Grafsammenhæng i netværk

Anders Meng Hansen s971577

LYNGBY 2004
EKSAMENSPROJEKT
NR. 61/04

IMM

Trykt af IMM, DTU

Forord

Dette projekt er skrevet som afsluttende projekt for opnåelse af civilingeniørgraden fra Danmarks Tekniske Universitet. Opgaven er udført på Institut for Informatik og Matematisk Modellering (IMM) ved Operationsanalyse Sektionen. Den omhandler kantsammenhæng i grafer og heuristikker til at øge kantsammenhængen i eksisterende grafer.

Jeg vil gerne takke min vejleder Jens Clausen for god vejledning og støtte gennem projektforløbet. Desuden vil jeg gerne takke min familie for god støtte undervejs.

Lyngby 1. juni, 2004

Anders Meng Hansen

Abstract

This report investigates heuristics for increasing edge connectivity in undirected graphs. Three simple heuristics, one metaheuristic and one advanced heuristic are considered. Four of the heuristics are tested with randomly generated graphs. The implementation and testing are done with MatLab. Results are analyzed. It is found that the heuristic using minimum spanning trees are the best of the four heuristics. It is also found that the running time of all 4 heuristics is very dependent on the computational efficiency of finding the minimum cut in a graph.

Keywords Edge connectivity, Tabu search, Network design, minimum spanning tree, FSM.

Indhold

Figurer

Kapitel 1

Indledning

I forbindelse med design af f.eks. telenetværk og udbygning af disse, er det vigtigt at kunne vurdere hvor robust netværket er over for fejl på ledningerne. Denne robusthed eller sammenhæng er afgørende for hvor sårbart et netværk er for at forbindelser mellem to "abonnenter" bliver afbrudt.

Hvis sårbarheden er for stor kan man forsøge at forbedre sammenhængen, f.eks. når netværket skal udbygges. Hvis man ser på, hvor mange ledninger der skal "klippes over" for at afbryde forbindelsen mellem to abonnenter og gør dette for alle par af abonnenter i netværket kan man definere kantsammenhængen som det mindste antal kanter, der skal afbrydes for at afbryde en forbindelse. Tilsvarende kaldes det mindste antal punkter (telefoncentraler) i netværket der skal gå i stykker for at der ikke kan kommunikeres mellem to abonnenter for punktsammenhængen. Jeg bruger, hvis ikke andet er nævnt, sammenhæng om kantsammenhængen.

Når man har fundet sammenhængen i et netværk kan man vurdere hvor sårbart det er over for fejl på linier. For at gøre netværket mere robust over for fejl på linierne, kan man have ønske om at udbygge netværket så sammenhængen i dette bliver større. Hvis man kender prisen på at tilføje de enkelte linier der ikke allerede er med i netværket kan man være interesseret i at finde den billigste løsning for at opnå en bestemt sammenhæng. Hvis prisen for at tilføje kanter er forskellig for forskellige kanter kaldes problemet "weighted-k-edge connectivity augmentation problem"(w-k-eca). Hvis prisen på at tilføje kanter er den samme for alle kanter kaldes det "unweighted k-edge augmentation connectivity problem"(uw-k-eca). I det følgende betragtes kun w-k-eca problemet.

1.1 Problemformulering

Der ønskes udført en eksperimentel analyse af graf-sammenhængsproblemet for at belyse, hvilke metoder der er bedst egnede til at løse w-k-eca i praksis.

Jeg har implementeret og undersøgt:

- En af de kendte avancerede huristikker til at løse w-k-eca.
- Metaheuristikker til at løse w-k-eca.
- Nogle heuristikker jeg selv har udviklet til at løse w-k-eca.

Heuristikkerne testes og sammenlignes med henblik på at finde hvilke heuristikker der giver de bedste resultater og hvilke af heuristikkerne der er hurtigst til at løse w-k-eca.

1.2 Afgrænsning og konventioner

Jeg har valgt kun at arbejde med w-k-eca. Priserne knyttet til grafens forskellige kanter er tilfældigt generede heltal mellem 1 og 100. Priserne afspejler således ikke geografiske forhold. Da dette projekt kun beskæftiger sig med w-k-eca vil det være nok at teste på denne måde for at afgøre hvilke heuristikker der finder de bedste løsninger.

Der findes algoritmer til at løse sammenhængsproblemt til optimalitet. De er i praksis uanvendelige, da de er så beregningstunge at selv et mindre problem ikke vil kunne løses inden for rimelig tid [1]. Jeg har derfor valgt kun at betragte heuristikker i dette projekt. Jeg arbejder kun med ikke-orienterede grafer. Hvis andet ikke er nævnt bruger jeg udtrykket graf om en ikke orienteret graf.

Kapitel 2

Teori

En graf er givet ved en mængde punkter og en mængde kanter. Punkterne kan eksempelvis repræsentere telefoncentraler og kanterne linier mellem centralerne. Når man skal arbejde med eksempelvis et telenetværk er det praktisk at tænke på det som en graf med kanter og punkter. Dette muliggør at man kan bruge kendte algoritmer fra grafteori og giver mulighed for at bruge programmer til at arbejde med netværket.

Jeg vil her beskrive nogle vigtige begreber som bliver brugt i det følgende. En mere udførlig beskrivelse af alle de begreber der anvendes kan f.eks. læses i [3] eller i [16]. Selve notationen følger [1]. Notationen er ikke den almindelige kendte som i eksempelvis [3]. Jeg har valgt at bruge notationen fra [1] da det gør foklaringen af de implementerede heuristikker lettere at relatere til artikelarbejderne.

2.1 Grundlæggende begreber

Det forudsættesat læseren er bekendt de grundlægende begreber i grafteori. De kan eksempelvis læses i [1] eller i [16].

En ikke orienteret graf G består af en mængde af punkter V(G), og en mængde af kanter E(G), hvor hver kant forbinder to punkter [3].

En delgraf kaldes en vej når dens punkter kan navngives p_1, p_2,...p_m så p_i og p_j er forbundet med en kant, |j-i|=1. En vej indeholder igen multiple kanter [3].

En vigtig egenskab ved en graf er er den såkaldte kantsammenhæng (undertiden blot betegnet sammenhæng). Kantsammenhængen mellem to punkter u og v er antallet af kantdisjunkte veje mellem u og v. Sammenhængen i en graf G' er den mindste sammenhæng mellem noget punktpar. Denne kaldes λ.

Kantsammenhængsproblemet kan defineres som:

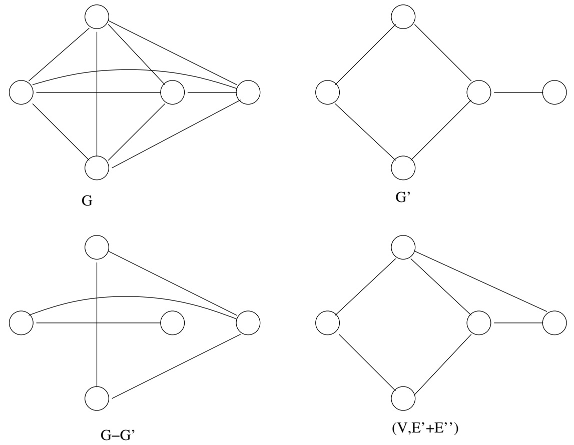
Find et antal kanter med laveste samlet pris der skal tilføjes en graf for at få kantsammenhængen til at vokse til den kantsammenhæng, man ønsker.

Jeg har i det følgende brugt notationen fra [1] for at gører sammenligningen mellem de udviklede heuristikker og heuristikkerne fra [1] nemmere. Den komplette graf K_n med n punkter, kaldes G(V,E). Den har |V|=n punkter og |E|=$\frac{1}{2}$n*(n-1) kanter. Graf G'(V,E') består af et antal kanter m og n punkter. E' $\subseteq$ E og punkterne er de samme som i G. Grafen G"(V,E-E') er komplementærgrafen til G'. Den indeholder alle de kanter der ikke er med i den graf vi ønsker at øge sammenhængen i. Det vil sige at man kan kalde G" "løsningsrummet"da det er fra den der skal findes en eller flere kanter som tilføjes til G'. På figur 2.1 er vist et eksempel på de forskellige grafer.

Målet kan nu mere detaljeret beskrives som at øge sammenhængen i grafen G' med δ, så sammenhæng efter at der er tilføjet en eller flere kanter er $\lambda + \delta$.

De kanter der tilføjes til G' har alle en pris. Prisen på en kant e, er 0 hvis e $\in$ E', det vil sige den allerede er i G'. Prisen er større end 0 hvis e $\in$ E-E'. Der er defor muligt at beregne prisen på at øge sammenhængen i G' med δ da prisen på kanterne der tilføjes er kendt. Som nævnt gælder det om at minimere prisen samtidig med at sammenhængen i G' bliver $\lambda + \delta$.

Som tidligere nævnt er udtømmende søgning efter den bedste løsning ikke mulig selv i en relativt lille graf. Et eksempel på kompleksiteten er: G' har som nævnt m kanter. G" har derfor $\frac{1}{2}n(n-1)-m$ kanter. Hvis man ønsker at undersøge alle muligheder for at tilføje kanter til grafen, for at finde den billigste pris for at øge sammenhængen med δ, og man først undersøger alle muligheder for at tilføje 1 kant til grafen er der derfor $\frac{1}{2}n(n-1)-m$ kanter at vælge imellem og dermed det samme antal muligheder. Hvis der skal tilføjes 2 kanter er der$(\frac{1}{2}n(n-1)-m)(\frac{1}{2}n(n-1)-(m-1))$ muligheder. Det ses



Figur 2.1: Illustration af den komplette graf G, start grafen G' med sammenhæng 1, de mulige kanter E-E' vi kan tilføje til G' for at øge sammenhængen, og (V,E'+E") som er den oprindelige graf og en ny kant som efter tilføjelsen af denne har sammenhæng 2.

at der hurtigt bliver et meget stort antal muligheder der skal undersøges selvom der er ret få punkter i grafen.

Hvis δ er større end 1 er det ikke sikkert at den optimale løsning kan findes ved at øge sammenhængsgraden med 1 ad gangen ([10] side 699.). Dette er interessant når man ønsker at øge sammenhængen i en graf med mere end én. I FSM heuristikken som beskrives senere øges sammenhængen med én ad gangen.

I FSM heuristikke som beskrives senere bruges bla. andet en kaktus. Derfor vil jeg her kort give en definitionen af en kaktus. En kaktus er en sammenhængende graf, hvor hver blok er en cykel eller en kant. En blok af en graf G', er en sammenhængende delgraf uden hængsler.

Til FSM heuristikken som beskrives senere er en strukturgraf nødvendig. Den fremstilles ud fra den graf man skal forbedre sammenhængen i, og repræsenterer alle mindste snit i den graf den er fremstillet ud fra. Strukturgrafen er en kaktus hvor hver kant har værdien λ hvis den ikke er med i en cykel, og $\lambda/2$ hvis den er med i en cykel. Jeg vil i forbindelse med FSM algoritmen komme mere ind på strukturgrafer.

2.2 Grafsammenhæng - metoder

Der findes flere måder at finde sammenhængen i et netværk på. Den jeg kort vil beskrive her er baseret på strømme i netværk. Den arbejder på orienterede netværk. Det er derfor nødvendigt at omforme grafen til et orienteret netværk inden man bruger f.eks. Edmonds-Karps algoritme. Dette er dog ikke svært idet hver kant i den ikke orienterede graf erstattes med to kanter i den orienterede graf - en hver vej.

For at finde kantsammenhængen mellem to punkter i en graf bruges Max-flow min-cut sætningen:

For ethvert netværk er værdien af en maksimale strømning lig kapaciteten af et minimalt snit [3].

Hvis kapaciteten af hver kant sættes til 1 kan man beregne kantsammenhængen mellem to punkter u og v ved at finde det maksimale flow mellem dem. Den udgave af maxflow algoritmen jeg har brugt er Edmonds-Karps algoritme [3] side 107. Den har en kompleksitet $O(m^2 * n)$. Edmonds-Karps algoritme finder et maksimalt flow fra et punkt til et andet. Man kan ud fra dette maksimale flow finde det mindste snit i grafen mellem disse to punkter. Hvis man ønsker at finde det mindste snit i hele grafen kan man finde det maksimale flow mellem alle punktpar. Ud fra det mindste af de maksimale flow kan det minimale snit i hele grafen findes. Det vil i princippet kræve at man kører Edmonds-karps algoritme n(n-1) gange. Det kan dog indses at det kun er nødvendigt at køre den fra et punkt til alle andre ([15] side 3) altså kun n-1 gange. Køretiden for at finde det mindste snit i hele grafen er derfor $O(m^2n^2)$.

Der findes en del andre metoder til at bestemme sammenhængen i en graf. Nogle af disse er beskrevte i [15]. De hurtigste algoritmer til at finde sammenhængen i en graf der beskrives i [15] har kørertiden på O(mn). Denne køretid er noget hurtigere end at bruge Edmonds-karps algortime som beskrevet ovenfor. Disse algoritmer er dog også mere komplicerede end Edmonds-Karps algoritme. Jeg har brugt Edmonds-Karps da den er let at implementere og arbejde med.

2.3 Kriterier til sammenligning af heuristikker

Når de forskellige heuristikker skal sammenlignes er der flere kriterier der skal overvejes.

- Køretiden.
- Kvaliteten/prisen på løsningen der findes.
- Kompleksiteten af heuristikken med hensyn til implementering.

Det er svært at måle køretiden eksakt da der ofte foregår en del andre processer på computeren mens heuristikken kører, som også tager CPU tid. Jeg vil dog alligevel sammenligne køretiden og bruge et gennemsnit af flere kørsler. Da implementeringen af heuristikkerne er lavet i MatLab er det muligt at bruge MatLab funktionen "profile"der viser køretider for de forskellige kald Matlab laver. Den giver køretid for de forskellige funktioner og filer og antallet af gange de bliver kaldt. Jeg har brugt den under udviklingsarbejdet med de forskellige programmer for at forbedre køretiden.

Jeg vil hvor det er muligt forsøge at give nogle teoretiske køretider for programmerne.

Til at vurdere kvaliteten (prisen) af den fundne løsning kan man sammenligne den med den optimale løsning for det givne problem eller den bedste fundne løsning hvis det er uoverkommeligt at finde den optimale løsning til problemet. Jeg har dog ikke nogle eksempler, hvor jeg kender den optimale løsning.

Jeg har derfor brugt en øvre og en nedre grænse for at vurdere kvaliteten af løsningen. Idéen med grænser er at have et mål for hvor god og hvor dårlig en optimal løsning kan være. Den optimale løsning til et problem har altid en pris der er større end eller lig med den nedre grænse og mindre end eller lig med den øvre grænse. I de fleste tilfælde vil den optimale løsning hverken være lig den øvre eller nedre grænse, men have en værdi et sted midt i mellem.

Den nedre grænse er et mål for hvor god den bedste løsning kan være. Det er ikke sikkert at der findes nogen brugbar løsning der lige så billig som den nedre grænse. Omvendt er en løsning optimal hvis hvis prisen på denne er lig med den nedre grænse. En løsning kan godt være optimal selvom den er dyrere end den nedre grænse.

Den øvre grænse er et mål for hvor "dårlig"den optimale løsning kan være. Den optimale løsning til et bestemt problem kan aldrig være dårligere end den øvre grænse.

For at finde en nedre grænse findes alle punkter med valens mindre end $\lambda+\delta$. Som nævnt skal alle punkter have valens større end eller lig $\lambda + \delta$ for at sammenhængen i grafen er $\lambda + \delta$. I det optimale tilfælde skal valensen i disse punkter kun hæves til $\lambda + \delta$ for at sammenhængen i hele grafen bliver $\lambda + \delta$. Hvis man antager at kanterne der skal tilføjes alle går mellem to punkter med valens mindre end $\lambda + \delta$ findes antallet af kanter der behøves på følgende måde:

Først identificeres alle punkter med valens mindre end $\lambda + \delta$. Disse punkter kaldes k. Antallet af kanter der behøves er:

(($\lambda + \delta$)-k)/2

rundert op til nærmeste heltal. Man skal herefter udvælge dette antal kanter med den laveste pris blandt alle kanter.

Det er bestemt ikke sikkert at man ved at udvælge de billigste kanter på ovennævnte måde får en mulig løsning men hvis den er mulig er den optimal. I tilfældet hvor der ikke er nogen punkter der har valens mindre end $\lambda + \delta$ bruger jeg den billigste kant som nedre grænse da der altid skal tilføjes minmum en kant for at øge sammenhængen i grafen.

Den øvre grænse for hvor dårlig den bedste løsning der findes er, findes ved hjælp af δ minimum udspændende træer af kanterne i G”. Hvis δ er større end én gøres det ved først at finde et minimum udspændende træ i G” og derefter fjerne kanterne i dette udspændende træ fra G”.

Med kanterne fjernet fra G” findes igen et nyt minimalt udspændende træ i G”. Igen slettes kanterne i det ny udspændende træ fra G”. Dette gøres δ gange så der er δ udspændende træ. Den nedre grænse er så summen af priserne af kanterne i de δ minimum udspændende træer. Det kan forekomme at det ikke er muligt at lave et minimalt udspændene træ i G”. I dette tilfælde bruger jeg δ udspændende skove. Man bliver så nødt til at undersøge om disse øger sammenhængen med δ. Hvis de udspændende skove ikke øger sammenhængen med δ findes endnu en udspændende skov. Igen undersøges om sammenhængen så er øget til $\delta + \lambda$. hvis ikke findes igen en udspændende skov og så fremdeleles. Når de udspændende skove og grafen tilsammen har sammenhængen er $\delta + \lambda$, findes den øvre grænse summen af priserne på kanterne i de udspændende skove.

Kompleksiteten af de forskellige metoder jeg har arbejdet er svær at kvantificeres. Jeg vil give mit bud på hvor meget arbejde der er forbundet med at implementere og bruge dem.

For at kunne vurdere de forskellige heuristikker har jeg under kørslerne opsamlet kørertiden og kvaliteten af løsningerne heuristikkerne finder. Derudover er informationer om graferne der arbejdes på også interessante når heuristikkerne skal vurderes. Disse informationer er: den øvere grænse, nedre grænse, sammenhængen i grafen der arbejdes på, antallet af punkter og antallet af kanter. I tilfælde hvor der er andre informationer jeg finder der kan være interessant vil jeg angive disse.

Kapitel 3

Implementerede heuristikker

Der er forskellige strategier for at opnå gode løsninger til w-k-eca. Jeg vil her kort forklare idéen i de 5 heuristikker jeg har implementeret og grunden til jeg har valgt dem.

Jeg har implementeret en udgave at metahuristiken Tabu search. Den er generelt anderkendt til at løse en bred vifte af optimerings problemer. Jeg har modificeret den for at undersøge om den også give gode resultater på w-k-eca problemet.

En kendt heuristik til at løse w-k-eca problemet er FSM heuristikken. Strategien her er at omforme problemet til et mindre vanskeligt problem. Strukturgrafen er mere simpel end den originale graf og indeholder stadigvæk alle minimale snit i G'. FSM algoritmen er den mest komplicerede af de heuristikker jeg har arbejdet med. FSM er beskrevet i kapitel 5 for sig selv, da den er en meget omfattende heuristik.

Den første af de heuristiker jeg selv har udviklet arbejder med minimalt udspændende træer. Ved at tilføje δ minimalt udspændende træer fra G" til G' og herefter undersøge om nogle af kanterne i disse træer kan fjernes uden at sammenhængen falder, opnås en løsning.

Den anden bruger minimalt udspændende træer mellem de punkter i G' der har mindst valens. Den tilføjer et minimalt udspændende træ mellem de punkter der har lavest valens indtil sammenhængen i grafen bliver $\delta+\lambda$. Som i den første heuristik jeg har udviklet, undersøges igen, om nogle af de tilføjede kanter er unødvendige for at opnå den ønskede sammenhæng i grafen.

Den sidste er en udgave af den grådige algoritme. Jeg tilføjer den billigst mulige kant til hvert punkt og undersøger om sammenhængen er $\delta+\lambda$. Dette fortages til sammenhængen er $\delta+\lambda$. Herefter undersøger jeg om nogle af de tilføjede kanter er overflødige som i de to ovenstående heuristikker.

Under beskrivelsen af hver heuristik har jeg vist pseudokoden for dem.

3.1 Tabu search

Idéen i tabu search er at bevæge sig rundt fra løsning til løsning. For at undgå at komme tilbage til en løsning der har været besøgt fornylig tilføjes nyligt besøgte løsninger til en tabuliste.

Tabu search har jeg implementeret i følgende udgave. Der er to muligheder for at finde en initiel løsning. Man kan bruge en løsning fra en af de andre heuristikker eller man kan finde en initiel løsning på følgende måde:

En initielle løsning F, findes ved at tilføje en del af kanterne i G" (de billigste) til G'. Herefter undersøges kanterne fra G" tilføjet til G' én efter én startende med den dyreste for at se om de kan fjernes uden at den ønskede sammenhæng ødelægges.

Det er værd at bemærke at man ikke kan være sikker på at den initielle løsning som beskrevet oven for faktisk eksisterer med de kanter man har valgt. Hvis den ikke findes må man bruge en større del af kanterne fra G" og lede igen. Ovenstående metode er ikke særlig effektiv. Jeg har derfor brugt løsninger fra de andre heuristikker som initiel løsning. Kun hvis man ikke har andre muligheder for at finde en løsning bør ovenstående metode bruges.

Når den initielle løsning F er fundet foretages iterationerne på følgende måde:

Tilføj et på forhånd valgt antal kanter, der ikke er i F og undersøg om nogle af de andre kanter i løsningen kan fjernes uden at sammenhængen formindskes. Dette giver en ny løsning. Det tillades ikke at fjerne de tilføjede kanter igen når kanterne undersøges. De tilføjede kanterne tilføjes en tabuliste og kan ikke tilføjes i de næste iterationer. Hvis den nye fundne løsning er bedre end den hidtil bedst fundne løsning opdateres den bedst fundne løsning.

Jeg har brugt sliding window som stop kriterie. Dvs at heuristikken stopper hvis den bedste løsning ikke er blevet opdateret i et vist antal iterationer.

Længden af tabulisten og antallet af kanter der tilføjes er de vigtigste parametre. Jeg har i resultat afsnittet undersøgt længden af tabu listen og antallet af kanter der skiftes.

Pseudo kode for tabu search

Input: En initiel løsning , en graf G',en prismatrix pm og længden af tabulisten k og størrelsen af slindingwindow.

Output: Et antal kanter der opfylder at de tilsammen med G' har sammenhængen $\delta+\lambda$.

1.Laveste-pris = pris(initiel løsning)

2. Sæt løsning = initiel løsning, antal-kørsler=0.

3. Find en muligløsning ved at tage løsning og tilføje k tilfældigt valgte kanter i E(G")\ { løsning } til denne.

4. Sorter kanterne i muligløsning efter pris.

5. Undersøg om nogle af kanterne i mulig løsning kan fjernes uden at sammenhængen falder. Start med den dyreste. De nyeste tilføjede kanter må ikke fjernes.

6. løsning=muligløsning

7. Hvis (pris(løsning) < laveste-pris):

- antal-kørsler=0.
- Laveste-pris=pris(nyløsning).
- Løsning=nyløsning.
- Opdater tabuliste med den fundne løsning.
- Gå til 3.

8. Hvis (antal-kørsler < slidingwindow)

- antal-kørsler = antal-kørsler+1.

- Opdater tabuliste med den fundne løsning.
- Gå til 3.

9. Hvis antal-kørsler = slidingwindow, output = løsning.

En mere udførlig beskrivelse af tabusearch generelt kan f.eks læses i [12] eller [13].

3.2 Heuristik baseret på minimum udspændende træer

Der findes en del specifikke heuristikker til at løse graf sammenhængsproblemet der benytter minimum udspændende træer.

Jeg har valgt at implementere en udgave der først finder et minimum udspændende træ i G". Kanterne i det minimalt udspændende træ slettes fra G". Et nyt minimalt udspændende træ findes igen i G" og så fremdeles. Dette gøres δ gange. De udspændende træer tilføjes til G'. De tilføjede kanter undersøges én for én startende med den dyreste. Hvis en kant kan fjernes uden at sammenhængen bliver mindre end $\lambda+\delta$ fjernes den.

Man kan ikke være sikker på at der findes δ udspændende træer i G". Hvis der ikke findes et udspændende træ i G" bruges istedet en slags grådig algoritme. Den billigste kant i G" for hvert punkt tilføjes til G'. Disse kanter fjernes fra G". Indtil sammenhængen er $\lambda+\delta$ findes den billigste kant i G" for hvert punkt som tilføjes til G' og kanterne slettes fra G". Herefter kan kanterne der er tilføjet undersøges én efter én som ovenfor for at se om de har nogen betydning for om sammenhængen er $\delta+\lambda$.

Ulempen ved denne heuristik er at den er beregningstung, da det at afgøre om en kant er med i det minimale snit er ret beregningstungt. Jeg bruger max flow, som har køretiden O(m^2n) (n-1) gange. Kruskals algoritme som jeg bruger til at finde minimale træer med, har køretiden O(m(log(m))).

Pseudokode for heuristik baseret på minimum udspændende træer

---

Input: En graf G' og en prismatrix pm.

Output: Et antalkanter der opfylder at de tilføjet til G' giver en sammenhæng på δ+λ.

1. Bestem sammenhængen i grafen ved hjælp af eksempelvis Edmonds-Karps algoritme. Denne kaldes λ.

2. Find et minimalt udspændende træ i G" (kanter der ikke er med i grafen). Fjern kanterne i dette fra G". Gentag dette δ gange.

3. Tilføj kanterne i de fundne minimalt udspændende træer til G'. Sorter kanterne i de minimalt udspændende træer efter prisen.

4. Undersøg om nogen af de tilføjede kanter er overflødige på følgende måde. Startende med den dyreste kant undersøges om kanten kan fjernes uden at dette ændrer den opnåede sammenhængsgrad (grafen med de tilføjede kanter har sammenhængsgraden $\lambda + \delta$). Hvis dette er muligt fjernes kanten. Alle kanter testes på denne måde én efter én.

3.3 Heuristik baseret på punkter med valens mindre end $\lambda + \delta$

Som beskrevet i teoriafsnittet skal der gælde at valensen af alle punkter skal være mindst λ+δ for at sammenhængen i grafen er λ+δ. Det er derfor oplagt forsøge at tilføje kanter til de punkter der har for lav valens.

Selve heuristikken tilføjer kanter til det eller de punkter der har mindst valens. Dette gøres ved at lave et minimum udspændende træ af kanterne i G" mellem de punkter der har mindste valens. Disse kanter tilføjes grafen G' og fjernes fra G". Et nyt minimum udspændende træ findes på de punkter der har lavest valens efter at de første kanter er tilføjet. Dette fortsættes indtil sammenhængen i G' er λ+δ. Hvis det ikke er muligt at lave et minimum udspændende træ mellem punkterne med mindst valens vælges den billigste kant i G" til hvert punkt med mindst valens. Dette gør sig eksempelvis gældende hvis der netop er ét punkt der har mindst valens og derfor kun en kant der skal tilføjes. Hvis der er to punkter med laveste valens kan det også forekomme, at kanten mellem to punkter allerede er i grafen G'.

De tilføjede kanter undersøges nu én efter én startende med den dyreste for at se om de kan fjernes fra grafen uden at sammenhængen falder under $\lambda+\delta$. De kanter der ikke ødelægger sammenhængen i grafen fjernes.

Pseudo kode for heuristik baseret på punkter med valens mindre end $\lambda + \delta$

Input: En graf G', en pris matrix pm.

Output: Et antal kanter der tilføjet til G' giver en sammenhæng på $\lambda+\delta$.

1. Så længe sammenhængen er mindre end $\lambda+\delta$ gør følgende:

   a Find punkter med lavest valens i G'.
   b Lav et minimalt udspændende træ på disse punkter i G". Hvis det ikke er muligt at lave et minmalt udspændende findes den billigste kant til hvert punkt i G".
   c Fjern de fundne kanter fra G" og tilføj dem til G'.

2. Sorter de fundne kanter efter pris.

3. Undersøg om nogen af de fundne kanter er overflødige. Dette gøres ved at fjerne en kant og undersøge om sammenhængen i grafen bliver mindre end $\lambda+\delta$. Hvis sammenhængen er mindre end $\lambda+\delta$ tilføjes kanten igen. Hvis den ikke er mindre end $\lambda+\delta$ tilføjer man ikke denne kant. Start med den dyreste kant.

3.4 Den grådige heuristik

Den version af den grådige algoritme jeg har brugt er implementeret på følgende måde. For alle punkter i grafen G' tilføjes den billigste kant fra G" til hvert punkt. Kanterne der tilføjes til G' slettes fra G". Det undersøges om sammenhængen i G' sammen med de tilføjede kanter er $\lambda+\delta$. Hvis sammenhængen er mindre end $\lambda+\delta$ tilføjes igen den billigste kant fra G" til hvert punkt og så fremdeles.

Når sammenhængen i grafen er $\lambda+\delta$ undersøges de tilføjede kanter én for én startende med den dyreste kant. Hvis en kant kan fjernes uden at sammenhængen bliver mindre end $\lambda+\delta$ fjernes den.

Det kan forekomme at der ikke er nogen kanter i G" til et punkt. I dette tilfælde tilføjes der ikke nogen kant til dette punkt.

Pseudo kode for den grådige heuristik

Input: En graf G' og en pris matrix pm.

Output: En mængde af kanter der tilføjet til G' hæver sammenhængen med δ.

1. Tilføj den billigste kant i G" til hvert punkt i grafen G' og slet dem fra G". Undersøg om grafen tilføjet disse kanter har sammenhængen $\lambda+\delta$. Hvis sammenhængen er mindre end $\lambda + \delta$ gentages processen, og de ny fundne kanter tilføjes til grafen.

2. Undersøg om nogen af de fundne kanter kan fjernes på følgende måde: Sorter kanterne der skal tilføjes grafen efter prisen. Fjern den dyreste kant og undersøg om sammenhængen falder under $\lambda + \delta$. Hvis den ikke falder fjernes kanten ellers tilføjes den igen. På ovenstående måde undersøges alle tilføjede kanter.

Kapitel 4

FSM

Dette kapitel indeholder en beskrivelse af FSM heuristikken fra [1]. Jeg vil først give heuristikken fra artiklen som den står beskrevet og herefter vise vha. et eksempel hvad der rent faktisk sker. Herefter vil jeg komme med mine kommentarer til artiklen og heuristikken.

Ideen i FSM heuristikken er at forbinde ethvert blad i en kaktus med et af de andre blade i kaktusen vha. af en maksimum pris matching. For at kunne køre FSM heuristikken må man først generere en sådan kaktus. Den skal repræsentere alle minimale snit i den graf der arbejdes ud fra. Den betegnes i artiklen strukturgrafen. Der er ikke angivet nogen metode til at generere strukturgrafen i [1] men blot en henvisning til de 2 artikler [5] og [6]. De indeholder dog ikke nogen algoritme til at fremstille en strukturgraf. Jeg har foruden de 2 nævnte artikler ledt efter en algoritme til at fremstille en strukturgraf på internettet og i Danmarks Tekniske Videncenters databaser. Dette gav nogle artikler der omhandler området med ikke nogen egentlig algoritme. De fundne artikler er [6], [7], [8], [9], [10] og [11].

4.1 FSM heuristikken fra artiklen

Pseudo kode for FSM heuristikken

Input: En komplet ikke orienteret sammenhængende graf G_s, en pris funktion C':$E_s \rightarrow Z^+$, en backpointer b:$E_s \rightarrow E$ og en delgraf G'_s.

Output: Et sæt af kanter E''_s som er en løsning til G'_s. Det vil sige at hvis E''_s tilføjes til G'_s fremkommer en graf med kantsammenhæng større end λ.

Inden algoritmen kan køres skal der fremstilles en strukturgraf svarende til G_s'. Punkt 0 i algoritmen kunne derfor være:

0. Lav en strukturgraf svarende til G'_s. I [1] benyttes betegnelsen G'_s både for delgrafen og for den tilhørende strukturgraf. I resten af denne beskrivelse betegner G'_s strukturgrafen.

1.Initialisering af variablene: $G_0 = G_s, G'_0 = G'_s, c'_0 = c', i = 0$.

2. Kontruer et træ G'_b fra G'_i ved hjælp af af funktionen REMAKE (se nedenfor) og lav en komplet graf G_b ud fra G_i.

3. $H_i = G_b$, $H'_i = G'_b$

4. Find distancen $d_i : E(H_i)$ vha. funktionen DIST (Se nedenfor). For hver cykel C i G'_s tilføjes et dummy punkt w_c, og $d_i(w_c, v) = \infty$ for alle v tilhørende C.

5. Beregn $d_i : E(H_i) \in Z^+ \cup \infty$ ved at finde den korteste vej P(u,v) af H_i for alle par af punkter (u,v) med valens 1 i H'_i. For kanten E_p af P(u,v) sættes $d_i(u,v) = d_i(E_p)$.

6. Kontruer en komplet delgraf S_i ud fra alle punkter i H'_i med valens = 1, E(S_i) $\subseteq$ E(H_i).

7. For alle priser $d'_i(u,v)$ med $(u,v) \in E(S_i)$ sættes $d''_i(u,v) = MAX + 1 - d'_i(u,v))$ hvor MAX er den største pris værdi for nogen kant i S_i.

Find en maksimum pris matching $M_i \subseteq E(S_i)$ ud fra d''_i.

8. For hver kant (u,v) $\in M_i$ indsæt de tilsvarende kanter $b'_i(b''_i(u,v))$ i E''_i, og slet evt. multiple kanter.

9.Hvis E''_i ikke er en løsning til G'_i sæt da $\Gamma = E''_i$ og kør i-v igennem:

- i $G'_{i+1} = G'_i$
- ii Sålænge $\Gamma \neq \emptyset$ gør følgende
  - (a) Vælg $(u,v) \in \Gamma$, og sæt $\Gamma = \Gamma - \{(u,v)\}$
  - (b) G'_{i+1} = den graf der fremkommer ved at sammentrække (u,v)-augmenting set A(u,v) af G'_{i+1}.

iii Lad G_{i+1} være en mindre komplet graf på V(G'_{i+1}) ud fra G_i pga. sammentrækningen.

iv Definer en ny pris funktion $c_{i+1} : E(G_{i+1}) \rightarrow Z^+ \cup \infty$ vha. $c'_{i+1}(u,v) = min\{c'_i(x,y) \mid (x,y) \in E(G_i), x \in S_u, y \in S_v)\}$, hvor S_w er $G'_i + \Gamma$ sammentrukket til w $\in V(G_{i+1})$ og der refereres til kanten (x,y) med en backpointer $b_{i+1}(u,v) = (x,y)$ hvor $b_{i+1} : E(G_{i+1}) \rightarrow E(G_i)$.

v i=i+1, gå til trin 2.

10. Hvis E''_i er en løsning til G'_i gør følgende:

(i) $E''_s = E''_0$

(ii) hvis $i \geq 1$ så for hver j (j=1...i) sæt $E''_s = E''_s \cup b_1(...(b_j(E''_j))...)$.

4.1.1 REMAKE

Pseudo kode for REMAKE

---

1. Hvis λ er ulige $G'_b = G'_s$, stop.

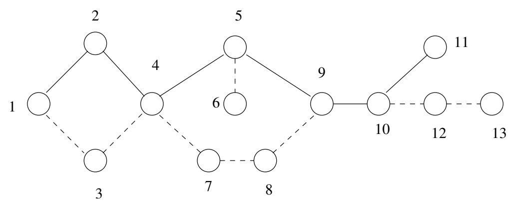
2. Slet multiple kanter i G'_s. Find alle cycler i G'_s vha. en dybde først søgning.

3. For hver cykel C i grafen tilføjes et dummy punkt w_c. Tilføj en kant fra w_c til alle punkter i cyklen og slet de originale kanter i cyklen.

G'_b er den graf der fremkommer heraf. For alle dummy punkter w_c og alle andre punkter i G'_b c'$(w_c, u) = \infty$.

I REMAKE er det nødvendigt at finde alle cykler i en strukturgraf for at kunne omforme strukturgrafen til et træ. Dette er relativt besværligt. Jeg vil her give den metode jeg har brugt til det.

I artiklen bruges dybde først søgning (DFS) uden dog at beskrive hvordan DFS skal bruges til at identificere alle cykler i strukturgrafen. Jeg har med udgangspunkt i DFS søgt i grafen på følgende måde for at finde alle cykler i strukturgrafen. Når DFS søger i grafen skal man blive ved med at tilføje kanter til vejen indtil man ikke kan længere og derefter undersøge om den



Figur 4.1: DFS søgning i strukturgrafen. Punkterne på vejen er 1,2,4,5,9,10,11.

vej man har opbygget indeholder nogen cykler. Et eksempel på en vej er vist på figur 4.1.

For at undersøge om en vej indeholder nogen cykel må man undersøge om et punkt er med i vejen mere end en gang. Det ses at vejen på figuren ikke indeholder nogen cykel da ingen punkter er med mere end en gang. Da det ikke er muligt at komme videre fra punkt 11 må man gå baglæns til punkt 10. Dette gøres ved at slette kanten mellem punkt 10 og 11 fra vejen. Fra punkt 10 kan man bevæge sig videre til punkt 12 og 13. Vejen bliver så 1,2,4,5,9,10,12,13.

Da ingen punkter er med mere end en gang i denne vej er der heller ikke nogen cykel på denne vej. Man må derfor søge videre. Når man skal finde den nye vej fra punkt 13 bliver man nødt til at gå baglæns ved at slette den sidste kant i vejen. Da man ikke kan komme videre fra punkt 12 slettes endnu en kant så vejen stopper i punkt 10. Da kanten mellem punkt 10 og 11 har været brugt må den ikke bruges igen, og man er derfor nødt til at gå baglæns igen. Nå man først søger så dybt man kan i grafen og herefter går baglæns ved at slette en kant fra vejen og ser om man kan komme nogen steder fra dette punkt kan en kant der har været brugt og blevet slettet aldrig komme med i vejen igen. Dette udnyttes i programmeringen af DFS, jeg bruger en boolsk variabel der fortæller om en kant har været brugt eller kan benyttes.

Den næste vej der findes er 1,2,4,5,9,8,7,4,3,1. Da den starter og slutter i samme punkt er der mindst en cykel. Der kan også være flere cykler i vejen som der er i dette eksempel. Det er derfor nødvendigt at undersøge om der er andre punkter der er med i vejen mere end en gang. Punkt 4 er med to gange så punkterne 4,5,9,8,7,4 er en cykel. De tilbageværende punkter på vejen er 1,2,4,1 og de er også en cykel. Der kunne være flere cykler så man må blive ved med at undersøge om nogen punkter er med mere end en gang. for at finde alle cykler. Da alle kanter nu er brugt er der ikke flere cykler i grafen og DFS slutter.

4.1.2 DIST

Pseudo kode for DIST

---

1. For hvert par af punkter u og v, beregn a(u,v) som antallet af kanter på vejen mellem u og v i G'_b. Find nabo punkterne s(u,v) til v og s(v,u) til u på denne vej. d(u,v)=c'(u,v) og b'(u,v)=(u,v).

2. Sorter kanterne (u,v) i $E_b - E'_b$ vha. eksempelvis buketsort, i aftagende orden efter a(u,v). For hver kant (u,v) i $E_b - E'_b$ gør følgende i den sorterede rækkefølge:

d(u,s(u,v))=d(u,v) og b'(u,s(u,v))=b'(u,v) hvis d(u,v) $<$ d(u,s(u,v))

d(s(v,u),v)=d(u,v) og b'(s(v,u),v)=b'(u,v) hvis d(u,v) $<$ d(s(v,u),v).

4.2 Eksempel på FSM heuristikken

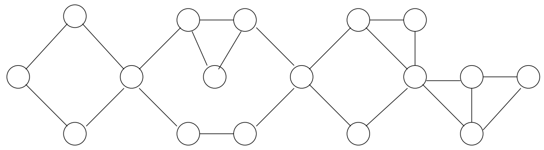
Her gives et eksempel på hvordan algoritmen ændrer på grafen undervejs i de forskellige trin.

Udgangs punktet for FSM er som nævnt en strukturgraf, dvs. en graf der indeholder alle minimale snit i den graf, hvis sammenhæng skal forbedres.

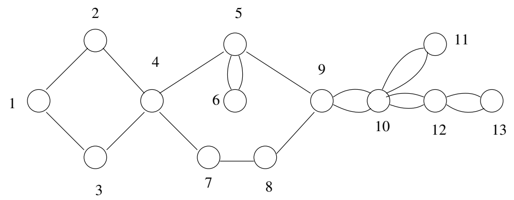
I artiklen vises et eksempel på en graf 4.2 og dens strukturgraf 4.3. Jeg benytter den som eksempel.

Undervejs i algoritmen foretages mange omformninger af grafen. Overalt hvor det er nødvendigt bruger forfatterne backpointere, så man kan finde tilbage til den orginale graf. Backpointere er ikke nærmere beskrevet i artiklen.

Strukturgrafen er en kaktus dvs. en ikke orienteret sammenhængende graf i hvilken ethvert par af cykler højest har ét punkt tilfælles. Som det ses af

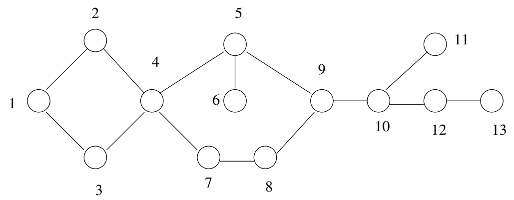


Figur 4.2: Grafen som man ønsker at hæve kantsammenhængen i.



Figur 4.3: Den tilhørende strukturgraf.

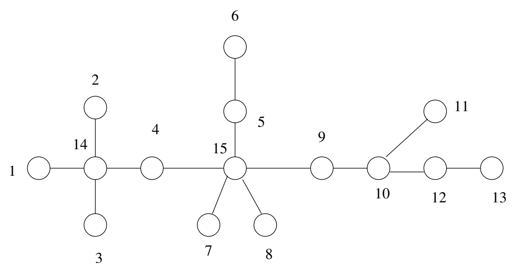
4.3 er en del af kanterne i eksemplet multiple. Dette er ikke interessant når det drejer sig om at forbedre kantsammenhængen. Når de multiple kanter er fjernet ser strukturgrafen ud som på figur 4.4.



Figur 4.4: Strukturgraf efter de multiple kanter er slettet

Nu køres REMAKE som finder alle cykler i grafen vha. depth-first search. En nærmere beskrivelse af hvordan cyclerne findes gives i afsnit REMAKE ovenfor. For hver cykel indføres et dummy punkt samt nye kanter fra alle punkter i cyklen til dummy punket. De originale kanter i cyklen fjernes. Der fremkommer herved et træ som på figur 4.5. Det ses at de punkter der havde valens 2 i strukturgrafen nu har valens 1 efter REMAKE er kørt og derfor også skal have en kant tilføjet, når sammenhængen skal forbedres.

Antallet af kanter er det samme som før REMAKE blev kørt men der er indført to nye punkter. Hvis stukturgrafen er et træ i forvejen foretages



Figur 4.5: Omformet strukturgraf efter REMAKE er kørt.

ingen omformning.

Det ses på figur 4.5 at for at forbedre kantsammenhængen er det nødvendigt at tilføje nye kanter til de punkter der valens 1, og til de punkter i cyklerne der har valens 2. Det er punkterne 1,2,3,6,7,8,11 og 13.

Da det gælder om at minimere omkostningerne af at tilføje kanter, bruges en pris funktion. Denne angiver omkostningerne ved at tilføje en ny kant til grafen. Den er nemmest repræsenteret som en matrix n x n. I eksemplet er der 13 punkter. De anvendte prisværdier er heltal genereret tilfældigt mellem 1-100.

For at forbedre sammenhængen i grafen er det nødvendigt at indføre nogle ekstra kanter i grafen. De skal forbinde bladene 1,2,3,6,7,8, 11 og 13 i grafen. Her benyttes en komplet delgraf G'_b på alle punkter med valens 1 som beskrivet i pkt. 6 i algoritmen. I pkt 7 findes en maksimum pris matching. Herved tilføjer man kanter med de laveste omkostninger der er større end 0 (0 svarer til de eksisterende kanter i strukturgrafen). Evt flere kanter kan tilføjes samtidig hvis der er flere med samme pris.

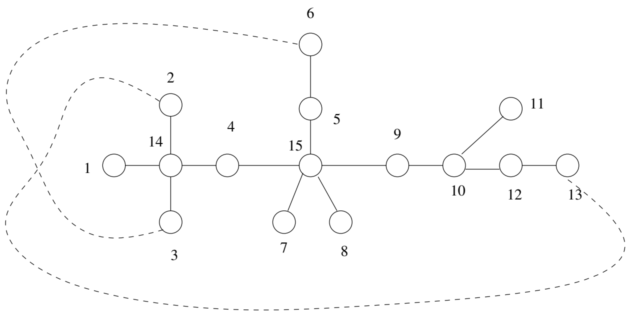
I eksemplet finder algoritmen at en kant fra punkt 2 til punkt 13 og fra punkt 3 til 6 skal tilføjes. De har begge pris værdien 17. Grafen kommer da til at se således ud:

Der mangler dog stadigvæk kanter til punkterne 1, 7, 8 og 11 for at øge sammenhængen i grafen.

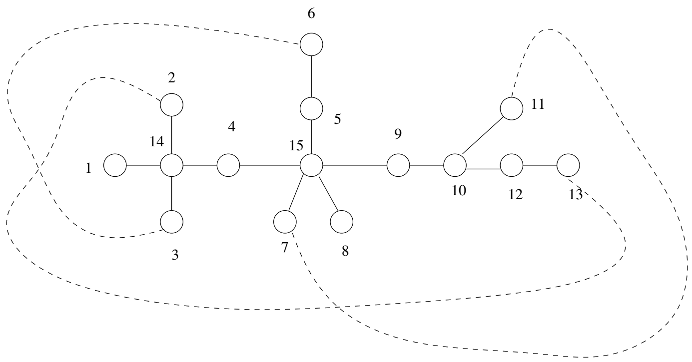
Heuristikken finder herefter ved brug af pris matricen at den billigste kant der kan tilføjes er fra 7 til 11 med en pris på 30.

C_m	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	15	17	36	82	79	99	67	51	82	20
2	0	0	86	0	83	26	78	36	80	38	80	57	17
3	0	86	0	0	22	17	95	44	67	62	35	39	75
4	15	0	0	0	0	17	0	40	28	10	40	28	55
5	17	83	22	0	0	0	52	92	0	8	66	95	24
6	36	26	17	17	0	0	43	51	17	58	52	58	51
7	82	78	95	0	52	43	0	0	66	41	30	10	60
8	79	36	44	40	92	51	0	0	0	61	31	82	32
9	99	80	67	28	0	17	66	0	0	0	65	4	89
10	67	38	62	10	8	58	41	61	0	0	0	0	31
11	51	80	35	40	66	52	30	31	65	0	0	19	97
12	82	57	39	28	95	58	10	82	4	0	19	0	0
13	20	17	75	55	24	51	60	32	89	31	97	0	0

Tabel 4.1: Pris matrix. Der er 0 de steder hvor de originale kanter er. Prismatricen er symmetrisk om diagonalen.

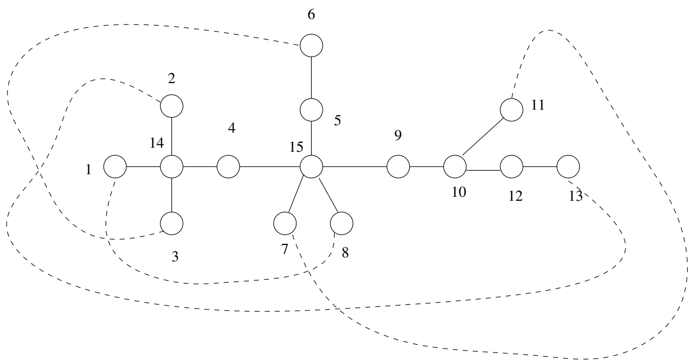


Figur 4.6: Strukturgraf med kanterne 2 til 13 og 3 til 6 tilføjet med stiplet linie.



Figur 4.7: Strukturgraf med kanterne 2 til 13, 3 til 6 og 7 til 11 tilføjet med stiplet linie.

Der mangler nu én kant for at kant sammenhængen er øget med en nemlig en kant fra 1 til 8 som har prisen 79. Den samlede omkostning ved at tilføje kanterne bliver 143.



Figur 4.8: Strukturgraf med kanterne 2 til 13, 3 til 6, 7 til 11 og 1 til 8 tilføjet med stiplet linie.

Jeg har ikke omtalt funktionen DIST i eksemplet ovenfor fordi jeg har fundet ud af, at den ikke er nødvendig !. Grunden til at DIST ikke er nødvendig er at [1] ikke beskriver hvordan man kommer tilbage til den oprindelige graf når man har fundet de kanter der skal tilføjes til strukturgrafen for at øge sammenhængen i strukturgrafen. For at finde de kanter der skal tilføjes den oprindelige graf skal man have en backpointer eller en anden metode til at komme tilbage til den oprindelige graf. DIST er nødvendig når de fundne kanter skal "oversættes"og tilføjes til den oprindelige graf. Denne del af algoritmen er ikke beskrevet i [1] da den kun arbejder med strukturgrafen. Jeg fandt dog først ud af dette efter at jeg havde programmeret DIST. Man kan på eksemplet på DIST nedenfor se at det kun er priserne på punkter inde i træet og ikke i punkter med valens en der ændres. Da det kun er punkter med valens én der skal have en kant tilføjet ændrer DIST ikke på prisen på de kanter der kan tilføjes til strukturgrafen.

For fuldstændighedens skyld er her beskrevet hvordan DIST ændrer prismatricen.

Hvis man f.eks. ønsker at indføre en ny kant fra punkt 6 til 13 koster dette en pris på 51. Hvis man i stedet lod kanten gå til punkt 12 som er nabo punkt til 13 vil det koste 58 altså en højere pris. Da antallet af kanter i grafen på vejen fra punkt 6 til punkt 13 er større end til punkt 12 (6 mod 5), er det med henblik på at forbedre af sammenhængen i grafen ligeså godt at lade kanten gå til punkt 13. Da det er billigere er det at foretrække. Prisen ved at tilføje en kant fra punkt 6 til 12 bør derfor være den samme som ved at tilføje en kant fra 6 til 13.

På samme måde undersøges om det er dyrere at tilføje en kant fra punkt fra 5 til 13 end fra nabopunktet 6 til 13. Dette er ikke tilfældet så der foretages ikke nogen ændringer i pris matricen.

For alle punktpar i grafen udensøges på denne måde de to nabopunkter der ligger på vejen mellem punkterne i punktparret. Hvis det er dyrere at lade en evt. ny kant gå til et sådant nabopunkt nedsættes prisen så den bliver den samme uanset om den går til det betragtede punkt eller dets nabo.

Den modificererde pris matrix ser sådan ud:

$C(u,v)$	1	2	3	4	5	6	7	8	9	10	11	12	13
1	0	0	0	15	17	36	82	79	20	20	51	20	20
2	0	0	86	0	26	26	78	36	17	17	80	17	17
3	0	86	0	0	17	17	95	44	35	35	35	39	75
4	15	0	0	0	0	17	0	36	10	10	36	17	17
5	17	26	17	0	0	0	43	51	0	8	52	24	24
6	36	26	17	17	0	0	43	51	17	51	52	51	51
7	82	78	95	0	53	43	0	0	10	10	30	10	60
8	79	36	44	36	51	51	0	0	0	31	31	32	32
9	20	17	35	10	0	17	10	0	0	0	30	4	17
10	20	17	35	10	8	51	10	31	0	0	0	0	17
11	51	80	35	35	52	52	30	31	30	0	0	19	97
12	20	17	39	17	24	51	10	32	4	0	19	0	0
13	20	17	75	17	24	51	60	32	17	17	97	0	0

Tabel 4.2: Den modificerede prismatrix efter kørsel af funktion DIST. Man ser at en del af værdierne er ændret til lavere værdier men værdierne af bladene er ikke ændret.

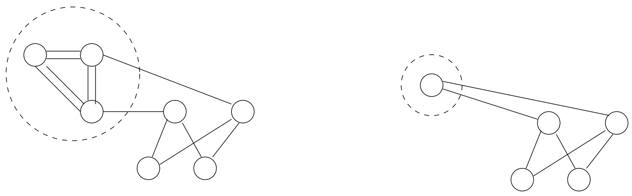
4.3 Strukturgrafen

Som nævnt bruger FSM heuristikken en strukturgraf som input. Det er derfor nødvendigt at generere en strukturgraf ud fra grafen man ønsker at øge sammenhængen i.

Jeg vil her give nogle betragtninger over hvordan en strukturgraf kan fremstilles ud fra en given graf. De er dog langtfra nok til at lave en generel metode til at generere en strukturgraf ud fra en given graf.

Når man har fundet den generelle sammenhæng λ i grafen gælder det om at finde dele af grafen der har en større sammenhæng end λ. Er sammenhængen mellem 2 punkter $\lambda+1$ kan de trækkes sammen til et punkt i strukturgrafen da sammenhængen mellem de to punkter er større end den generelle sammenhæng i grafen. På figur 4.9 er vist 3 punkter med en stiplet cirkel om på grafen til venstre. De har en sammenhæng på 4. Ved at trække dem sammen til et punkt kommer grafen til at se ud som den til højre. Punktet med den stiplede cirkel om er de tre punkter trukket sammen til ét punkt. Sammenhængen i grafen til højre er stadigvæk 2. Ved på denne måde at trække punkter sammen der indbyrdes har en sammenhæng større end λ kan man måske fremskaffe en strukturgraf. Det kræver dog at man finder det minimale snit mellem alle punktpar. Dette giver n(n-1) kald af eksempelvis Edmonds-Karps algoritme, der kan beregne sammenhængen mellem to punkter. Hvis nogle punkter trækkes sammen bliver der lidt færre kald da antallet af punkter falder. Dette er dog stadigvæk en meget beregningskrævende men dog en mulig måde at få fremskaffet en graf der indeholder alle minimalesnit og ingen snit der har en værdi større end λ. Den graf der fremkommer ved ovenstående metode er ikke altid en strukturgraf, men ved en sammenhængsgrad på 2 vil resultatet blive en strukturgraf. Den metode jeg har skitseret her er dog ikke nok til at fremstille en strukturgraf i alle tilfælde. Det kan dog være en del af en algoritme til at fremstille strukturgrafer.

Den bedste køretid jeg har set refereret til at lave en strukturgraf er i [8]. Den opgives til O(nm log n^2/m).



Figur 4.9: Et eksempel på en graf til højre med en delgraf i den stiplede cirkel med sammenhæng på 4. Hele grafen har en sammenhæng på 2, så man kan derfor sammentrække de 3 punkter i cirklen til et punkt som på figuren til højre.

Kapitel 5

Implementering

Jeg vil i dette kapitel beskrive de vigtigste datastrukturer jeg har brugt i implementeringen af de forskellige heuristikker, og de forskellige programmer jeg har udviklet.

5.1 Datastrukturer

Jeg vil her kort beskrive de to vigtigste datastrukturer jeg har brugt i forbindelse med implementeringen af de forskellige heuristikker. Herefter vil jeg beskrive de programmer jeg har lavet.

5.1.1 Incidensmatrix

Incidensmatricen repræsenterer en graf på med en n x n matrix der indeholder følgende værdier:

1 når en kant går mellem punktet p1 og p2.

0 ellers.

Incidensmatricen suppleret med en vektor med kapaciteterne giver de fornødne informationer til at beskrive en graf.

Denne repræsentation af en graf har den fordel at der er direkte adgang til kanterne. En n x n matrix fylder dog en del mere end en kantliste repræsentation når grafen er tynd.

5.1.2 Kantliste repræsentation

En anden måde at beskrive en graf på er en vektor med følgende indhold:

[startpunkt, slutpunkt, kapacitet.]

Denne metode til at beskrive grafen på bruger mindre hukommelse hvis grafen ikke er for tæt og er på den måde mere attraktiv end incidensmatricen. Den struktur har de samme informationer som incidensmatricen. For at få en køretid der er funktion af antallet af punkter og ikke antallet af kanter er det nødvendigt at listen er sorteret efter startpunktet. Dette kombineret med en vektor der fortæller hvor i vektoren de forskellige punkter er placeret giver en køretid som er en funktion af antallet af punkter. Begge datastrukturer er brugt i mine programmer.

5.1.3 Implementering i MatLab

Inden jeg vil beskrive de udviklede programmer vil jeg her kort fortælle lidt om optimering af MatLab programmer.

Jeg har brugt mcc komandoen i MatLab til at omforme MatLab koden til et MatLab script for at forbedre køretiden. Dette gøres ved følgende komando i MatLab prompten:

»mcc -x filenavn

filenavn er navnet på den file man ønsker at omforme.

Mcc giver bedre køretid i de stykker kode der indeholder mange for løkker. Mere information om mcc findes i [14] eller ved at skrive: "help mcc" i MatLab komando prompten. Ved at bruge mcc er det muligt at opnå forbedringer på 50 procent i køretiden.

5.2 Programmer

De forskellige programmer jeg har fremstillet kan findes på www.menghansen.dk som en zip-file med navnet graf.zip. Jeg vil her kort beskrive de programmerne jer har fremstillet og hvad de indeholder.

5.2.1 Edmondskarps

Min implementering af Edmonds-Karps algoritme.

Edmondskarps retunerer sammenhængsgraden af en graf. Den benytter Edmonds-Karps algoritme til at finde det maximale flow mellem to punkter. Herefter kan det minimale snit findes ved hjælp af maxflow mincut sætningen. Alle kanter har kapacitet 1. Algoritmen køres fra ét punkt til alle andre i grafen.

5.2.2 Grafg

Mit program til at generere graf og pris matricer med. Ved at kalde grafg med et antal af punkter og en procentdel af den komplette grafs kanter returneres en graf bestående af en procentdel af tilfældig udvalgte kanter. Desuden returneres også en pris matrix svarende til de kanter der ikke er i grafen. Prismatricen indeholder heltal mellem 1-100.

5.2.3 FSM

Implementering af FSM som beskrevet i kapitel 5.

5.2.4 Tabu

Implementering af tabu search med sliding window som beskrevet i afsnit 3.1. Det er muligt at variere længden af tabulisten og antallet af kanter der tilføjes i hver iteration.

5.2.5 Mut

Mit program der bruger minimum udspændende træer til at øge sammenhængen i grafer. Heuristikken er beskrevet i afsnit 3.2.

5.2.6 Valens

Valens er implementeringen af heuristikken på punkter med mindst valens som beskrevet i afsnit 3.3.

5.2.7 Ga

Implementering af den grådige heuristik.

5.2.8 Tabutest.

Jeg har for at gøre det lettere at udarbejde resultater lavet en del små hjælpeprogrammer. Tabutest er et program der kalder Tabu med forskellige datasæt og ændrer på antallet af kanter der tilføjes i hver iteration og på længden på tabulisten.

5.2.9 Res.

Res er et program der kalder Mut, Valens og Ga med forskellige datasæt. Der opsamles køretid for hver heuristik, prisen på den fundne løsning og øvre og nedre grænse for hvert datasæt.

Kapitel 6

Resultater

Jeg vil i dette afsnit præsentere resultaterne fra de 4 heuristikker jeg har implementeret og kørt med. For FSM heuristikken har jeg ikke nogen kørsler da jeg ikke har andre grafer med tilhørende strukturgrafer end den jeg har brugt som eksempel i FSM afsnittet. Jeg sammenligner istedet for med nogle af de resultater der findes i [1].

Alle programmer er lavet i MatLab. MatLab er ikke så hurtigt som eksempelvis c++ selvom jeg har compileret noget af koden til et MatLab script. Det er dog vigtig at huske at selvom jeg ville have kunnet løse større problemer hvis jeg havde brugt c++ kan man alligevel sammenligne resultaterne inbyrdes, graferne der arbejdes på er bare mindre.

Heuristikkerne der er implementeret er beskrevet i de tidligere afsnit. I tabellerne med resultater bruges følgende navne om heuristikkerne. Heuristikken, beskrevet i afsnit 3.2, der arbejder med minimalt udspændende træer på de kanter der ikke er med i grafen kaldes **mut**. Heuristikken, beskrevet i afsnit 3.3, der arbejder med minimalt udspændende træer på punkter med laveste valens i grafen kaldes **valens**. Den grådige heuristik, beskrevet i afsnit 3.4, kaldes **ga**. Tabu search heuristikken, beskrevet i afsnit 3.1, kaldes **tabu**.

Jeg har to hovedkriterier jeg sammenligner:

- Køretiden for heuristikkerne.
- Kvaliteten af løsningen de finder, givet ved prisen.

Inden jeg går igang med at beskrive de forskllige kørsler vil jeg kort beskrive hvad de forskellige afsnit herunder indeholder.

I afsnit 6.1 udføres en sammenligning af løsningerne fundet af **mut**, **ga** og **valens** heuristikkerne. Dette er udført ved at sammenligne de 3 heuristikker i ialt 70 kørsler. Jeg har lavet et gennemsnit over 10 kørsler og giver 7 tabeller med priserne på løsningerne.

I afsnit 6.2 sammenligner jeg køretiderne for **valens**, **ga** og **mut** heuristikkerne.

I afsnit 6.3 undersøges længden af tabulisten og antallet af kanter der skal skiftes i tabu search.

I afsnit 6.4 undersøges løsningen og kørertiden af **tabu**. Kvaliteten af løsningen og køretiden sammenlignes med **mut** og **valens** heuristikken.

I afsnit 6.5 sammenligner jeg **mut** heuristikken med FSM heuristikken på baggrund af kørsler med **mut** og restultater fra [1] med FSM. Det undersøges hvor stor en procentdel af løsningerne der er optimale når graferne der arbejdes på er små.

I afsnit 6.6 gives et eksempel på **valens** og en uheldig egenskab ved denne.

6.1 Sammenligning af valens, ga og mut.

For at finde ud af hvilke af **valens**, **ga** og **mut** heuristikkerne der giver de bedste resulater er her vist 7 tabeller med gennemsnittet af 10 kørsler. Jeg har fremstillet resultaterne på følgende måde. Først genereres en graf med et bestemt antal punkter, eksempelvis 25. Dernæst findes sammenhængen, antallet af kanter, øvre grænse og nedre grænse. Dernæst har jeg kørt de tre heuristikker på denne graf og opsamlet prisen på de løsninger de finder. Derefter genereres en ny graf osv. Dette gøres ialt 10 gange.

Tabellerne i dette afsnit er opbygget så der angives den mindste værdi for de 10 kørsler, gennemsnittet af de 10 kørsler, og den største værdi af de 10 kørsler. Jeg angiver i første linie sammenhængen (λ), i anden linie antallet af kanter. I tredje linie er den øvre grænse for alle tre heuristikker angivet. I fjerde linie angives nedregrænse. De sidste tre rækker er resultaterne for de tre heuristikker angivet. Alle grafer i hver tabel har samme antal punkter. I alle 70 kørsler er δ lig 1.

Resulaterne er grupperet efter hvor stor sammenhæng graferne har. Eksempelvis er der en tabel med relativt lille sammenhæng, en med en noget større sammenhæng og en med en relativt stor sammenhæng. Man kan derfor undersøge om heuristikkerne virker på både tætte og tynde grafer. Der er grafer med 25, 50 og 75 punkter. Igen kan det undersøges om der er forskel mellem heuristikkerne når antallet af punkter i grafen ændres.

Jeg har for 2 af tabellerne vist resultaterne for hver af de 10 kørsler i appendix A.

Antal punkter	**25**		
	min	gennemsnit	max
λ	1	1.9	3
Antal kanter	68	74.3	84
Øvre grænse	109	158.4	222
Nedre grænse	1	1.3	3
Valens	1	17.9	72
mut	1	6.4	30
Ga	1	6.4	30

Tabel 6.1: Gennemsnit af 10 kørsler med 25 punkter. De 10 grafer har en relativt lille sammenhængsgrad med et gennemsnit på 1.9. Det ses at **ga** og **mut** giver de samme løsninger men **valens** heuristikken giver noget dårligere løsninger. Alle 3 heuristikker finder i flere tilfælde den optimale løsning selvom gennemsnittet er et stykke fra den nedre grænse. **Mut** og **ga** giver for alle 10 kørsler de samme resultater. Resultaterne er vist i appendix A1.

Antal punkter	25		
	min	gennemsnit	max
λ	2	4	5
Antal kanter	91	105.2	115
Øvre grænse	68	210.9	265
Nedre grænse	1	1	1
Valens	2	43	88
Mut	2	24.5	204
Ga	2	6.3	22

Tabel 6.2: Gennemsnit af 10 kørsler med 25 punkter. De 10 grafer har en sammenhæng med et gennemsnit på 4. Ingen af heuristikkerne finder en løsning der er lig den nedre grænse. Det er derfor ikke muligt at sige om nogen har fundet den optimale løsning. De er dog alle relativt tæt på med en mindste værdi på 2. **Ga** og **mut** giver ikke helt de samme løsninger som det var tilfældet i den ovenstående kørsel. Der er også stor forskel på gennemsnittet af de to heuristikker. Dette skyldes at **mut** finder en meget dårlig løsning på 204 i en af de 10 kørsler, denne trækker gennemsnittet meget op. **Ga** giver de bedste løsninger.

Antal punkter	25		
	min	gennemsnit	max
λ	5	6.1	8
Antal kanter	121	137	153
Øvre grænse	162	211.6	247
Nedre grænse	1	1.1	2
Valens	3	28.7	100
Mut	1	8.1	20
Ga	1	8.1	20

Tabel 6.3: Gennemsnit af 10 kørsler med 25 punkter. De 10 grafer har en sammenhængsgrad med et gennemsnit på 6.1. Ingen af de tre heuristikker finder den optimale løsning i nogen af kørslerne. **Mut** og **Ga** finder løsninger tæt på den optimale løsning i nogle af de 10 kørsler. **Valens** giver noget dårligere løsninger end **mut** og **ga**. Resultater for hver af de 10 kørsler kan ses i appendix A2.

Antal punkter	**50**		
	min	gennemsnit	max
λ	3	4.2	5
Antal kanter	227	244.1	258
Øvre grænse	141	166.4	185
Nedre grænse	1	1.1	2
Valens	1	17.3	71
Mut	1	4.1	15
Ga	1	4.1	15

Tabel 6.4: Gennemsnit af 10 kørsler med 50 punkter. De 10 grafer har en sammenhængsgrad med et gennemsnit på 4.2. Det ses at **ga** og **mut** giver de samme løsninger. **Valens** giver meget dårligere løsninger end **mut** og **ga** med et gennemsnit på 17.3. Dette er større end den dårligste løsning på 15 som **ga** og **mut** finder. **Mut** og **ga** finder i flere af de 10 kørsler en løsning lig den nedre grænse som derfor er en optimal løsning.

Antal punkter	**50**		
	min	gennemsnit	max
λ	6	7.8	9
Antal kanter	360	371.5	394
Øvre grænse	166	206.7	278
Nedre grænse	1	1.2	2
Valens	1	28	149
Mut	1	6	13
Ga	1	6	13

Tabel 6.5: Gennemsnit af 10 kørsler med 50 punkter. De 10 grafer har en sammenhængsgrad med et gennemsnit på 7.8. **Ga** og **mut** giver de samme løsninger. Flere af **ga** og **mut**'s løsninger er på den nedre grænse, dvs at de er optimale. **Valens** finder også den optimale løsning i flere af kørslerne men har et meget højere gennemsnit end **mut** og **ga**.

Antal punkter	**75**		
	min	gennemsnit	max
λ	5	7.2	9
Antal kanter	530	554.8	574
Øvre grænse	156	181.9	213
Nedre grænse	1	1	1
Valens	1	15.1	97
Mut	1	2.1	4
Ga	1	2.1	4

Tabel 6.6: Gennemsnit af 10 kørsler med 75 punkter. De 10 grafer har en sammenhængsgrad med et gennemsnit på 7.2. **Ga** og **mut** giver de samme løsninger som er ret tæt på den optimale løsning i næsten alle kørsler. Alle tre heuristikker finder den optimale løsning i mindst et tilfælde. **Valens** har dog også en løsning der er meget dårlig med en værdi på 97 mod 4 som er **mut** og **ga's** dårligste løsning.

Antal punkter	**75**		
	min	gennemsnit	max
λ	3	4.6	6
Antal kanter	399	422.6	441
Øvre grænse	149	183.5	206
Nedre graænse	1	1.4	3
Valens	1	41.6	105
Mut	1	5.3	15
Ga	1	5.3	15

Tabel 6.7: Gennemsnit af 10 kørsler med 75 punkter. De 10 grafer har en sammenhængsgrad med et gennemsnit på 4.6. Det ses at **ga** og **mut** giver de samme løsninger. I flere af kørslerne give alle tre heuristikker den optimale løsning. **Valens** giver meget dårligere løsninger end **ga** og **mut**.

Det ses at **ga** og **mut** heuristikken ofte giver ens løsninger. Dette skyldes at begge finder mange af de billigste kanter i G" inden kanterne undersøges én for én. Da der ikke skal bruges ret mange kanter for at øge sammenhængen med én i de eksempler jeg har kørt, er det tit de samme kanter der er tilbage eller kanter der har samme pris, når heuristikkerne har undersøgt kanterne én for én.

Valens giver i langt de fleste kørsler en noget dårligere løsning end både **mut** og **ga**. Desuden har **valens** en tendens til nogle gange at have meget dyre kanter i løsningen. I resultaterne ovenfor ses at den dårligste løsning, **valens** har, ofte er meget stor. Dette er en dårlig klart egenskab.

6.2 Køretider for valens, ga og mut.

Jeg har i tabellerne nedenfor vist de 3 huristikkers løsninger og køretider når sammenhængen øges med henholdsvis 1,2 og 3. I føste søjle er angivet prisen på den løsning heuristikken finder. I anden søjle er angivet den øvre grænse og i tredje søjle er angivet køretiden i sekunder. I den øverste tredjedel af tabellen er δ=1, i den midterste tredje del er δ=2 og i den nederste tredjedel er δ=3.

Jeg har gjort følgende for at fremstille resultaterne i dette afsnit. Først genereres en graf og den øvre grænse findes. Herefter er heuristikkerne kørt med henholdsvis δ=1, δ=2 og δ=3. Køretiden for hver kørsel og prisen på løsningen der findes er opsamlet.

	pris	øvre grænse	køretid
	δ=**1**		
Mut	4.9	171.9	98.8
Valens	31.9	171.9	13.5
Ga	11.6	171.9	103
	δ=**2**		
Mut	21.9	584.6	224.5
Valens	126.6	584.6	5.4
Ga	70.9	584.6	120.4
	δ=**3**		
Mut	69	1261.5	391.7
Valens	273.6	1261.5	61.8
Ga	197.6	1261.5	177.5

Tabel 6.8: Gennemsnit af 10 kørsler for en graf med 30 punkter. Køretiden vokser meget når δ øges. **Valens** er den heuristik der er hurtigst men også den der giver de dårligste løsninger. **Mut** giver de bedste løsninger i alle tilfælde. Den har også den højeste køretid af de tre heuristikker. **Ga** giver bedre resultater end **valens** men dårligere resulater end **mut**. **Ga** har en køretid der er bedre end **mut** men dårligere end **valens**.

	pris	øvre grænse	køretid
	δ=**1**		
Mut	10	181.8	203.8
Valens	23.6	181.8	23.4
Ga	10	181.8	245
	δ=**2**		
Mut	33	612.8	509.2
Valens	106	612.8	13.1
Ga	164.8	612.8	269
	δ=**3**		
Mut	69.8	1306.4	876.3
Valens	242.6	1306.4	84.3
Ga	405.2	1306.4	558.8

Tabel 6.9: Gennemsnit af 5 kørsler for en graf med med 35 punkter. Køretiden vokser kraftigt når δ øges. **Valens** er den heuristik der hurtigst finder en løsning men også den der giver den dårligste løsning af de tre heuristikker på nær i tilfældet med δ =2. Her giver **ga** den dårligste løsning. **Mut** giver de bedste løsninger men har samtidig den længste køretid.

	pris	øvre grænse	køretid
	δ=**1**		
Mut	1	173	11250
Valens	1	173	354
Ga	1	173	10977
	δ=**2**		
Mut	4	520	24898
Valens	52	520	523
Ga	190	520	11974
	δ=**3**		
Mut	15	1148	40596
Valens	163	1148	1395
Ga	16	1148	24301

Tabel 6.10: En kørsel med en graf med 100 punkter. Køretiden er temmelig lang specielt når δ=3. Her har **valens** en stor fordel da køretiden er mindre end 10 procent af de to andre heuristikker. Løsningen når $\delta = 3$ er til gengæld også meget dårligere. **Mut** finder den bedste løsning i alle tilfælde. **Mut** bruger når δ er tre også over 11 timer på at finde denne løsninng. Når δ er én finder alle heuristikker den optimale løsning. **Ga** finder en løsning med noget højere pris når δ =2 end de to andre heuristikker.

Generelt er **mut** den heuristik der giver de bedste resultater men den har også den længste køretid. Specielt i 6.10 bruger **mut** over 11 timer mens **valens** bruger mindre end 25 minutter, en meget stor forskel i køretid. **Valens** har dog en løsning der er dyrere end **mut** så køretid og kvaliteten af løsning hænger sammen. **Ga** har en løsning der er billigere end **valens** og dyrere end **mut**. Køretiden er ca det halve at **mut** men stadigvæk over 6 timer i 6.10 med $\delta = 3$.

I kørslerne i afsnit 6.1 giver **mut** og **ga** ofte de samme løsninger. Når δ er 2 eller 3 er dette ikke tilfældet længere. Da antallet af kanter der skal tilføjes er noget større når δ er 2 eller 3 er der mange flere muligheder for at kombinere de forskellige kanter. Derfor finder heuristikkerne ikke så tit den samme løsning, når δ>1.

6.3 Undersøgelse af parametre i tabu search

Inden **tabu** kan bruges skal længden af tabulisten og antallet af kanter der skiftes i hver iteration undersøges. Til dette har jeg lavet nogen kørsler med forskllige længder af tabulister og antallet af kanter der skiftes i hver iteration.

For at afgøre længden og antallet af kanter der skal skiftes i hver iteration under forskellige mulighede, har jeg valgt at undersøge længden af tabulisten med værdier på 2,4,6,8,10, og antallet af kanter der skiftes i hver iteration med værdierne 1, 2, 4, 6.

De nedenstående tabeller er vises antallet at kanter der skiftes henad og længden af tabulisten nedad. Jeg angiver prisen for hver kørsel med en given længde af tabulisten og et givent antal af kanter der skiftes. Det vil sige at i tabel 6.11 er prisen på løsningen 75 når der skiftes 1 kant i hver iteration og tabulisten har længden 4.

	1	2	4	6
2	33	32	58	75
4	75	122	122	122
6	122	122	122	122
8	122	122	122	122
10	122	122	122	122

Tabel 6.11: En kørsel af **tabu** med 20 punkter. δ er øget med én. Den initielle løsning har en pris på 122. Den bedste løsning der findes af tabusearch er med en tabuliste på 2 og 2 kanter der skiftes i hver iteration. Denne løsning har en en pris på 32.

Jeg har udført flere kørsler end beskrevet her. En del af disse kørsler har ligesom i 6.13 ikke forbedret den initielle løsning. Jeg har valgt kun at medtage 6.13 som eksempel på disse kørsler da de ikke give mere information om længden af tabulisten og antallet af kanter der skal skiftes.

Antallet af kanter der skal skiftes i hver iteration og længden af tabulisten afhænger størrelsen af grafen der arbejdes med. Med relativt små grafer er en længde af tabu listen på 2-4 og 2 kanter der skiftes, det der giver de bedste resulater.

	1	2	4	6
2	61	42	50	42
4	38	97	143	143
6	143	143	143	143
8	143	143	143	143
10	143	143	143	143

Tabel 6.12: En kørsel af **tabu** med 35 punkter. δ er øget med én. Den initielle løsning har en pris på 143. Den bedst fundne løsning er når længden af tabulisten er 2 og antallet af kanter der skiftes er enten 2 eller 6. Denne løsning har en pris på 42.

	1	2	4	6
2	21	21	21	21
4	21	21	21	21
6	21	21	21	21
8	21	21	21	21
10	21	21	21	21

Tabel 6.13: En kørsel af **tabu** med 40 punkter. δ er øget med én. Den initielle løsning har en pris på 21. **Tabu** finder ingen løsning der er bedre end den initielle løsning.

	1	2	4	6
2	39	51	62	33
4	30	54	141	136
6	146	181	191	157
8	228	209	288	282
10	267	323	332	332

Tabel 6.14: En kørsel af **tabu** med 40 punkter. δ er øget med én. Den initielle løsning har en pris på 332. Den bedst fundne løsning er når én kant skiftes og tabulisten har en længde på 4.

6.4 Resultater opnået med tabu search

Efter at have undersøgt længden af tabulisten og antallet af kanter der skal skiftes vil jeg her vise nogle resultater af kørsler med tabu search.

Jeg har brugt en løsning fra **valens** heuristikken som initiel løsining til **tabu** da **valens** giver de dårligste løsninger af **ga**, **mut** og **valens**. Dette er valgt da **tabu** har større mulighed for at finde en løsning der er bedre end **valens'** løsning til problemet. Jeg har også kørt **mut** for at have noget at sammenligne den løsning **tabu** finder med.

I tabellerne nedenfor svarer hver række til en graf. I den første søjle nævnes nummeret på kørslen. I anden søjle vises prisen på løsningen **mut** heuristikken finder.I trejde søjle vises prisen på løsningen **tabu** finder. I fjerde søjle vises den nedre grænse. I femte søjle vises prisen af den løsning **valens** heuristikken finder.

Kørsel	**Mut**	**Tabu**	Nedre grænse	**Valens**
1	2	2	1	2
2	32	125	4	125
3	1	62	1	87
4	5	5	1	5
5	7	7	2	7
6	10	56	2	56
7	5	5	1	5
8	20	20	2	20
9	8	8	1	8
10	1	1	1	1

Tabel 6.15: 10 kørsler af **tabu** med 20 punkter. δ er øget med én. **Tabu** finder ikke i nogen af kørslerne en løsning der er bedre end **mut**. I nr. 10 kørsel finder **tabu** den optimale løsning. Denne er dog ikke fundet af **tabu** da den initielle løsning også er optimal. Det er derfor **valens** der har fundet den optimale løsning. I en del tilfælde kan **tabu** ikke forbedre den initielle løsning selvom den har en pris der er meget højere en **mut** f.eks. i kørsel nr. 6.

Tabu giver i langt de fleste tilfælde dårligere løsninger end **mut**. I en del tilfælde kan **tabu** ikke forbedre den initielle løsning. **Tabu's** køretid er generelt længere end **mut's**. Der bruges som bekendt sliding window.

Kørsel	**Mut**	**Tabu**	Nedre grænse	**Valens**
1	17	17	1	17
2	1	1	1	1
3	4	38	1	38
4	4	31	1	31
5	7	11	1	11
6	2	2	1	2
7	3	3	1	3
8	1	1	1	1
9	1	1	1	1
10	10	10	1	10

Tabel 6.16: 10 kørsler af **tabu** med 35 punkter. δ er øget med én. I alle 10 kørsler finder **tabu** ingen løsning der er bedre end den initielle løsning. **Mut** finder i flere tilfælde løsninger der er bedre end den initielle løsning **tabu** bruger som input. Det er derfor muligt for **tabu** at finde en bedre løsning.

Kørsel	**Mut**	**Tabu**	Nedre grænse	**Valens**
1	10	120	2	163
2	170	136	1	139
3	1	1	1	1

Tabel 6.17: 3 kørsler af **tabu** med 50 punkter. δ er øget med én. I den tredje kørsel finder både **tabu** og **mut** den optimale løsning, den initielle løsning er dog også optimal. I anden kørsel er **mut's** løsning dårligere end **tabus** løsning.

Dette medfører at **tabu** bliver ved at køre hvis den bedste løsning **tabu** har fundet bliver opdateret.

6.5 Sammeligning af resultater med FSM

I artiklen [1] er en kort beskrivelse af de resultater der er opnået med FSM. Her vises blandt andet en tabel over 360 kørsler. I 79.2 procent af tilfældene finder FSM den optimale løsning. De 360 grafer der er testgrundlaget er en blanding af grafer med 10, 15 og 20 punkter. I disse små grafer kan den optimale løsning findes ved hjælp af udtømmende søgning. Den komplette graf med 20 punkter har 190 kanter. Dette er ikke et særligt stort løsningsrum. Derfor tror jeg at mange heuristikker vil finde den optimale løsning i disse små grafer. Jeg har for at kunne sammenligne disse resultater kørt 180 kørsler med **mut** heuristikken. **Mut** er den der giver flest optimale løsninger i de kørsler hvor **mut** er sammenlignet med **ga** og **valens**. Jeg har kørt på grafer med 10,15,20 og 25 punkter. **Mut** finder en løsning der er lig den nedre grænse i 11 procent af tilfældene. Dette er et stykke fra resultaterne for FSM. Jeg har som tidligere nævnt ikke nogen test data hvor jeg kender den optimale løsning. Den nedre grænse jeg bruger er i mange tilfælde bedre end den optimale løsning. Jeg har set på de fundne løsninger og mange af dem ligger ikke ret langt fra den nedre grænse. Der er derfor grund til at antage at en del af disse også er optimale. Hvor stor en procentdel dette drejer sig om ved jeg ikke.

Punkter	Antal kørsler	antal optimale løsninger
10	50	2
15	50	8
20	50	5
25	30	5

Tabel 6.18: 180 kørsler af **mut** med 10, 15,20 og 25 punkter.I første søjle angives antallet af punkter, i anden søjle vises det totale antal kørsler og i sidste søjle antallet af kørsler der har givet en løsning med pris lig den nedre grænse. Ialt er 20 af de 180 kørsler lig den nedre grænse og derfor også optimale.

6.6 Eksempel på valens heuristikken

I 6.19 er vist en kørsel med 50 punkter og 122 kanter. Denne kørsel er medtaget for at vise en uheldig egenskab ved **valens** heuristikken.

	nedregrænse	pris	øvregrænse
δ=1 **Valens**	2	95	133
δ=1 **Mut** og **ga**	2	2	133
δ=2 **Valens**	6	84	433
δ=2 **Mut** og **ga**	6	33	433

Tabel 6.19: Resultater for **mut** og **valens** heuristikkerne på en graf med 50 punkter og 122 kanter og λ=2. Der er to kørsler for hver heuristik hvor sammenhængen sættes op med henholdvis 1 og 2.

Det ses i tabel 6.19 at løsningen for at øge sammenhængen i grafen er billigere ved at øge sammenhængen med to end med én når **valens** heuristikken bruges i dette tilfælde. Efter at have undersøgt de løsninger **valens** heuristikken har fundet er grunden følgende: Når **valens** skal øge sammenhængen i grafen med en er det kun nødvendigt at tilføje en kant da der er to punkter der har for lav **valens**. Kanten mellem disse to punkter er ikke med i grafen G' og kan derfor tilføjes til G'. Den har i eksemplet en pris på 95. Da det kun er nødvendigt at tilføje denne ene kant til G' for at øge sammenhængen med en bliver den ikke fjernet når heuristikken undersøger om kanten er nødvendig for at øge sammenhængen i grafen.

Derfor bliver kanten mellem disse to punkter tilføjet og dette er nok til at øge sammenhængen i hele grafen med 1. Kanten har prisen 95. Når man bruger **valens** til at øge sammenhængen med to, bliver den samme kant tilføjet først. Dernæst får punkter med valensen 3 tilføjet kanter og til slut får punkter med valensen 4 tilføjet kanter. Da prisen på den omtalte kant mellem punkterne er meget dyr bliver kanten fjernet i den efterfølgende afprøvning af om kanterne kan fjernes uden at ødelægge sammenhængen i grafen. Derfor finder **valens** heuristikken en løsninger der er billigere når den øger valensen med 2 end med én i dette tilfælde.

6.7 Opsumering af resultater

Jeg har observeret at **mut** giver de bedste løsninger under alle de testede data. **Valens** giver noget dårligere løsninger men har til gengæld en langt bedre køretid. **Ga** giver generelt bedre løsninger end **Valens** men dårligere løsninger end **mut**. Det er opserveret at kvaliteten af løsningen og køretiden hænger sammen så **mut** har læmgt køretid og giver de bedste løsninger af de tre. **Valens** er hurtigst af de tre og giver generelt de dårligste løsninger af de tre heuristikker.

Tabu giver ikke lige så gode resultater som de tre andre og har en lang køretid.

Det er generelt observeret at en stor del af kørertiden går med at finde det minimale snit i grafen. MatLab har en funktion "prfile", der angiver hvor langtid de forskellige dele af et program tager at køre. Det er her opserveret at kald til Edmonds-karps algoritme tage en tror del af køretiden i alle fire heuristikker.

Kapitel 7

Konklusion

Jeg har implementeret 4 heuristikker til at øge sammenhængen i ikke orienterede grafer. De bedste resultater fås af **mut** og **ga** heuristikkerne. Den hurtigste af heuristikkerne er **valens** men denne giver også noget dårligere resultater end **ga** og **mut**. **Tabu** har vist sig ikke at give lige så gode resultater som **mut** og **ga** heuristikkerne men ligeså gode resultater som **valens**, da **valens** bruges som den initielle løsning. **Tabu** er den langsomste af de 4 heuristikker jeg har testet. **FSM** heuristikken har det ikke været muligt at sammenligne direkte med de andre heuristikker da jeg ikke har data fra [1] som resultaterne er kørt på i artiklen. **Tabu** er bedst til at forsøge at forbedre en given løsning fra en af de andre heuristikker. Det er dog sjældent at **tabu** finder en løsning der er bedre end den initielle løsning hvis den er god. Tabu listen bør ikke være for lang ca 2-6 og antallet af kanter der skiftes i hver iteration bør være 2-4 når der arbejdes med grafer med 20-75 punkter.

FSM heuristikken er klart den mest komplicerede af de 5 heuristikker. Den er ikke fuldt beskrevet i [1] idet kun er den del af heuristikken der arbejder med strukturgrafen er beskrevet. De dele der mangler for at kunne bruge **FSM** i praksis er betydelige og vil sandsynligvis tage meget tid og arbejde at forstå og implementere. De 4 andre heuristikker er forholdsvis lette at forstå og også til at implementere på en rimelig tid.

7.1 Forslag til videre arbejde.

Jeg mener at der er gode muligheder i at arbejde med strukturgrafer til at løse w-k-eca. Man kan prøve at bruge **FSM**, men jeg mener også at der er gode muligheder i selv at prøve at udvikle heuristikker der bruger strukturgrafens egenskaber. Det kræver dog at man finder en metode der kan fremstille strukturgrafer ud fra en given graf som er hurtig, og som samtidig er let at implementere.

En implementering af de heuristikker jeg har arbejdet med i dette projekt i c/c++ vil give en bedre køretid for alle heuristikkerne og større problemer vil derfor kunne løses.

Kapitel 8

Appendix

Appendix A1. Alle resultater som tabel 6.1 er et gennemsnit af: Hver række i nedenstående tabel svarer til en graf. 1. søjle indeholder sammenhængen i grafen, 2. søjle indeholder antallet af kanter, 3. søjle indeholder den øvre grænse, 4. søjle prisen af løsningen valens heuristikken finder, 5. søjle indeholder prisen på løsningen mut heuristikken finder, 6. søjle indeholder prisen på løsningen ga heuristikken finder, 7. søjle indeholder den nedre grænse.

2	70	135	22	1	1	1
2	82	151	2	2	2	1
3	73	153	72	16	16	2
3	73	133	68	30	30	3
2	78	180	1	1	1	1
1	68	222	6	6	6	1
2	78	132	1	1	1	1
1	69	209	1	1	1	1
1	84	109	2	2	2	1
2	68	160	4	4	4	1

Appendix A2. Alle resultater som tabel 6.3 er et gennemsnit af: Hver række i nedenstående tabel svarer til en graf. 1. søjle indeholder sammenhængen i grafen, 2. søjle indeholder antallet af kanter, 3. søjle indeholder den øvre grænse, 4. søjle prisen af løsningen valens heuristikken finder, 5. søjle indeholder prisen på løsningen mut heuristikken finder, 6. søjle indeholder prisen på løsningen ga heuristikken finder, 7. søjle indeholder den nedre grænse.

5	143	214	8	8	8	1
6	135	243	7	7	7	1
6	137	162	47	4	4	1
7	146	199	72	11	11	1
7	135	246	25	3	3	1
5	132	186	5	5	5	2
5	128	182	3	3	3	1
7	153	219	17	17	17	1
8	140	218	100	20	20	1
5	121	247	3	3	3	1

Litteratur

[1] Toshimasa Watanabe, Toshiya Mashima and Satoshi Taoka, *The k-Edge-Connectivity Augmentation Problem of Weighted Graphs* , 1992.

[2] William J. cook, William H. Cunningham, William R. Pulleyblank, Alexander Schrijver, *Combinatorial Optimization* , 1998.

[3] Frank Nielsen *Grafteori - algoritmer og netværk* , 1995.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest *Introduction to Algorithms* , 1997.

[5] H. N. Garbov *Applications of a poset representation to edge connectivity and graph rigidity* , IEEE 1991 812-821.

[6] J. E. Hopcroft, R. E. Tarjan *Dividing a graph into triconnected components* , SIAM 1973 136-158.

[7] Hiroshi Nagamochi, Yoshitaka Nakao and Toshihide Ibaraki *A fast algorithm for cactus representations of minimum cuts* , 1997.

[8] Lisa Fleischer *Building chain and cactus representations of all minimum cuts from Hao-Orlin in the same asymptotic time* , Journal of algorithms 33, 51-72 1999.

[9] András A. Benczúr *Cut structures and randomized algorithms in edge-connectivity problems.* , MIT 1993.

[10] Dalit Naor, Dan Gusfield, Charles Martel *A fast algorithm for optimally increasing the edge-connectivity* , IEEE 1990.

[11] Kapali P. Eswaran R. Endre Tarjan *Augmentation problems* , SIAM 1976.

[12] Fred Glover, Manuel Laguna *Tabu Search* , SIAM 1973 136-158.

[13] B. Fortz, M. Labbé *A tubu search heuristic for the design of two-connected networks with bounded rings* .

[14] George Almási and David Padua *MaJic: Compiling MatLab for speed and responsiveness* , University of illinois at Urbana-Campaign.

[15] Abdol-Hossein Esfahanian *On the evolution of graph connectivity algorithms* , Computer science and Engineering Department Michigan State University.

[16] Jørgen Bang-Jensen, Gregory Gutin *Digraphs: Theory, Algorithms and Applications*, Springer 2001.