgang} > A[t-1]_{avgang}$:
      $A[t]_{forsinkelse} = A[t-1]_{avgang} + 1 - A[t]_{ankomst}$
    hvis $A[t]_{avgang} > A[0]_{avgang} + Rammestorrelse - 1$:
      $retur \rightarrow feil(kryss)$
)
$retur \rightarrow ok$

Siden rammen er repeterende må en regne ut forsinkelsen en gang til ved å starte på nytt med neste innkommende tidluke. Måten å gjøre dette på er

å ta en rotering av inngående ramme, den første tidluken legges da bakerst i listen som utgjør rammen og det regnes ut ny ankomsttid ved å legge til rammestørrelsen. På denne måten slipper man å benytte modulens ved utregning av forsinkelse. Det vil fungere siden $A[siste]_{ankomst} - A[0]_{ankomst} <= Rammestørrelsen$. Rotering av figur 6.6(b) vil egentlig bli som i 6.7.

Inn

Ut	Tidluker	L_2	L_3	L_4	L_1
		2	2	3	4
	3	1	1	4	3
	4	2	2	1	4
	5	3	3	2	1
	6	4	4	3	2

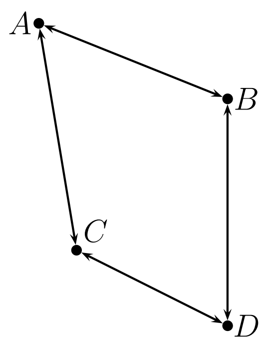
Figur 6.7: *Etter rotering av rammen.*

6.4 Løse for hele nettverket

Prøver følgende strategi for å finne svitsjetabeller samtlige svitsjer i nettverket; løse for både inn- og utgående linker for en og en svitsj av gangen. Da det bli fire følgende situasjoner for dataord som skal overføres gjennom svitsj.

1. Har allerede en fast plassering i rammen både på inn- og utgående link.
2. Har allered en fast plassering på inngåend link, men ikke på utgående.
3. Har allered en fast plassering på utgående link, men ikke inngående.
4. Har ingen faste plasseringer, hverken på inn- eller utgående link.

For 1 er det klart at det ikke er noe å gjøre med og akseptere som den er. Helst bør en slik situasjon unngås. For å unngå en slik situasjon må det taes hensyn til rekkefølgen av svitsjene som løses for eksempel i figur 6.8 vil det hvis en løser først for svitsj A, her har vi kun situasjonen i 4. Alle linker mellom A og B, samt mellom A og C er nå løst. Deretter løses det for svitsj B også for C. Da er samtlig linker fått hver sin fordeling av rammene. Men for trafikk som passerer gjennom D vil situasjonen være som i 1. Det er kan antas at strategien om å løse for en og en svitsj ikke fører frem.



Figur 6.8: *Enkelt nettverk*

Kapittel 7

Diskusjon

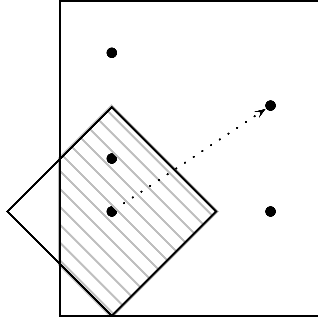
7.1 Oppsummering

Med utgangspunkt i tilfeldig plassering av svitsjene i arealet og et gitt kommunikasjonsbehov mellom disse, har jeg foreslått en algoritme for å finne et gunstig nettverk for å knytte svitsjene sammen. I stedet for å søke seg igjennom samtlige nettverkskombinasjoner for å oppfylle kommunikasjonsbehovene har den foreslåtte algoritmen en raskere kjøretid. Det er også foreslått enkelte måter å generere svitsjetabell for statisk TDMA svitsjing.

7.2 Nettverksgenerering

7.2.1 Valg av område

Jeg har sett på å velge ut deler av område som en måte å begrense søket på, og har valgt å se på utvelgelsen som en funksjon av avstanden mellom to svitsjer. Dette innebærer at det ikke er noen begrensning på hvor lange ledningene må være. RC effekter for ledninger vil kort forklart begrense spenningen på ledninger når en operer på høye frekvenser, dette er en funksjon som bestemmes av ledningslengden. Av den grunn kan det være ønskelig å begrense lengden på ledningene. Dette bør gjøres når en velger ut område, og kan velges ut som en union av to områder. Det ene området sørger for å ta deg nærmere, mens det andre er en avgrensning av ledninglengden rundt svitsjen vi starter fra. Område utgjør fortsatt et polygon, dette betyr at det ikke vil være noen endring i metodene for å generere et nettverk. Det skraverte området i figur 7.1 er et eksempel på slik utvelgelse.



Figur 7.1: *Valg av område, med begrensning av lengden*

7.2.2 Alle veier

Valg av område påvirker resultatet, med utgangspunktet i figur 5.7 vil kanskje det beste nettverket være en stjerne-topologi(se figur 2.2(b) side 14) med svitsjen E i senter. Av samtlige kombinasjoner som er blitt generert i kapittel 5.1.3 vil ingen av disse utgjøre en stjernetopologi. For å finne veiene her er det benyttet en firkant som areal for utvelgelse av alternative svitsjer og for trafikkbehovet mellom A til B finner man aldri et alternativ som sørger for å gå mellom A til E.

Det bør vurderes å ikke benytte kun en måte å velge ut et område på av gangen. Et forslag er å benytte andre former på området hvis for eksempel en svitsj ligger internt. Eller la det være basert på avstanden mellom to svitsjene.

Å generere alle kombinasjoner for å finne et nettverk tilsvarer å prøve ut (nesten) alle kombinasjoner av nettverket en kan ha.

7.2.3 Grådig metode

"Grådig metode" benytter seg av mer kode for å fungere. Mer kode betyr større mulighet for feil. Jeg har ikke implementert grådig metode med mellomlagring av delløsninger som foreslått på side 43. Grådig metode benytter mer antagelser for å fungere. For eksempel når en skal finne en ny svitsj å optimalisere fra, må en gjette seg til hvor stor sannsynlighet det er for at annen trafikk vil gå igjennom akkurat den svitsjen. I tillegg vil den kun velge å generere videre etter å løst en liten del av problemet. Den delen som den genererer videre på trenger ikke nødvendigvis føre frem til den optimale løsningen.

7.2.4 Generelt

Ved valg av enten "alle kombinasjoner" eller "grådig metode" vil komme an på antall svitsjer og kommunikasjonsbehov som er til stede. Er dette antallet lavt kan det være hensiktsmessig å benytte "alle kombinasjoner", siden det vil være større garanti for at et godt nettverk kan bli funnet. Hvis det er et fåtall svitsjer kan andre kommunikasjonstrukturer være mer hensiktsmessig. Jeg har fått forespeilet at antall svitsjer kan ligge opp mot 200. Hvis det er slik at en må sammenligne et høyt antall kombinasjoner vil "grådig metode" være å foretrekke. Sjansen for å finne et optimalt nettverk regnes som lavere her enn ved "alle kombinasjoner", men det antas at det er en god sannsynlighet for å finne et godt nettverk.

7.3 Svitsje tabeller

For å finne svitsjetabeller ser det ikke ut til at dette kan løses ved å se på en og en svitsj av gangen. Skal ikke se bort fra at jeg har hengt meg opp i en eller annen problemstilling, som allikevel er ubetydelig eller lettere å løse enn det jeg antar.

En annen strategi her er å prøve å løse det for hele nettverket som en matematisk ligning. Ligningen beskriver med ukjente forsinkelsen til de ulike tidlukene og skal unngå kollisjoner ved at to forskjellige dataord ender på samme tidluke. Her må det opereres på heltall, siden en forsinkelse ikke kan regnes ut som desimaltall samt at en ramme kun har en viss størrelse. Jeg antar at dette havner innen et matematisk felt som kalles *diskre optimalisering*. Et felt jeg ikke vil gå nærmere inn på.

7.4 Forslag til videre arbeid

7.4.1 Vurderinger av andre kommunikasjonstrukturer

I forhold til bussbasert design har nettverket presentert her fordeler ved at den vil mest sannsynlig ha kortere ledninger, men også kanskje det viktigste, en mulighet for å utføre kommunikasjon parallelt. Til tross for ulempene med buss som ble introdusert i avsnitt 2.1, betyr det ikke at bussbasert kommunikasjonstruktur har utspilt sin rolle riktig enda.

Den vanlige oppfatningen av tverrkobling er at den knytter ledninger til samtlige funksjonsblokker på kretsen, men dette er noe den slett ikke trenger å gjøre siden det ikke er sikkert at alle porter trenger å kobles sammen. En

tverrkobling kan like så godt kun legge ledninger mellom de funksjonsblokkene som trenger å kunne kommunisere sammen.

En måte å lage et enkelt tverrkoblet nettverk er å benytte en svitsj og for fordelingen benytte algoritmen i kapittel 6.2. Jeg er ikke sikker på hvor stor forskjell det er i ytelse mellom tverrkobling og nettverket foreslått her.

Hovedpoenget med en sammenligning er å gi et grunnlag for hva slags kommunikasjonstruktur som vil være hensiktsmessig å velge til enhver tid.

7.4.2 Effektivitet av svitsjer

Nettverksgenereringen har kun sett på hvor effektive linkene blir. Hvor effektive svitsjene er bør også taes med i betraktningen.

7.4.3 Deling av kommunikasjonsbehov

Kommunikasjonsbehov representert her består av at trafikken skal kun ta en vei gjennom nettverket. Båndbreddebehovet kan være forskjellig og kan gi en skjev fordeling mellom linkene. Hvis trafikken fordeles mer jevnt mellom linkene burde det være mulig å gjøre dette ved å dele kommunikasjonsbehovene i mindre deler. En må da finne kriterier for når et kommunikasjonsbehov skal deles. Samtidig setter dette krav til at dataordene må bygges opp igjen i riktig rekkefølge ved mottaksporten. Dette endrer ikke måten å generere nettverk på slik som er foreslått.

7.4.4 Ruting for irregulære nett

Det finnes allerede prinsipper for ruting i nettverk av arbeidstasjoner disse kan kanskje tilpasses for bruk i NoC's. Irregulær ruting kan være en fordel for å la nettet i seg selv håndtere variasjoner i trafikken. Men merk at hvis det innføres en annen ruting antar jeg at den vil benytte seg anderledes av nettverket enn det som den er generert etter. Nettetverket er generert ut fra en gitt trafikkfordelingen, å innføre en annen rutingalgoritme kan krenke dette.

7.4.5 Trafikk av eller på

For videre arbeid er det ikke nødvendigvis å implementere nettverket for å støtte en mer dynamisk trafikk (ved hjelp av irregulær ruting) nødvendigvis den rette veien å gå. Med inspirasjon fra Prophid [17], som er en krets designet for multimedia, er det kanskje en ide å la nettverket støtte trafikk som enten er der (konstant) eller ingen trafikk. Når det er ingen trafikk unngås svitsjing

i nettverket - dette med tanke på effektforbruket. Det er nettopp slik det fungerer i bærbare elektroniske apparater, enten er en tjeneste på eller av[1].

7.5 Annet relatert arbeid

Det er ikke til å unngå at ved slutten av hovedfaget sitter med litteratur og andre innfallsvinkler på oppgaven.

De aller fleste bøker omkring samkjørte nettverk som [23] behandler nettverk for parallelle systemer. Litteratur omkring nettverk for irregulære systemer er svært lite. Det finnes noen eksempler som analyserer irregulære nettverk [3], men her tar en for seg nettverk av arbeidstasjoner.

Det finnes noen eksempler på generering av irregulære nettverk. Jeg har ikke lykkes i å få anledning til å studere andres arbeid på dette feltet i noe særlig grad. Eksempler her er [24] som tar for seg nettverksgenerering generelt (som LAN, WAN, trådløst etc), med utgangspunkt i et rigid matematisk fundament. Den tar også for seg generering av nettverk for integrerte kretser, men tar ikke utgangspunkt i at svitsjer kun kan ha en fast plassering i nettverket, men heller at det legges til en svitsj der hvor det er mest optimalt.

Et annet forsøk på genering av nettverk for integrerte kretser[25], ved å kartlegge mulige konflikter mellom kommunikasjonsbehov og resurser i nettverket. Denne metoden benytter en splitt og hersk metode som fremgangsmåte. Her ser det heller ikke ut til at plasseringen av svitsjer i særlig grad blir behandlet.

Design metoden brukt i [26] kan være verdt en dypere studie. Her gir de en kartlegging av kommunikasjonsbehovet, og foreslår en topologi som kan bestå av buss eller egne dedikerte kanaler med tilhørende protokoll.

[1]Eksempel: mobiltelefonen NOKIA 7700 kommer med noen av følgende tjenester: video avspilling, TCP/IP(IPv4 & IPv6), VGA kamera, Radio, MMS, Bluetooth, GPRS, E-GPRS, HSCSD (High-Speed Circuit-Switched Data) og MP3 spiller. Noen av disse trenger muligens egen hardware. Alle tjeneste brukes selvfølgelig ikke samtidig.

Kapittel 8

Konklusjon

Hvis en skal generere nettverk ved å prøve ut samtlige kombinasjoner av topologien vil dette for store nettverk ikke være hensiktsmessig. Dette vil gi såpass mange kombinasjoner at en ikke kan finne et nettverk innen rimelig tid.

Som en mer effektiv måte å generere nettverk på er det foreslått å benytte en fremgangsmåte som går ut på å løse for en og en svitsj av gangen. Denne fremgangsmåten er langt mer hensiktsmessig med tanke på tidsforbruk. Når det kun løses for en liten del av gangen er generingen mer basert på antagelser. Derfor vil det også være rimelig å anta at metoden ikke finner den optimale løsningen, men forhåpentligvis en løsning som allikevel er tilfredstillende.

For et tilfeldig nettverk med statisk TDMA svitsjing er det ønskelig at svitsjetabellene er med på å holde ventetiden så lav som mulig. Det er foreslått enkelte algoritmer som delvis kan løse oppgaven med å generere svitsjetabeller for en slik ruting, men har ikke lykkes med å finne en god metode som kan løse dette for et helt nettverk.

Bibliografi

[1] Tom Shanley and Don Anderson. *PCI System Architecture*. MindShare Inc. ISBN-0-201-40993-3, 1995.

[2] David E. Culler and Jaswinder Pal Sing with Anoop Gupto. *Parallel computer achitecture: a hardware / software approach*. Morgan Kaufmann Publishers, Inc. (ISBN 1-55860-343-3), 1999.

[3] Wai Hong Ho and Timothy Mark Pinkston. A clustering apporach for identifying and quantifying irregularities in interconnection networks. *IEEE Transactions on parallel and distributed systems*, December 2003.

[4] M. Schwartz. *Telecommunication networks : protocols, modeling and analysis*. Reading, Mass. : Addison-Wesley, 1987.

[5] William Stallings. *ISDN, an introduction*. Macmillian Publishing Company, a division of Macmillian , Inc., 1989.

[6] http://www.wikipedia.com.

[7] Shashi Kumar, Axel Jantsch, Juha-Pekka Soininen, Martti Forsell, Mikael Millberg, Johny Oberg, Kari Tiensyrja, and Ahmed Heman. A network on chip architexcture and design methodology. *IEEE Computer society Annual Symposium on VLSI*, 2002.

[8] Luca Benini and Giovanni De Micheli. Networks on chips: A new SoC paradigm. *IEEE Computer*, Jan 2002.

[9] Jörg Henkel, Wayne Wolf, and Srimat Chakradhar. On-chip networks: A scalable, communicatiion-centric embedded system design paradigm. *Proceedings of the 17th International Conference on VLSI Design (VLSID'04) IEEE*, 2004.

[10] WilliamJ. Dally and Brian Towles. Route packets, not wires: On-chip interconnection networks. *38th. DAC*, June 2001.

[11] Jari Nurmi, editor. Kluwer Academic, 2004. Kapittel 14: Socket based design using decoupled interconnects - Drew Wingard.

[12] ARM (http://www.arm.com). *AMBA(tm) specification Rev. 2.0*, mai 1999.

[13] IBM. Coreconnect (tm) bus architecture, 1999.

[14] palmchip corp. White papers. http://www.palmchip.com.

[15] Silicore Corp. *Wishbone Specification*, 2001 October.

[16] Faraydon Karim, Anh Ngyen, and Sujit Dey. On-chip communication architecture for OC-768 network processors. *38th DAC*, June 2001.

[17] J.A.J. Leijten, J.L van Meerbergen, A.H. Timmer, and J.A.G. Jess. Prophid: A platform-based design method. *Design Automation for Embedded Systems, 6,5-37*, 2000.

[18] Margherita Barile. Taxicab metric. http://mathworld.wolfram.com/TaxicabMetric.html.

[19] Ellis Horowitz and Sartaj Sahni. *Fundamentals of computer algorithms.* Computer Science Press, ISBN 0-914894-22-6, 1984.

[20] Jon Orwant, Jarkko Hietaniemi, and John Macdonald. *Mastering Algorithms with Perl.* O'Reilly ISBN: 1-56592-398-7, 1999.

[21] Alan Dolan and Joan Aldous. *Networks and algorithms : an introductory approach.* John Wiley & Sons Ltd. ISBN-0 471 93992 7, 1993.

[22] Ronald L. Graham. The shortest network problem. Videogram, 1988.

[23] José Duato, Sudhakar Yalamanchili, and Lionel Ni. *Interconnection Networks an engineering approach.* IEEE Society Press, ISBN 0-8186-7800-3, 1997.

[24] Alessandro Pinto, Luca Carloni, and Alberto SangiovanniVincentelli. Constraint-driven communication synthesis. In *Proceedings of the 39th. Design Automation Conference 2002 (DAC'02), New Orelans, Lousiana USA*, June 2002.

[25] Wai Hong Ho and Thimithy Mark Pinkston. A methodology for designing efficient on-chip interconnects on well-behaved communication patterns. *IEEE: The Ninth International Symposium on High-Performance Computer Architecture*, 2003.

[26] K. Lahiri, A. Raghunathan, and S. Dey. Efficient exploration of the soc communication architecture design space. *Proc. Intl. Conf. on Computer-Aided Design, p 424-430*, 2000.

Tillegg A

Forslag til analyse / optimalisering

Det meste av litteratur omkring samkjørte nettverk baserer seg på parallelle systemer. En er nødt til se på hvordan dette egentlig burde tilpasses for dette nettverket.

A.1 Ventetid og diameter

Den gjennomsnittlig *ventetid* regnes ut for hvert individuelle kommunikasjonskrav. Dette bli i forskjell fra tradisjonell utregning, hvor man tar utgangspunkt i gjennomsnittlig ventetid mellom samtlige noder i nettverket. Dette vil ikke være hensiktsmessig her, siden vi kun ser på individuelle kommunikasjonsbehov. Den faktiske ventetiden blir ikke funnet før svitsjetabeller er utregnet, fordi det er mulig at dataord blir hold tilbake mer enn en klokkesykel i svitsjen. Men det er mulig å forholde seg til relativ ventetid ved å kun se på antall svitsjer veien går igjennom, dvs. *diameteren* i stedet.

For generering med "alle kombinasjoner" kan dette lett finnes. For grådig metode derimot blir en nødt til å ta å gjette med sannsynlighet. En måte å gjøre dette på blir å se på avstanden som er igjen mellom en kandidat og målsvitsjen.

A.2 Antall linker

Under sammenligning vil vi kunne finne ut hvor stor overføring det skal være i forbindelsen mellom to svitsjer. Hvis vi har predefinert hvor mange ledninger det er i hver link og hvilken klokkefrekvensen det skal være i forbindelsen kan vi regne ut hvor mange linker det skal gå mellom disse to. Det vi skal finne ut

er hvor mange linker som må til for å opprettholde overføringshastigheten. Det finner man ved å ta båndbreddebehovet og dele på båndbredden for en link og avrunde resultatet "oppover" (vist med "tak funksjonen" $\lceil\ \rceil$).

$$L = \left\lceil \frac{B_f}{W \times f} \right\rceil$$

L : Antall linker.
B_f : Båndbreddebehov mellom forbindelsen
f : Klokkehastighet
W : Antall ledninger for en link.

A.3 Ubrukt båndbredde

Det vil være ubrukt båndbredde i nettverket. Dette er ubrukt kapasitet i nettverket og noe som selvfølgelig skal holdes til et minimum. For å finne ubrukt båndbredde mellom en forbindelse regner man ut den totale båndbredden og trekker fra båndbreddebehovet.

$$B_u = W \times L \times f - B_f$$

Tillegg B

Eksempler på kjøringer

B.0.1 All veier

Det skal være mulig å generere veier som ikke fører til en vei direkte til målsvitsjen, men tar en "god omvei" for å nå målsvitsjen. Det skal allikevel være mulig å finne et godt nettverk allikevel. Ved å velge ut et område slik som i figur 5.6(b) og med et stort avvik kan dette vises. Hvis vi benytter dette for nettverket i figur B.1(a) med avvik lik 1 vil dette dekke et stort nok område til å inkludere alle svitsjene som i figur B.1(b). På denne måten vil en få veier som vist i figur B.1(c).

Merk at fra C til A,B og D er det på grunn av måten område blir valgt kun disse veien :

```
C-A
C-B
C-D
```

Målet er å finne færrest mulig ledninger i bruk med utganspunkt i at hver link kun består av en ledning og opererer med en klokkehastighet på 6 Mhz, dvs med en overføringskapasitet på 6 Mbit/s. Hvert av overføringbehovene er på 8 Mbit/s. Det ble generert 512000 kombinasjoner[1]. Av dette var det 375 kombinasjoner som kun benyttet 24 ledninger for hele nettverket. Kun en av disse løsningene hadde ingen ubrukt kapasitet i nettverket, vist i figur B.1(e).

[1]Det tok ca. 27 minutter å gå igjennom alle kombinasjonene. `Perl` er brukt som programmeringspråk, med et annet programmeringspråk eller en annen implementering er det rimelig å anta at tidenforbruket kan reduseres betraktelig

Start - Slutt	Trafikk	Ledninger	Kapasitet	Ubrukt
A - C	24000000	4	24000000	0
C - B	24000000	4	24000000	0
C - D	24000000	4	24000000	0
B - C	24000000	4	24000000	0
C - A	24000000	4	24000000	0
D - C	24000000	4	24000000	0
Sum:	144000000	24	144000000	0

Tabell B.1: *Resultat av kjøringen*

Grådig metode

Ved forsøk med grådig metode og samme mål for sammenligningen fåes nettverket i figur B.1(f). Dette er forskjellig fra nettverket i figur B.1(e), men svaret er korrekt med hensyn til antall ledninger i bruk. I tillegg, til tross for at det ikke var med hensikt, ingen ubrukt kapasitet i nettverket. Det skulle ha vært 4 forskjellige løsninger for nettverket her. Hver av løsningene har en av svitsjene i "sentrum", det vil si for figur B.1(e) er C i sentrum.

Det som er viktig å legge merke til er at ingen av disse to metodene klarte å finne samtlige 4 løsninger. For "alle kombinasjoner" blir det feil på grunn av at den finner for få alternativer fra C, veier som C-B-A burde vært inkludert. Problemet med grådig metode (på den måten som jeg har implementert løsningen) vil ikke flere delsvar bli tatt vare på under forsøket på å finne nettverket, den vil kun velge ut en av de best løsningene og gå videre på denne.

B.1 Kjøringer grådig metode

Benytter netverket fra figur 5.7, overføringene som skal være mellom svitsjene er på 100 bit. Tester kjøringene med å benytte en link med 1 ldning og operere på 300Hz. Ved å se på kommunikasjonsbehov som er symmetriske kan vi få en viss ide om hvordan algoritmen fungerer.

Figur B.2(a)

Prøver for hver sammenligning å først finne færrest mulig ledninger, hvis det er likt antall ledninger sees det etter.

1. Minst mulig ubrukt kapasitet, hvis det er mindre ubrukt kapasitet forkastet forrige løsning.

(a) Posisjoner

(b) Arealvalg fra B til C. Vil inkludere A og C.

(c) Ett veivalg for B til C

(d) Arealvalg for C til D

(e) Løsning for "alle kombinasjoner"

(f) Løsning for grådig metode

Figur B.1: *Nett*

2. Hvis det ikke er dårligere ubrukt kapasitet, sjekkes det om det er minst mulig distanse igjen. Her sees det etter total distanse som er igjen, ikke gjennomsnittet.

For min kjøring fikk jeg følgende resultat. Svitsjene ble løst i følgende Rekkefølgen C, I, A, G, F, H, B. Merk at E ikke er med, det er siste svitsjen som står igjen. Det er ikke nødvendig å løse for denne. Ubrukt kapasitet er 14.3%, og benytter totalt 42 ledninger. Alle forbindelsene benytter 1 ledning unntatt E→F, E→H, B→E, D→A og F→E. Alle veiene passerer to eller tre svitsjer å gå igjennom, unntaket er for H til A som tar en litt unødvendig vei igjennom $H \rightarrow B \rightarrow E \rightarrow A$.

Figur B.2(b)

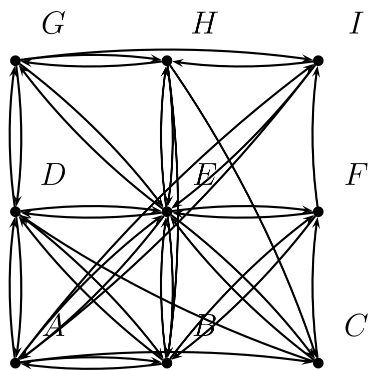
Her prøver jeg for hver sammenligning å først finne færrest mulig ledninger i bruk og deretter se etter:

- minst mulig ubrukt kapasitet.
- færrest mulig forbindelser.
- minst mulig avstand.

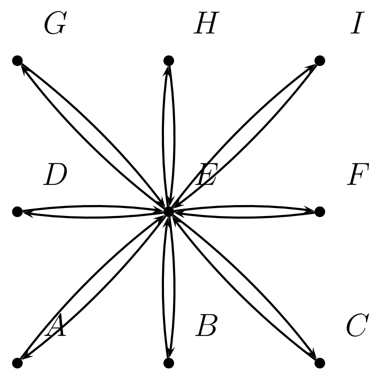
For denne kjøringen ble det løst i følgende rekkefølge C, I, A, G, F, H, D, B. Det er totalt 48 ledninger i bruk, 3 ledninger pr. forbindelse. Totalt 11 % ubrukt kapasitet i nettet. Det som er avgjørende for resultatet her er at den ser etter minst mulig forbindelser. Til tross for at denne topologien ser symmetrisk ut betyr det ikke nødvendigvis at denne måten å sammenligne på er den beste.

Figur B.2(c)

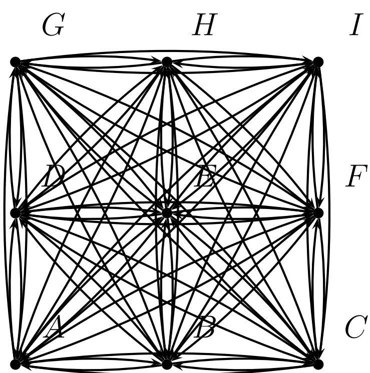
Ser kun etter minste diameter. Alle svitsjene er koblet sammen tydelig at ventetiden er lav. Totalt 72 ledninger og den ubrukt kapasitet er 66 %.



(a)



(b)



(c)

Figur B.2: *Eksempler på resultater på kjøring med grådig metode*

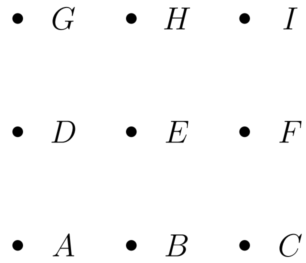
Tillegg C

Kombinasjoner

Viser kombinasjoner av nettverket fra svitsj A til resten av nettverket. Vi har følgende lignende speilinger (f.eks. AH betyr fra A til H):

Fra A til B: AD, BE, BA, BC, CB, CF, DA, DE, DG, EB, ED, EF, EH, FA, FC, FI, GD, GH, HE, HG, HI, IH, IF (24 STK)
Fra A til C: AG, BH, CI, CA, DF, FD, GA, GI, HB, IC, IG (12 STK)
Fra A til E: BD, BF, CE, DB, DH, EA, EC, EG, EI, FB, FH, GE, HD, HF, IE (16 stk)
Fra A til F: AH, BG, BI, CH, CD, DC, DI, FA, FG, GB, GF, HA, HC, IB, ID (16 stk)
Fra A til I: CG, GC, IA

```
Ch: From: A Transmit: 100 targets: B
A-B
Ch: From: A Transmit: 100 targets: C
A-B,B-C
A-C
Ch: From: A Transmit: 100 targets: D
A-D
Ch: From: A Transmit: 100 targets: E
A-B,B-E
A-D,D-E
A-E
Ch: From: A Transmit: 100 targets: F
A-B,B-C,C-F
A-B,B-E,E-F
A-B,B-F
A-C,C-F
```



Figur C.1:

```
A-D,D-E,E-F
A-D,D-F
A-E,E-F
A-F
Ch: From: A Transmit: 100 targets: G
A-D,D-G
A-G
Ch: From: A Transmit: 100 targets: H
A-B,B-E,E-H
A-B,B-H
A-D,D-E,E-H
A-D,D-G,G-H
A-D,D-H
A-E,E-H
A-G,G-H
A-H
Ch: From: A Transmit: 100 targets: I
A-B,B-C,C-F,F-I
A-B,B-C,C-I
A-B,B-E,E-F,F-I
A-B,B-E,E-H,H-I
A-B,B-E,E-I
```

```
A-B,B-F,F-I
A-B,B-H,H-I
A-B,B-I
A-C,C-F,F-I
A-C,C-I
A-D,D-E,E-F,F-I
A-D,D-E,E-H,H-I
A-D,D-E,E-I
A-D,D-F,F-I
A-D,D-G,G-H,H-I
A-D,D-G,G-I
A-D,D-H,H-I
A-D,D-I
A-E,E-F,F-I
A-E,E-H,H-I
A-E,E-I
A-F,F-I
A-G,G-H,H-I
A-G,G-I
A-H,H-I
A-I
.
.
.

Total number of combinations: 2.26794859793077e+31
```

Tillegg D

Implementasjon

D.1 Perl

`Perl` er et script språk som har fint egnet seg som test programmeringsverktøy. Fordelen med `Perl` er at den har en enklere måte å utføre enkelte oppgaver på, som f.eks. lesing og skriving til fil, sammenligne data, og å bygge datastrukturer. På den måten har jeg funnet `Perl` som et fint verktøy for å teste ut ideer av algoritmer.

`Perl` er et script språk. Det legges vekt på at programmet forstatt skal kunne kjøre hvis for eksempel variabler ikke har tilegnede verdier. Andre programmeringspråk ville ha stoppet med en feilmelding ved en slik kjøring. Med `Perl` kan resultat bli at programmet kjører selv om det er feil i koden. Den vanlige regelen ved valg av `Perl` som plattform for utvikling er når en lager enkle programmer og programmet skal forkastet etter at det har spilt sin rolle (denne regelen har en mange unntak). Det er ikke uvanlig at størrelsen på et program vokser betraktelig og et annet programmeringspråk bør velges dersom en implementering skal gjennomføres. Det er bedre at et program stopper kjøringen ved småfeil enn at det fortsetter.

Koden har dokumentasjon på engelsk[1]. Har prøvd å skrive det meste av koden slik at det er mulig for ikke-`Perl` programmerere å kunne forstå hva som skjer. Litt varierende kodekvalitet Håper at kommentarene også vil være til hjelp.

[1] For litt trening i dette språket

D.2 Kommentarer til kode

Merk at pakken for kommunikasjonsbehovene heter `IcNet::LiP::Channel` og kalles for "channel" i programkoden.

D.2.1 Graf

Benytter Graph::Base lastet ned fra CPAN (`http://www.cpan.org/`) og er lisensiert under GNU public license[2].

For å representere veier benyttes graf. Merk at utskrift av grafen, og utskrifter av testkjøring (ved reproduksjon) ikke holder oversikt over rekkefølgen som skrives ut. Ved utskriften vises kantene av grafen. En vei som jeg til vanlig representerer som "A-B-C", (dvs. Start i A gjennom B så til C som er sluttsvitsj), vil dette skrives ut som for eksempel "B-C,A-B", merk at rekkefølgen av utskriften er tilfeldig.

Aksesstid til grafen mener jeg er den største grunnen til treghet i programvaren. Ved generering av "alle kombinasjoner" vil den kun sammenligne et sted mellom 300 og 400 kombinasjoner av gangen. Dette kan forbedres ved å lage egen datastruktur.

[2]Se http://www.gnu.org

Tillegg E

Program

Innhold

1 Program Documentation **3**
1.1 Introduction 3
1.2 Runnable Code 5
1.3 Common Code Base 8
1.3.1 IcNet::LiP::Channel; 8
1.3.2 IcNet::LiP::Path 9
1.3.3 SearchAofA.pm 10
1.3.4 IcNet::FastSearchAofA.pm 16
1.3.5 IcNet::LiP::CommunicationsGraph 16
1.3.6 IcNet::LiP::AreaSelctMethods.pm 17
1.3.7 IcNet::LiP::RestrictArea 20
1.3.8 IsNet::LiP::GraphOperations 20
1.3.9 IcNet::Geometry.pm 22
1.3.10 IcNet::LiP::MapDemands 24
1.3.11 IcNet::LiP::NetAnalyze 25
1.4 All Combinations 25
1.4.1 IcNet::LiP::AllCombinations::AllCombinations 25
1.4.2 AllCombinations::WireOptimize 25
1.4.3 IcNet::LiP::FindPathModule 26
1.5 Greedy 28
1.5.1 IcNet::LiP::Greedy::ExploreNet.pm 28
1.5.2 IcNet::LiP::Greedy::Optimize.pm 29
1.5.3 IcNet::LiP::AllCombinations::OptHelper; 29
1.5.4 IcNet::LiP::Greedy::FMWireUsageDistance 29
1.5.5 IcNet::LiP::Greedy::THDiame 29
1.6 Switching 30
1.6.1 IcNet::LiP::MBG_matching 30
1.7 For visualization 30
1.7.1 IcNet::LiP::SVGImage.pm 30
1.7.2 IcNet::LiP::PSTricksPrintout 31

2 Code **32**

2.0.3 IcNet::LiP::Channel 32
2.0.4 IcNet::LiP::CommunicationsGraph 35
2.0.5 IcNet::LiP::AreaSelectMethods 37
2.0.6 IcNet::LiP::Path 40
2.0.7 IcNet::LiP::Restrictarea 43
2.0.8 IcNet::LiP::FindPathModule 44
2.0.9 IcNet::LiP::Greedy::ExploreNetPM 47
2.0.10 IcNet::LiP::Greedy::OptHelper 56
2.0.11 IcNet::LiP::Greedy::THDiam 57
2.0.12 IcNet::LiP::Greedy::FSWireUnusedDist 59
2.0.13 IcNet::LiP::Greedy::OptimizePM 62
2.0.14 IcNet::LiP::MBG_matching 63
2.0.15 IcNet::Geometry 66
2.0.16 IcNet::SearchAofA 70
2.0.17 TestTimeSlots.pm 74
2.0.18 TestObjects::ExploreNetTestGraphs 75

Kapittel 1

Program Documentation

1.1 Introduction

DESCRIPTION

This is demonstration files for configuring *System On Chip Networks*. The programs is written in PERL.

Code style

PERL provides several different "smart ways" in the code writing so just a few lines of code would be needed to do certain tasks. The result is that most PERL code may have a quite elaborate style compared to other languages (eg. Java or Pascal). For the matter of clarity I've tried to write most of the code in a style that makes it easier for non-Perl programmers to read the code. In addition I've also added pretty much comments throughout the source code.

Notes on debugging

All code needs to be tested in one way or another. I've used "inlinedebugging, this means that I've put in printstatements in the code to print intermediate variables to screen. Some of the code still available in the source, but commented out. All of them should have a comment "# DBGbehind each line.

Package hierarchy

Packages in `IcNet::*` are packages that (hopefully) is general for most SoC networks. Packages `IcNet::LiP::*` are packages that are used for a specialized SoC network.

Documentation

All (or most) documentation is written inn POD. POD is short for "Plain Old Ducomentation" format used by PERL. The advantage of this is that the documentation itself may easily be adopted to various formats such as as HTML, LaTeX or man pages.

Package List

IcNet::Geometry.pm

IcNet::IcNet.pm

IcNet::Link.pm

IcNet::Port.pm

IcNet::Resource.pm

IcNet::SlackReader.pm

IcNet::LiP::Channel.pm

IcNet::LiP::Path.pm

IcNet::LiP::SimpleInitSystem.pm

IcNet::LiP::WriteRepository.pm

IcNet::LiP::MBG_matching.pm

IcNet::LiP::ReadRepository.pm

IcNet::LiP::Switch.pm

IcNet::LiP::MBG_switch.pm

IcNet::LiP::Repository.pm

IcNet::LiP::SwitchSolving.pm

Other packages

Graph.pm (Available from CPAN archive)

1.2 Runnable Code

```perl
use strict;
use TestObjects::ExploreNetTestGraphs;

use IcNet::LiP::AreaSelectMethods;
#use IcNet::LiP::Network;
use IcNet::LiP::NetAnalyze;
use IcNet::LiP::SVGImage;
use IcNet::LiP::AllCombinations::AllCombinations;
use IcNet::LiP::AllCombinations::AllPaths_AofA;
use IcNet::LiP::AllCombinations::Optimize;
use IcNet::LiP::AllCombinations::WireOptimize;

my $net;
my @channels;


## Sets the maximum number of wires for one link
## and the clock frequency.
my $LinkWidth = $ARGV[0]?$ARGV[0]:1; # Width of link
my $LinkFrequency = $ARGV[1]?$ARGV[1]:100; # Clock frequency


##########################################
#
## The TestObjects::ExploreNetTestGraphs package
## holds test obects.
## Add and remove for testing
#($net, @channels) = PMgraph1; # T-shape, point-to-multipoint
#($net, @channels) = T_shaped_pp; # T-shape
#($net, @channels) = PMgraph3;
#($net, @channels) = PMgraph4_square;
($net, @channels) = grid_3x3; # Example thesis
#($net, @channels) = (exampleNet1, channelsExampleNet1);
#($net, @channels) = exampleNet2;
#($net, @channels) = exampleNet3;
#($net, @channels) = exampleThesis1;


################################################
#
## Experiment:
## Default method of findingpaths is using a
## dimond shaped area.
## This may be changed.
## IcNet::LiP::AreaSelectNethods has different area
## shapes to select.
## Use Icnet::LiP::AllCombinations::changeSelectMethod
## to change the way areas should be selected.

changeSelectMethod( \&IcNet::LiP::AreaSelectMethods::square );
#changeSelectMethod( \&IcNet::LiP::AreaSelectMethods::bigSquare );
#changeSelectMethod( \&IcNet::LiP::AreaSelectMethods::selfDiamond );
#changeSelectMethod( \&IcNet::LiP::AreaSelectMethods::rectangle );

#SetASM_fixed(0,0,600,600);
#changeSelectMethod( \&IcNet::LiP::AreaSelectMethods::fixedSquare );

## Experiment:
## Set the offset for the area select method.
SetASM_offset(10);


## ##########################################
##
## ##########################################

## Step 1
## Create all combinations for all channels
       my $all = AllCombinations( $net, \@channels );

## Debug:
## This print combinations to screen.
#      my $optimize = IcNet::LiP::AllCombinations::AllPaths_AofA->start(1);
#      exit(0);

## Step 2:
## Search for "best" network.
## One may select optimization method.
#      my $optimize = IcNet::LiP::AllCombinations::Optimize->start(1);
       my $optimize = IcNet::LiP::AllCombinations::WireOptimize->start(1);
```

```
        $optimize->AofA( $all );
        print "Total number of combinations: ", $optimize->totalCmb,"\n";
        $optimize->doSearch( $net, $LinkWidth,  $LinkFrequency );


## Step 3:
## Make results; make new channels
## each solution is a set of channels
## the paths for each channel represents the network
my @solutions = ();

# for each solution ...
foreach my $r ( $optimize->result ) {
        my @indieSolution = ();

        # for each channel
        foreach my $res ( @$r ) {
                # $$res[0] : IcNet::LiP::Channel
                # $$res[1] : the path. IcNet::LiP::Path
                my $ch = $$res[0]->subClone();
                $ch->Path( $$res[1] );
                push @indieSolution, $ch;
        }
        push @solutions, [@indieSolution];
}

## Step 4:
## printout
foreach my $indieSolution ( @solutions ) {
        foreach my $ch ( @$indieSolution ) {
                $ch->printAll;
                print "Path: ",$ch->Path(), "\n";
        }
        netAnalyze( $LinkWidth, $LinkFrequency, @$indieSolution);
}


use strict;
use Graph;
use IcNet::LiP::Greedy::ExploreNetPM;
use TestObjects::ExploreNetTestGraphs;
use IcNet::LiP::NetAnalyze;
use IcNet::LiP::SVGImage;
use IcNet::LiP::PSTricksPrintout;

my $net;
my @channels;

        ## $net : Network, graph with position attribute
        ##        for each svitch
        ## @channels : traffic demands, as IcNet::LiP::Channel

## Add and remove for testing
#($net, @channels) = PMgraph1; # T- pm
#($net, @channels) = T_shaped_pp(); # T- pp
#($net, @channels) = PMgraph3;
#($net, @channels) = PMgraph4_square;
($net, @channels) = grid_3x3;
#($net, @channels) = PMgraph_grid(2); # out of order

## Maps traffic demands and switch positions into one graph.
#addDemands($net,@channels);


## Set global variables for ExploreNetPM
my $linkWidth = $ARGV[0]?$ARGV[0]:8;
my $linkFrequency = $ARGV[1]?$ARGV[1]:100;

## Find a solution
my $solution = ExploreNet( $net,  $linkWidth, $linkFrequency, @channels );
print "\nThe solution looks like:\n";
print $solution,"\n\n";

## Prints solution
foreach my $ch ( @channels ) {
        $ch->printAll;
        print "Path: ",$ch->Path(), "\n";
}

        netAnalyze( $linkWidth, $linkFrequency, @channels);



## use to draw an image of the network
```

```
#       SVGImage( $solution, "network.png");

## Draws the network
## prints an PS tricks (LaTeX) picture

#       PSprint ($solution,100,"psprint.pst.tex"); # prints to file psprint.pst.tex
#       PSprint ($solution,10); # prints to stdout

print "\n";
```

1.3 Common Code Base

1.3.1 IcNet::LiP::Channel;

SYNOPSIS

DESCRIPTION

This package is obsolete and documentation is not up-to-date. It has been replaced by the Path.pm package.

```
$self = { From      => "",
          target    => [],
          Transmit  => 0,
          Path      => IcNet::LiP::Path->new(),
          ID        => "",
          @_
        };
```

Methods

new

Constructor.

From

An resource that the channel starts from.

target

An resource that the channel ends to.

Transmit

Path

Set or read the *Path* for this channel.

Examples:

```
$ph = Path(); # Return path

@arr = [["a->b"}, ["b->c","a->b"] ];
$ch->Path( @arr ); # Set the path,
                   # replaces the old one.
```

addPath

Adds (or in the right Perl therm: push) a new set of links at the end of the Path. In other words you are expanding the path.

Examples: @list = (Link1",Link2"); $ch->addPath(@list);

getPath

Return the *Path* or the array represented at a given time. You may also give it a number representing the clock cycle and the links in use at that clock cycle is returned.

Examples:

```
# Gets the links in use at
      # clock cycle 2.
# Returns empty list if no
# links is in use at that clock cycle.

@links = getPath( 2 );

# gets the whole path
@path = getPath();
```

StartTime

Set or read the StartTime.

EndTime

Read the EndTime. EndTime is the StartTime + the length of the Path. Thus an EndTime cannot be set.

ID

An ID for a channel will be automatically set by joining the ID from the *From port* and *To port*, separated with a '_' to be sure to make the ID unique.

printAll

Prints the Path. Handy for debugging.

Length

Return the length of the Path

incStartTime

Increases the start time by one.

1.3.2 IcNet::LiP::Path

SYNOPSIS

```
use IcNet::LiP::Path;
```

DESCRIPTION

To read and modify a path through the network.

```
$self = { Path => $p,
          ID => "",
          Resource => "",
          @_
        };
```

Methods

new

Constructor.

Root

The start Resource.

Leaf

The resources that the paths ends to.

Path

addPath

ID

An ID for a channel will be automatically set by joining the ID from the *From port* and *To port*, separated with a '_' to be sure to make the ID unique.

printAll

Prints the Path. Handy for debugging.

1.3.3 SearchAofA.pm

SYNOPSYS

```
use SearchAofA;
@AofA = (['A','B'], ['C','D','E'],
         ['F','G'], ['H','I','J']);

# Calls constructor, and search through the AofA
$offset = 0; # (Control for extra local parameters)
$combinations = SearchAofA->start($offset,  \@AofA );
```

```
# Show result
for ( $combinations->result  ) {
  print  @$_ ,"\n";
}

# Shows:
#  ACFH
#  ACFI
#  ACFJ
#   ...etc...
#  BEGI
#  BEGJ
```

DESCRIPTION

This module is intended to compares all combinations between several arrays, bundled in an array-of-array data structure. It is a very handy module when you want to have a quick way of comparing several lists of objects together.

The comparing is done with the `compare` method and might be rewritten to tailor your need. Rewriting of the `compare` method is simply done by exporting this package (class). See the example later to see how this might be done. Reuse is the main idea behind this package.

For the sake of clarity in this text, the array holding the arrays is the **Base Array**, the arrays containing data will simply be called **Data Arrays**.

However this module does not permutes (reorder) the different data arrays.

Local and global parameters To get more flexibility during comparing it is possible to add local and global parameters.

The *local parameters* is parameters that belong to the individual Data Array. Local parameters must be set in the very beginning of the data array and should have the same number of parameters for all data arrays. The module uses an *offset* parameter to tell where objects that should be compared starts, all items below this offset number are local parameters. The example in the SYNOPSYS section shows an offset equal to 0, hence no local parameters. If the *$offset* value is set to 1, it will in this example just ignore the first character in the data arrays and print `BDGI BDGJ BEGI BEGJ`.

Global Parameters are all parameters that follows the array-of-array reference of the `start` method. All global parameters are sent to the `compare` method.

Methods

start($offset, $ref_AofA, @globalParams)

CONSTRUCTOR

It will start the search through the array of array. *@globalParams* is other parameters that may be used with the `compare` method. Somtimes you does not want the comparing to start immediatly, then *$ref_AofA* and i<$globalParams> should not be added as input parameters.

```
# Start comparing immediatly
$comb = SearchAofA->start(0, \@arrayArray);

# Do not start comparing immediatly
$comb = SearchAofA->start(0);
$comb->AofA( $ref_arrayArray); # adds array
print $comb->totalCmb(),"\n"; # prints number of combinations
$comb->doSearch( @globalParams ); # start searching
```

doSearch(@globalParams)

Start exploring each combinations of the array-of-array datastructure *AofA*, for each new combination it calls the `compare` method. See the `compare` method regarding how it stores combinations.

compare(@globalParams)

Looks at the current combination of elements of the array-of-array datastrucure. To store a combination it must return 0. If it should delete all previous combinations (eg. the combination is betterthat the previous) and store a new combinations it should return a positive value. If the current combination is worse that the existing one it should return a negative value.

It's default function is that it returns 0, as a result all combinations will be stored.

As mentioned, this class (module) should be exported and this method rewritten to tailor your needs.

combination

Returns the current combination or the one specified with the input parameter. If you have local parameters use *localCombination* instead.

```
@return = $o->combination();

@ret = $o->combination(1,2,3,4);
```

localCombination

Returns a list with local parameters and its current combination. The returned list is an array of array.

```
# This example has an 'offset' to 1
$c = SearchAofA->new(1, $ref_AofA );
@return = $c->localCombination;

foreach my $local ( @return ) {
  # print local:
  print $$local[0];
  # print item
  print $$local[1];
}
```

getValueResult

Returns values of some calculations of the combinations. To store a value the `compare` method should return with an second parameter like this:

```
return (0, $value);
```

(TODO. write an sxample of how this is done..)

result

Return all "approved"combinations from array-of-array datastructure.

For (un)practical resons, if you use local parameters the result method should be rewritten into something like the code below. This returns a list that have both the local parameters and the result. The returned list returns a list with arrayreferences and looks like:

```
([$local1, $result1],[$local2,$result2]...)
```

Example:

```
To be done......
```

The rewritten code:

```
sub result {
  my $self = shift;
```

```
    return map {
      my $return = [];
      my @combination = $self->combination( @$_ );

      for my $i ( 0 .. $#$_ ) {
        my $localP = [];
        for my $L ( 0 .. $self->{offset}-1 ) {
          push @$localP, $self->localParameter($i,$L);
        }
        push @$localP, $combination[ $i ];
        push @$return, $localP;
      }

      $return;

    } @{ $self->{cmpResult} };

  }
```

totalCmb

Return the total number of combinations.

Data fiddling methods

AofA

Internally used methods

addCmpResult

Stores a combination.

newCmpResult

Deletes all previous combinations and store a new one.

nextCombination

Increase the $self->{indexArr} for the next combination.

(TODO: explain the use of $self->{indexArr}, this helps to understand the code as well....)

offset

The offset value used for local parameters.

Example

In this example we make all combinations of boys and girls that will dance togheter. Some aliens from outher space are not allowed on the dance floor, these are Znorrrky and ET. (how discriminating).

```
package Dancers;

use IcNet::SearchAofA;

@ISA = ("IcNet::SearchAofA");

sub compare {

  $self = shift; # Always

  # Get global parameters
  $forbidlist = shift; # String

  # (Cleaner code with this line)
  @currentCombination = $self->combination();

  # Do test to forbid-list
  foreach $person ( @$forbidlist ) {
                      # if true, result is not stored
    return -1 if  grep /$person/, @currentCombination;
  }

  return 0; # Return and store result
}

1;

#use Dancers;

$boys = ['Donald','George','ET'];
$girls = ['Nancy','Fiona','Znorrrky'];

# People not allowed on dance floor
$forbidDancersList = ['Znorrrky','ET'];

$dancefloor = Dancers->start(0,[$boys, $girls], $forbidDancersList );
foreach my $comb ( $dancefloor->result ) {
  print @$comb,"\n";
}
```

1.3.4 IcNet::FastSearchAofA.pm

SYNOPSYS

```
use FastSearchAofA;

# an "array-of-array"
@AofA = (['A','B'], ['C','D','E'],
         ['F','G'], ['H','I','J']);

# Calls constructor, and search through the AofA
$offset = 0; # (Control for extra local parameters)
$combinations = FastSearchAofA->start($offset,  \@AofA );

# Show result
for ( $combinations->result  ) {
  print  @$_ ,"\n";
}

# Shows:
#  ACFH
#  ACFI
#  ACFJ
#   ...etc...
#  BEGI
#  BEGJ
```

DESCRIPTION

Does the same as *IcNet::SearchAofA*, the difference is that you must keep control over each new item added or subracted from your "array-of-array". It tries to avoid that values that already have a calculated value is recalculated later.

To control this you must overwrite the methods *add()* and *subtract()*.

TODO

Actually a re-do of this module might makes it clearer to understand. Part of the code is somewhat ad-hoc.

1.3.5 IcNet::LiP::CommunicationsGraph

SYNOPSIS

```
use IcNet::LiP::CommunicationGraph;
$Network = new IcNet::LiP::CommunicationGraph
```

DESCRIPTION

A *communicationg graph* has the following link attributes:

demand

This is the communicationg needed between the start end end switches. The demand itself is typically in this case a *IcNet::Channel* object.

Note: This package is extended fram class Grap::Directed.

Subrutines

new

clone

Clones a communicationg graph. Clone may have bug.

addDemandsToDestinations($startSwitch, $demand, @endSwitches)

addDemands($startSwitch, $endSwitch, $demand)

setDemands($startSwitch, $endSwitch, @demands)

getDemands($startSwitch, $endSwitch)

getIncomingDemands($switch)

getOutgoingDemands($switch)

getOutgoingDemandsPM($switch)

Demands that are pointing outwards may be a point-to-multiopoint demands. In this case the *demand* is located in more that one edge and needs to be collected.

```
The returned list is from calling this method is:
( [$demand1, $endVertex1, $endVertex2..],
  [$demand2, $endVertex3, $endVertex1], ....etc )
```

removeDemands($startSwitch, $endSwitchm @demands)

1.3.6 IcNet::LiP::AreaSelctMethods.pm

SYNOPSIS

```
use IcNet::LiP::AreaSelectMethods;
```

DESCRIPTION

Subrutines

SetASM_offset($value)

Changes the *$offset* value.

Areaselect subrutines All other subrutines will be called with as (@startPoint, @endPoint). All methods return an polygon that represent the selected area.

eg.

```
@polygonArea = square( $startX, $startY, $endX, $endY);
```

square

Selects a square. The *$offset* parameter may be used to shrink or increase the area.

Warning: If *$offset* value is 0, the code may not include the endpoint or the startpoint. It is recommended to set the *$offset* to 1 to aviod that the start- and endpoints points lies at the very edge of the square..

```
 ----e
|    |
|    |
s-----
```

diamond

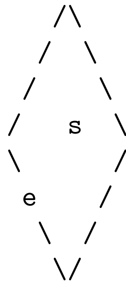
Selects a diamond shaped area around the end point and will include a area behind the endpoint. Endpoint is in the center of the area. The *$offset* value may be used to shrink or increase the area.

```
       /\
      /  \
     /    \
    /  e   \
    \      /
     s    /
      \  /
       \/
```

selfDiamond

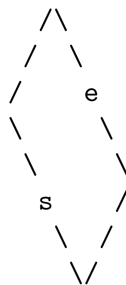
The opposite of *diamond* sub.

Selects a diamond shaped area around the start point and will include a area behind the startpoint. Startpoint is in the center of the area. The *$offset* value may be used to shrink or increase the area.



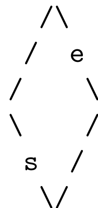
rectangle

Selects a rectangle shaped area between startpoint and endpoint.



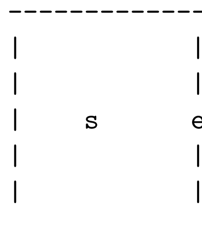
rectangeleHalf

Similar to *rectangle* but half the width of *rectangle*.



bigSquare

Makes a square around the first point. The sides are 2 times the manahttan distance between the start and end point.



fixedSquare

Always select a square of a predefined size. Set the predifend corners with the *SetASM_fixed($x0,$y0,$x1,$y1)* subroutine.

See also

IcNet::LiP::RestrictArea

1.3.7 IcNet::LiP::RestrictArea

SYNOPSIS

DESCRIPTION

Subrutines

NOTE the network graph *$N* har 'position' attributes associated with it's vertices. See `$IcNet::LiP::GraphOperations(vertexPosition)` about how to retrive and set position attributes..

def_findArea($startX, $startY, $endX, $endY)

The default subroutine to search for subgraphs.

findSubGraph($N, $startvertex, $endvertex, $subref_findArea)

Finds a subgraph betweem the *$startvertex* and *$endvertex* from the network graph *$N*.

The *$findArea_subrutine* is a *reference* to a subrutine. This subrutine will restrict the area that the search for intermediate point will emerge. A default subrutine is provided in this package if this parameter is set to 0 (zero) or undef. The default subrutine will select points around a diamond shaped rectangle around the `endVertex` (or end point if u'd like),(and this will represent a *subgraph* of the NetworkGraph.)

verticesInPolygon($N, $ref_polygon)

$N network graph, each vertex holds an 'position' attribute. The *$ref_polygon* is an reference to an array representing (x,y) coordinates of the polygon. Returns an subgraph with all vertices covered by the polygon.

```
my $subG = vetricesInPolygon( $N, [0,0,50,50,100,100,0,0] );
```

1.3.8 IsNet::LiP::GraphOperations

SYNOPSYS

```
use IcNet::LiP::GraphOperations;
```

DESCRIPTION

Various graph operations.

Subroutines

copyVertex($ToGraph, $FromGraph, $vertex)

Copies all the vertex ($vertex) with all attributes and values associated with this vertex to graph $ToGraph from graph $FromGraph. Makes a new clone of the graph.

vertexDistance($graph, $vertex_A, $vertex_B)

Returns the manhatten distance between $vertex_A and $vertex_B.

vertexPosition($graph, $vertex)

vertexPosition($graph, $vertex, $Xval, $Yval)

Return or sets the X ($Xvar) and Y ($Yval) values for the `position` attribute for the vertex *$vertex* in graph *$graph*.

```
vertexPosition($G, $v, 1050, 300);
($X,$y) = vertexPosition($G, $v);
```

completeNetwork($graph)

Returns a complete graph (all vertices connected to all other vertices by one edge).

cloneGraph

Returns an exact clone of the graph. All attributes and it's associated values for both graph, edges, and vertices are copied to the new graph.

```
$newNetworkGraph = cloneGraph( $G );
```

mergeGraph

vertexDemand

_vertexAttribute

vertexSend

vertexReceive

_edgeAttribute

edgeTraffic

getAllPositions

scanOrder

scanDownToUp

scanUpToDown

scanDownLeftToUpperRight

Makes a scan over a graph with vertices that has a position attribute. Returns a list of vertices in a scanned order. eg. from down to up (lowest y-position to highest y-position).

```
# calls scanDownToUp, scans from down(lowest y-pos) to
# highest y-pos)
@order = scanOrder($graph, "downToUp");

# calls scanUpToDown
@order = scanOrder($graph, "upToDown")

# diagonal scan
@order = scanOrder($graph, "downLeftToUpperRight");
```

allEdgesDistance

switchLenghts

Calculates a Hash_of_Hash element that holds the lenghts between all pairs of switches in a Communicationgraph (network)

```
# calculate lenghts
%lenghts = switchLenghts( $Com_Graph );

# get lengths
$lenght = $lenghts{$switch_A, $switch_B);
```

1.3.9 IcNet::Geometry.pm

DESCRIPTION

Geometric algorithms from chapter 10 of "Mastering Algorithms with PERL" edited by Jon Orwant. **License** GNU public License.

Subroutines

rectilinear_area($x, $y, $distance)

Returns a polygon representing the area around the point $x and $y. A rectilinear area is a diamond shaped.

manhattan_intersection(@lines)

Find the intersections of strictly horizontal and vertical lines. Requires basic_tree_add(), basic_tree_del(), and basic_tree_find(), all defined in Chapter 3, Advanced Data Structures.

range_check_tree($tree_link, $horizontal, $compare)

Returns the list of tree nodes that are within the limits $horizontal->[0] and $horizontal->[1]. Depends on the binary trees of Chapter 3, Advanced Data Structures.

basic_tree_find

Usage: ($link, $node) = basic_tree_find(\$tree, $target, $cmp)

Search the tree \$tree for $target. The optional $cmp argument specifies an alternative comparison routine (called as $cmp->($item1, $item2) to be used instead of the default numeric comparison. It should return a value consistent with the <=> or cmp operators.

Return two items:

a reference to the link that points to the node (if it was found) or to the place where it should go (if it was not found)

the node itself (or undef if it does not exist)

$node = basic_tree_add(\$tree, $target, $cmp);

If there is not already a node in the tree \$tree that has the value $target, create one. Return the new or previously existing node. The third argument is an optional comparison routine and is simply passed on to basic_tree_find.

$val = basic_tree_del(\$tree, $target[, $cmp]);

Find the element of \$tree that has the value $val and remove it from the tree. Return the value, or return undef if there was no appropriate element on the tree.

MERGE_SOMEHOW

Make $tree_link point to both $found->{left} and $found->{right}.

Attach $found->{left} to the leftmost child of $found->{right} and then attach $found->{right} to $$tree_link.

traverse($tree, $func)

Traverse $tree in order, calling $func() for each element. in turn

manhattan_distance(@p)

Computes the Manhattan distance between two points in the plane.

convex_hull_graham(@xy)

Compute the convex hull of the points @xy using the Graham's scan. Returns the convex hull points as a list of ($x,$y,...).

distance(@p)

distance(@p) computes the Euclidean distance between two d-dimensional points, given 2 * d coordinates. For example, a pair of 3-D points should be provided as ($x0, $y0, $z0, $x1, $y1, $z1).

clockwise($x0, $y0, $x1, $y1, $x2, $y2)

Return positive if one must turn clockwise (right) when moving from p0 (x0, y0) to p1 to p2, negative if counterclockwise (left). It returns zero if the three points lie on the same line – but beware of floating point errors.

point_in_polygon ($x, $y, @xy)

Point ($x,$y), polygon ($x0, $y0, $x1, $y1, ...) in @xy. Returns 1 for strictly interior points, 0 for strictly exterior points. For the boundary points the situation is more complex and beyond the scope of this book. The boundary points are exact, however: if a plane is divided into several polygons, any given point belongs to exactly one polygon.

Derived from the comp.graphics.algorithms FAQ, courtesy of Wm. Randolph Franklin.

1.3.10 IcNet::LiP::MapDemands

SYNOPSIS

DESCRIPTION

Subrutines

addDemands($CGraph, @demands)

Adds the demands (@demands) to the communication graph $CGraph.

1.3.11 IcNet::LiP::NetAnalyze

SYNOPSIS

DESCRIPTION

Analyse network for derived channels and its paths.

Subrutines

1.4 All Combinations

1.4.1 IcNet::LiP::AllCombinations::AllCombinations

SYNOPSIS

```
use IcNet::LiP::AllCombinations::AllCombinations
```

DESCRIPTION

Generates all combinations of paths between a startswitsj and end switch.

Subrutines

AllCombinations($ComGraph, $@demands)

Returns all paths given between switches in the Communication Graph, and by the given demands.

```
@allPaths = AllCombinations( $C, \@channels);
```

changeSelectMethod($ref_to_subrutine)

To select the method used to find an area. The subrutine is used to find the area to select out betwwen start- and end switch. See *IcNet::LiP::AreaSelectMethods;*.

```
# Change the search to select squares.
changeSelectMethod( \&IcNet::LiP::AreaSelectMethods::square )
```

1.4.2 AllCombinations::WireOptimize

SYNOPSIS

```
use IcNet::LiP::AllCombinations::WireOptimize;
```

DESCRIPTION

Overriden Methods

compare($Network, $maxWidth, $maxFrequency)

result

Prints the combination.

1.4.3 IcNet::LiP::FindPathModule

SYNOPSIS

```
use IcNet::LiP::FindPathModule;

$allPaths = FindAllPath( $NetworkGraph,
                         $startVertex, $endVertex,
                         $findArea_subroutine,
                         $acceptPoint_subroutine);
```

DESCRIPTION

Finds paths between two points in the plane by starting at one point (the startpoint) and finding the end point by selecting intermediate points by some criteria (eg. always pick the next point that get me closer to to the end point).

The points and the resulting *path* in the plane is represented as a *Graph.* See IcNet::GraphOperations module for a description of parameters of the graph. Basically each vertex of the graph has a 'point' parameter, this means that the word end vertex"and end point"may have similar meaning.

Subroutines

FindAllPaths(...)

```
FindAllPaths($NetworkGraph,
             $startVertex, $endVertex,
             $findArea_subroutine,
             $acceptPoint_subroutine)
```

Finds all paths between the $startVertex and the $endVertex. However this one is deprecated, and not recommended for use. Use the *AreaSearch* method instead.

AreaSearch($N, $startvertex, $endvertex, $subref_findarea)

Search for all paths between $startvertex, $endvertex. The *$subref_findarea* is an optinally referece to a subrutine that take the position to $startvertex and $endvertex as arguments and returns the polygon of the area where a path should be found.

Internal subroutines

EdgeSeek($subGraph, $startVertex, $endVertex)

Starts the exploring for paths.

_initFindPath($subGraph)

Sets attribute 'visited' to 0 (zero) for all vertices.

I_MinSpan_EdgeSeek(..)

```
I_MinSpan_EdgeSeek($subGraph,
                   $currentVertex,
                   $endVertex,
                   $ref_to_path,
                   $ref_to_StorePaths )
```

Finds paths after some criteria. New paths found is stored in $ref_to_StorePaths.

recursiveAreaSearch

Anonymeous subroutines

findArea_subroutine

See IcNet::RestricArea documentation page.

acceptPoint_subrutine

Whether a point is accepted in the path or not is determined by a the *$acceptPoint_subroutine*, which is a *reference* to a subrutine. The default is to find points that always are closer to the end point. If you want to use the default subrutine set this parameter to 0 (zero) or undef.

TODO: write about how to call these subrutine parameters..

1.5 Greedy

1.5.1 IcNet::LiP::Greedy::ExploreNet.pm

SYNOPSIS

DESCRIPTION

note I've tried to write the code more dependend on the communication graph, rather that the IcNet::Channel (communication demand) object.

Subrutines

ExploreNet

ExploreVertex

collectOutgoingCandidates

optimizeOutgoingCandidates

outgoingUpdate

collectIncomingCandidates

optimizeIncomingCandidates

incomingUpdate

candidateVertices

optimizeCandidates

squareArea

findOuterVertices($C)

Return a list of switches that have the least probability to become a switch with a path through it.

_vertexPoints($C)

Return a hash that has the switches as keys and points as values. All points are set to 0. The switches returned are the one that are at the outer bound of the area.

1.5.2 IcNet::LiP::Greedy::Optimize.pm

SYNOPSIS

DESCRIPTION

Overriden Methods

compare

Prints the combination.

1.5.3 IcNet::LiP::AllCombinations::OptHelper;

SYNOPSIS

```
use IcNet::LiP::AllCombinations::OptHelper;
```

DESCRIPTION

Helper subrutines for optimising a LiP network.

Subrutines

Efficiency($traffic, $maxWidth, $maxFrequency)

1.5.4 IcNet::LiP::Greedy::FMWireUsageDistance

SYNOPSIS

DESCRIPTION

Tries to optimize with respect to low wires, low unused traffic, and low distance left toward en switch.

Overriden Methods

compare

add

subtract

result

1.5.5 IcNet::LiP::Greedy::THDiame

SYNOPSIS

DESCRIPTION

Tries to optimize give the network by minimizing the diameter.

Overriden Methods

compare

add

subtract

result

1.6 Switching

1.6.1 IcNet::LiP::MBG_matching

SYNOPSIS

```
use IcNet::LiP::MBG_matching;
```

DESCRITION

An maximum bipartite graph (MBG) implementation from the book "Algorithms Handbook".

1.7 For visualization

1.7.1 IcNet::LiP::SVGImage.pm

SYNOPSIS

```
use IcNet::LiP::SVGImage;

# $C is IcNet::LiP::CommunicationGraph
SVGImage( $C );
```

DESCRIPTION

See source for details.

See also IcNet::LiP::GraphOperations.

Subrutines

SVGImage($ComGraph, [$filename])

1.7.2 IcNet::LiP::PSTricksPrintout

SYNOPSIS

```
use IcNet::LiP::PSpicture;
# $C is IcNet::LiP::CommunicationGraph

PSprint( $C, 10, "network.tex" );

# Usage
PSprint( $C, $divide, $fileName );
```

$C a graph. Each vertex has a Position attribute. See also IcNet::LiP::GraphOperations.
$divide for division of position arguments. Will bi set to 1 of omitted.
$fileName to print out to a filename. Will print out to stdout if omitted.

Kapittel 2

Code

2.0.3 IcNet::LiP::Channel

```
package IcNet::LiP::Channel;
2
#
# POD documentation at end of file.
#

7  use strict;
use IcNet::LiP::Path;

sub new {
  my $invoce = shift;
12  my $class = ref($invoce) || $invoce;
  my $self = {};

  $self = { From       => "",
            target     => [],
17          Transmit   => 0,
            Path       => IcNet::LiP::Path->new(),
            ID => "",
            @_
          };
22

  bless ($self, $class);
  return $self;
}
27
##########################################################
#

sub clone {
32  my $self = shift;
  my $clone = (ref($self))->new;
  $clone->{From} = $self->{From};
  $clone->target( $self->target );
  $clone->Transmit( $self->Transmit );
37 # $clone->Path( $self->Path->Path->copy() ); # uncertain about this one
  $clone->ID( $self->ID );

  # other params
  my %other = @_;
42  foreach my $k ( keys %other ) {
    $clone->{$k} = $other{$k};
  }

  return $clone;
47 }

sub subClone {
  my $self = shift;
  my $clone = (ref($self))->new;
52  $clone->{From} = $self->{From};
  $clone->target( $self->target );
  $clone->Transmit( $self->Transmit );
  $clone->{Path} = IcNet::LiP::Path->new();
  $clone->ID( $self->ID );
```

```perl

  # other params
  my %other = @_;
  foreach my $k ( keys %other ) {
    $clone->{$k} = $other{$k};
  }

  return $clone;
}

############################################################
# From

sub From {
  my $self = shift;
  $self->{From} = shift if @_;

  return $self->{From};
}

############################################################
# Target

sub target {
  my $self = shift;

  if ( @_ ) { push @{ $self->{target} }, @_;   };

  return @{ $self->{target} };
}

############################################################
# removeTarget

sub removeTarget {
  my $self = shift;
  my $removeThis = shift;
  my @newTarget = grep { $_ ne $removeThis }
                        @{ $self->{target} };

  $self->{target} = [@newTarget];

  return $self->{target};
}

############################################################
# Transmit

sub Transmit {
  my $self = shift;

  $self->{Transmit} = shift if @_;

  return $self->{Transmit};
}
############################################################
# Path

sub Path {
  my $self = shift;

  $self->{Path}  = IcNet::LiP::Path->new( @_ ) if @_;

  return $self->{Path}->Path;
}

############################################################
# addPath

sub addPath {
  my $self = shift;

  return $self->{Path}->addPath( @_ );
}

############################################################
# endTime

#sub EndTime {
# my $self = shift;
# return $#{$self->{Path}}+$self->{StartTime};
#}

############################################################
```

```
    # ID
142
    sub ID {
      my $self = shift;

      $self->{ID} = shift if @_;
147
      return $self->{ID};
    #  return join('_', $self->{To}->ID, $self->{From}->ID);
    }

152 ############################################################
    # printAll

    sub printAll {
      my $self = shift;
157
      # Now print
      print "Ch:",$self->ID," From: ",$self->{From},
            " Transmit: ",$self->{Transmit},
            " targets: ",$self->target,"\n";
162
    }

    ############################################################
    # Length
167 sub Length {
      my $self = shift;
      return $#{ $self->{Path}} +1;
    }

172 ############################################################
    # incStartTime

    sub incStartTime {  # deprecated
      my $self = shift;
177   $self->{StartTime}++;
    }

    ############################################################
    # linkConflict
182
    sub linkConflict {
      print "IcNet::LiP::Channep.pm: linkConflict deprecated\n";
      exit(0);
    } # END linkConflict
187
    ############################################################
    # getConflicts
    sub getConflicts {
      print "IcNet::LiP::Channep.pm: getConflicts deprecated\n";
192   exit(0);
    } # end sub getConflicts

    ############################################################
    # conflictIndex
197 sub conflictIndex {
      print "IcNet::LiP::Channep.pm: conflictIndex deprecated\n";
      exit(0);
    } # end sub conflictIndex

202 ############################################################
    # conflictFrameIndex
    # Compares conflict relative to starttime and
    # frame.

207 sub conflictFrameIndex {
      print "IcNet::LiP::Channep.pm: conflictFrameIndex deprecated\n";
      exit(0);
    } # end sub conflictFrameIndex

212 ############################################################
    # conflictAt

    sub conflictAt {
      print "IcNet::LiP::Channep.pm: conflictAt deprecated\n";
217   exit(0);
    }

    1;

222 #
    # End of code.
    #
```

```
##########################################################
227 ##########################################################
#
# POD documentation.
#
```

2.0.4 IcNet::LiP::CommunicationsGraph

```
package IcNet::LiP::CommunicationGraph;

##########################################################
  4 #
#
#
# POD documentation at end of file

  9 use strict;
use base qw(Graph::Directed);

sub new {
  my $proto = shift;
 14   my $class = ref($proto) || $proto;
  my $C  = $class->SUPER::new( @_ );
  bless ($C, $class);          # reconsecrate
  return $C;
}
 19
sub clone {
  my $C = shift;

  return $C->new();;
 24
}

##########################################################

 29 sub addDemandsToDestinations {
  my $C    = shift;
  my $from = shift;
  my $d    = pop; # demand
  my @towards = @_;
 34
  foreach my $t ( @towards ) {
    $C->addDemands($from, $t, $d);
  }

 39   return $C;

}

##########################################################
 44
sub addDemands {
  my $C = shift;
  my $from = shift;
  my $to = shift;
 49   # @_ new demands

# my @currentDemand = $C->getDemands($from,$to);
# push @currentDemand, @_;
# $C->setDemands($from, $to, @currentDemand );
 54
  $C->setDemands($from, $to, $C->getDemands($from,$to), @_ );

  return $C;
}
 59
##########################################################

sub setDemands {
  my $C = shift;
 64   my $from = shift;
  my $to = shift;
  my @d = @_; # demand

  unless ( $C->has_edge($from,$to) ) {
 69     $C->add_edge($from,$to);
  }

  $C->delete_attribute('demand',$from,$to);
  return $C->set_attribute('demand',$from,$to,[@d]);
 74
```

```
    }

    ###########################################################

79  sub getDemands {
      my $C = shift;
      my ($a , $b) = @_;

      if ( $C->has_edge($a, $b) && $C->has_attribute('demand',$a,$b) ) {
84      return @{ $C->get_attribute( 'demand', $a,$b) };
      }

      # return empty
      return ();
89  }

    ###########################################################

    sub getIncomingDemands {
94    my $C = shift;
      my $v = shift; #vertex
      my @incomingDemands = ();

      my @incomingEdges = $C->in_edges($v);
99    my ($a,$b);
      while ( ($a,$b) = splice( @incomingEdges, 0, 2) ) {
        push @incomingDemands, $C->getDemands($a,$b);
      }

104   return @incomingDemands;

    }

    ###########################################################
109
    sub getOutgoingDemands {
      my $C = shift;
      my $v = shift; #vertex
      my @outgoingDemands = ();
114
      my @outgoingEdges = $C->out_edges($v);
      my ($a,$b);
      while ( ($a,$b) = splice( @outgoingEdges, 0, 2) ) {
        push @outgoingDemands, $C->getDemands($a,$b);
119   }

      return @outgoingDemands;

    }
124
    ##########################################################

    sub getOutgoingDemandsPM {
      my $C = shift;
129   my $v = shift; # vertex / switch
      my %out = ();

      # walk through outgoing edges.
      my @outgoingEdges = $C->out_edges($v);
134   my ($s,$e);
      while  ( ($s,$e) = splice( @outgoingEdges, 0, 2) ) {

        # Look for similar demands fpr each edge.
        foreach my $demand ( $C->getDemands( $s,$e) ) {
139       if ( exists $out{$demand} ) {
            push @{$out{$demand}}, $e;
          }
          else {
            push @{$out{$demand}}, $demand, $e;
144       }
        }
      }

      # the returned list is
149   # ( [$demand1, $endVertex1, $endVertex2..],
      #   [$demand2, $endVertex3, $endVertex1], .... )

      return values %out;

154 }

    ##########################################################

    sub removeDemands {
```

```
  my $self = shift;
  my $a = shift;
  my $b = shift;
  my @remove = @_;
  my @newDemands = $self->getDemands($a,$b);

  foreach my $r ( @remove ) {
    my @newDemands = grep { $_ ne $r } @newDemands;
  }

# print @demands; # DBG

  $self->setDemands($a,$b,@newDemands);

  return $self;

}

############################################################
#

sub vertexPosition {
  my $self = shift;
  my $v    = shift;

  if ( @_ ) {
    my $x = shift;
    my $y = shift;
    $self->{ Attr }->{ V }->{ $v }->{ position } = [$x,$y];
  }

  my $p = $self->{ Attr }->{ V }->{ $v }->{ position };
  return ($$p[0],$$p[1]);

}

1;

__END__
```

2.0.5 IcNet::LiP::AreaSelectMethods

```
package IcNet::LiP::AreaSelectMethods;

############################################################
# Methods for selecting an area between two points.
#
# POD documentation at end of file

use IcNet::Geometry;
use POSIX;
use strict;

BEGIN {
  require Exporter;

  our @ISA = qw(Exporter);
  our @EXPORT = qw( &square &diamond &rectangle &rectangleHalf &bigSquare &
       fixedSquare &SetASM_offset &SetASM_fixed);
  our %EXPORT_TAGS = ( );
  our @EXPORT_OK = qw( &AreaDebug );
  our $VERSION = 0.1;

}

our @EXPORT_OK;

# non-exported package globals
our $offset = 0;
our ($fixedX0, $fixedY0, $fixedX1, $fixedY1);

sub SetASM_offset {
  $offset = shift;
}

sub SetASM_fixed {
  ($fixedX0, $fixedY0, $fixedX1, $fixedY1) = @_;
}

sub fixedSquare {

  return square( $fixedX0, $fixedY0, $fixedX1, $fixedY1);
```

```
    }
 42
    sub square {
      my ($x1,$y1,$x2,$y2) = @_;
      # center point
      my $Cx = (($x2-$x1)/2)+$x1;
 47   my $Cy = (($y2-$y1)/2)+$y1;

      # add a distance
      my $Dx = abs(($x2-$x1)/2)+$offset;
      my $Dy = abs(($y2-$y1)/2)+$offset;
 52   my @p = (); # polygon

      $p[0] = $Cx+$Dx;
      $p[1] = $Cy-$Dy;
      $p[2] = $Cx+$Dx;
 57   $p[3] = $Cy+$Dy;
      $p[4] = $Cx-$Dx;
      $p[5] = $Cy+$Dy;
      $p[6] = $Cx-$Dx;
      $p[7] = $Cy-$Dy;
 62
      # This is first quadrant coordinates :
      # downleft cordner $p[0],$p[1]
      # downright cordner $p[2],$p[3]
      # upright cordner $p[4],$p[5]
 67   # upleft cordner $p[6],$p[7]

    # deprecated code:
    #  if ($x1<$x2 ) {
    #     $p[0]=$x1-$offset;
 72 #     $p[6]=$x1-$offset;
    #     $p[2]=$x2+$offset;
    #     $p[4]=$x2+$offset;
    #  }
    #  else {
 77 #     $p[0]=$x2-$offset;
    #     $p[6]=$x2-$offset;
    #     $p[2]=$x1+$offset;
    #     $p[4]=$x1+$offset;
    #  }
 82
    #  if ( $y1<$y2) {
    #     $p[1]=$y1-$offset;
    #     $p[3]=$y1-$offset;
    #     $p[5]=$y2+$offset;
 87 #     $p[7]=$y2+$offset;
    #  }
    #  else {
    #     $p[1]=$y2-$offset;
    #     $p[3]=$y2-$offset;
 92 #     $p[5]=$y1+$offset;
    #     $p[7]=$y1+$offset;
    #  }
    # print "@p \n"; # DBG

 97   return @p;

    } # End square

    ####################################################################
102 #

    sub selfDiamond {
      my $startX = shift; # Start point (switch)
      my $startY = shift;
107   my $endX   = shift; # End point (switch)
      my $endY   = shift;

      # 1) Find distance from start point to end point.
      my $distance = manhattan_distance( $startX, $startY, $endX, $endY );
112
      ## EXPERIEMENTAL:
      ## A (possible) way to recover from the problem that
      ## the point_in_polygon (called at a later stage)
      ## subrutine may not include points laying at the
117   ## edge of the polygon.

      $distance += $offset;


122   # 2) Find the rectilinear area around the end vertex
      return rectilinear_area($startX, $startY, $distance);
    }
```

```
    ####################################################################
127 #

    sub diamond {
      my $startX = shift; # Start point (switch)
      my $startY = shift;
132   my $endX   = shift; # End point (switch)
      my $endY   = shift;

      # 1) Find distance from start point to end point.
      my $distance = manhattan_distance( $startX, $startY, $endX, $endY );
137
      ## EXPERIEMENTAL:
      ## A (possible) way to recover from the problem that
      ## the point_in_polygon (called at a later stage)
      ## subrutine may not include points laying at the
142   ## edge of the polygon.

      $distance += $offset;


147   # 2) Find the rectilinear area around the end vertex
      return rectilinear_area($endX, $endY, $distance);
    }

    ####################################################################
152 #

    sub bigSquare {

      my ($x0,$y0,$x1,$y1) = @_;
157   my $D = manhattan_distance( $x0, $y0, $x1, $y1 );

      return  ( $x0+$D, $y0-$D,
                $x0+$D, $y0+$D,
                $x0-$D, $y0+$D,
162             $x0-$D, $y0-$D);

    }

    ####################################################################
167 #

    sub rectangle {
      my ($x0,$y0,$x1,$y1) = @_;
      my $Dx = $x1-($x0+$offset);
172   my $Dy = $y1-($y0+$offset);
      my @p = ();

      $p[0] = $x0+$Dy; # P0
      $p[1] = $y0-$Dx;
177
      $p[2] = $x1+$Dy; # P1
      $p[3] = $y1-$Dx;

      $p[4] = $x1-$Dy; # P2
182   $p[5] = $y1+$Dx;

      $p[6] = $x0-$Dy; # P3
      $p[7] = $y0+$Dx;

187   return @p;

    }

    ###################################################################
192 #

    sub rectangleHalf {
      my ($x0,$y0,$x1,$y1) = @_;
      my $Dx = ceil( ($x1-$x0)/2 );
197   my $Dy = ceil( ($y1-$y0)/2 );
      my @p = ();

      $p[0] = $x0+$Dy; # P0
      $p[1] = $y0-$Dx;
202
      $p[2] = $x1+$Dy; # P1
      $p[3] = $y1-$Dx;

      $p[4] = $x1-$Dy; # P3
207   $p[5] = $y1+$Dx;
```

```
      $p[6] = $x0-$Dy; # P4
      $p[7] = $y0+$Dx;

212   return @p;

    }

    ####################################################################
217 #

    ####################################################################
    #

222 sub AreaDebug {
      # @_ points (x1,y1,x2,y2,...);
      my @p = @_;

      for (my $i=0; $i < $#p; $i+=2 ) {
227     print "($p[$i], $p[$i+1]) ";
      }

      print "\n";

232 }

    1;

    __END__
```

2.0.6 IcNet::LiP::Path

```
    package IcNet::LiP::Path;

  3 #
    # POD documentation at end of file.
    #

    use strict;
  8 use Graph;
    use overload '""' => \&stringify;

    sub new {
      my $invoce = shift;
 13   my $class = ref($invoce) || $invoce;
      my $self = {};
      my $p = Graph::Directed->new;

      $self = { Path => $p,
 18             ID => "",
                Resource => "",
                @_
              };

 23   $self->{Path} = shift if @_;

      bless ($self, $class);
      return $self;
    }
 28
    ############################################################
    #Resource

    sub Resource  {
 33   my $self = shift;

      $self->{Resource} = shift if @_;

      return $self->{Resource};
 38 }

    #############################################################
    # From

 43 sub Root {

      # Find root nodes
      my $self = shift;
      my @s = $self->{Path}->source_vertices();
 48
      # Assuming that there is only one source vertex.
      # (and it should be..)
      @s?return $s[0]:undef;
```

```
 53  }

     ############################################################
     # To

 58  sub Leaf {
       # Find leaf nodes
       my $self = shift;
       my @s = $self->{Path}->sink_vertices();

 63    return @s;
     }

     ############################################################
     # Path
 68
     sub Path {
       my $self = shift;

       if ( @_ ) {
 73       $self->{Path}  = shift;
       };

       return $self->{Path};
     }
 78
     ############################################################
     # addEdge

     sub addEdge {
 83    my $self = shift;
       $self->{Path}->add_edge($_[0],$_[1]);
     }

     ############################################################
 88  #

     sub addPath {
       my $self = shift;
       # @_ => edges
 93
       while ( my ($s,$e) = splice(@_,0,2) ) {
         $self->{Path}->add_edge($s,$e);
       }

 98    return $self->{Path};
     }

     ############################################################
     # insertPath
103  sub insertPath {

     }
     ############################################################
     # ID
108
     sub ID {
       my $self = shift;

       if ( @_ ) {
113      $self->{ID} = shift;
       }

       return $self->{ID};
     #  return join('_', $self->{To}->ID, $self->{From}->ID);
118  }

     ############################################################
     # PrintAll

123  sub PrintAll {

     }

     ############################################################
128  # Length
     sub Length {
       my $self = shift;
     }

133  ############################################################
     # conflictAt

     sub conflictAt {
```

```
  my $self = shift;
138 }

    ############################################################
    # links

143 sub links {
      my $self = shift;

      $self->{Path}->add_edges( @_ ) if @_;

148   return $self->{Path}->edges;

    }

    ############################################################
153 # through
    sub through {
      my $self = shift;
      my $switch = shift; # Switch ID
      my @out = ();       # Result
158   my $in = "";        # Incoming link ID
      my @e;              # tmp, edges
      my ($startV, $endV );

      if ( $self->{Path}->has_vertex( $switch ) ) {
163     @e = $self->{Path}->in_edges( $switch );

        # There is always just one in edge,
        # That's why I can find in link like this.
        $in = $self->{Path}->get_attribute( "link", $e[0], $e[1] );
168
        @e = $self->{Path}->out_edges( $switch );
        while ( ($startV, $endV) = splice(@e,0,2) ) {
          push @out, $self->{Path}->get_attribute( "link", $startV, $endV );
        }
173
      }

      return $in, \@out;

178 }

    ############################################################
    # MBG_edge

183 sub MBG_edge {
      my $self = shift;
      my $switch = shift; # The switch we're looking for.
      my $mbg_edge = Graph::Directed->new(); # The result.

188   my $sV_start; # Temp vars..
      my $sV_end;
      my $link_S;
      my $eV_start;
      my $eV_end;
193   my $link_E;

      # Test: make sure that we have the
      # switch we are looking for before continuing.
      return undef unless $self->{Path}->has_vertex( $switch );
198
      # This will be start and end vertices of the
      # resulting graph.
      my @startV = $self->{Path}->in_edges( $switch );
      my @endV = $self->{Path}->out_edges( $switch );
203
      while ( @startV ) {
        $sV_start = shift @startV;
        $sV_end = shift @startV;
        $link_S = $self->{Path}->get_attribute('link',$sV_start, $sV_end);
208     for ( my $i=0; $i <= $#endV; $i+=2 ) {
          $eV_start = $endV[$i];
          $eV_end = $endV[$i+1];
          $link_E = $self->{Path}->get_attribute('link',$eV_start, $eV_end);

213       $mbg_edge->add_edge( $link_S, $link_E );
          # add attributes
#         $mbg_edge->set_attribute('capacity', $link_S, $link_E, 1 );
          $mbg_edge->set_attribute('pathid', $link_S, $link_E, $self->{ID} );
#         $mbg_edge->set_attribute('cy', $sV, $eV, 1 ); # Reserve
218 #     $mbg_edge->set_attribute('ct', $sV, $eV, 1 ); # Reserve
        }
      }
```

```
  return $mbg_edge;

}

############################################################
# stringify

sub stringify {
  my $self = shift;
  my $string = "Path: ";
  $string .= "$self->{Path}";

# if ( ref( $self->{Resource}) ) {
#   $string .=  $self->{Resource}->ID;
# } else {
#   $string .= $self->{Resource};
# }

  return $string;

}

1;

#
# End of code.
#
#############################################################

#############################################################
#
# POD documentation.
#
```

2.0.7 IcNet::LiP::Restrictarea

```
package IcNet::LiP::RestrictArea;

############################################################
# Helper module to find switches in different areas.
# This is returned as an subgraph.
#
# POD documentation at end of file.
#
############################################################

use IcNet::Geometry;
use strict;
use Graph;
use Graph::Undirected;
use IcNet::LiP::GraphOperations;

BEGIN {
  use strict;
  require Exporter;

  our @ISA = qw(Exporter);
  our @EXPORT = qw(&FindSubGraph &verticesInPolygon);
  our %EXPORT_TAGS = ( );
  our @EXPORT_OK = qw( );
  our $VERSION = 0.1;

}

sub def_findArea {

  my $startX = shift; # Start point (switch)
  my $startY = shift;
  my $endX   = shift; # End point (switch)
  my $endY   = shift;

  # 1) Find distance from start point to end point.
  my $distance = manhattan_distance( $startX , $startY , $endX, $endY );

  ## EXPERIEMENTAL:
  ## A (possible) way to recover from the problem that
  ## the point_in_polygon (called at a later stage)
  ## subrutine may not include points laying at the
  ## edge of the polygon.

#  $distance += 1; # Experiment 1 (includes)
  $distance -= 1; # Experimen 2 (exclude)
```

```
    # 2) Find the rectilinear area around the end vertex
    return rectilinear_area($endX, $endY, $distance);
50
  };

  sub FindSubGraph {
    my $N           = shift; # Network Graph
55  my $startVertex = shift; # Start Vertex (switch)
    my $endVertex   = shift; # End Vertex (switch)

    my $subref_findArea = shift; # Code to find polygon
                                  # area around points.
60
    # unless specified with 0 or undef then use default
    $subref_findArea = \&def_findArea unless $subref_findArea;

    # 1) Find the area around the end vertex
65  my @polygon = &$subref_findArea(vertexPosition($N, $startVertex) ,
                                     vertexPosition($N, $endVertex) );

    # 2) Retrive all vertices in that rectilinear areas
    #    This gives us a subgraph that we will explore
70  my $subG = verticesInPolygon( $N, \@polygon );

    # HACK: poor code recovery !
    # The point_in_polygon() method called in sub
    # _vertexsInPolygon does have a  problem to determine
75  # whether a point on the edge of the polygon should be
    # included as points in the polygon or not. The startVertex
    # should always appear on one of the edges.
    #
    # I need to test if the $startVertex is included or
80  # not; it must be included !!
    copyVertex($subG,$N, $startVertex)
      unless  $subG->has_vertex( $startVertex ) ;

    return $subG;
85
  }

  ##########################################################
  #
90 #

  sub verticesInPolygon {
    my $N = shift;
    my $poly = shift;
95  my @p;     # tmp var, point
    my %tmpA; # tmp var, attributes
    my $subG = new Graph::Undirected;

  # print "start verticesInPolygon\n"; # DBG
100
    foreach my $n ( $N->vertices ) {
      @p = vertexPosition($N, $n );

      # If vertex is inside the polygon, add it to the subGraph.
105   if ( point_in_polygon( $p[0], $p[1], @$poly ) ) {
        copyVertex($subG,$N, $n);
      }

    }
110
  # print "end verticesInPolygon\n"; # DBG

    return $subG;

115 }

  1;

  __END__
120
  ##########################################################
  #
  # POD Section
  #
```

2.0.8 IcNet::LiP::FindPathModule

```
package IcNet::LiP::FindPathModule;
```

```
##############################################################
# Helper module to find different paths in a Graph
# representing the LIP network.
#
# POD documentation at end of file.
#
##############################################################

use IcNet::Geometry;
use strict;
use Graph;
use Graph::Undirected;
use IcNet::LiP::GraphOperations;
use IcNet::LiP::RestrictArea;

BEGIN {
  use strict;
  require Exporter;

  our @ISA = qw(Exporter);
  our @EXPORT = qw( &FindAllPath &AreaSearch);
  our %EXPORT_TAGS = ( );
  our @EXPORT_OK = qw( );
  our $VERSION = 0.1;

}

# This would tell if a point is closer to the enpoint or not
# (..and also serve as an example of uninterpretable Perl code. )
my $def_acceptPoint = sub {
  return ( vertexDistance($_[0], $_[4], $_[2]) <=
           vertexDistance($_[0], $_[3], $_[2]) );
};

##############################################################
#
#

sub FindAllPath {
  my $G           = shift; # Network Graph
  my $startVertex = shift; # Start Vertex (switch)
  my $endVertex   = shift; # End Vertex (switch)

  my $acceptPoint_sub = shift; # Code to limit the
                               # search criteria.
  my $subref_findArea = shift; # Code to find polygon
                               # area around points.

  # unless specified with 0 or undef then use default
  $acceptPoint_sub = $def_acceptPoint
    unless $acceptPoint_sub;

  # 2) Retrive all vertexs in that rectilinear areas
  #    This gives us a subgraph that we will explore
  my $subG = FindSubGraph( $G, $startVertex, $endVertex, $subref_findArea);

  # 3) Create all possible "connections" in the subgraph
  completeNetwork( $subG );

  # 4) find Paths
# print "$acceptPoint_sub \n"; exit(1); # DBG
  my $paths = EdgeSeek( $subG, $startVertex, $endVertex, $acceptPoint_sub );

  return $paths;

}

##############################################################
#
#

sub AreaSearch {
  my $N = shift;
  my $startVertex = shift;
  my $endVertex = shift;
  my $subref_findArea = shift;
  my $pathCollection = [];
  my $path = Graph::Directed->new();
  $path->add_vertex( $startVertex );

  recursiveAreaSearch($N, $startVertex, $endVertex,
                      $subref_findArea, $pathCollection,
                      $path);
```

```
      return $pathCollection;
 89
    }

    sub recursiveAreaSearch {
      my $N                = shift;
 94   my $lastVertex       = shift;
      my $endVertex        = shift;
      my $subref_findArea  = shift;
      my $pathCollection   = shift;
      my $path             = shift;
 99
      # if I've arrived at the endvertex I
      # have found a path from start to end.
      if ( $lastVertex eq $endVertex ) {
        my $C = cloneGraph( $path );
104     push @$pathCollection, $C;
      }
      else {
        my $subG = FindSubGraph($N, $lastVertex, $endVertex,
                                $subref_findArea);
109
        # Do not visit last vertex again
        $subG->delete_vertex( $lastVertex );

        foreach my $nextV ( $subG->vertices() ) {
114       # $path->add_vertex( $nextV );
          $path->add_edge( $lastVertex, $nextV );

          # Not yet at the end vertex, continue
          # search.
119       recursiveAreaSearch($subG, $nextV, $endVertex,
                              $subref_findArea,
                              $pathCollection, $path);

          $path->delete_vertex( $nextV );
124     }

      }

    }
129
    ############################################################
    #
    #

134 sub EdgeSeek {
      my $N = shift; # Network Graph
      my $s = shift; # start vertex
      my $e = shift; # End vertex
      my $crit_ref = shift;
139   my $pathCollection = [];
      my $path = Graph::Directed->new();
      $path->add_vertex( $s );

      _initFindPath( $N );
144
    # print "$crit_ref \n"; exit(1); # DBG
      I_MinSpan_EdgeSeek($N, $s, $e, $path, $pathCollection, $crit_ref);

      return $pathCollection;
149
    }

    ############################################################
    #
154 #

    sub _initFindPath {
      my $N = shift; # Network Graph

159   foreach my $v ( $N->vertices ) {
        $N->set_attribute('visited',$v,0);
      }

    }
164
    ############################################################
    # I_MinSpan_EdgeSeek( $N_Graph, $currentVertex, $endVertex,
    #    $path_ref, $ref_toStorePath )
    #
169
    sub I_MinSpan_EdgeSeek {
```

```
      my $N = shift;  # Network graph
      my $explVertex = shift; # Current vertex to explore
      my $endV = shift; # End vertex
174   my $path  = shift;
      my $pathCollection = shift;
      my $crit_ref = shift;

      # Set this vertex as visited
179   $N->set_attribute('visited', $explVertex, 1);

      # Add new vertex to path.

      # End vertex or not ?
184   if ( $endV eq $explVertex ) {
        # make a clone to store as a result.
        my $C = cloneGraph( $path );
        push (@$pathCollection , $C );
      }
189   else {

        # Explore each neighbor vertices, if they are not visited
        # then explore this vertex.
        # This is an Depth First Search
194     foreach my $nextVertex ( $N->neighbors( $explVertex ) ) {

          # vertex already visited ??
          if ( $N->get_attribute('visited', $nextVertex ) == 0) {

199         # NOTE: if it is desirable to have other criterias
            # for paths, this is the place to hack !!

            my @sourceV = $path->source_vertices();
            if ( &$crit_ref($N, $sourceV[0], $endV, $explVertex, $nextVertex) ) {
204           $path->add_vertex( $nextVertex );
              $path->add_edge( $explVertex, $nextVertex );
              I_MinSpan_EdgeSeek($N, $nextVertex, $endV, $path,
                                 $pathCollection, $crit_ref);
              $path->delete_vertex( $nextVertex );
209         }
          }
        } # end foreach ...
      } # end else ...

214   $N->set_attribute('visited', $explVertex, 0);
    #  pop @$path;

    } # End I_MinSpan_EdgeSeek

219 1;

    __END__
```

2.0.9 IcNet::LiP::Greedy::ExploreNetPM

```
    package IcNet::LiP::Greedy::ExploreNetPM;

  3 ##########################################################
    # Tries to generate a network, with point-to-multipoint (PM)
    # connections.
    #
    # POD documentation at end of file
  8 #

    BEGIN {
      use strict;
      require Exporter;
 13
      our @ISA = qw(Exporter);
      our @EXPORT = qw( &ExploreNet &ExploreVertex );
      our %EXPORT_TAGS = ( );
      our @EXPORT_OK = qw( $ExploreNetValues );
 18   our $VERSION = 0.1;

    }

    use strict;
 23 #use diagnostics; # DBG - debug
    use Data::Dumper; # for debugging

    use IcNet::Geometry;
    #use IcNet::RecursiveAA;
 28 use IcNet::SearchAofA;
    use IcNet::LiP::GraphOperations;
```

```
    use IcNet::LiP::RestrictArea;
    use IcNet::LiP::Channel;
    use IcNet::LiP::CommunicationGraph;
33  use IcNet::LiP::AreaSelectMethods;
    use IcNet::LiP::MapDemands;

    #use Print_AofA; # Experimental package
    use IcNet::LiP::Greedy::OptimizePM;
38  use IcNet::LiP::Greedy::WireOptimize;
    use IcNet::LiP::Greedy::WireUsageOptimize;
    use IcNet::LiP::Greedy::FSWireUsageOptimize;
    use IcNet::LiP::Greedy::FSWireUsageDistance;
    use IcNet::LiP::Greedy::FSWireUnusedDist;
43  use IcNet::LiP::Greedy::FSWireUnusedAvgDist;
    use IcNet::LiP::Greedy::THDiam;
    use IcNet::LiP::Greedy::RemoveIdentical;
    use IcNet::LiP::Greedy::CandidateWrapper;

48  #  package globals
    my $linkFrequency = 1000;
    my $linkWidth     = 8;
    our @EXPORT_OK;

53  ################################################
    ## !!!!!                                      ##
    ## HACK here to change area select method     ##
    ##                                            ##
    #my $ref_area = \&IcNet::LiP::AreaSelectMethods::square;
58  my $ref_area = \&IcNet::LiP::AreaSelectMethods::diamond;

    BEGIN {
      SetASM_offset(2);
    }
63  ##                                            ##
    ################################################

    ##########################################################

68  sub ExploreNet {
      my $net = shift; # initial network graph
      $linkWidth = shift;
      $linkFrequency = shift;
      my @demands = @_;
73
      # a copy to work with
      my $ComNet = cloneGraph( $net );

      # Map demands(channels)
78    addDemands($ComNet, @demands);

      # the returned result
      my $solution = (ref $net)->new( $net->vertices );

83    # Copy positions to $solution,
      # this step is not necessary, but handy later
      # if I want to graphically display the result.
      foreach ( $net->vertices ) {
        vertexPosition($solution , $_, vertexPosition( $net, $_ ));
88    }

      # ( BUILD HEAD OF TREE HERE )

      my @outerV = ();
93    my $vertex;

      ## START HACK #######################
      ##
      ## To manually set the order
98    ## Mark out to use findOuterVertices( .. )
      ##

    #    @outerV = ('C','A','B','D');
    # while ( $vertex = shift @outerV ) {  # HACK A
103
      ##
      ###### ( or ....)
      ##
      ## Mark out the 2 lines below for predefined
108   ## way of order

      while ( @outerV = findOuterVertices( $ComNet ) ) {
      $vertex = $outerV[0];

113   ##
```

```
    ##### ( or ....) EXPERIMENT:

  # while ( @outerV = findOuterVertices_2( $ComNet ) ) {
  # $vertex = $outerV[0];
118
    ##
    ## END HACK ###########################

      my ($optOut, $optIn) = ExploreVertex( $vertex, $ComNet );
123
      # A simple but probably an error prone way of checking
      # if an optimal outgoing candidate has been found.
      unless ( @$optOut ) {
        print "No need to update outgoing solution \n";
128   }
      else {
        outgoingUpdate( $ComNet, $solution, $vertex, $$optOut[0] );
      }

133   unless ( @$optIn ) {
        print "No need to update incoming solution \n";
      }
      else {
        incomingUpdate( $ComNet, $solution, $vertex, $$optIn[0]  );
138   }

      # (BUILD NEW NODES OF TREE HERE)

      $ComNet->delete_vertex( $vertex );
143   print "ComNet:", $ComNet,"\n"; # DBG
      print "solution: ", $solution,"\n"; # DBG

    } ## END while

148 return $solution; #, @demands;
  }

  ##########################################################
  # Explore vertex
153
  sub ExploreVertex {
    my $exploreVertex = shift;
    my $ComNet = shift;   # communication network
  # my $subSolution = new IcNet::LiP::CommunicationGraph;
158 my @optOutCandidates = ();
    my @optInCandidates = ();

    print "======== test Vertex $exploreVertex: \n";
    print "- optimizing for outgoing traffic $exploreVertex \n"; # DBG
163 my @outDemand = $ComNet->getOutgoingDemandsPM( $exploreVertex );

    # 1a) Find canditates for the
    # partial networks this $exploreVertex will send to.
    my $outCandidates =
168    collectOutgoingCandidates( $ComNet, $exploreVertex, \@outDemand );

    # optimize
    if ( @$outCandidates ) {
      print "optimize outgoing:\n"; # DBG
173 #     exit(0); # DBG
      @optOutCandidates =
        optimizeOutgoingCandidates( $outCandidates, $ComNet, $exploreVertex );
    }
    else {
178   print "no outgoing candidates found.\n";
    }

    # ---- for receive params :
    print "- optimizing for incoming traffic $exploreVertex \n";
183 my $inCandidates = collectIncomingCandidates( $ComNet, $exploreVertex);

    # optimize
    if ( @$inCandidates ) {
      print "optimize incoming:\n"; # DBG
188   @optInCandidates =
         optimizeIncomingCandidates( $inCandidates, $ComNet, $exploreVertex );
    }
    else {
      print "no incomng candidates found.\n";
193 }

  # print @optInCandidates;  # DBG

    return (\@optOutCandidates, \@optInCandidates);
```

```
198
    }
    ##########################################################

    sub candidateVertices {
203   # $_[0] graph - network
      # $_[1] end vertex
      # $_[2] start vertex
      # $_[3] reference to subrutine for finding an area

208   # ( IcNet::LiP::RestrictArea::FindSubGraph )
      return FindSubGraph(@_);

    }

213 ##########################################################
    #

    sub collectOutgoingCandidates {
      my $solarSystem = shift;    # Network graph
218   my $dockingStation = shift; # Vertex to optimize from
      my $spaceShips = shift;     # Demands as arrayRef (Channels)


      my $returnCandidates = [];  # Ret val. AofA
223
      foreach my $Targets ( @$spaceShips ) {

        ## Note the array is organized as :
        ## $$Targets[0], channel/demand
228     ## $$Targets[1 .. n], endtargets (vertices/switches) for point to
        ##   multipoint connections.

        ## The code will handle point-to-multipoint communication.

233     my $candidate = [];
        $$Targets[0]->printAll(); # (Print to screen) DBG

        # We explore each endtarget individually
        foreach my $i ( 1 .. $#$Targets ) {
238
          # If demands are equal we have arrived,
          # no further planning of notwork (flight route) is needed
          unless ( $$Targets[$i] eq $dockingStation ) {
            print "start : $dockingStation end: ", $$Targets[$i]," \n"; # DBG
243
            # Selects point in an area.
            my $subG = candidateVertices( $solarSystem,
                                          $dockingStation,
                                          $$Targets[$i], $ref_area   );
248
            # code recovery:
            # remove vertex we are optimizing from
            #  as a canditate vertex.
            $subG->delete_vertex( $dockingStation );
253
            # Show candidates
            print $subG->vertices,"\n"; # DBG

            # Map the destination and the new vertex i might
258         # connect to. ie. we keep it as an new edge to connect
            # to the Communication Graph later.
            my @tmpC = map {
              my $candidate = { Destination=> $$Targets[$i],
                                NewVertex=> $_ };
263         } $subG->vertices;

            # store candidates
            push @$candidate, [ @tmpC ];

268       } # END unless ..
        } # End foreach ..

        # Now we have explored all candidates each end point might
        # have. Now we make all combinations of point to multipoint
273     # candidates
        my $allCombinations =
          IcNet::SearchAofA->start( 0, $candidate );

        # Now just make all combinations as a
278     # ''candiadateWrapper''. This comes in handy during the
        # optimalization method. (See doc for
        # CandidateWrapper.pm for details)
        my @candidateWrap = ();
```

```
      foreach my $R ( $allCombinations->result ) {
283        my $CL = new IcNet::LiP::Greedy::CandidateWrapper;
           foreach ( @$R ) {
             $CL->addCandidate( $_ );
           }
         push @candidateWrap, $CL;
288   }

      # ... and save everything
      push @$returnCandidates, [ $$Targets[0], @candidateWrap ]
           if @candidateWrap;
293
    } # END my $Targets ( @$spaceShips ) { ...

    return $returnCandidates;

298 }

##############################################################
#

303 sub collectIncomingCandidates {

    my $ComNet     = shift; # Communication graph
    my $towards       = shift; # Vertex to optimize towards

308 my @in_edges = $ComNet->in_edges($towards);
    my ($start_vertex, $end_vertex);

    my $returnCandidates = [];

313 # NOTE: $end_vertex is always the same as $towards.
    # $start_vertex and $end_vertex represent an edge
    # in the graph.
    while( ($start_vertex,$end_vertex) = splice(@in_edges,0,2) )
    {
318   # If equal vertices, we have arrived - then do nothing
      unless ( $start_vertex eq $end_vertex ) {
        print "outgoing:", $start_vertex," towards: $end_vertex \n";

        my $subG = candidateVertices( $ComNet, $towards,
323                                   $start_vertex,
                                      $ref_area);
        $subG->delete_vertex( $towards );
        print $subG->vertices,"\n";

328     print "Channels in this connection is :\n";
        foreach my $demands
          ( $ComNet->getDemands($start_vertex, $end_vertex) )
        {
          $demands->printAll;
333       my @candidates = ();
          push @candidates, $demands;

          foreach my $c ( $subG->vertices ) {

338 #         my $candidate = { Destination=> $towards,
    #                           NewVertex=> $c };

            my $candidate = { Destination=> $start_vertex,
                              NewVertex=> $c };
343
            my $CL = new IcNet::LiP::Greedy::CandidateWrapper;
            $CL->addCandidate( $candidate );

            push @candidates, $CL;
348       }

          #.. and finally save everyting
          push @$returnCandidates, [@candidates];

353     } # END foreach my $demands ...

      } # END unless ..

    } # END While
358
# foreach( @$returnCandidates) { print @$_;print "\n";} # DBG

    return $returnCandidates;

363 }

##############################################################
```

```
    #

368 sub optimizeCandidates {
      my $Combinations = shift;     # AofA
      my $net = shift;  # Network
      my $switch = shift; # optimising to/from this switch

373   ###### START HACK  ################################
      ##
      ## Be experimental here with different optimalization
      ## methods:
      ##
378
      ## exprimental
    # my $opt =
    #    IcNet::LiP::Greedy::OptimizePM->start(1, $Combinations,
    #                                            $net, $switch);
383
      ##
      ## Minimum Wires
      ##
    # my $opt =
388 #    IcNet::LiP::Greedy::WireOptimize->start(1, $Combinations,
    #                                            $net, $switch,
    #                                            $linkWidth,
    #                                            $linkFrequency);
      #####################
393   ##
      ## min. wires, and min unused traffic
      ##

    # my $opt = IcNet::LiP::Greedy::WireUsageOptimize->start(1);
398 # $opt->AofA( $Combinations );
    # print "Number of combinations ", $opt->totalCmb(),"\n";
    # $opt->doSearch( $net, $switch, $linkWidth, $linkFrequency);

      ############################
403   ## EXPERIMENTAL: FastSearch
      ## min. wires, min connections, and min unused traffic
      ##

    #    my $opt = IcNet::LiP::Greedy::FSWireUsageOptimize->start(1);
408 #    $opt->AofA( $Combinations );
    #    print "Number of combinations ", $opt->totalCmb(),"\n";
    #    $opt->doSearch( $net, $switch, $linkWidth, $linkFrequency);

      ##
413   ##
      #################################
      ## min. wires, min connections, min unused traffic
      ## and min distances left
      ##
418
    #    my $opt = IcNet::LiP::Greedy::FSWireUsageDistance->start(1);
    #    $opt->AofA( $Combinations );
    #    my %dist = IcNet::LiP::GraphOperations::switchLenghts( $net );
    #    $opt->setDistance( \%dist );
423 #    print "Number of combinations ", $opt->totalCmb(),"\n";
    #    $opt->doSearch( $net, $switch, $linkWidth, $linkFrequency);

      ##
      ########################################
428   ## min. wires, min unused traffic
      ## and min distances left
      ##

    #    my $opt = IcNet::LiP::Greedy::FSWireUnusedDist->start(1);
433 #    $opt->AofA( $Combinations );
    #    my %dist = IcNet::LiP::GraphOperations::switchLenghts( $net );
    #    $opt->setDistance( \%dist );
    #    print "Number of combinations ", $opt->totalCmb(),"\n";
    #    $opt->doSearch( $net, $switch, $linkWidth, $linkFrequency);
438
      #################################
      ## min. wires, min unused traffic
      ## and min distances left
      ##
443
    #    my $opt = IcNet::LiP::Greedy::FSWireUnusedDist->start(1);
    #    $opt->AofA( $Combinations );
    #    my %dist = IcNet::LiP::GraphOperations::switchLenghts( $net );
    #    $opt->setDistance( \%dist );
448 #    print "Number of combinations ", $opt->totalCmb(),"\n";
    #    $opt->doSearch( $net, $switch, $linkWidth, $linkFrequency);
```

```
      ##
      #################################
453   ## min. diameter
      ##
      ##

        my $opt = IcNet::LiP::Greedy::THDiam->start(1);
458     $opt->AofA( $Combinations );
        my %dist = IcNet::LiP::GraphOperations::switchLenghts( $net );
        $opt->setDistance( \%dist );
        print "Number of combinations ", $opt->totalCmb(),"\n";
        $opt->doSearch( $net, $switch, $linkWidth, $linkFrequency);
463
      ##
      ###################
      ## min. wires, min unused traffic
      ## and min average distances left
468   ##

    #   my $opt = IcNet::LiP::Greedy::FSWireUnusedAvgDist->start(1);
    #   $opt->AofA( $Combinations );
    #   my %dist = IcNet::LiP::GraphOperations::switchLenghts( $net );
473 #   $opt->setDistance( \%dist );
    #   print "Number of combinations ", $opt->totalCmb(),"\n";
    #   $opt->doSearch( $net, $switch, $linkWidth, $linkFrequency);

      ##
478   ##
      ###### END HACK #################################

      # Test DBG
      print "Number of combinations tried : ",
483         $opt->totalCmb," vertex: $switch\n";

      print "Results: \n";

      # Show solution(s).
488   # This also shows how to break down the solution results:
      my @solutions = $opt->result();
      print "Number of solutions: ", $#solutions+1 ,"\n";

      return $opt->result;
493
    }

    #########################################################

498 sub optimizeOutgoingCandidates {
      return optimizeCandidates( @_ );
    }

    #########################################################
503
    sub optimizeIncomingCandidates {
      return optimizeCandidates( @_ );
    }

508 #########################################################

    sub outgoingUpdate {
      my $ComNet = shift;  # Net - Communication needs
      my $S = shift;       # Net - solution
513   my $switch  = shift; # Switch we're optimizing from
      my $opt = shift;     # Thee optimal solution
      my $tempSolution = (ref $ComNet)->new();

      # Update
518   foreach my $o ( @$opt ) {
        # $$o[0] : demand (IcNet::LiP::Channel)
        # $$o[1] : CandidateWrapper,
        # Update Paths
        foreach my $nl ( $$o[1]->NewLinks ) {
523       $$o[0]->addPath($switch, $nl);

          # Update solution network
          $S->add_edge($switch, $nl)
            unless $S->has_edge($switch,$nl);
528
          # Update Communication Network,
          foreach my $nd ( $$o[1]->Candidates ) {
            $ComNet->addDemands($nl, $nd->{Destination}, $$o[0]);
          }
533     }
```

```
    }

    print "out S: $S \n"; # DBG
538
# print Dumper( $S ); exit(0);# DBG

    # (IMPLEMENTATION OPTION:
    # I might have code to delete outgoing edges from the
543 # Commuication Graph  here.
    # The vertex I've optimized will be deleted in the
    # end of the (while) loop in ScanExplore. This will
    # delete all edges as well. It will depend on
    # your Graph-library implementation. )
548
}

###########################################################

553 sub incomingUpdate {
    my $ComNet = shift; # Net - Communication needs
    my $S = shift;      # Net - solution
    my $switch = shift; # Switch we're optimizing towards
    my $opt = shift;    # Thee optimal solution
558
    foreach my $o ( @$opt ) {
      foreach my $newlink ( $$o[1]->NewLinks ) {
        # Update Paths
        $$o[0]->addPath($newlink,$switch);
563
        # Update solution network
        $S->add_edge($newlink, $switch)
          unless $S->has_edge($newlink, $switch);

568     # Update Communication Network,
        # (do not add new demands/edges if the new
        # updated demand is pointing towards the same
        # vertex.)
        foreach my $nd ( $$o[1]->Candidates ) {
573       $ComNet->addDemands( $nd->{Destination},$newlink, $$o[0]);
#            unless ( $newlink eq $nd->{Destination} );
        }
      }

578 }

    print "in S: $S \n"; # DBG

}
583
sub findOuterVertices {
    my $C = shift; # Communication graph

    # Code recovery;
588 # Possible bug in the Graph.pm package.
    # It won't delete the last vertex, the result is that
    # this subrouting will always return a vertex.
    # With one vertex left there is no need to go further.
    return () unless $C->vertices() > 1;
593
    # get outer vertices as keys
    my %vertexPoints = _vertexPoints( $C );

    # Find which vertices that would make the "best"
598 # choice.
    foreach ( keys %vertexPoints ) {
      # get communication targets
      my @targets = $C->successors( $_ );

603   # calc point for each target
      foreach my $target ( @targets ) {
        # use areaselect method

        my $S = FindSubGraph( $C, $_, $target, $ref_area );
608
        # remove target and start vertex
        # we should not take these into account
        $S->delete_vertex( $_ );
        $S->delete_vertex( $target );
613
        # give points
        foreach my $p ( $S->vertices ) {
          # unforseen code recovery;
          # to ensure that I just add values to vertices that
```

```
618          # are at located outermost I must check that
             # it exist in the %vertexPoint. If it doesn't
             # it is an interior vertex.
             ++$vertexPoints{ $p } if exists $vertexPoints{ $p };
           }
623      }
       }

       # DBG
     #  foreach ( keys %vertexPoints ) {
628  #    print $_," points", $vertexPoints{ $_ },"\n";
     #  }

       # sort by points;
       # maps by temp using array : ([sw_name,points],[sw_name,points],..)
633    my @sortPoints = sort { $$a[1] <=> $$b[1] }
                        map { [$_,$vertexPoints{$_}]} keys %vertexPoints;

       ##
       print "POINTS: ";
638    foreach ( @sortPoints) {print "$$_[0]($$_[1]) ";}
       print "\n";

       # return, switches with least points are first in list.
       return map { $$_[0] } @sortPoints;
643
     }

     ########################################################################
     #
648
     sub _vertexPoints {
       my  $C = shift;

       # Map points and switches,
653    # this is needed to perform the graham scan.
       my %h_map = ();
       my @xypos = ();
       foreach ( $C->vertices ) {
     #     my ($x,$y) = vertexPosition($C,$_ );
658      my ($x, $y) = $C->vertexPosition( $_ );
         $h_map{$x}{$y} = $_;
         push @xypos, $x, $y;
       }

663    # Find outer points, and map back to switches
       my @xyouter = convex_hull_graham( @xypos );
       my %vertexPoints = ();
       for ( my $i=0; $i < $#xyouter; $i+=2) {
         my ($x,$y) = ($xyouter[$i], $xyouter[$i+1]);
668      $vertexPoints{ $h_map{$x}{$y}}  = 0;
       }

       return %vertexPoints;

673  }

     ##############################################################
     #

678  sub findOuterVertices_2 {
       my $C = shift; # Communication graph

       return () unless $C->vertices() > 1;

683    # get outer vertices as keys
       my %vertexPoints = _vertexPoints( $C );

       # Find which vertices that would make the "best"
       # choice.
688    foreach ( keys %vertexPoints ) {
         # get communication targets
         my @targets = $C->successors( $_ );

         # calc point for each target
693      foreach my $target ( @targets ) {
           # use areaselect method
           my $S = FindSubGraph( $C, $_, $target, $ref_area );

           # remove target and start vertex
698        # we should not take these into account
           $S->delete_vertex( $_ );
           $S->delete_vertex( $target );
```

```
        # give points
703     foreach my $p ( $S->vertices ) {
          ++$vertexPoints{ $p } if exists $vertexPoints{ $p };
        }
      }
    }
708
    # take additional info about number of
    # demands regarding this switch.
    my %nmbPoints = ();
    foreach ( keys %vertexPoints ) {
713   $nmbPoints{$_} = $C->in_degree($_)+$C->out_degree($_);
    }

    # sort by points;
    # maps by temp using array : ([sw_name,points],[sw_name,points],..)
718 my @sortPoints = sort { $$a[1] <=> $$b[1] or
                            $$a[2] <=> $$b[2] }
                     map { [$_,$vertexPoints{$_},$nmbPoints{$_}]}
                     keys %vertexPoints;

723 ## verbose printout DBG:
    print "POINTS: ";
    foreach ( @sortPoints) {print "$$_[0]($$_[1],$$_[2]) ";}
    print "\n";

728 # return, switches with least points are first in list.
    return map { $$_[0] } @sortPoints;

  }

733 1;

__END__
```

2.0.10 IcNet::LiP::Greedy::OptHelper

```
package IcNet::LiP::Greedy::OptHelper;

################################################################
  4 #
# POD documentation at end of file

BEGIN {
  require Exporter;
  9
  our @ISA = qw( Exporter );
  our @EXPORT = qw( &NetEfficiency &TotalWires);
  our %EXPORT_TAGS = ( );
  our @EXPORT_OK = qw( );
 14 our $VERSION = 0.1;

}

use strict;
 19 use POSIX;

sub NetEfficiency {
  my ($traffic, $maxW, $maxF ) = @_;
  my $maxBW = $maxW * $maxF;
 24 my $usedBW = 0;
  my $nmbLinks = 0;

  foreach my $linkBW ( values %$traffic ) {
    $nmbLinks += ceil($linkBW / $maxBW );
 29 $usedBW += $linkBW;
  }

  return ($usedBW / ( $maxBW*$nmbLinks*$nmbLinks));

 34 }

sub TotalWires {
   my ($traffic, $maxW, $maxF ) = @_;
  my $maxBW = $maxW * $maxF;
 39 my $usedBW = 0;
  my $wires = 0;

  foreach my $linkBW ( values %$traffic ) {
    $wires += $maxW * ceil($linkBW / $maxBW );
 44 }

  return $wires;
```

```
}

1;

__END__
```

2.0.11 IcNet::LiP::Greedy::THDiam

```
package IcNet::LiP::Greedy::THDiam;

use strict;
use Data::Dumper;
use IcNet::FastSearchAofA;
use IcNet::LiP::GraphOperations;
use IcNet::LiP::Greedy::OptHelper;
use vars ('@ISA');

@ISA = ("IcNet::FastSearchAofA");

#my $tries=0; # (for DBG counts 1 each time compare is called)

sub start {
  my $proto = shift;
  my $class = ref($proto) || $proto;
  my $self = $class->SUPER::start( @_ );
  $self->{traffic} = {};
  $self->{connections} = 0;
  $self->{totalDistance} = 0;
  $self->{totalDiameter} = 0;
  $self->{distances} = {};
  $self->{diameter} = {};
  bless( $self, $class );
  return $self;
}

# adds the hash that has distance values
sub setDistance {
  my $self = shift;
  my $d = shift;
# print Dumper( $d ); die; # DBG
  $self->{distances} = $d;
  $self->setDiameter(); # now, calc diameter..
}

# to easier retrive distances
sub distance {
  my $self = shift;
  my ($s,$e) = @_;
# print "$s $e\n";
# print Dumper( $self->{distances}->{$s}->{$e} ); die; # DBG
  return ${$self->{distances}}{$s}{$e};
}

# shuld be set after we've set the distance
sub setDiameter {
  my $self = shift;

  # find max length
  my $max = 0;
  foreach my $K ( values %{$self->{distances}} ) {
    foreach my $d ( values %$K ) {
      $max = $d if $d > $max;
    }
  }

# print $max,"\n"; die; # DBG

  foreach my $k ( keys %{$self->{distances}} ) {
    foreach my $d ( keys %{$self->{distances}{$k}} ) {
      ${$self->{diameter}}{$k}{$d} = $self->distance($k,$d) / $max;

    }
  }

}

sub diameter {
  my $self = shift;
  my ($s,$e) = @_;
  return ${$self->{diameter}}{$s}{$e};
}
```

```
    sub compare {
      my $self = shift;
77    my $net = shift;
      my $optSwitch = shift;
      my $linkWidth = shift;
      my $linkFreq = shift;
      my $distances = shift; #..between switches
82    my $newValues = { diameter=>0
                          };

    # my %Traffic = %{$self->{traffic}};
      $newValues->{diameter} = $self->{totalDiameter};
87
    # $newValues->{connections} = $self->{connections};
    # $newValues->{wires} = TotalWires(\%Traffic,$linkWidth,$linkFreq);
    # $newValues->{totalDistance} = $self->{totalDistance};

92    # calculate unused traffic
      # (unused = total capacity - used capacity )
    # my  $totalTraffic = 0;
    # foreach ( values %Traffic ) {
        # find total traffic
97  #     $totalTraffic += $_;
    # }
    # $newValues->{unused} = $newValues->{wires}*$linkFreq - $totalTraffic;

      #
102   # Do comparing
      #

      # previous values exists, then compare with new values
      my $presentValues = ${ $self->{valueResult}}[0];
107
      # does previous values exists ?
      unless( exists  $presentValues->{diameter} ) {
        print "no exists, $newValues->{diameter}\n ";
        return (1,$newValues);
112   }

      # DBG
      ## Keep attention when we pass a certain number of combinations.
      ## Waiting for something to happen is tedious. A print now and
117   ## then helps a lot.....
      # $tries++;
    # unless ( $tries % 50000 ) {
    #   my $total  =$self->totalCmb;
    #   print "tried $tries of ", $total," combinations\n";
122 #   print " Wires $newValues->{wires} \n";
    #   printf " time left: %.2f \n", ((time()-$Tstart)/$tries)*($total-$tries);
    # }


127 # print "W |$newValues->{wires}| $linkWidth $linkFreq ",keys %Traffic,"\n";

      # Compare diameter
      if ( $presentValues->{diameter} > $newValues->{diameter} ) {
    #   print "but smaller diameter $newValues->{diameter}\n";
132     return (1, $newValues);
      }
      elsif ( $presentValues->{diameter} == $newValues->{diameter} ) {
        return (0,undef);
      }
137   else {
        return (-1,undef);
      }

      # if we get her there is something fataly wrong with my code
142   print "Warning  : comparing failed ?\n";
      return (0);

    }

147 ########################################################

    sub add {
      my $self = shift;
      my $value = $self->getValueAtBaseIndex();
152
      # Calc. efficiensy etc...
    #   $$value[0]->printAll; # DBG
    #   print "$$value[1]","\n"; # DBG

157   for my $newNode ( $$value[1]->NewLinks ) {
        unless ( exists ${$self->{traffic}}{$newNode} ) {
```

```
        ${$self->{traffic}}{$newNode} = $$value[0]->Transmit;
        $self->{connections}++;
      }
162   else {
        ${$self->{traffic}}{$newNode} += $$value[0]->Transmit;
      }
    }

167 for my $candidate ( $$value[1]->Candidates ) {
      $self->{totalDistance} += $self->distance(
                                  $candidate->{NewVertex},
                                  $candidate->{Destination} );
      $self->{totalDiameter} += $self->diameter(
172                               $candidate->{NewVertex},
                                  $candidate->{Destination} );
    }


177 }

####

sub subtract {
182
  my $self = shift;
  my $value = $self->getValueAtBaseIndex();

  for my $newNode ( $$value[1]->NewLinks ) {
187
    ${$self->{traffic}}{$newNode} -= $$value[0]->Transmit;
    # if no traffic, delete
    if ( ${$self->{traffic}}{$newNode} <= 0 ) {
      delete ${$self->{traffic}}{$newNode};
192   $self->{connections}--;
    }

  }

197 for my $candidate ( $$value[1]->Candidates ) {
    $self->{totalDistance} -= $self->distance(
                                $candidate->{NewVertex},
                                $candidate->{Destination} );

202 $self->{totalDiameter} -= $self->diameter(
                                $candidate->{NewVertex},
                                $candidate->{Destination} );
  }
}
207
###############################################
# Overwrited method from IcNet::SearchAofA
sub result {
  my $self = shift;
212
  return map {
    my $return = [];
    my @combination = $self->combination( @$_ );

217   for my $i ( 0 .. $#$_ ) {
      my $localP = [];
      for my $L ( 0 .. $self->{offset}-1 ) {
        push @$localP, $self->localParameter($i,$L);
      }
222   push @$localP, $combination[ $i ];
      push @$return, $localP;
    }

    $return;
227
  } @{ $self->{cmpResult} };

}

232 1;

__END__
```

2.0.12 IcNet::LiP::Greedy::FSWireUnusedDist

```
package IcNet::LiP::Greedy::FSWireUnusedDist;

use strict;
use Data::Dumper;
```

```
5   use IcNet::FastSearchAofA;
    use IcNet::LiP::GraphOperations;
    use IcNet::LiP::Greedy::OptHelper;
    use vars ('@ISA');

10  @ISA = ("IcNet::FastSearchAofA");

    #my $tries=0; # (for DBG counts 1 each time compare is called)

    sub start {
15    my $proto = shift;
      my $class = ref($proto) || $proto;
      my $self = $class->SUPER::start( @_ );
      $self->{traffic} = {};
      $self->{connections} = 0;
20    $self->{totalDistance} = 0;
      $self->{distances} = {};
      bless( $self, $class );
      return $self;
    }
25
    # adds the hash that has distance values
    sub setDistance {
      my $self = shift;
      my $d = shift;
30  # print Dumper( $d ); die; # DBG
      $self->{distances} = $d;
    }

    # to easier retrive distances
35  sub distance {
      my $self = shift;
      my ($s,$e) = @_;
    # print "$s $e\n";
    # print Dumper( $self->{distances}->{$s}->{$e} ); die; # DBG
40    return ${$self->{distances}}{$s}{$e};
    }

    sub compare {
      my $self = shift;
45    my $net = shift;
      my $optSwitch = shift;
      my $linkWidth = shift;
      my $linkFreq = shift;
      my $distances = shift; #..between switches
50    my $newValues = { connections=>0,
                        totalLength=>0,
                        efficiency=>0,
                        wires => 0,
                        unused => 0   };
55
      my %Traffic = %{$self->{traffic}};

      $newValues->{connections} = $self->{connections};
      $newValues->{wires} = TotalWires(\%Traffic,$linkWidth,$linkFreq);
60    $newValues->{totalDistance} = $self->{totalDistance};

      # calculate unused traffic
      # (unused = total capacity - used capacity )
      my  $totalTraffic = 0;
65    foreach ( values %Traffic ) {
        # find total traffic
          $totalTraffic += $_;
      }
      $newValues->{unused} = $newValues->{wires}*$linkFreq - $totalTraffic;
70
      #
      # Do comparing
      #

75    # previous values exists, then compare with new values
      my $presentValues = ${ $self->{valueResult}}[0];

      # does previous values exists ?
      unless( exists  $presentValues->{wires} ) {
80      print "no exists, $newValues->{wires} ";
        print keys %Traffic,"\n";
        return (1,$newValues);
      }

85    # DBG
      ## Keep attention when we pass a certain number of combinations.
      ## Waiting for something to happen is tedious. A print now and
      ## then helps a lot.....
```

```
  # $tries++;
90 # unless ( $tries % 50000 ) {
#   my $total  =$self->totalCmb;
#   print "tried $tries of ", $total," combinations\n";
#   print " Wires $newValues->{wires} \n";
#   printf " time left: %.2f \n", ((time()-$Tstart)/$tries)*($total-$tries);
95 # }

  # print "W |$newValues->{wires}| $linkWidth $linkFreq ",keys %Traffic,"\n";

100  # Compare wires
  if ( $presentValues->{wires} > $newValues->{wires} ) {
    print "better fewer wires $newValues->{wires}\n";
    return (1, $newValues);
  }
105  elsif ( $presentValues->{wires} == $newValues->{wires} ) {
#   print "same value $newValues->{wires}";
    if ( $presentValues->{unused} > $newValues->{unused} )
    {
      print "but better usage, unused: $newValues->{unused}\n";
110     return (1,$newValues);
    }

    if (  $presentValues->{totalDistance} > $newValues->{totalDistance})
    {
115     print "but shorter $newValues->{totalDistance}\n";
      return (1,$newValues);
    }

    # else ..
120 #   print " else $newValues->{wires}",keys %Traffic," \n";
    return (0,undef);
  }
  else {
#   print " worse $newValues->{wires} \n";
125   return (-1,undef);
  }

  # if we get her there is something fataly wrong with my code
  print "Warning  : comparing failed ?\n";
130  return (0);

}

#######################################################
135
sub add {
  my $self = shift;
  my $value = $self->getValueAtBaseIndex();

140  # Calc. efficiensy etc...
#   $$value[0]->printAll; # DBG
#   print "$$value[1]","\n"; # DBG

  for my $newNode ( $$value[1]->NewLinks ) {
145   unless ( exists ${$self->{traffic}}{$newNode} ) {
      ${$self->{traffic}}{$newNode} = $$value[0]->Transmit;
      $self->{connections}++;
    }
    else {
150     ${$self->{traffic}}{$newNode} += $$value[0]->Transmit;
    }
  }

  for my $candidate ( $$value[1]->Candidates ) {
155   $self->{totalDistance} += $self->distance(
                                $candidate->{NewVertex},
                                $candidate->{Destination} );
  }

160
}

####

165 sub subtract {

  my $self = shift;
  my $value = $self->getValueAtBaseIndex();

170  for my $newNode ( $$value[1]->NewLinks ) {

    ${$self->{traffic}}{$newNode} -= $$value[0]->Transmit;
```

```
        # if no traffic , delete
        if ( ${$self->{traffic}}{$newNode} <= 0 ) {
175       delete ${$self->{traffic}}{$newNode};
          $self->{connections}--;
        }

      }
180
      for my $candidate ( $$value[1]->Candidates ) {
        $self->{totalDistance} -= $self->distance(
                                      $candidate->{NewVertex},
                                      $candidate->{Destination} );
185   }
    }

    ##############################################
    # Overwrited method from IcNet::SearchAofA
190 sub result {
      my $self = shift;

      return map {
        my $return = [];
195     my @combination = $self->combination( @$_ );

        for my $i ( 0 .. $#$_ ) {
          my $localP = [];
          for my $L ( 0 .. $self->{offset}-1 ) {
200         push @$localP, $self->localParameter($i,$L);
          }
          push @$localP, $combination[ $i ];
          push @$return, $localP;
        }
205
        $return;

      } @{ $self->{cmpResult} };

210 }

    1;

    __END__
```

2.0.13 IcNet::LiP::Greedy::OptimizePM

```
    package IcNet::LiP::Greedy::OptimizePM;

    #use strict;
    use IcNet::SearchAofA;
  5 use IcNet::LiP::GraphOperations;
    use IcNet::LiP::Greedy::OptHelper;

    @ISA = ("IcNet::SearchAofA");

 10 sub compare {
      my $self = shift;
      my $net = shift;
      my $optSwitch = shift;
      my $newValues = { edges=>0,
 15                     totalLength=>0,
                        efficiency=>0 };

      my @comb = $self->localCombination;
      my %Traffic = ();
 20   my $number_edges = 0;

      # Calc. efficiensy etc...
      foreach my $CM ( @comb ) {

 25 #   $$CM[0]->printAll; # DBG
    #   print "$$CM[1]","\n"; # DBG


        for my $c ( $$CM[1]->NewLinks ) {
 30       unless ( exists $Traffic{$c} ) {
            $Traffic{$c} = $$CM[0]->Transmit;
            $number_edges++;
          }
          else {
 35         $Traffic{$c} += $$CM[0]->Transmit;
          }
        }
      }
```

```
40    $newValues->{edges} = $number_edges;
      $newValues->{efficiency} = NetEfficiency(\%Traffic,1,1);

      #
      # Do comparing
45    #

      my $presentValues = ${ $self->{valueResult}}[0];

      # does previous values exists ?
50    return (1,$newValues) unless exists $presentValues->{edges};

      # minimum number of edges is more important
      # compare this first
      if ( $presentValues->{edges} > $newValues->{edges} ) {
55      print "better $number_edges\n";
        return (1, $newValues);
      }
      elsif ( $presentValues->{edges} == $newValues->{edges} ) {
        return (0,undef);
60    }
      else {
        return (-1,undef);
      }

65    # if we get her there is something wring with my code
      print "Warning  : comparing failed ?\n";
      return (0);

    }
70
    sub result {
      my $self = shift;

      return map {
75      my $return = [];
        my @combination = $self->combination( @$_ );

        for my $i ( 0 .. $#$_ ) {
          my $localP = [];
80        for my $L ( 0 .. $self->{offset}-1 ) {
            push @$localP, $self->localParameter($i,$L);
          }
          push @$localP, $combination[ $i ];
          push @$return, $localP;
85      }

        $return;

      } @{ $self->{cmpResult} };
90
    }

    1;
```

2.0.14 IcNet::LiP::MBG_matching

```
 1  package IcNet::LiP::MBG_matching;

    #
    # POD at eof
    #
 6
    BEGIN {
      use strict;
      use Exporter ();
    #  use IcNet::Port;
11  #  use IcNet::Resource;
    #  use IcNet::Link;
    #  use IcNet::LiP::Switch;
    #  use IcNet::LiP::Channel;
      use Graph;
16
      our @ISA         = qw(Exporter);
      our @EXPORT      = qw(MBG_solve MBG_solve_original);
      our %EXPORT_TAGS = ();
      our @EXPORT_OK   = qw( );
21
    }

    ###############################################
    # MBG_solve_original
```

```
26
     sub MBG_solve_original {
       my $MBG = shift;
       my @X = $MBG->source_vertices();
       my @Y = $MBG->sink_vertices();
31     my @edges;
       my ($startV, $endV); # temps
       my %mate;
       my %free;
       my %label;
36     my @A;
       my @Q; # Queue
       my $done = 0;
       my $found = 0;

41     foreach ( $MBG->vertices() ) {
         $mate{ $_ } = "";
       }

       while ( not $done ) {
46
         # INITIALIZE
         foreach ( @X ) { $free{ $_ } = "";  }
         @edges = $MBG->edges;
         while ( ($startV, $endV) = splice( @edges, 0, 2) ) {
51         if ( $mate{ $endV } eq "" ) {
             $free{ $startV } = $endV;
           }
           elsif ( $mate{ $endV } ne $startV ) {
             push ( @A, $startV, $mate{ $endV } );
56         }
         }
         # END INITIALIZE

         # Empty queue
61       @Q = ();

         # Add unmatched vertices to Q
         foreach ( @X ) {
           if ( $mate{ $_ } eq "" ) {
66           push @Q, $_;
             $label{ $_ } = "";
           }
         }
     #
71   # org code
     #
         $found = 0;
         while ( (not $found) && scalar( @Q ) ) {
           $startV = pop @Q;
76         if ( $free{ $startV } ne "" ) {
             _MBG_augment_original( $startV , \%mate, \%free, \%label );
             $found = 1;
           }
           else {
81           for ( my $i=0; $i < $#A; $i+=2 ) {
               if ( $A[ $i ] = $startV ) {
                 if ( $label{ $A[$i+1] } eq "" ) {
                   $label{ $A[$i+1] } = $startV;
                   push @Q, $A[ $i+1 ];
86               }
               }
             } # END for
           } # END if .. else ..

91         $done = 1 unless @Q;

         } # END while

       } # END outer while (not $done)
96
       # Remove twins
       foreach ( @Y ) {
         delete $mate{$_};
       }
101
       # Clean mate
       foreach ( keys %mate ) {
         delete $mate{ $_ } if $mate{ $_ } eq "";
       }
106
       print "Mate start\n";     # DBG
       _deb_hash( \%mate );      # DBG
       print "Mate end_____\n"; # DBG
```

```
111     return %mate;

      } # END sub

      sub _MBG_augment_original {
116     my $x = shift; # vertex
        my $mate = shift;
        my $free = shift;
        my $label = shift;

121     # AUGMENT
        if ( $$label{ $x } eq "" ) {
      #    print $$free{ $x },"\n"; exit(0); # DB # DBG
          $$mate{ $x } = $$free{ $x };
          $$mate{ $$free{ $x } } = $x;
126     }
        else {
          $$free{ $$label{ $x }} = $$mate{ $x };
          $$mate{ $x } = $$free{ $x };
          $$mate{ $$free{ $x } } = $x;
131       _MBG_augment( $$label{ $x }, $mate, $free, $label );
        }
      }

      ###############################################
136   # MBG_solve

      sub MBG_solve {
        my $MBG = shift;
        my @X = $MBG->source_vertices();
141     my @Y = $MBG->sink_vertices();
        my @edges = $MBG->edges();
        my ($startV, $endV); # temps
        my %mate;
        my %free;
146     my %label;
        my @A;
        my @Q; # Queue
        my $notFound = 1;

151   # print $MBG; #DBG
      # print @X; print "\n"; # DBG
      # print @Y; print "\n"; # DBG

        # No need to do anything if the Graph itself
156     # has no edges
      # my $notDone = $MBG->edges;
        my $notDone = 1;

        while ( $notDone ) {
161
          # INITIALIZE
          foreach ( @X ) { delete $free{ $_ };  }

          for (my $i=0; $i<$#edges; $i+=2 ) {
166   #       print "$i", $edges[$i]," ", $edges[$i+1],"\n"; # DBG
            if ( not defined $mate{ $edges[ $i+1 ] }  ) {
              $free{ $edges[ $i ] } = $edges[ $i+1 ];
            }
            elsif ( $mate{ $edges[ $i+1] } ne $edges[ $i ] ) {
171           push ( @A, $edges[$i], $mate{ $edges[$i+1] } );
            }
          }
          # END INITIALIZE

176   #    print "free:";_deb_hash( \%free ); # DBG
      #    print "mateS: "; _deb_hash( \%mate ); # DBG
      #    exit(0); # DBG

          # Empty queue
181       @Q = ();

          # Add unmatched vertices to Q
          foreach ( @X ) {
            unless ( defined $mate{ $_ } ) {
186           push @Q, $_;
              delete $label{ $_ };
            }
          }
      #    print "label: "; _deb_hash( \%label ); # DBG
191   #    print "mateS: "; _deb_hash( \%mate ); # DBG
      #    print "free:";    _deb_hash( \%free ); # DBG
```

```
      $notFound = 1;
#     print "start w $#Q $notDone\n"; # DBG
196   while ( $notFound && @Q ) {
        $startV = pop @Q;
#       print "VVVVV:",$startV,"A ";
        if ( defined $free{ $startV } ) {
          _MBG_augment( $startV , \%mate, \%free, \%label );
201       $notFound = 0;
        }
        else {
          for ( my $i=0; $i < $#A; $i+=2 ) {
            if ( $A[ $i ] eq $startV ) {
206           unless ( defined $label{ $A[$i+1] } ) {
                $label{ $A[$i+1] } = $startV;
                push @Q, $A[ $i+1 ];
              }
            }
211       } # END for
        } # END if .. else ..

        $notDone = 0 unless @Q;

216   } # END while

      $notDone = 0 unless @Q;
#     print "mateL: "; _deb_hash( \%mate); # DBG
    } # END outer while ($notDone)
221
    # delete uneccesary matching
    foreach ( @Y ) {
      delete $mate{$_};
    }
226
    return %mate;

} # END sub

231 sub _MBG_augment {
    my $x = shift; # vertex
    my $mate = shift;
    my $free = shift;
    my $label = shift;
236
    # AUGMENT
    if ( not defined $$label{ $x } ) {
      $$mate{ $x } = $$free{ $x };
      $$mate{ $$free{ $x } } = $x;
241 }
    else {
      $$free{ $$label{ $x }} = $$mate{ $x };
      $$mate{ $x } = $$free{ $x };
      $$mate{ $$free{ $x } } = $x;
246   _MBG_augment( $$label{ $x }, $mate, $free, $label );
    }
}

sub _deb_hash {
251 my $h=shift;
    foreach ( keys %$h ) {
      print "$_=>$$h{$_}  ";
    }
    print "\n";
256 }

1;

__END__
```

2.0.15 IcNet::Geometry

```
package IcNet::Geometry;

BEGIN {
  4 require Exporter;

    our @ISA = qw(Exporter);
    our @EXPORT = qw( &distance  &manhattan_distance &manhattan_intersection &
         range_check_tree &basic_tree_find &basic_tree_add &basic_tree_del &traverse
          &convex_hull_graham &point_in_polygon &rectilinear_area);
    our %EXPORT_TAGS = ( );
  9 our @EXPORT_OK = qw( );
    our $VERSION = 0.1;
```

```
  use constant epsilon => 1e-14;
}

use strict;

sub rectilinear_area {
  my $x =shift;
  my $y = shift;
  my $distance = shift;

  return ($x+$distance, $y,
          $x          , $y+$distance,
          $x-$distance, $y,
          $x          , $y-$distance);

}

sub manhattan_intersection {
  my @op; # The coordinates are transformed here as operations.

  while (@_) {
    my @line = splice @_, 0, 4;

    if ($line[1] == $line[3]) {     # Horizontal.
      push @op, [ @line, \&range_check_tree ];
    } else {                        # Vertical.
      # Swap if upside down.
      @line = @line[0, 3, 2, 1] if $line[1] > $line[3];

      push @op, [ @line[0, 1, 2, 1], \&basic_tree_add ];
      push @op, [ @line[0, 3, 2, 3], \&basic_tree_del ];
    }
  }

  my $x_tree; # The range check tree.
  # The x coordinate comparison routine.
  my $compare_x = sub { $_[0]->[0] <=> $_[1]->[0] };
  my @intersect; # The intersections.

  foreach my $op (sort { $a->[1] <=> $b->[1] ||
                         $a->[4] == \&range_check_tree ||
                         $a->[0] <=> $b->[0] }
                  @op) {
    if ($op->[4] == \&range_check_tree) {
      push @intersect, $op->[4]->( \$x_tree, $op, $compare_x );
    } else { # Add or delete.
      $op->[4]->( \$x_tree, $op, $compare_x );
    }
  }

  return @intersect;
}

sub range_check_tree {
  my ( $tree, $horizontal, $compare ) = @_;

  my @range          = ( );   # The return value.
  my $node           = $$tree;
  my $vertical_x     = $node->{val};
  my $horizontal_lo  = [ $horizontal->[ 0 ] ];
  my $horizontal_hi  = [ $horizontal->[ 1 ] ];

  return unless defined $$tree;

  push @range, range_check_tree( \$node->{left}, $horizontal, $compare )
      if defined $node->{left};

  push @range, $vertical_x->[ 0 ], $horizontal->[ 1 ]
      if $compare->( $horizontal_lo, $horizontal ) <= 0 &&
      $compare->( $horizontal_hi, $horizontal ) >= 0;

  push @range, range_check_tree( \$node->{right}, $horizontal,
                                 $compare )
      if defined $node->{right};

  return @range;
}

sub basic_tree_find {
  my ($tree_link, $target, $cmp) = @_;
  my $node;

  # $tree_link is the next pointer to be followed.
  # It will be undef if we reach the bottom of the tree.
```

```
      while ( $node = $$tree_link ) {
        local $^W = 0;      # no warnings, we expect undef values

 99     my $relation = ( defined $cmp
                         ? $cmp->( $target, $node->{val} )
                         : $target <=> $node->{val} );

        # If we found it, return the answer.
104     return ($tree_link, $node) if $relation == 0;

        # Nope - prepare to descend further - decide which way we go.
        $tree_link = $relation > 0 ? \$node->{left} : \$node->{right};
      }
109
      # We fell off the bottom, so the element isn't there, but we
      # tell caller where to create a new element (if desired).
      return ($tree_link, undef);
    }
114
    sub basic_tree_add {
      my ($tree_link, $target, $cmp) = @_;
      my $found;

119   ($tree_link, $found) = basic_tree_find( $tree_link, $target, $cmp );

      unless ($found) {
        $found = {
          left  => undef,
124       right => undef,
          val   => $target
            };
        $$tree_link = $found;
      }
129
      return $found;
    }

    sub basic_tree_del {
134   my ($tree_link, $target, $cmp) = @_;
      my $found;

      ($tree_link, $found) = basic_tree_find ( $tree_link, $target, $cmp );

139   return undef unless $found;

      # tree_link has to be made to point to any children of $found:
      #  if there are no children, make it null
      #  if there is only one child, it can just take the place
144   #    of $found
      #  But, if there are two children, they have to be merged somehow
      #    to fit in the one reference.
      #
      if ( ! defined $found->{left} ) {
149     $$tree_link = $found->{right};
      } elsif ( ! defined $found->{right} ) {
        $$tree_link = $found->{left};
      } else {
        MERGE_SOMEHOW( $tree_link, $found );
154   }

      return $found->{val};
    }

159 sub MERGE_SOMEHOW {
      my ($tree_link, $found) = @_;
      my $left_of_right = $found->{right};
      my $next_left;

164   $left_of_right = $next_left
          while $next_left = $left_of_right->{left};

      $left_of_right->{left} = $found->{left};

169   $$tree_link = $found->{right};
    }

    sub traverse {
      my $tree = shift or return;   # skip undef pointers
174   my $func = shift;

      traverse( $tree->{left}, $func );
      &$func( $tree );
      traverse( $tree->{right}, $func );
179 }
```

```
     sub manhattan_distance {
       my @p = @_;                    # The coordinates of the points.
       return abs( $p[0] - $p[2] ) + abs( $p[1] - $p[3] );
184  }

     sub distance {
       my @p = @_;                    # The coordinates of the points.
       my $d = @p / 2;                # The number of dimensions.
189
       # The case of two dimensions is optimized.
       return sqrt( ($_[0] - $_[2])**2 + ($_[1] - $_[3])**2 )
           if $d == 2;

194    my $S = 0;                     # The sum of the squares.
       my @p0 = splice @p, 0, $d;  # The starting point.

       for ( my $i = 0; $i < $d; $i++ ) {
         my $di = $p0[ $i ] - $p[ $i ];  # Difference...
199      $S += $di * $di;                # ...squared and summed.
       }

       return sqrt( $S );
     }
204
     sub convex_hull_graham {
       my ( @xy ) = @_;

       my $n = @xy / 2;
209    my @i = map { 2 * $_ } 0 .. ( $#xy / 2 ); # The even indices.
       my @x = map { $xy[ $_ ]     } @i;
       my @y = map { $xy[ $_ + 1 ] } @i;

       # First find the smallest y that has the smallest x.
214
       # $ymin is the smallest y so far, @xmini holds the indices
       # of the smallest y(s) so far, $xmini will the index of the
       # smallest x, $xmin the smallest x.
       my ($ymax, $ymin, $xmini, $xmin, $i );
219
     # for ( $ymin = $ymax = $y[ 0 ], $i = 1; $i < $n; $i++ ) { # BUG ?
       for ( $ymin = $ymax = $y[ 0 ], $xmini=0,
             $i = 1; $i < $n; $i++ ) {
         if ( $y[ $i ] + epsilon < $ymin ) {
224        $ymin  = $y[ $i ];
           $xmini = $i;
     #     @xmini = ( $i ); # ( BUG?: )
         } elsif ( abs( $y[ $i ] - $ymin ) < epsilon ) {
           $xmini = $i # Remember the index of the smallest x.
229            if not defined $xmini or $x[ $i ] < $xmini;
         }
       }

       $xmin  = $x[ $xmini ];
234    splice @x, $xmini, 1;          # Remove the minimum point.
       splice @y, $xmini, 1;

       my @a = map {  # Sort the points according to angle with that point.
         atan2( $y[ $_ ] - $ymin,
239             $x[ $_ ] - $xmin)
           } 0 .. $#x;

       # An unusual Schwartzian Transform.  This leaves us the sorted
       # indices so that we can apply the sort multiple times -- a permutation.
244
       my @j = map { $_->[ 0 ] }
       sort {      # Sort by the angles, then by x, then by y.
         return $a->[ 1 ] <=> $b->[ 1 ]              ||
             $x[ $a->[ 0 ] ] <=> $x[ $b->[ 0 ] ] ||
249          $y[ $a->[ 0 ] ] <=> $y[ $b->[ 0 ] ];
       }
       map { [ $_, $a[ $_ ] ] } 0 .. $#a;

       @x = @x[ @j ];           # Permute.
254    @y = @y[ @j ];
       @a = @a[ @j ];

       unshift @x, $xmin;       # Put back the minimum point.
       unshift @y, $ymin;
259    unshift @a, 0;

       my @h = ( 0, 1 );         # The hull.
       my $cw;
```

```
264    # Backtrack: while there are right turns or no turns, shrink the hull.
       for ( $i = 2; $i < $n; $i++ ) {
         while (
                clockwise( $x[ $h[ $#h - 1 ] ],
                           $y[ $h[ $#h - 1 ] ],
269                        $x[ $h[ $#h ] ],
                           $y[ $h[ $#h ] ],
                           $x[ $i ],
     # From Errata         $y[ $i ] ) < epsilon
                           $y[ $i ] ) > epsilon
274             and @h >= 2 ) {  # Keep two points in hull at all times.
           pop @h;
         }
         push @h, $i;  # Grow the hull.
       }
279
       # Interlace x's and y's of the hull back into one list, and return.

     # From Errata return map { ( $x[ $_ ], $y[ $_ ] ) } 0 .. $#h;
       return map { ( $x[ $_ ], $y[ $_ ] ) } @h;
284  }

     sub point_in_polygon {
         my ( $x, $y, @xy ) = @_;

289      my $n = @xy / 2;                         # Number of points in polygon.
         my @i = map { 2 * $_ } 0 .. (@xy/2);  # The even indices of @xy.
         my @x = map { $xy[ $_ ]      } @i;      # Even indices: x-coordinates.
         my @y = map { $xy[ $_ + 1 ] } @i;      # Odd indices: y-coordinates.

294      my ( $i, $j );                           # Indices.

         my $side = 0;                            # 0 = outside, 1 = inside.

         for ( $i = 0, $j = $n - 1 ; $i < $n; $j = $i++ ) {
299          if (
                (

                 # If the y is between the (y-) borders ...
                 ( ( $y[ $i ] <= $y ) && ( $y < $y[ $j ] ) ) ||
304              ( ( $y[ $j ] <= $y ) && ( $y < $y[ $i ] ) )
                )
                and
                # ...the (x,y) to infinity line crosses the edge
                # from the ith point to the jth point...
309             ($x
                 <
                 ( $x[ $j ] - $x[ $i ] ) *
                 ( $y - $y[ $i ] ) / ( $y[ $j ] - $y[ $i ] ) + $x[ $i ] )) {
              $side = not $side; # Jump the fence.
314      }
         }

         return $side ? 1 : 0;
     }
319
     sub clockwise {
         my ( $x0, $y0, $x1, $y1, $x2, $y2 ) = @_;
         return ( $x2 - $x0 ) * ( $y1 - $y0 ) - ( $x1 - $x0 ) * ( $y2 - $y0 );
     }
324
     1; # Fine

     ##########################################################
     #
329  # Start POD section
     #
```

2.0.16 IcNet::SearchAofA

```
     package IcNet::SearchAofA;

     ##########################################################
  4  # Recursively tries all combinations in an Array of Array.
     #
     # POD documentation at end of file.

     use strict;
  9
     sub start {
       my $invoce = shift;
       my $class = ref($invoce) || $invoce;
```

```
14      # Init
        my $self = {
          cmpResult   => [],
          valueResult => [],
          indexArr    => [],
19        offset      => 0,
          AofA        => []
        };

        bless($self, $class); # Halleluja !!
24
        $self->{offset} = shift if @_;

        # if other parameters, then start search
        if ( @_ ) {
29        $self->{AofA} = shift;
          $self->doSearch( @_ );
        }

        return $self;
34
      } # END sub new

      ##############################################################
      #
39    sub offset {
        my $self = shift;

        $self->{offset} = shift if @_;

44      return $self->{offset};
      }

      ##############################################################
      #
49    sub AofA {
        my $self = shift;

        $self->{AofA} = shift if @_;

54      return $self->{AofA};
      }

      sub localParameter {
        my $self = shift;
59      my $baseIndex = shift;
        my $dataIndex = @_?shift:-1;

        if ( $dataIndex+1 ) {
          return ${ $self->{AofA} }[$baseIndex][$dataIndex];
64      }
        else {
          return @{ $self->{AofA} }[$baseIndex];
        }

69      return ();

      }

      ##############################################################
74    #

      sub compare {
        my $self = shift;

79      # uncomment if U want to.... _debug_
      # print $self->combination,"\n";

        # retval = 0 -> store all combinations
        return 0;
84    }

      #############################################################
      #
      sub getValueResult {
89      my $self = shift;
        return @{ $self->{valueResult} };
      }

      sub haveValueResult {
94      my $self = shift;

        return exists ${ $self->{valueResult}}[0]?1:0;
```

```
}
##########################################################
#

sub combination {
  my $self = shift;

  # Use internal index array or an
  # index by input parameter.
  my @i = @_?@_:@{ $self->{indexArr} };

  #( The commented out code will return empty elements as
  # well).
  # return map { ${ $self->{AofA}}[$_][ $i[$_] ]} 0 .. $#i;

  return map
  {
    ${ $self->{AofA}}[$_][ $i[$_] ]?
    ${ $self->{AofA}}[$_][ $i[$_] ]:
    ();
  } 0 .. $#i;

}

##########################################################
#

sub localCombination {
  my $self = shift;

  # Use internal index array or an
  # index by input parameter.
  my @i = @_?@_:@{ $self->{indexArr} };

  return map
  {
    if ( ${ $self->{AofA}}[$_][ $i[$_] ] ) {
      my $tmp = [];
      # add local parameters
      for my $L ( 0 .. $self->{offset}-1 ) {
        push @$tmp, $self->localParameter($_, $L);
      }
      push @$tmp, ${ $self->{AofA}}[$_][ $i[$_] ];
      $tmp;
    }
    else {
      ();
    }
  } 0 .. $#i;

}

##########################################################
#

sub result {
  my $self = shift;

  return map {
   [$self->combination(@$_)];
  } @{ $self->{cmpResult} };
}

##########################################################
#

sub addCmpResult {
  my $self = shift;
  my $value = shift;

  push (@{$self->{valueResult}}, $value) if $value;
  push @{$self->{cmpResult}},[ @{ $self->{indexArr} } ] ;
}

##########################################################
#

sub newCmpResult {
  my $self = shift;
  # Deletes old values
  $self->{cmpResult} = [];
  $self->{valueResult} = [];
  # Add new values.
```

```
      $self->addCmpResult( @_ );
    }
184
    ############################################################
    #

    sub nextCombination{
189   my $self = shift;
      my $i = @_ ?
        shift:
        $#{ $self->{indexArr}};

194   if ( ${ $self->{indexArr}}[$i]  < $#{$self->{AofA}->[$i]} ) {
        ${ $self->{indexArr}}[$i]++;
        return 1;
      }

199   # Restart this index:

      ${ $self->{indexArr}}[$i] = $self->{offset};

      $i--;
204   return $i<0? 0 :$self->nextCombination( $i);

    }

    ############################################################
209 #

    sub doSearch {
      my $self       = shift;
      my @globalParams = @_;
214
      # Reset
      $self->{cmpResult} = [];

      # Init ''indexArr''
219   $self->{indexArr} = [ map { $self->{offset} } 0..$#{ $self->{AofA} }  ] ;
    # $#{ $self->{indexArr} } = $#{ $self->{AofA} };
      #print $#{ $self->{ indexArr } }, "\n"; # DBG

      # execute if ''indexArr'' has value, otherwise
224   # the input array is empty.
      if ( @{$self->{indexArr}} ) {
        do {

          my ($return,$value) = $self->compare( @globalParams);
229       # 0; The result of comparing is equal as the first one.
          # => store this one.
          if ( $return == 0) {
            $self->addCmpResult( $value );
          }
234       # positive value; The result is better than the previous.
          # => delete the old ones and store the new one
          elsif ( $return > 0 ) {
            $self->newCmpResult( $value );
          }
239       # Negative value; the result is worse than the previous.
          # => do nothing.

        } while ( $self->nextCombination() );
      } # END if
244
    }

    ############################################################
    #
249
    sub totalCmb {
      my $self = shift;
      my $offset = 0;
    # $offset = shift if @_;
254   my $combinations = 0;

      # Code below might look odd....
      # It's written to return a value even if one or more of
      # the arrays are empty. eg. should work for :
259   # $self->{AofA} = [ [], ['a','b']]; (return 2)
      # and
      # $self->{AofA} = []; (return 0)

      foreach my $a ( @{ $self->{AofA} } ) {
264     # test for empty array
        if ( @$a ) {
```

```
        # unless $combinations has a value then it's init value is
        # the size of the first array with some length.
        unless ( $combinations ) {
269       $combinations = ($#$a-$self->{offset})+1;
        }
        else {
          $combinations *= ($#$a-$self->{offset})+1;
        }
274     }
    }

    return $combinations;

279
}

1;

284 __END__
```

2.0.17 TestTimeSlots.pm

```
package TestTimeSlot;

##########################################################
#
5 # POD documentation at end of file


BEGIN {
  use strict;
10  require Exporter;

  our @ISA = qw(Exporter);
  our @EXPORT = qw( &initTSD1 &initTSD2 &initTSD3 &initTSD4 &initTSD5 &initTSD6 )
      ;
  our %EXPORT_TAGS = ( );
15  our @EXPORT_OK = qw( );
  our $VERSION = 0.1;

}

20 use strict;


sub initTSD1 {

25  return
  (
   IcNet::LiP::TimeSlotData->new( 0),
   IcNet::LiP::TimeSlotData->new( 2),
   IcNet::LiP::TimeSlotData->new( 2),
30   IcNet::LiP::TimeSlotData->new( 3)
  );


}
35
sub initTSD2 {
  return
  (
    IcNet::LiP::TimeSlotData->new( 2),
40    IcNet::LiP::TimeSlotData->new( 2),
    IcNet::LiP::TimeSlotData->new( 3),
    IcNet::LiP::TimeSlotData->new( 4)
  );

45
}


sub initTSD3 {
50  return
  (
    IcNet::LiP::TimeSlotData->new( 0),
    IcNet::LiP::TimeSlotData->new( 2)
  );
55
}

sub initTSD4 {
  return
60  (
```

```
      IcNet::LiP::TimeSlotData->new( 0),
      IcNet::LiP::TimeSlotData->new( 0),
      IcNet::LiP::TimeSlotData->new( 1),
      IcNet::LiP::TimeSlotData->new( 1)
65  );


  }

70  sub initTSD5 {
    return
    (
      IcNet::LiP::TimeSlotData->new( 1),
      IcNet::LiP::TimeSlotData->new( 2),
75    IcNet::LiP::TimeSlotData->new( 3),
      IcNet::LiP::TimeSlotData->new( 3)
    );


80  }


    sub initTSD6 {
      return
85    (
        IcNet::LiP::TimeSlotData->new(0),
        IcNet::LiP::TimeSlotData->new(0),
        IcNet::LiP::TimeSlotData->new(1),
        IcNet::LiP::TimeSlotData->new(1),
90      IcNet::LiP::TimeSlotData->new(6),
        IcNet::LiP::TimeSlotData->new(6)
      );


95  }


    1;

100 __END__

    =pod

    =head1 NAME
105
    =head1 SYNOPSIS

    =head1 DESCRIPTION

110 =head1 Subrutines

    =over 4

    =back
115
    =cut
```

2.0.18 TestObjects::ExploreNetTestGraphs

```
    package TestObjects::ExploreNetTestGraphs;

    ##########################################################
 4  # Test networks for ExploreNetPM.
    #
    # POD documentation at end of file

    BEGIN {
 9    use strict;
      require Exporter;

      our @ISA = qw(Exporter);
      our @EXPORT = qw( &PMgraph1 &T_shaped_pp &PMgraph3
14                      &PMgraph4_square &PMgraph_grid
                        &grid_3x3 mapDemands &exampleNet1
                        &channelsExampleNet1 &exampleNet2
                        &exampleNet3 &exampleThesis1);
      our %EXPORT_TAGS = ( );
19    our @EXPORT_OK = qw( );
      our $VERSION = 0.1;

    }

24  use strict;
```

```perl
use Graph;
#use DemandContainer;
use IcNet::LiP::Channel;
use IcNet::LiP::CommunicationGraph;
use IcNet::LiP::GraphOperations;

#sub mapDemands {
# my $C = shift; # Communication Graph
# my @channels = @_;

# foreach my $demand ( @channels ) {
#   foreach my $target ( $demand->target ) {
#     $C->addDemands($demand->From, $target, $demand);
#   }
# }

#}


############################################################

sub PMgraph1 {

  #
  # "T shaped", A to the rest, point to multipoint
  #
  #   B  C  D
  #
  #      A
  # (0,0)
  #

  my $G = new IcNet::LiP::CommunicationGraph;

  $G->add_vertices('A','B','C','D');

  vertexPosition($G,'A',300,50);
  vertexPosition($G,'B',100,250);
  vertexPosition($G,'C',300,250);
  vertexPosition($G,'D',500,250);

  my @Channels = (
    IcNet::LiP::Channel->new( From => 'A', target => ["B","C","D"], Transmit => 8
        e+06),
  );


  return ($G,@Channels);

}

############################################################

sub T_shaped_pp {

  #
  # "T shaped", point to point
  #
  #   B  C  D
  #
  #      A
  # (0,0)
  #

  my $G = new IcNet::LiP::CommunicationGraph;

  $G->add_vertices('A','B','C','D');

  vertexPosition($G,'A',300,50);
  vertexPosition($G,'B',100,250);
  vertexPosition($G,'C',300,250);
  vertexPosition($G,'D',500,250);

  my @Channels = (
    IcNet::LiP::Channel->new( From => 'A', target => ["B"],
                              Transmit => 8000000),
    IcNet::LiP::Channel->new( From => 'A', target => ["C"],
                              Transmit => 8000000),
    IcNet::LiP::Channel->new( From => 'A', target => ["D"],
                              Transmit => 8000000),
    IcNet::LiP::Channel->new( From => 'B', target => ["A"],
                              Transmit => 8000000),
    IcNet::LiP::Channel->new( From => 'B', target => ["C"],
                              Transmit => 8000000),
```

```
      IcNet::LiP::Channel->new( From => 'B', target => ["D"],
109                             Transmit => 8000000),
      IcNet::LiP::Channel->new( From => 'C', target => ["B"],
                                Transmit => 8000000),
      IcNet::LiP::Channel->new( From => 'C', target => ["A"],
                                Transmit => 8000000),
114   IcNet::LiP::Channel->new( From => 'C', target => ["D"],
                                Transmit => 8000000),
      IcNet::LiP::Channel->new( From => 'D', target => ["B"],
                                Transmit => 8000000),
      IcNet::LiP::Channel->new( From => 'D', target => ["C"],
119                             Transmit => 8000000),
      IcNet::LiP::Channel->new( From => 'D', target => ["A"],
                                Transmit => 8000000),
    );

124
    return ($G,@Channels);

  }

129
  ############################################################

  sub PMgraph3 {

134 #
    # "T shaped", point to multipoint
    #
    #   B  C  D
    #
139 #      A
    # (0,0)
    #

    my $G = new IcNet::LiP::CommunicationGraph;
144
    $G->add_vertices('A','B','C','D');

    vertexPosition($G,'A',300,50);
    vertexPosition($G,'B',100,250);
149 vertexPosition($G,'C',300,250);
    vertexPosition($G,'D',500,250);

    my @Channels = (
      IcNet::LiP::Channel->new( From => 'A',
154                             target => ["B","C","D"],
                                Transmit => 8000000),
      IcNet::LiP::Channel->new( From => 'B',
                                target => ["A","C","D"],
                                Transmit => 8000000),
159   IcNet::LiP::Channel->new( From => 'C',
                                target => ["B","A","D"],
                                Transmit => 8000000),
      IcNet::LiP::Channel->new( From => 'D',
                                target => ["B","C","A"],
164                             Transmit => 8000000)
    );


    return ($G,@Channels);
169
  }

  ############################################################

174 sub PMgraph4_square {

    #
    # "square shaped"
    #
179 #    D  C
    #
    #    A  B
    # (0,0)
    #
184
    my $G = new IcNet::LiP::CommunicationGraph;

    $G->add_vertices('A','B','C','D');

189 vertexPosition($G,'A',100,100);
    vertexPosition($G,'B',300,100);
    vertexPosition($G,'C',300,300);
```

```
  vertexPosition($G,'D',100,300);

194

  my @Channels = (
    IcNet::LiP::Channel->new( From => 'A', target => ["B"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'A', target => ["C"], Transmit => 100),
199 IcNet::LiP::Channel->new( From => 'A', target => ["D"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'B', target => ["A"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'B', target => ["C"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'B', target => ["D"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'C', target => ["B"], Transmit => 100),
204 IcNet::LiP::Channel->new( From => 'C', target => ["A"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'C', target => ["D"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'D', target => ["B"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'D', target => ["C"], Transmit => 100),
    IcNet::LiP::Channel->new( From => 'D', target => ["A"], Transmit => 100)
209 );


  return ($G,@Channels);

214 }

sub PMgraph_grid {
  my $d = shift; # Dimension
  my $spacing = 100;
219 my $transmit = 100;
  my $G = new IcNet::LiP::CommunicationGraph;

  for my $x ( 0..$d ) {
    for my $y (0..$d ) {
224   my $name = ($x*$spacing)."x".($y*$spacing);
      $G->add_vertex( $name );
      vertexPosition( $G, $name, $x * $spacing, $y * $spacing);
    }
  }
229
  my @Channels = ();

  foreach my $from ( $G->vertices ) {
    foreach my $target ( $G->vertices ) {
234   push @Channels, IcNet::LiP::Channel->new(
                      From => $from,
                      target => [$target],
                      Transmit => $transmit)
         if $from ne $target;
239 }}

  foreach my $c ( @Channels ) {
    foreach my $t ( $c->target ) {
      $G->addDemands($c->From, $t, $c);
244 }}

  return $G;

}
249
##############################################################

sub grid_3x3 {

254 #
  # "square shaped", "one-to-one" connections
  #
  #   G  H  I
  #
259 #   D  E  F
  #
  #   A  B  C
  # (0,0)
  #
264
  my $G = new IcNet::LiP::CommunicationGraph;

  $G->add_vertices('A','B','C','D','E','F','G','H','I');

269 vertexPosition($G,'A',100,100);
  vertexPosition($G,'B',300,100);
  vertexPosition($G,'C',500,100);
  vertexPosition($G,'D',100,300);
  vertexPosition($G,'E',300,300);
274 vertexPosition($G,'F',500,300);
  vertexPosition($G,'G',100,500);
```

```
      vertexPosition($G,'H',300,500);
      vertexPosition($G,'I',500,500);

279   my @Channels = (
        IcNet::LiP::Channel->new( From => 'A', target => ["B"], Transmit => 100, ID
            => 'A->B'),
        IcNet::LiP::Channel->new( From => 'A', target => ["C"], Transmit => 100, ID
            => 'A->C'),
        IcNet::LiP::Channel->new( From => 'A', target => ["D"], Transmit => 100, ID
            => 'A->D'),
        IcNet::LiP::Channel->new( From => 'A', target => ["E"], Transmit => 100, ID
            => 'A->E'),
284     IcNet::LiP::Channel->new( From => 'A', target => ["F"], Transmit => 100, ID
            => 'A->F'),
        IcNet::LiP::Channel->new( From => 'A', target => ["G"], Transmit => 100, ID
            => 'A->G'),
        IcNet::LiP::Channel->new( From => 'A', target => ["H"], Transmit => 100, ID
            => 'A->H'),
        IcNet::LiP::Channel->new( From => 'A', target => ["I"], Transmit => 100, ID
            => 'A->I'),

289     IcNet::LiP::Channel->new( From => 'B', target => ["A"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'B', target => ["C"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'B', target => ["D"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'B', target => ["E"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'B', target => ["F"], Transmit => 100),
294     IcNet::LiP::Channel->new( From => 'B', target => ["G"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'B', target => ["H"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'B', target => ["I"], Transmit => 100),

        IcNet::LiP::Channel->new( From => 'C', target => ["A"], Transmit => 100),
299     IcNet::LiP::Channel->new( From => 'C', target => ["B"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'C', target => ["D"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'C', target => ["E"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'C', target => ["F"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'C', target => ["G"], Transmit => 100),
304     IcNet::LiP::Channel->new( From => 'C', target => ["H"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'C', target => ["I"], Transmit => 100),

        IcNet::LiP::Channel->new( From => 'D', target => ["A"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'D', target => ["B"], Transmit => 100),
309     IcNet::LiP::Channel->new( From => 'D', target => ["C"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'D', target => ["E"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'D', target => ["F"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'D', target => ["G"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'D', target => ["H"], Transmit => 100),
314     IcNet::LiP::Channel->new( From => 'D', target => ["I"], Transmit => 100),

        IcNet::LiP::Channel->new( From => 'E', target => ["A"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'E', target => ["B"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'E', target => ["C"], Transmit => 100),
319     IcNet::LiP::Channel->new( From => 'E', target => ["D"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'E', target => ["F"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'E', target => ["G"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'E', target => ["H"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'E', target => ["I"], Transmit => 100),
324

        IcNet::LiP::Channel->new( From => 'F', target => ["A"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'F', target => ["B"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'F', target => ["C"], Transmit => 100),
329     IcNet::LiP::Channel->new( From => 'F', target => ["D"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'F', target => ["E"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'F', target => ["G"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'F', target => ["H"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'F', target => ["I"], Transmit => 100),
334
        IcNet::LiP::Channel->new( From => 'G', target => ["A"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'G', target => ["B"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'G', target => ["C"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'G', target => ["D"], Transmit => 100),
339     IcNet::LiP::Channel->new( From => 'G', target => ["E"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'G', target => ["F"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'G', target => ["H"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'G', target => ["I"], Transmit => 100),

344     IcNet::LiP::Channel->new( From => 'H', target => ["A"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'H', target => ["B"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'H', target => ["C"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'H', target => ["D"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'H', target => ["E"], Transmit => 100),
349     IcNet::LiP::Channel->new( From => 'H', target => ["F"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'H', target => ["G"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'H', target => ["I"], Transmit => 100),
```

```
        IcNet::LiP::Channel->new( From => 'I', target => ["A"], Transmit => 100),
354     IcNet::LiP::Channel->new( From => 'I', target => ["B"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'I', target => ["C"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'I', target => ["D"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'I', target => ["E"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'I', target => ["F"], Transmit => 100),
359     IcNet::LiP::Channel->new( From => 'I', target => ["G"], Transmit => 100),
        IcNet::LiP::Channel->new( From => 'I', target => ["H"], Transmit => 100)
      );

      return ($G,@Channels);
364
    }


    ##########################################################
369
    sub exampleNet1 {

      # grid
      #         P5
374   #     P3
      # P1
      #
      # P2     P4
      my $G = new Graph::Undirected;
379
      print "network:\n";
      print "         P5\n";
      print "     P3\n";
      print " P1 \n\n";
384   print " P2     P4\n\n";


      $G->add_vertices('P1','P2','P3','P4','P5');

389   vertexPosition($G,'P1', 150, 300);
      vertexPosition($G,'P2', 150, 100);
      vertexPosition($G,'P3', 350, 350);
      vertexPosition($G,'P4', 500, 100);
      vertexPosition($G,'P5', 600, 400);
394
      return $G;
    }

    sub channelsExampleNet1 {
399   # test channels for sub testGraph4
      return (
        IcNet::LiP::Channel->new( From => 'P1', target => ["P3"], Transmit => 10),
        IcNet::LiP::Channel->new( From => 'P2', target => ["P3","P4","P5"], Transmit
            => 5),
        IcNet::LiP::Channel->new( From => 'P3', target => ["P1"], Transmit => 2),
404     IcNet::LiP::Channel->new( From => 'P4', target => ["P5"], Transmit => 10),
        IcNet::LiP::Channel->new( From => 'P4', target => ["P5"], Transmit => 40)
      );
    }

409 ##########################################################

    sub exampleNet2 {

      # grid
414   #        P3
      # P2
      #
      #     P4
      # P1
419   my $G = new Graph::Undirected;

      $G->add_vertices('P2','P3','P1');

    #  vertexPosition($G,'P1', 100, 100);
424   vertexPosition($G,'P2', 100, 300);
      vertexPosition($G,'P3', 250, 350);
      vertexPosition($G,'P1', 300, 200);

      return $G,
429   (
        IcNet::LiP::Channel->new( From => 'P1', target => ["P2"], Transmit => 5),
        IcNet::LiP::Channel->new( From => 'P1', target => ["P2","P3"], Transmit => 5)
            ,
        IcNet::LiP::Channel->new( From => 'P3', target => ["P1"], Transmit => 5),
        IcNet::LiP::Channel->new( From => 'P2', target => ["P3"], Transmit => 5),
```

```
434   );
    }

    ###########################################################
439
    sub exampleNet3 {

      # grid
      #       P3
444   # P2
      #
      #     P4
      # P1
      my $G = new Graph::Undirected;
449
      $G->add_vertices('A','B','C','D','E');

      vertexPosition($G,'A', 200, 100);
      vertexPosition($G,'B', 650, 200);
454   vertexPosition($G,'C', 400, 200);
      vertexPosition($G,'D', 100, 200);
      vertexPosition($G,'E', 300, 300);

      return $G,
459   (
        IcNet::LiP::Channel->new( From => 'A', target => ["D"], Transmit => 25),
        IcNet::LiP::Channel->new( From => 'A', target => ["C"], Transmit => 15),
        IcNet::LiP::Channel->new( From => 'E', target => ["A"], Transmit => 25),
        IcNet::LiP::Channel->new( From => 'B', target => ["A"], Transmit => 25),
464   );
    }

    sub exampleThesis1 {

469   #
      #       P3
      # P2
      #
      #     P1
474   #
      my $G = new Graph::Undirected;

      $G->add_vertices('P1','P2','P3');

479   vertexPosition($G,'P2', 100, 300);
      vertexPosition($G,'P1', 500, 100);
      vertexPosition($G,'P3', 400, 400);

      return $G,
484   (
        IcNet::LiP::Channel->new( From => 'P1', target => ["P2"], Transmit => 4),
        IcNet::LiP::Channel->new( From => 'P1', target => ["P3"], Transmit => 4),
        IcNet::LiP::Channel->new( From => 'P1', target => ["P2","P3"],
                                  Transmit => 4),
489   );
    }


    1;
494
    __END__

    =pod

499 =head1 NAME

    PMTestGraphs.pm

    =head1 SYNOPSIS
504
      use PMTestGraphs;
      ($networkGraph, @channels) = PMgraph1;

    =head1 DESCRIPTION
509
    Various network to test ExploreNetPM package. Read source subrutines for how a
        test network is build.

    =cut
```