Aalborg University

Department of Computer Science.
Fredrik Bajers Vej 7E, 9220 Aalborg Ø.

Titel:
Traveling Salesman Problem

Projektperiode:
16. maj 2003 til 20. juni 2003

Semester:
BOS03

Gruppebetegnelse:
E2-216

Gruppemedlemmer:
Kenneth Vittrup
Morten Zinck
Thomas Winterberg
Kasper Ørum Nielsen
Frederik Dannemare
Peter Sönder

Vejleder:
Helen Urban

Synopsis

Problemstillingen i dette projekt bygger på Traveling Salesman Problemet. Rapporten starter med kort at beskrive, hvad dette problem omhandler. Efterfølgende gennemgås den teori, som skal munde ud i de algoritmer, der kan give et fornuftig estimat for en løsning af problemet.

Inden der træffes et valg omkring, hvilken algoritme vi ville vælge ud til en evt. efterfølgende implementering af de få, vi kan nå at gennemgå, vil der blive gennemgået en del teori. Denne teori omhandler bl.a. emnerne Store-O notation, bevisteknikker, mængder, grafer, træer og endelig kompleksitetsteori.

Indhold

Kapitel 1

Indledning

Med en forudgående beskrivelse af anvendt teori har denne rapport til formål at munde ud i forskellige løsningsforslag til den meget omdiskuterede matematiske og datalogiske problemstilling "Traveling Salesman Problem" (forkortet TSP).

Problemet kan beskrives således: "En handelsrejsende skal på sin rejse besøge et antal byer netop én gang og samtidig opnå dette via den kortest mulige rute. En yderligere betingelse er, at startbyen er også slutbyen, da han jo gerne skulle komme hjem igen efter sin handelstur".

Problematikken i TSP kategoriseres af matematikere som et emne, der er meget svært at løse. Matematikere har endnu ikke formået at fremstille en metode, som løser problemet *effektivt* (definitionen af effektivt forklares senere i rapporten), men forskellige matematikere har fremstillet metoder, som er i stand til at producere en løsning inden for rimelig tid. Denne løsning er ikke nødvendigvis den optimale løsning (den korteste rute), men tæt på.

Hvorvidt TSP (samt andre matematiske problemer i samme kategori) rent faktisk kan løses effektivt, er et af de mest omdiskuterede emner blandt matematikere. De fleste er dog enige om, at problemtypen ikke kan løses effektivt, selv om ingen med sikkerhed ved dette.

Kapitel 2

Funktionsvækst

For at forstå TSP-problematikken er det vigtigt at have en forståelse for funktioner.

Væksten af funktioner har en speciel notation, benævnt *Store-O*. Dette er en måde at estimere, hvorledes en funktion udvikler sig, når mængden af input tiltager. Det er muligt at lave en funktion, som er hurtig på én computer men langsom på en anden. Derfor vil det ikke være praktisk at snakke om køretid for funktionen, da dette vil afhænge af computerkraft.

Derimod kan man analysere en funktion, og ud fra dette kan man bestemme funktionens udvikling uafhængig af underliggende soft– og hardware. Under analysen er det ligeledes ikke nødvendigt at tage højde for, hvilket programmeringssprog løsningsforslaget er implementeret i. Dette skyldes, at Store-O udregnes fra simple operationer som f.eks. antallet af sammenligninger, multipliceringer og/eller additioner.

Store-O blev introduceret i 1892 af Paul Bachmann (1837-1920) i forbindelse med funktionsvækst og talteori. Denne notation er i mere end et århundrede blevet brugt som en generel retningslinie for at beskrive kompleksitet af funktioner.[7]

2.1 Store-O

Hvis f er en funktion, vil funktionen g estimere worst-case gennemløbet af funktionen f.

Definition 1 *Store-O*

Lad f og g være funktioner fra heltal eller reelle tal over i reelle tal. Så siges $f(x)$ at være $O(g(x))$, hvis der findes konstanter C og k, således at

$$|f(x)| \leq C|g(x)|$$

når $x > k$ (dette læses som "$f(x)$ er Store-O af $g(x)$").

Med andre ord vil $g(x)$ være af minimum samme kompleksitet som $f(x)$. Til at beskrive denne relation bruges to konstanter C og k, som kaldes "vidner". For at finde sammenhængen mellem $f(x)$ og $g(x)$ skal vi blot finde ét par vidner, således at $|f(x)| \leq C|g(x)|$, for alle $x > k$.

Eksempel

Find $O(g(x))$ for $f(x) = x^2 + 2x + 1$, for $x > k$

Først sættes dette større end eller lig 0:

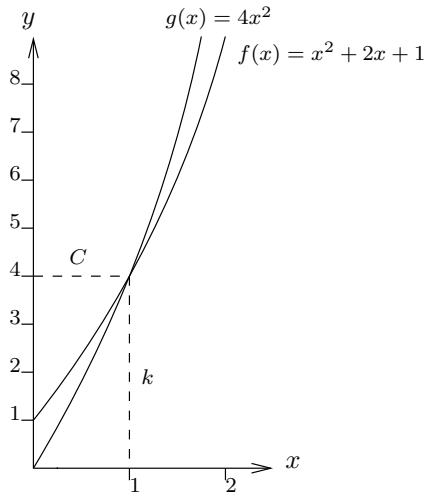
$$0 \leq x^2 + 2x + 1$$

Eftersom det største led er x^2, får vi:

$$x^2 + 2x + 1 \leq x^2 + 2x^2 + x^2 = 4x^2$$

Hermed er $C = 4$ og $k = 1$ et par vidner til at vise, at $f(x)$ er $O(x^2)$. Dette kan præciseres, idet $f(x) = x^2 + 2x + 1$ er mindre end $g(x) = 4x^2$ når $x > 1$. Dette er illustreret i figur 2.1.

◀



Figur 2.1: Sammenhæng mellem graferne i eksemplet.

Tabel 2.1 illustrerer de mest brugte betegnelser for funktionsvækst.

For at løse et problem skal man bruge en *algoritme* til at udregne en løsning med.

Definition 2 *Algoritme*

En algoritme er et tælleligt antal præcise instruktioner, der anvendes til at udføre en beregning eller til at løse et problem.

En algoritme er effektiv, når den kan gennemløbes i polynomiel tid. Dette er ikke tilfældet for $O(x!)$ og $O(b^x)$, hvor b er en konstant. Selv en meget lille værdi af x vil resultere i, at algoritmer af denne type ikke vil blive gennemløbet effektivt. Dette faktum er illutreret i tabel 2.2, der viser kompleksiteter af algoritmerne fra tabel 2.1 ved forskellige lave værdier af x.

Kompleksitet	Betegnelse
$O(1)$	Konstant kompleksitet
$O(\log\ x)$	Logaritmisk kompleksitet
$O(x)$	Linær kompleksitet
$O(x\ \log\ x)$	x log x kompleksitet
$O(x^b)$	Polynomiel kompleksitet
$O(b^x)$, for $b > 1$	Eksponintiel kompleksitet
$O(x!)$	Faktoriel kompleksitet

Tabel 2.1: Liste over de mest anvendte funktionsvækster.

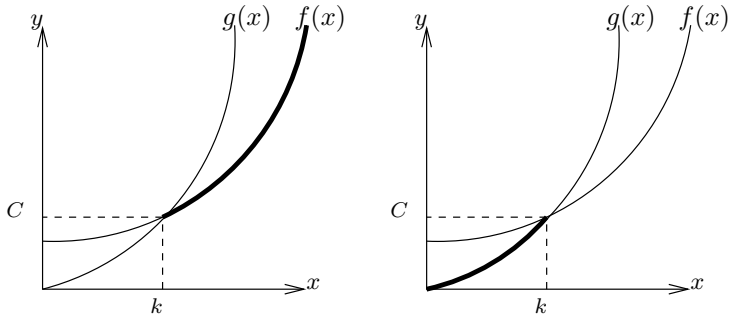
x	$O(1)$	$O(\log x)$	$O(x)$	$O(x\log x)$	$O(x^2)$	$O(2^x)$	$O(x!)$
1	1	0.00	1	0.00	1	2	1
2	1	0.30	2	0.60	4	4	2
3	1	0.48	3	1.43	9	8	6
4	1	0.60	4	2.41	16	16	24
5	1	0.70	5	3.49	25	32	120
6	1	0.78	6	4.67	36	64	720
7	1	0.85	7	5.92	49	128	5040
8	1	0.90	8	7.22	64	256	40320
9	1	0.95	9	8.59	81	512	362880
10	1	1.00	10	10.00	100	1024	3628800

Tabel 2.2: Eksempler på algoritmers kompleksitet ved forskellige værdier af x.

2.2 Store-Theta og Store-Omega

Store-O er den notation, som oftest bruges, når væksten af en funktion skal beskrives. Store-O har dog nogle begrænsninger, da denne kun giver en øvre grænse af kompleksiteten af $f(x)$. Store-O er derved beregnet til at beskrive *worst-case* gennemløb af funktioner. Ligeledes findes notationer for at beskrive *best-case* og *average-case* gennemløb af funktioner, benævnt hhv. *Store-Theta* (Θ) og *Store-Omega* (Ω).

For at illustrere sammenhængen mellem Store-O og Store-Theta er der i figur 2.2 med en fed streg indtegnet, hvilke dele af funktionerne $f(x)$ og $g(x)$, der er interessante i de enkelte tilfælde.



(a) Θ for $x > k$ markeret

(b) Θ for $x < k$ markeret

(c) Store-O for $x > k$ markeret

(d) Store-O for $x < k$ markeret

Figur 2.2: Eksempel på Store-Omega og Store-O vist i en graf

TSP vil minimum være en eksponentielt voksende funktion, hvis en optimal løsning skal findes. Det interessante i problemet er, hvad *worst-case* gennemløb af funktioner er (punkt c i figur 2.2). Derfor vil dette projekt ikke involvere Θ og Ω yderligere.

Kapitel 3

Bevisstrategier

At kunne benytte beviser til at verificere en løsning til en problemstilling er centralt i dette projekt.

Inden for matematikken skal man kunne bevise, at en bestemt antagelse er korrekt. I dette afsnit vil vi gennemgå nogle af de værktøjer, man kan bruge til at bevise, om en matematisk påstand er korrekt. Vi viser nogle "relativt" simple eksempler til de vigtigste bevisteknikker, der gennemgåes. Vi starter med at definere, hvad et udsagn er.

Definition 3 *Udsagn*

Et udsagn er et deklarativt udtryk, der enten kan være sandt eller falsk.

Man kan bruge udsagn som hjælpeværktøjer til at bevise, om en matematisk påstand er korrekt. Et udsagn p kunne f.eks. være "Det regner". Udsagnet p kan også *negeres* ved at skrive $\neg p$, og dette betyder "Det regner ikke".

Definition 4 *Negation*

Lad p være et udsagn. Lad "Det er ikke tilfældet at p" være et andet udsagn, kaldet negationen *af p. Negationen af p er benævnt ved $\neg p$. Udsagnet $\neg p$ læses som "not p".*

I forbindelse med beviser er definitionen af en implikation den første, som skal kendes.

Definition 5 *Implikation*

Lad p og q være udsagn. Implikationen $p \rightarrow q$ er kun falsk, hvis udsagnet p er sandt, samtidig med at q er falsk. Ellers er implikationen sand. I implikationen $p \rightarrow q$ kaldes p hypotesen, og q kaldes konklusionen.

Et eksempel på en implikation kan vi lade hypotesen p være "Det er søndag" og lade konklusionen q være "Vi har fri". Implikationen $p \rightarrow q$ bliver til sætningen "Hvis det er søndag, så har vi fri".

3.1 Direkte beviser

Når det skal bevises, om en hypotese er korrekt, kan man først prøve med et *direkte bevis*. Vi vil i følgende tage et eksempel, hvor man bruger et direkte bevis til at bevise implikationen $p \rightarrow q$. Dette gøres ved at vise, at hvis q er sand, kan implikationen aldrig blive falsk. Et bevis for dette kan udføres ved at antage, at udsagnet p er sandt. Derefter kan man inddrage beviste teoretiske og matematiske sætninger for derved at bevise, at q aldrig er falsk samtidig med at p er sand.

Eksempel
Det kan bevises, at når n er et ulige heltal, så er n^2 også et ulige heltal. Hvis hypotesen p er, at n er et ulige heltal, så er konklusionen q, at n også kan skrives på formen $2k+1$, hvor k er et heltal. Dette giver implikationen $p \rightarrow q$.

Der gælder den matematiske sætning der siger at et lige heltal, som adderes med tallet 1, altid giver et ulige heltal.

Dette betyder, at n^2 kan omskrives til:

$$\begin{aligned} n^2 &= (2k+1)^2 \\ &= (2k+1)(2k+1) \\ &= 4k^2 + 4k + 1 \\ &= 2(2k^2 + 2k) + 1 \end{aligned}$$

Vi beskriver her kun den sidste udregning i eksemplet. Parentesens indhold multipliceret med to vil give et lige heltal. Dette kommer af at et heltal (lige eller ulige), som multipliceres med to, altid giver et lige heltal. Når man adderer tallet 1 til dette, vil det endelige resultat give et ulige heltal.

Dermed er implikation bevist.

◀

3.2 Indirekte beviser

Hvis man har svært ved at bevise en implikation med et direkte bevis, kan man forsøge at lave et *indirekte bevis*. Dette gøres ved at bevise, at følgende implikation er sand:

$$\neg q \rightarrow \neg p$$

Denne implikation er den *omvendte* af implikationen $p \rightarrow q$, og disse to implikationer har samme sandhedstabel (se tabel 3.1).

Definition 6 *Sandhedstabel*

En sandhedstabel er en tabel over alle sandhedsværdier for en given implikation.

I sandhedstabellerne (og enkelte steder i teksten) er S en forkortelse for ordet sand(t), og F er en forkortelse for ordet falsk(t).

p	q	$p \to q$	$\neg q$	$\neg p$	$\neg q \to \neg p$
S	S	S	F	F	S
S	F	F	S	F	F
F	S	S	F	S	S
F	F	S	S	S	S

Tabel 3.1: Sandhedstabel for $\neg q \to \neg p$.

Hvis hypotesen p er "Det regner", og konklusionen q er "Man bliver våd", betyder $p \to q$, at "Når det regner, bliver man våd". Den omvendte implikation $\neg q \to \neg p$ betyder da "Hvis man ikke er våd, så regner det ikke".

Fremgangsmåden, som bruges til at bevise en implikation ved hjælp af et indirekte bevis, minder om den, som bruges med et direkte bevis. Der benyttes blot den omvendte implikation $\neg q \to \neg p$ i stedet for den oprindelige implikation $p \to q$.

Implikationen $\neg q \to \neg p$ bevises ved at bruge den direkte bevisteknik på denne omvendte implikation. Man antager altså, at $\neg p$ er sand og beviser, at $\neg q$ derved også bliver sand.

Eksempel
Vi vil give et indirekte bevis på, at når hypotesen p er, at $3n+2$ giver et ulige heltal, så må konklusionen q være, at n er et ulige heltal. Nu ved vi, hvad implikationen $p \to q$ er. Det antages, at konklusionen q er falsk. $\neg q$ svarer til, at n er et lige heltal, som kan skrives som $n = 2k$, hvor k er et heltal.

Dette indsættes i $3n + 2$, og konklusionen negeres:

$$\begin{aligned} 3n + 2 &= 3(2k) + 2 \\ &= 6k + 2 \\ &= 2(3k + 1) \end{aligned}$$

Det fremgår at, $\neg q$ medfører, at $3n + 2$ er et lige heltal.

Derved er implikationen bevist.

◀

3.3 Bevis med svag induktion

Svag induktion er en meget anvendt bevisteknik, idet den beviser, at et bestemt udsagn r er rigtigt. Dette gøres ved at se på det første element og derefter bevise, at implikationen $p \rightarrow q$ er rigtig, når p er et hvilket som helst element, og q er det efterfølgende element $(p + 1)$.

Der er to skridt i et svagt induktionsbevis, hvoraf det sidste er rekursivt:

Basisskridt: Vis at udsagnet r er sandt for det første element, f.eks. $r(1)$.

Induktionsskridt: Hypotesen p er, at udsagnet $r(n)$ er sandt, hvor n er et vilkårligt element. Det bevises efterfølgende, at $r(n + 1)$ er sand, forudsat at $r(n)$ er sand. jævnfør implikationen $p \rightarrow q$.

Eksempel
Ved at bruge svag induktion vil vi bevise følgende sætningen:

$$2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

Ovenstående udsagn gælder for alle n, som ikke er negative.

Basisskridt: $r(0)$ er et sandt udsagn, da $2^0 = 1 = 2^1 - 1$.

Induktionsskridt: Hypotesen p er, at udsagnet $r(n)$ er sandt. Det skal så ud fra dette bevises, at $r(n + 1)$ er sand.

Det vil sige, at det skal vises, at følgende er korrekt:

$$1 + 2 + 2^2 + \ldots + 2^n + 2^{n+1} = 2^{(n+1)+1} - 1 = 2^{n+2} - 1$$

Udregningerne for induktionsskridtet bliver da:

$$\begin{aligned} 1+2+2^2+2^3+\ldots+2^n+2^{n+1} &= (1+2+2^2+\ldots+2^n)+2^{n+1} \\ &= (2^{n+1}-1)+2^{n+1} \\ &= 2*2^{n+1}-1 \\ &= 2^{n+2}-1 \end{aligned}$$

I udregningerne ovenfor vises det, at når værdierne for tallet 2 opløftet i henholdsvis 0, 1, 2 osv. op til og med n'te potens summeres, så giver det samme resultat som $2^{n+1}-1$. Det er bevist, at udsagnet r er sandt, når 2 opløftes i en potens, der er én højere (2^{n+1}) end den forrige (2^n).

◀

3.4 Bevis for stærk induktion

Stærk induktion er en variant af svag induktion, og det er næsten lig et svagt induktionsbevis, da kun induktionsskridtet er ændret. I induktionsskridtet benyttes *konjunktion*.

Definition 7 *Konjunktion*

Lad p og q være udsagn. Udsagnet "p og q" er sandt når både p og q er sande, og falsk ellers. En konjunktion er angivet som $p \wedge q$.

Fremgangsmåden for stærk induktion følger:

Basisskridt: Vis at udsagnet r er sandt for det første element, f.eks. $r(1)$.

Induktionsskridt: Hypotesen p er, at når alle udsagnene $r(1)$, $r(2)$,...,$r(n)$ er sande, så er $r(n+1)$ også sand. Dette giver implikationen $[r(1) \wedge r(2) \wedge \ldots \wedge r(n)] \rightarrow r(n+1)$, hvor n er et vilkårligt element. Med andre ord bevises det, at udsagnet $r(i)$ er sandt for $i = 1, 2, \ldots, n$. Ud fra dette er konklusionen q, at $r(n+1)$ også må være sandt.

Det kan bevises, at den ene metode er en pålidelig og brugbar bevisteknik forudsat, at den anden også er det.

Eksempel
Man har et spil med to spillere og to lige store stakke med n tændstikker i hver. Den første spiller starter med at tage én eller flere tændstikker fra den en stakkene. Derefter tager den anden spiller også én eller flere tændstikker fra den ene eller den anden af stakkene. Spillerne skiftes på denne måde med at tage et antal tændstikker. Den, der tager den sidste tændstik, har vundet. I dette spil kan man garantere, at spiller nr. to konsekvent kan vinde hver gang.

Basisskridt: Hvis der er en tændstik i hver stak, så kan den første spiller tage tændstikken i den ene stak, og så kan den anden spiller vinde ved at tage tændstikken i den anden stak.

Induktionsskridt: Hypotesen p er, at spiller nr. to kan vinde hver gang med stakke af $1, 2, \ldots, n$ tændstikker, hvor n er et vilkårligt antal større end én.

Det kan bevises, at når spiller nr. to kan vinde for basisskridtet med stakke bestående af én tændstik, så kan det også lade sig gøre for stakke med en størrelse op til n tændstikker, da de kan komme tilbage til basisskridtet. Konklusionen q siger dermed, at det kan lade sig gøre, at spiller nr. to vinder ved stakke af $n + 1$ tændstikker.

Nu bevises tilfældet med $n+1$ tændstikker. Det vises, at når den første spiller fjerner j tændstikker fra en af stakkene, så vil der være $(n+1)-j$ tændstikker i netop den stak. Hvis spiller nr. to derefter fjerner samme antal tændstikker fra den anden stak, så vil hver stak derefter have $(n + 1) - j$ tændstikker. Det gælder så, at hver stak indeholder $1 \leq (n + 1) - j \leq n$ tændstikker. Derfor kan spiller nr. to altid vinde ifølge induktionshypotesen p, da antallet af tændstikker i hver stak efter første tur er indenfor det interval, vi allerede har bevist vores påstand gælder for. Det samme gælder for efterfølgende ture.

Som afslutning på beviset bemærkes det, at hvis den første spiller fjerner alle tændstikkerne i den ene stak, så kan spiller nr. to vinde ved at fjerne alle tændstikkerne i den anden stak.

◀

3.5 Andre bevisstrategier

Der er andre metoder for bevisfremførelse. I dette afsnit vil nogle af dem blive beskrevet kort.

Definition 8 *Biimplikation*

Lad s og t være udsagn. Biimplikationen $s \leftrightarrow t$ er udsagnet, der er sandt, når s og t har samme sandhedsværdier og ellers er biimplikationen falsk.

En *biimplikation* $s \leftrightarrow t$ er sand, hvis implikationerne $s \rightarrow t$ og $t \rightarrow s$ begge er sande. For denne siges det, at "s gælder, hvis t også gælder og omvendt".

Nogle gange kan man støde på et sammensat udsagn, som altid er sandt uanset sandhedsværdierne for udsagnene i det sammensatte udsagn. Denne type udsagn kaldes *tautologier* og er defineret som:

Definition 9 *Tautologi*

En tautologi er et udsagn, som er sat sammen af flere udsagn. En tautologi har den egenskab, at den altid er sand uanset sandhedsværdierne for udsagnene, som udgør tautologien.

Nogle gange har man brug for at udnytte, at to udsagn er lig hinanden. Dette er tilfældet, når deres sandhedstabeller er ens. Her kaldes udsagnene for *logisk ækvivalente.*

Definition 10 *Logisk ækvivalens*

Udsagnene p og q siges at være logisk ækvivalente, hvis biimplikationen $p \leftrightarrow q$ er en tautologi. Notationen $p \equiv q$ angiver, at p og q er logisk ækvivalente.

Nogle gange bruges $\Leftrightarrow$ i stedet for $\equiv$, når logisk ækvivalens benævnes.

Falskhedsbeviser og trivielle beviser

Falskhedsbeviser og *trivielle beviser* er teknikker, der tager udgangspunkt i implikationen $p \rightarrow q$. Ved falskhedsbeviser beviser man, at hypotesen p er falsk, og at implikationen derfor altid er sand (se række tre og fire i tabel 3.2). Ved trivielle beviser viser man, at konklusionen q er sand, og at implikationen derfor altid er sand (se række et og tre i tabel 3.2).

p	q	$p \rightarrow q$
S	S	S
S	F	F
F	S	S
F	F	S

Tabel 3.2: Sandheds tabel for implikationen $p \rightarrow q$

Bevis ved eksempler

Nogle gange er det lettere at bevise en implikation $p \rightarrow q$ ved at omskrive p til *disjunktion*, hvis omskrivningen kan lade sig gøre.

Definition 11 *Disjunktion*

Lad p og q være udsagn. Udsagnet "p eller q" er falsk, når både p og q er falske, og udsagnet er sandt ellers. En disjunktion er angivet som $p \vee q$.

Ved at omskrive p opnås implikationen $(p_1 \vee p_2 \vee p_3 \vee \ldots \vee p_n) \rightarrow q$. Denne substitution kan laves, fordi det kan bevises, at udsagnet $s = ((p_1 \rightarrow q) \wedge (p_2 \rightarrow q) \wedge (p_3 \rightarrow q) \wedge \ldots \wedge (p_n \rightarrow q))$ er en biimplikation med $t = ((p_1 \vee p_2 \vee p_3 \vee \ldots \vee p_n) \rightarrow q)$, og at denne er en tautologi.

$$[(p_1 \rightarrow q) \wedge (p_2 \rightarrow q) \wedge (p_3 \rightarrow q) \wedge \ldots \wedge (p_n \rightarrow q)] \leftrightarrow [(p_1 \vee p_2 \vee p_3 \vee \ldots \vee p_n) \rightarrow q]$$

Biimplikationen kan bevises ved at vise, at der blot skal findes ét p fra udsagnet s, der er sandt, for at q i udsagnet s bliver sand. Derfor er s og t logisk ækvivalente.

Et bevis ved eksempler kan derfor bruges til at bevise implikationer på formen:

$$(p_1 \vee p_2 \vee p_3 \vee \ldots \vee p_n) \rightarrow q$$

For bedre at forklare idéen bag denne type beviser, følger vi her op med et "relativt" simpelt eksempel.

Eksempel
Antag at en pakke spillekort er opstillet som et korthus. Kortene er angivet med henholdsvis k_1, k_2, k_3 osv. op til og med det sidste kort k_{52}. f_i er sætningen "Kortet k_i fjernes fra korthuset", hvor $1 \leq i \leq 52$, og q er "Korthuset falder sammen". Så gælder det, at fjernes et hvilket som helst kort fra korthuset, så vælter det. Dette bliver så den sammensatte sætning "Hvis kortet k_i fjernes fra korthuset, så vælter korthuset", som er det samme som implikationen $f_i \rightarrow q$.

Biimplikationen $s \leftrightarrow t$ givet ved $[(f_1 \rightarrow q) \wedge (f_2 \rightarrow q) \wedge (f_3 \rightarrow q) \wedge \ldots \wedge (f_n \rightarrow q)] \leftrightarrow [(f_1 \vee f_2 \vee f_3 \vee \ldots \vee f_n) \rightarrow q]$ viser, at der er to måder at anskue korthuset på:

1. Venstre side af ovenstående biimplikation (dvs. s) viser, at et hvilket som helst kort, der fjernes, individuelt medfører, at korthuset falder sammen. Her fokuseres der på de individuelle kort.

2. Højre side af ovenstående biimplikation (dvs. t) viser, at alle kortene er afhængige af hinanden. Dvs. at lige meget hvilket kort, der fjernes, så falder korthuset sammen.

◀

Eksemplet illustrerer, at det nogle gange kan bevises, at hypotesen p i en implikation $p \rightarrow q$ kan omskrives til en disjunktion, hvorefter det kan bevises, at når et f_i er sandt, så medfører det, at konklusionen q bliver sand.

Ækvivalensbeviser

Vi kan beskrive bevisteknikken med følgende biimplikation:

$$p \leftrightarrow q$$

Hvis en hypotese siger, at adskillige udsagn er ækvivalente, så kan teknikken udvides til at håndtere adskillige udsagn benævnt f_i, hvor $i = 1, 2, 3, \ldots, n$. Dette kan også skrives:

$$f_1 \leftrightarrow f_2 \leftrightarrow f_3 \leftrightarrow \ldots \leftrightarrow f_n$$

Man kan bevise, at alle disse f_i er ækvivalente ved at bruge tautologien:

$$[f_1 \leftrightarrow f_2 \leftrightarrow f_3 \leftrightarrow \ldots \leftrightarrow f_n] \leftrightarrow [(f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_3) \wedge \ldots \wedge (f_n \rightarrow f_1)]$$

Med tautologien kan man vise, at implikationerne $f_1 \rightarrow f_2$, $f_2 \rightarrow f_3$,..., $f_n \rightarrow f_1$ er sande, så er udsagnene f_1, f_2, f_3,..., f_n alle ækvivalente.

Givet et eksempel. Hvor vi lader sætningen "Hvis det er efterår, så falder bladene af træerne" være $p_1 \rightarrow p_2$ og sætningen "Hvis bladene falder af træerne, så ligger der blade på jorden" være $p_2 \rightarrow p_3$ og endelig "Hvis der ligger blade på jorden, så er det efterår" være $p_3 \rightarrow p_1$. Disse sætninger kan beskrives:

$$[f_1 \leftrightarrow f_2 \leftrightarrow f_3] \leftrightarrow [(f_1 \rightarrow f_2) \wedge (f_2 \rightarrow f_3) \wedge (f_3 \rightarrow f_1)]$$

Modstridsbeviser

Da vi bruger *negationslovene* i denne bevisteknik, definerer vi dem her:

Definition 12 *Negationslovene*

For udsagnene p og $\neg p$ gælder:

$$p \vee \neg p \equiv S$$
$$p \wedge \neg p \equiv F$$

Med et *modstridsbevis* forsøger man at bevise, at hypotesen p i implikationen $p \rightarrow q$ er sand ved at vise, at den negerede hypotese $\neg p$ i implikationen $\neg p \rightarrow q$ er falsk. Dette gøres ved at vise, at $\neg p$ medfører to forskellige og modstridende konklusioner r og $\neg r$, og at q derfor kan skrives som $(r \wedge \neg r)$. Da det også gælder, at $(r \wedge \neg r) \equiv F$, så implikationen kan omskrives til r $\neg p \rightarrow F$. For at implikationen kan være sand, skal $\neg p$ være falsk, og derfor må p være sand.

Eksempel
Vi vil lave et modstridsbevis for, at tallet 18 ikke er et primtal. I dette eksempel udgøres hypotesen p af sætningen "18 er ikke et primtal". For at bevise dette med et modstridsbevis negeres p, som så kommer til at være sætningen "18 er et primtal".

At *primtalsfaktorisere* betyder at opskrive et ikke-primtal som produkt af flere primtal (tallet 1 betragtes ikke som primtal).

Nu vises det, at baseret på vores antagelse om, at 18 er et primtal, så må det være sandt, at 18 ikke kan primtalsfaktoriseres svarende til, at $\neg r$ er sand. Men det kan også vises med udregning $2 \times 3 \times 3$, at 18 kan faktoriseres med primtal, som er i modstrid med $\neg r$, og betyder r også er sand.

Dette betyder, at r og $\neg r$ begge skal være sande på samme tid. $r \wedge \neg r$ bliver så omskrevet til F, og implikationen $\neg p \rightarrow F$ opnås. For at implikationen $\neg p \rightarrow F$ bliver sand, skal $\neg p$ være falsk. Dette resulterer i en modstrid, da vores hypotese p derfor også må være sand.

◀

Eksistensbeviser

Under *eksistensbeviser* bruges et *prædikat* til at forklare bevisteknikken, og definitionen følger:

Definition 13 *Prædikat*

Et prædikat P er en funktion, der gør, at udsagnet P(x), hvor x er en variabel, enten bliver sandt eller falskt baseret på værdien af x.

Ligeledes skal *kvantorer* defineres, for at forklare bevisteknikken.

Definition 14 *Kvantorer*

Den universelle kvantor $\forall$ af funktionen $P(x)$ er udsagnet "$P(x)$ er sandt for alle værdier af x".

Den eksistentielle kvantor $\exists$ af funktionen $P(x)$ er udsagnet "Der eksistere et element x for hvilket $P(x)$ er sandt".

Hvis man skal bevise et matematisk udsagn om, at et element af en specifik type findes blandt andre, så kan man bruge et eksistensbevis. Denne type bevis tager udgangspunkt i prædikatet P og antager, at der findes en værdi x, for hvilken $P(x)$ er sand. Beviset går ud på at vise, at dette er tilfældet,

hvilket vises som $\exists x P(x)$. En antagelse som denne kan bevises på flere måder. En måde kaldes *konstruktiv*, og en anden kaldes *ikke-konstruktiv*.

Den konstruktive måde går ud på at bevise, at der er et bestemt element a for hvilket $P(a)$ er sandt.

Den ikke-konstruktive måde beviser, at der findes et element med den ønskede egenskab uden at præcisere, hvilket element det er. Dette kan f.eks. gøres med et modstridsbevis ved at bevise, at $\neg(\exists x P(x))$ giver en modstrid, for det betyder så, at der virkelig findes et element af den type, som man leder efter.

Forskellen mellem den konstruktive og den ikke-konstruktive måde er, at den konstruktive beviser, at elementet eksisterer ved at finde det. Den ikke-konstruktive beviser blot, at elementet skal være der.

Unikthedsbeviser

Med et unikthedsbevis forsøger man at bevise, at der findes netop ét element blandt andre elementer, som er unikt, da det har en egenskab, som alle andre elementer ikke har.

Derfor består et bevis af denne type af to dele:

1. Bevis at der findes et element x med den eftersøgte egenskab $P(x)$ (opsummeret i $\exists x(P(x))$).

2. Bevis at alle andre elementer ikke har egenskaben $P(x)$ (opsummeret i $\forall y((y \neq x) \rightarrow \neg P(y))$).

Dette bevis kan beskrives på følgende måde:

$$\exists x(P(x) \wedge \forall y((y \neq x) \rightarrow \neg P(y)))$$

Ovenstående påstand skal tolkes som "Der findes et x, for hvilket det gælder, at $P(x)$ er sandt. Det gælder samtidig for alle y, som ikke er lig x, at $P(y)$ er falsk.

Beviser med mod-eksempler

Et mod-eksempel er et specifikt tilfælde, hvor et udsagn $P(x)$ er falsk. Mod-eksempler bruges derfor til at bevise, at et udsagn af formen $\forall x P(x)$ er falsk. Dette kan gøres, hvis der kan findes bare ét eksempel, hvor $P(x)$ er falsk. Når man støder på et udsagn af ovennævnte form, som man ikke har kunnet finde et bevis eller modbevis for, så kan man prøve at finde et mod-eksempel.

Kapitel 4

Mængder

I dette kapitel vil vi forklare, hvad mængder er, og hvordan man kan manipulere og kombinere dem for at danne nye *mængder*. Dette kan f.eks. benyttes, når man skal søge efter bestemte elementer i forskellige mængder. Da vi ved, at TSP indeholder mange elementer både i form af mulige ruter og antal byer, er dette yderst relevant for projektet.

4.1 Hvad er mængder

For at komme ordentligt i gang med mængder starter vi med at definere, hvad en mængde er.

Definition 15 *Mængde*

En mængde er en usorteret samling af elementer. En mængde siges også at indeholde elementer.

Mængder bruges i matematikken til at gruppere elementer. Inden for mængdeteori opererer man med begreberne *relationer* og *kombinationer*. Relationer er sorterede par af elementer, hvorimod kombinationer er usorterede samlinger af elementer. Vi vil i dette afsnit fokusere på *endelige mængder*, da *uendelige mængder* ikke er relevant for vores projekt. For at få en præcis

definition af, hvad en endelig og en uendelig mængde er, viser vi definitionen af disse.

Definition 16 *Endelig mængder og uendelige mængder*

En endelig mængde er en mængde, hvis elementer kan stilles op i listeform. En mængde siges at være uendelig, hvis den ikke er endelig.

For at lave et simpelt eksempel på dette, forestiller vi os, at alle biler i Danmark udgør én mængde A. En anden mængde B er alle de blå biler i verden. Hvis man vil have en liste over alle blå biler i Danmark, kan man lave en ny mængde C bestående af alle de elementer, som både er i mængderne A og B. Dette kaldes for en *delmængde*.

Nogle andre eksempler på matematiske uendelige mængder:

$\mathbf{N}$	$=$	$\{0, 1, 2, 3, \ldots\}$	Mængden af naturlige tal
$\mathbf{Z}$	$=$	$\{\ldots, -2, -1, 0, 1, 2, \ldots\}$	Mængden af alle heltal
$\mathbf{Q}$	$=$	$\{p/q \mid p \in \mathbf{Z}, q \in \mathbf{Z}, q \neq 0\}$	Mængden af rationelle tal
$\mathbf{R}$	$=$		Mængden af reelle tal.

Der findes derudover også den tomme mængde, angivet med $\emptyset$ eller $\{\}$, og den universelle mængde U, der beskriver alle elementerne, som er relevante.

4.2 Mængder og delmængder

Da uendelige mængder ikke kan skrives op på listeform, kan man bruge følgende form for at udtrykke kravene til mængdens elementer:

$$\mathbf{R} = \{x \mid x \text{ er et reelt tal } \}$$

Udtrykket læses som elementet x tilhører mængden R, hvis x er et reelt tal.

Definition 17 *Delmængde*

Mængden A er en delmængde af mængden B, hvis alle elementerne i A også er i B. Notationen for dette er $A \subseteq B$.

For at vise at to mængder A og B indeholder de samme elementer, kan man på følgende måde vise, at de er delmængder af hinanden, og implikation nedenfor viser sammenhængen:

$$((A \subseteq B) \wedge (B \subseteq A)) \rightarrow (A = B)$$

Det vil sige, hvis A er en delmængde af B og B er en delmængde af A så er A og B den samme mængde.

Definition 18 *Ægte delmængde*

Hvis A og B er mængder, og A er en ægte delmængde af B, så skrives dette $A \subset B$. Hvis A er en ægte delmængde af B, så kan B ikke være en delmængde af A, og derfor vil det gælde, at $A \neq B$.

Inden for mængdeteori bruger man også et begreb, der hedder det *kartetiske produkt*. Det kartetiske produkt er alle tænkelige produkter af to mængder A og B. Det er defineret som:

Definition 19 *Kartetisk produkt*

Hvis A og B er mængder, så er det kartetiske produkt angivet ved $A \times B$, som består af alle ordnede par (a, b), hvor $a \in A$ og $b \in B$. Derfor gælder det, at

$$A \times B = \{(a, b) \mid a \in A \wedge b \in B\}$$

En delmængde R af det kartetiske produkt $A \times B$ kaldes en relation fra mængden A til mængden B. Elementerne i R er sorterede par, og disse par er opstillet således, at elementerne fra A står først, og elementerne fra B står sidst.

Eksempelvis kunne det gælde for mængderne A og B, at $A = \{a, b, c\}$ og $B = \{1, 2, 3\}$. Så ville det kartetiske produkt $R = A \times B$ være parrene i tabel 4.1.

	1	2	3
a	$(a, 1)$	$(a, 2)$	$(a, 3)$
b	$(b, 1)$	$(b, 2)$	$(b, 3)$
c	$(c, 1)$	$(c, 2)$	$(c, 3)$

Tabel 4.1: Det kartetiske produkt $A \times B$

4.3 Operationer på mængder

Vi definerer, hvad en *foreningsmængde* og en *fællesmængde* er, da vi i dette afsnit bruger disse til at forklare hvordan man manipulerede mængder.

Definition 20 *Foreningsmængde*

Lad A og B være mængder. Så skrives foreningsmængden af A og B som $A \cup B$. Foreningsmængden er de elementer, der er i A og dem der er i B. Det gælder derfor, at $A \cup B = \{x \mid x \in A \vee x \in B\}$.

Foreningsmængden bruges til at sammenlægge mængder

Definition 21 *Fællesmængde*

Lad A og B være mængder. Så skrives fællesmængden af A og B som $A \cap B$. Fællesmængden er alle de elementer, som både er i A og i B. Det gælder derfor, at $A \cap B = \{x \mid x \in A \wedge x \in B\}$.

Fællesmængden bruges til at finde de elementer, der er fælles for to mængder. Hvis to mængder A og B har en tom fællesmængde, siges mængderne at være disjunkte. Dette kan angives som $A \cap B = \emptyset$.

Vi vil følgende definere, hvad *differencen* mellem to mængder A og B er, samt hvad *komplimentet* til en mængde er.

Definition 22 *Difference*

Lad A og B være mængder. Så skrives difference som $A - B$. Dette beskriver de elementer, der er med i mængden A, og som ikke er i mængden B. Det gælder så, at $A - B = \{x \mid x \in A \wedge x \notin B\}$.

Definition 23 *Komplement*

Lad U være alle mulige elementer i A. Så skrives A's komplement som $\bar{A}$. Dette er de elementer i U, der ikke er indeholdt i mængden A. Dette er differencen $U - A = \bar{A}$.

Et element x tilhører derfor den komplementære mængde $\bar{A}$, hvis x ikke er i mængden A. Dette forhold angives:

$$\bar{A} = \{x \mid x \notin A\}$$

En teknik til at bevise at to mængder A og B er lig med hinanden, går ud på at vise, at A er indeholdt i B, og at B samtidig er indeholdt i A.

Når man arbejder med operationer på mere end to mængder, så skal man bruge nogle specielle symboler for at skrive udtryk op på en kort form. Vi giver herunder et par definitioner af foreningsmængden og fællesmængden, når der arbejdes med mere end to mængder.

Definition 24 *Den generaliserede foreningsmængde*

Foreningsmængden af en samling af mængder er de elementer, der mindst er med i én af mængderne i samlingen. Dvs. alle elementerne i samlingen af mængder.

Den generaliserede foreningsmængde for mængderne A_1, A_2 og A_3 er angivet ved:

$$A_1 \cup A_2 \cup A_3 = \bigcup_{i=1}^{3} A_i$$

Definition 25 *Den generaliserede fællesmængde*

Fællesmængden af en samling af mængder er de elementer, der er med i alle mængderne i samlingen.

Den generaliserede fællesmængde for mængderne A_1, A_2 og A_3 er angivet ved:

$$A_1 \cap A_2 \cap A_3 = \bigcap_{i=1}^{3} A_i$$

Her vises hvad den generaliserede foreningsmængde og fællesmængde er for $A_i = \{i, i+1, i+2, \ldots\}$.

Foreningsmængden angives ved:

$$\bigcup_{i=1}^{n} A_i = \bigcup_{i=1}^{n} \{i, i+1, i+2, \ldots\} = \{1, 2, 3, \ldots\}$$

Foreningsmængden vil altid indeholde alle tal, da alle tal fra ét af og op efter er indeholdt i mængden A_1.

Fællesmængden angives ved:

$$\bigcap_{i=1}^{n} A_i = \bigcap_{i=1}^{n} \{i, i+1, i+2, \ldots\} = \{n, n+1, n+2, \ldots\}$$

Det gælder for fællesmængden i dette eksempel, at den kun vil indeholde talene fra den mindste mængde, som her er den sidste mængde A_n. Da denne mængdes elementer starter fra n og går mod uendelig, vil fællesmængden kun indeholde $\{n, n+1, n+2, \ldots\}$.

Kapitel 5

Grafteori

Grafteori er medtaget i dette projekt, fordi det er en meget brugt måde at anskueliggøre TSP problematikken på. Med grafer kan man modellere og visualisere ens problemer, og derved får man et godt overblik over det pågældende problem.

En *graf* er en matematisk konstruktion, som består af en eller flere punkter kaldet *knuder*. Hver knude har en eller flere egenskaber, som gør hver den unik. I en graf kan der eksistere forbindelser mellem knuder. Dette kaldes for *kanter*. Hvis der er en kant mellem to knuder, kaldes knuderne for naboknuder. Knudernes egenskaber og kanterne i grafen definerer, hvad den skal illustrere. En graf består derfor af en mængde knuder og en mængde kanter.

Definition 26 *Naboknuder*

Hvis V er mængden af knuder, og E er mængden af kanter, da er de to knuder u og v i grafen $G = (V, E)$ tilstødende eller naboer, hvis kanten $\{u, v\} \in E$.

Herunder vil vi gennemgå forskellige typer af grafer, der har forskellige karakteristika.

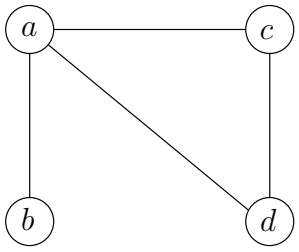
Simpel graf

Man kunne forestille sig en omgangskreds bestående af en mængde personer (knuder) med den egenskab, at ikke alle kender hinanden. Mængden af knuder i grafen udgøres af a, b, c og d. Mængden af kanter mellem disse knuder fremgår af tabel 5.1.

	a	b	c	d
a		x	x	x
b	x			
c	x			x
d	x		x	

Tabel 5.1: En række knuder hvor kanter imellem dem er markeret med "x".

Ud fra mængden af knuder og mængden af kanter kan vi fremstille figur 5.1.



Figur 5.1: Simpel graf

Definition 27 *Simpel graf*

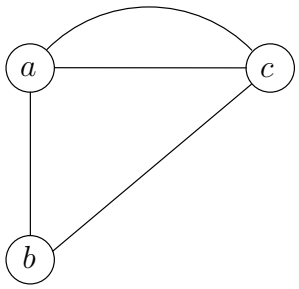
Hvis V er en mængde af knuder, som ikke er tom, og E er en mængde af uordnede par af knuder fra V kaldet kanter, da er en simpel graf givet ved $G = (V, E)$.

Multigraf

Ved at give grafen mulighed for at forbinde de samme knuder med flere kanter, fremkommer en multigraf. Denne er kendetegnet ved, at der mellem to knuder er mulighed for at vælge mellem flere kanter for at komme fra knude til knude. Et eksempel er vist på figur 5.2.

	a	b	c
a		x_1	x_2
b	x_1		x_1
c	x_2	x_1	

Tabel 5.2: En række knuder hvor kanter imellem dem er markeret med "x_i", hvor i giver antallet af kanter.



Figur 5.2: Eksempel på multigraf

Definition 28 *Multigraf*

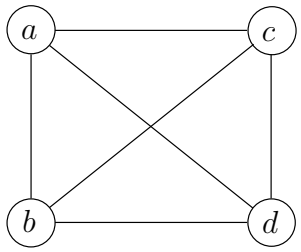
En multigraf $G = (V, E)$ består af V knuder, E kanter og funktionen f fra E til $\{\{u, v\} \mid u, v \in V, u \neq v\}$. Kanterne e_1 og e_2 fra E kaldes multiple eller parallelle kanter, hvis $f(e_1) = f(e_2)$.

Komplet graf

I de tilfælde, hvor alle knuder har en kant til alle andre knuder, fremkommer en komplet graf, som vist på figur 5.3.

	a	b	c	d
a		x	x	x
b	x		x	x
c	x	x		x
d	x	x	x	

Tabel 5.3: En række knuder hvor kanter imellem dem er markeret med "x".



Figur 5.3: Eksempel på en komplet graf

Definition 29 *Komplet graf*

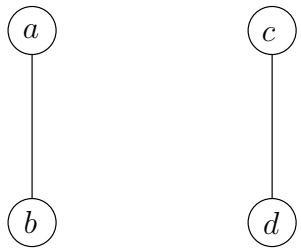
En komplet graf $G = (V, E)$ bestående af mængden $V \neq \emptyset$ og E, som er en fulstændig mængde af knuder fra V, der forbinder alle knuder i V med hinanden.

Sammenhængende grafer

En graf, hvor alle knuder er forbundne med resten af grafen, kaldes sammenhængende. Er dette ikke tilfældet, som følge af at en eller flere knuder ikke er forbundne med resten af grafen, kaldes grafen for ikke-sammenhængende. Oftest vil det være lettest at betragte en ikke-sammenhængende graf som en mængde sammenhængende grafer. Dette er vist på figur 5.4.

	a	b	c	d
a		x		
b	x			
c				x
d			x	

Tabel 5.4: En række knuder hvor kanter imellem dem er markeret med "x".



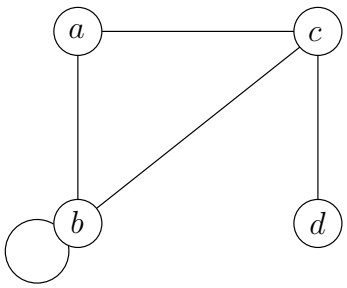
Figur 5.4: Eksempel på en ikke-sammenhængende graf

Pseudograf

En pseudograf er en graf, hvor en eller flere knuder har en eller flere kanter til sig selv, kaldet loop(s). Et eksempel på en pseudograf er vist i figur 5.5.

	a	b	c	d
a		x	x	
b	x	x	x	
c	x	x		x
d			x	

Tabel 5.5: En række knuder hvor kanter imellem dem er markeret med "x".



Figur 5.5: Eksempel på pseudograf

Definition 30 *Pseudograf*

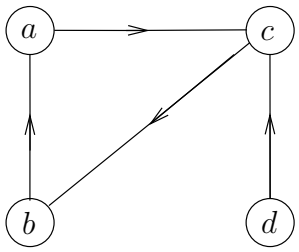
En pseudograf $G = (V, E)$ består af en mængde V af knuder, en mængde E af kanter samt funktionen f fra E til $\{\{u, v\} \mid u, v \in V\}$. En kant e er et loop, hvis $f(e) = \{u, u\} = \{u\}$ for et $u \in V$.

Orienteret graf

En orienteret graf er karakteriseret ved, at der er grænser for, hvilke kanter der må benyttes. Når der fra en given knude skal vælges en kant, må kun de kanter vælges, som er repræsenterede med en pil væk fra den nuværende kant. Et eksempel på en orienteret graf er vist i figur 5.6. Som med en simpel graf, må der ikke forekomme loops i grafen.

	a	b	c	d
a			x	
b	x			
c		x		
d			x	

Tabel 5.6: En række knuder hvor kanter imellem dem er markeret med "x".



Figur 5.6: Eksempel på en orienteret graf

Definition 31 *Orienteret graf*

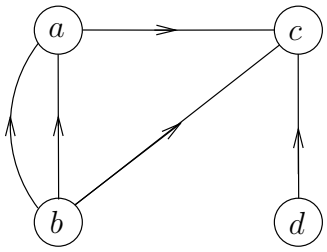
En orienteret graf $G = (V, E)$ består af en mængde V af knuder, og en mængde E af ordnede par af knuder fra V.

Muteret graf

Ved at kombinere flere af de grafer, som er blevet vist i figur 5.1 – 5.6, kan der laves flere forskellige typer mutationer af grafer. Et eksempel er vist i figur 5.7, hvor princippet for orienteret graf er blevet kombineret med princippet for en multigraf.

	a	b	c	d
a			x_1	
b	x_2		x_1	
c				
d			x_1	

Tabel 5.7: En række knuder hvor kanter imellem dem er markeret med "x_i", hvor i er antallet af kanter.



Figur 5.7: Eksempel på en orienteret multigraf

Definition 32 *Orienteret multigraf*

En orienteret multigraf $G = (V, E)$ består af en mængde V af knuder, en mængde E af kanter samt funktionen f fra E til $\{(u, v) \mid u, v \in V\}$. To kanter e_1 og e_2 er multiple kanter, hvis $f(e_1) = f(e_2)$.

Graden af en knude

Enhver knude i grafen G har en grad, der betegner, hvor mange naboer knuden har. Hvis f.eks. graden af en knude er n, betyder dette, at knuden har n naboer.

Definition 33 *Graden af en knude*

Graden af en knude i en ikke-orienteret graf er antallet af tilstødende kanter til den, hvor et loop vil bidrage med to. Graden af en knude v betegnes ved $deg(v)$.

Dette vil betyde, at $deg(a)$ på figur 5.8 er tre, og $deg(b)$ er en.

Vægtet graf

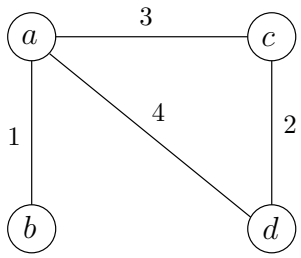
Tidligere i afsnittet blev sammenhængen mellem to knuder a og b, altså kanten, defineret som værende enten eksisterende eller ikke-eksisterende. For en vægtet graf gælder det ydermere, at kanterne hver får tildelt en værdi, kaldet en vægt. Denne værdi skrives $w(a, b)$ i tekst eller på kanten som vist i figur 5.8. I tabel 5.8 er illustreret, hvorledes dette kan laves.

	a	b	c	d
a		1	3	4
b	1			
c	3			2
d	4		2	

Tabel 5.8: En række knuder hvor kanter imellem dem er markeret med kantens vægt.

Sti og kreds

Nogle gange kan det være praktisk at bevæge sig fra knude til knude via kanterne. F.eks. hvis man skal finde ud af, om der er forbindelse mellem to



Figur 5.8: Eksempel på en vægtet graf

computere i et netværk. Dette kaldes for en *sti*. Hvis man danner en sti, som starter og slutter i samme knude, danner man en *kreds*.

Definition 34 *Sti og kreds*

Lad n være et heltal, som ikke er negativt, og lad G være en graf, der ikke er orienteret. En sti af længde n fra knude u til knude v i G er da en sekvens af n kanter $e_1, e_2, ..., e_n$ fra G, således at $f(e_1) = \{x_0, x_1\}, f(e_2) = \{x_1, x_2\}, \ldots, f(e_n) = \{x_{n-1}, x_n\}$, hvor $x_0 = u$, og $x_n = v$. Når grafen er simpel, angives stien med sin sekvens af knuder $x_0, x_1, \ldots, x_n$.

En sti bliver til en kreds, hvis den starter og slutter i samme knude, således at $u = v$ samtidig med, at den har en længde større end nul. En sti eller kreds siges at passere gennem knuderne $x_1, x_2, \ldots, x_{n-1}$, eller siges at traversere kanterne $e_1, e_2, \ldots, e_n$.

Simpel sti og simpel kreds

En sti eller kreds er *simpel*, hvis den ikke indeholder den samme kant mere end en gang.

Sætning 1 *Simpel sti*

Der eksisterer en simpel sti imellem alle tænkelige par af unikke knuder i en sammenhængende ikke-orienteret graf G.

Bevis for sætning 1
Lad u og v være to forskellige knuder i den sammenhængende ikke-orienterede graf G. Idet grafen er sammenhængende, er der mindst én sti imellem u og v. For at bevise at dette er sandt, bruges et modstridsbevis. Når man skal bevise sætningen for en simpel sti med et modstridsbevis (se afsnit 3.5), starter man med at antage, at den korteste sti i grafen ikke er simpel. Denne antagelse vil føre frem til to modstridende konklusione.

Lad knuderne $x_0, x_1, \ldots, x_n$ være følgen af knuder i en sti med minimum længde, hvor $x_0 = u$ og $x_n = v$. Derved følger, at $x_i = x_j$ for nogle i og j, hvor det gælder, at $0 \leq i < j$. Derfor er x_i og x_j en og samme knude.

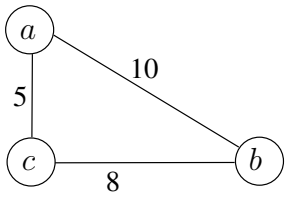
Dvs., at der er en kortere sti $x_0, x_1, \ldots, x_{i-1}, x_j, \ldots, x_n$ fra u til v end følgen af knuder $x_0 \ldots, x_n$, som opnås ved at slette kanten tilhørende følgen af knuder $x_i, \ldots, x_{j-1}$. Med andre ord vil det sige, at man ved at fjerne knuderne $x_i, \ldots, x_{j-1}$ vil få en sti, som er kortere, end den sti, man antog, var den mindste.

Det, at antagelsen nu medfører, at der i grafen både er en minimum sti og en sti, der er endnu mindre, betyder, at antagelsen må være forkert, og derfor må der nødvendigvis være en simpel sti i grafen.

$\triangleleft$

5.1 Trekantsuligheden

Trekantsuligheden tager udgangspunkt i et kordinatsystem med grafen $G = (V, E)$, hvor V er mængden af knuder og E er mængden af kanter. Den siger, at den direkte vej mellem to knuder maximalt er af samme længde eller mindre end den ville være, hvis man gik over en tredje knude. Matematisk set kan dette beskrives som $w(a, b) \leq w(a, c) + w(c, b)$, hvor a, b og c er punkter fra mængden V.



Figur 5.9: Eksempel på trekantsuligheden

Såfremt G er en komplet graf kan trekantsuligheden benyttes.

Sætning 2 *Trekantsuligheden*

For en komplet graf G, der indeholder punkterne $x_1, x_2, \ldots, x_{n-1}, x_n$, hvor $n \geq 3$ gælder, at $w(x_1, x_n) \leq w(x_1, x_2) + w(x_2, x_3) + \ldots + w(x_{n-1}, x_n)$.

Ved hjælp af et induktionsbevis vil vi bevise trekantsuligheden.

Bevis for sætning 2
Basisskridt: $w(x_1, x_3) \leq w(x_1, x_2) + w(x_2, x_3)$ er sandt jf. figur 5.9. Ved at indsætte værdierne fra figur 5.9 fåes: $10 \leq 5 + 8 = 13$, hvilket jo er sandt.

Induktionsskridt: Der antages, at

$w(x_1, x_n) \leq w(x_1, x_2) + w(x_2, x_3) + \ldots + w(x_{x-1}, x_n)$ gælder for en komplet graf med n kanter.

Induktionsskridtet består så af at bevise, at det også gælder for $n + 1$, altså

$$w(x_1, x_{n+1}) \leq w(x_1, x_2) + w(x_2, x_3) + \ldots + w(x_{n-1}, x_x) + w(x_n, x_{n+1})$$

Ifølge trekantsuligheden gælder følgende:

$$w(x_1, x_{n+1}) \leq w(x_1, x_2) + w(x_2, x_3) + \ldots + w(x_n, x_{n+1})$$

Det er hermed bevist, at trekantsuligheden også gælder ved kreds med mere

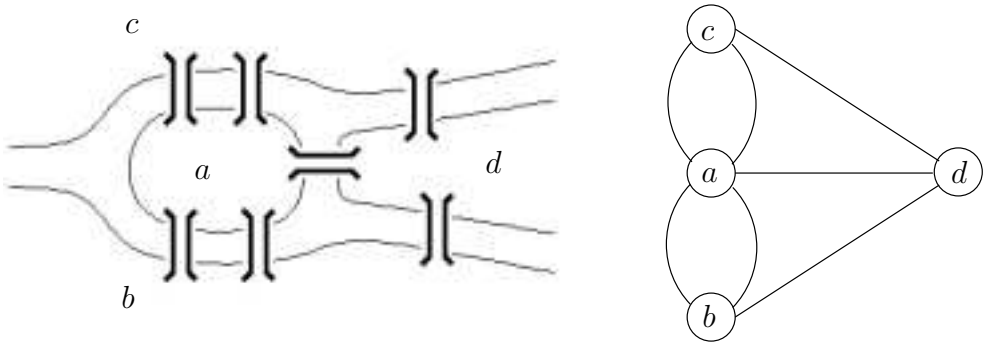
end 3 kanter. Vægten af den direkte vej, vil altså blive mindre end eller lig med summen af de resterende kanter i kredsen.

◁

5.2 Euler-kredse og -stier

En af de kendte problemer, som løses vha. grafteori, er broerne i Königsberg. Byen er delt i fire bydele af floden Pregel. Dette er illustreret i venstre side af figur 5.10. Problemet består i, at beboerne i byen ville gå hen over broerne hver søndag og kun passere hver bro én gang samt slutte på samme side, som de begyndte.

I 1736 løste den schweiziske matematiker Leonhard Euler (1707-1783) dette problem. Euler brugte en multigraf (se højre side af figur 5.10), hvor de fire regioner og broer var repræsenteret ved hjælp af hhv. knuder og kanter. [7]



Figur 5.10: Broerne i Königsberg

Det kan være hensigtsmæssigt at finde en sti i en graf, som besøger alle kanter netop én gang. Dette kaldes for en *Euler-sti*. Det kan også være praktisk at besøge alle kanter i en graf og slutte i den samme knude, som man startede i. Dette kaldes for en *Euler-kreds*. Euler-stier og Euler-kredse er definerede som følger.

Definition 35 *Euler-kreds og Euler-sti*

En Euler-sti i en graf G er en simpel sti, som indeholder alle kanter i G.
En Euler-kreds i en graf G er en simpel kreds, som indeholder alle kanter i G, og hvor startknuden er lig slutknuden.

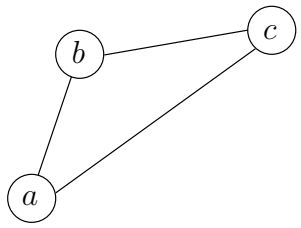
Leonhard Euler opdagede, at der findes to betingelser, der afgør, hvorvidt en graf med et endeligt antal kanter har en Euler-kreds eller en Euler-sti.

Nødvendige betingelser for Euler-kredse

Den første betingelse for at en forbundet multigraf G har en Euler-kreds er, at graden af alle knuder skal være lige.

Lad knuden x_0 være en del af grafen G, hvor G har et endeligt antal kanter. For at opbygge en kreds, tages der udgangspunkt i kanten, som går fra x_0 til x_1. Ved efterfølgende at fortsætte denne metode opbygges den simple sti, som går langs kanterne $\{x_0, x_1\}, \{x_1, x_2\}, \ldots, \{x_{n-1}, x_n\}$.

Et eksempel på en Euler-kreds kunne være, at der er en graf med tre knuder a, b, c, så stien med kanterne $\{a, b\}, \{b, c\}$ og $\{c, a\}$ opnås. Dette eksempel er illustreret i figur 5.11.



Figur 5.11: Trekant

Da grafen er endelig, vil et gennemløb af denne blive afsluttet. Samtidig kan det garanteres, at startknuden også vil være slutknuden. Dette underbygges af, at da graden af en knude er lige, vil opbygningen af kredsen bruge én af den givne knudes kanter til at komme til knuden og den anden kant til at komme væk fra knuden igen.

Sætning 3 *Forbundet multigraf med Euler-kreds*

En forbundet multigraf har en Euler-kreds, hvis og kun hvis graden af alle knuder er lige.

Bevis for sætning 3
En graf G har en Euler-kreds, hvis graden af alle knuderne i G er lige. Antag at der findes en graf G med en Euler-kreds, og at en knude v har ulige grad. Hvis man starter i v, vil man ikke kunne slutte i denne, da man skal bruge et lige antal kanter for at gøre dette (en kant ud fra knuden og en ind til den igen).

Hvis man ikke starter i v, skal man besøge denne på et tidspunkt, og man vil også slutte i denne (eksempelvis en kant ind til knuden, en ud fra den igen, og en ind til den igen). Dette medfører altså, at der findes en Euler-graf, hvis graden af alle knuder $v \in G$ er lige.

$\triangleleft$

Med sætning 3 kan man argumentere for, at broerne i Königsberg ikke har en Euler-kreds (se evt. figur 5.10), da graden af alle knuder ikke er lige.

Nødvendige betingelser for Euler-stier

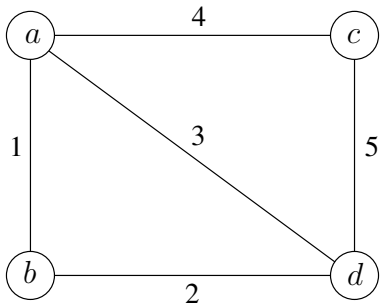
Den anden betingelse, som Leonhard Euler opdagede, var, at en forbundet multigraf, som indeholder præcis to knuder med ulige grad, ikke kan have en Euler-kreds, men kan have en Euler-sti.

Når man ved opbygning af en sti eksempelvis bevæger sig fra knude a til knude b, som ikke nødvendigvis er naboknuder, bruges en sti fra knuden a til knuden b. Herefter vil enhver passering gennem a og/eller b medføre, at der skal bruges yderligere to kanter. Konsekvensen af dette er, at både a og b er ulige.

Sætning 4 *Euler-sti*

En forbundet multigraf har en Euler-sti, men ikke en Euler-kreds, hvis og kun hvis den har præcis to knuder med ulige grad.

Bevis for sætningen 4
Denne bevises ud fra det eksempel, som er illustreret i figur 5.12.



Figur 5.12: Eksempel på Euler-sti

Hvis man i figur 5.12 skal lave en Euler-sti og vælger at starte i a, kan man eksempelvis lave stien a, b, d, a, c og til sidst d. Knuderne a og d har ulige grad. Idet deres grader er ulige, vil man, når man starter i a, ikke kunne slutte i denne knude.

I eksemplet er $deg(a) = 3$. Da der startes i a, vil der blive brugt en kant på at komme væk fra knuden og en på at komme tilbage dertil. Herved er der kun én kant tilbage, og da et krav er, at alle kanter skal besøges netop én gang, er man nødt til at bruge denne for at komme væk fra knuden igen. I d er det så lige modsat. Man vil derfor, såfremt man starter i a og skal lave en Euler-sti, slutte i d i figur 5.12.

$\triangleleft$

5.3 Hamilton-kredse og -stier

Hamilton-sti

Vi har tidligere i dette kapitel beskrevet kredse og stier i simple grafer, hvor alle kanterne i en graf skulle besøges. Her vil vi beskrive *Hamilton-stier* og *Hamilton-kreds*, som bygger på, at det i stedet for kanter er alle knuder i en graf, der skal besøges. Vi lægger ud med definitionerne for disse kredse og stier.

Definition 36 *Hamilton-sti*

En Hamilton-sti i grafen $G = (V, E)$ *er betegnet* $x_0, x_1, \ldots, x_{n-1}, x_n$, *hvis* $V = \{x_0, x_1, \ldots, x_{n-1}, x_n\}$ *og* $x_i \neq x_j$ *for* $0 \leq i < j \leq n$.

Hamilton-kredse

Meget lig en Euler-kreds findes en Hamilton-kreds. Denne baserer sig på, at alle knuder i grafen G skal besøges én gang, hvorefter der returneres til udgangsknuden. Løsningen til dette problem er det centrale i denne rapport og vil blive uddybet senere.

Definition 37 *Hamilton-kreds*

En Hamilton-kreds i grafen $G = (V, E)$ *er betegnet* $x_0, x_1, \ldots, x_{n-1}, x_n, x_0$ *(for* $n > 1$*), hvis* $x_0, x_1, \ldots, x_{n-1}, x_n$ *er en Hamilton-sti.*

Hvis man har fundet en Hamilton-kreds i en graf, kan man tilføje kanter til grafen og stadig have en Hamilton-kreds. Dette gør sig gældende, da kredsen

vil forblive den samme, selv om der kommer flere kanter (man bruger dem blot ikke). Sætning 5 bruges til at bevise, at en graf G med mere end to knuder har en Hamilton-kreds, hvis graden af alle knuder v er større end halvdelen af antallet af knuder. Hvis der eksempelvis er seks knuder i grafen, skal graden af hver knude mindst være tre.

Sætning 5 *Dirac's Sætning*

En graf G med n knuder ($n \geq 3$) har en Hamilton-kreds, hvis $deg(v) \geq \frac{n}{2}$.

Sætning 6 *Ore's Sætning*

En graf G med n knuder ($n \geq 3$) har en Hamilton-kreds, hvis $deg(v) + deg(u) \geq n$, hvor $(u, v \in E)$.

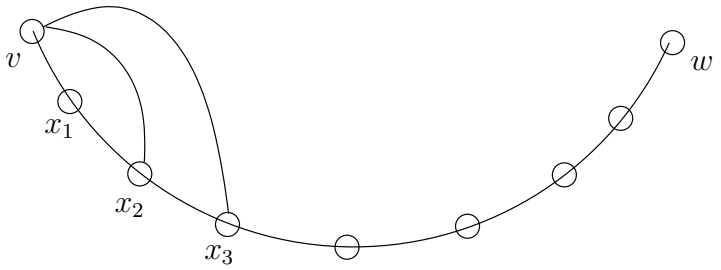
Der er en sammenhæng mellem Dirac's Sætning og Ore's Sætning. Dirac's Sætning kan bevises i forbindelse med Ore's Sætning, da Dirac's betingelser er underforstået i Ore's Sætning.

Bevis for sætning 6
Denne bevises med et modstridsbevis. Antag at $deg(v) + deg(w) \geq n$, hvor $n \geq 3$, for alle v og w, som ikke er naboer. Antag også at G ikke har en Hamilton-kreds.

Ved at tage udgangspunkt i grafen G og tilføje det maximale antal kanter til G, således at der ikke findes en Hamilton-kreds for G, må der findes to knuder v og w, som ikke er naboer. For disse gælder der, at deres samlede grad er større end eller lig med n. Det må desuden gælde, at der findes en Hamilton-sti mellem dem.

Sæt knuden $v = x_0$ og knuden $w = x_{n-1}$. Som følge af at der er n knuder, og fordi der startes i x_0, bliver w til x_{n-1}. Sæt dernæst $deg(v) = r$. Dette medfører, at $deg(w)$ må være større end eller lig med antallet af knuder minus

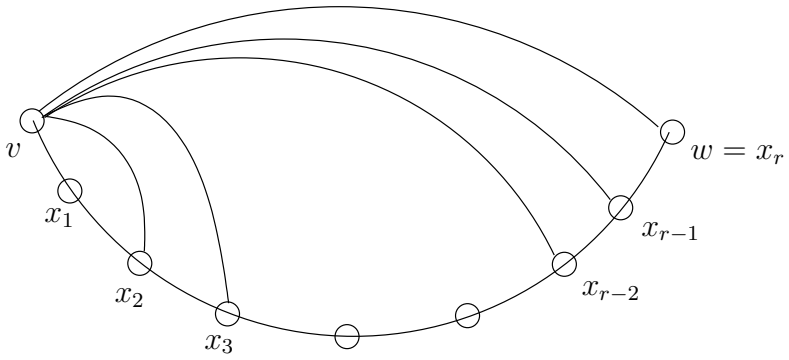
r, altså $deg(w) = n - r$. Dette er gældende, fordi $deg(v) + deg(w) \geq n$. Se evt. figur 5.13.



Figur 5.13: Illustration til Ore's Sætning.

Da w blandt andet har naboerne x_{n-2} og x_{n-3}, kan nabo-knuderne til w skrives som x_{n-1-q}, hvor $q \geq n$. Idet graden af w er $n - r$ fåes, at w's nærmeste nabo er $x_{n-1-(n-r)} = x_{r-1}$.

v har som nævnt r naboknuder, nemlig x_1 til x_r. Dermed er v også nabo til x_{r-1} (se evt. figur 5.14), hvilket w også er. Med andre ord, så eksisterer der en Hamilton-kreds, hvilket er en modstrid i henhold til vores antagelse.



Figur 5.14: Illustration til Ore's Sætning.

◁

Kapitel 6

Træteori

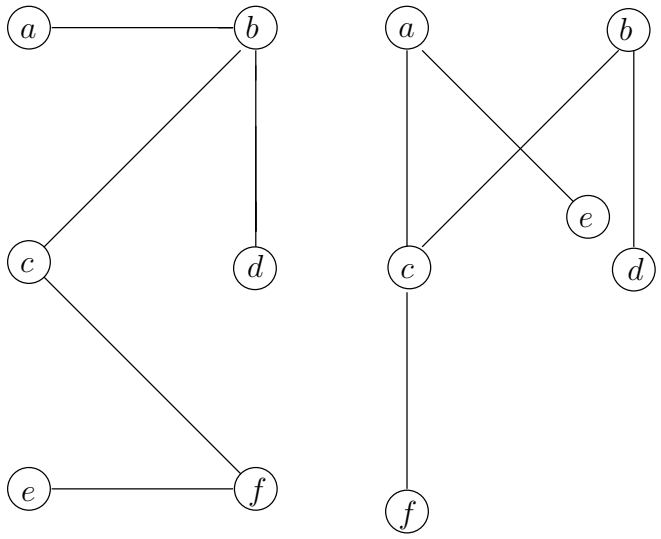
I dette kapitel vil vi beskæftige os med *træer* og teorien, der knytter sig til disse. Vi har valgt at beskæftige os med træer, fordi træer er en naturlig fortsættelse af grafer. Træer er en bestemt type af grafer. Vi har desuden valgt at beskæftige os med træer, idet træer har den egenskab, at de kan bruges til at konstruere forskellige algoritmer, hvilket vi vil se senere i vores rapport.

Definition 38 *Træ*

Et træ er en sammenhængende, ikke-orienteret simpel graf uden kredse.

Skov

Hvis en graf består af to eller flere forbundne træer, hvor imellem der ikke er en forbindelse, kaldes grafen en *skov*.



Figur 6.1: Eksempler på to træer i en skov

Rodtræer

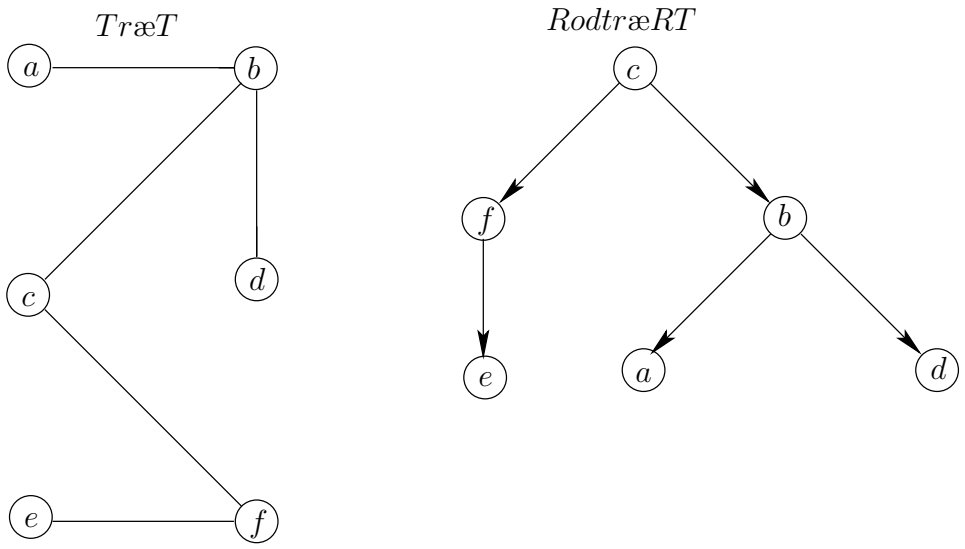
Definition 39 *Rodtræ*

Et træ kaldes et rodtræ, hvis en knude bestemmes til at være en rodknude og alle kanter i træet går ud fra denne knude, således at træets kanter i et rodtræ har en orientering.

Når rodtræer omtales og illustreres vha. figurer i denne rapport, er der altid tale om, at rodtræernes kanter altid har en orientering (fra *forælder* til *barn*); også selv om dette ikke eksplicit nævnes.

Relationer mellem knuder

Knuder i et træ benævnes forskelligt alt efter, hvor i træet en knude befinder sig. Roden er som allerede nævnt den øverste knude i et rodtræ. Roden er samtidig forælder til de knuder, hvortil der fra roden går en orienteret kant. Disse knuder benævnes rodens børn. Sagt på en anden måde: Hvis en knude



Figur 6.2: Knuden c i træet T vælges som rodknude og former derved rodtræet RT

v er forælder til en knude w, er w barn til v.

Knuder med samme forælder kaldes *søskende*, og en knudes forælder, samt dennes forælder, osv. (op til rodknuden) kaldes *forfædre* (ancestors). En knude v's *efterkommere* er alle de knuder, som har knuden v som forfader.

En knude i et træ kaldes et *blad*, hvis knuden ingen børn har. Knuder med børn kaldes *interne knuder*.

Deltræer

Et træ T kan inddeles i *deltræer*. Man kan lade en knude v i træet T være rod i dens eget deltræ UT. Således udgøres deltræet UT af rodknuden v samt alle dens efterkommere. Betragt eksempelvis rodtræet RT i figur 6.2. Lad nu b være roden af et nyt deltræ bestående af knuderne b, a og d.

Når man arbejder med træer, vil man som oftest vide noget om deres egenskaber i form af antal knuder, antal kanter, træets højde, osv.

Definition 40 *En knudes niveau i et rodtræ samt højden af et rodtræ*

En knude v's niveau i et rodtræ er længden af den unikke sti (antallet af kanter) fra rodknuden til knuden v. Det højeste niveau i et rodtræ benævnes samtidig rodtræets højde.

Forholdet mellem antallet af knuder og kanter

Sætning 7

Et træ med n knuder har $n-1$ kanter.

Bevis for sætning 7
Sætningen bevises med svag induktion.

Basisskridt: Når antallet af knuder i et træ er én ($n = 1$), har træet ingen kanter ($n-1 = 1-1 = 0$). Således er sætningen ovenfor sand for basisskridtet.

Induktionsskridt: Den induktive hypotese er, at ethvert træ med k knuder har $k-1$ kanter, hvor k er et positivt heltal.

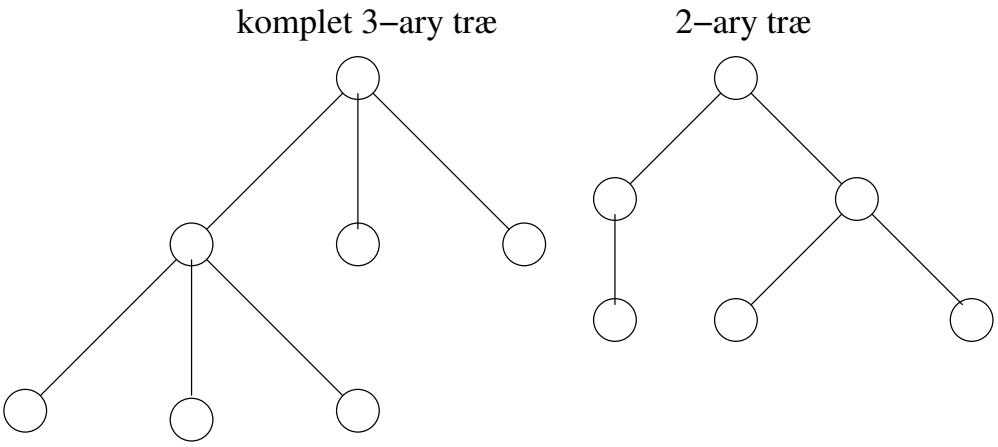
Antag nu, at et træ T har $k+1$ knuder, og at v er et blad i træet T. Lad samtidig w være forælder til v.

Ved at fjerne knuden (bladet) v fra træet T samt kanten, der forbinder w og v, fremkommer grafen T‘ med k knuder. Den nye graf T‘ forbliver en sammenhængende graf uden kredse og er således stadig et træ.

Gennem vores påstand ser vi, at T‘ har $k-1$ kanter. Herved følger det, at T har k kanter, idét T har én kant mere end T‘, nemlig kanten der forbinder w og v.

Dette slutter det induktive skridt, og sætningen er bevist.

◁



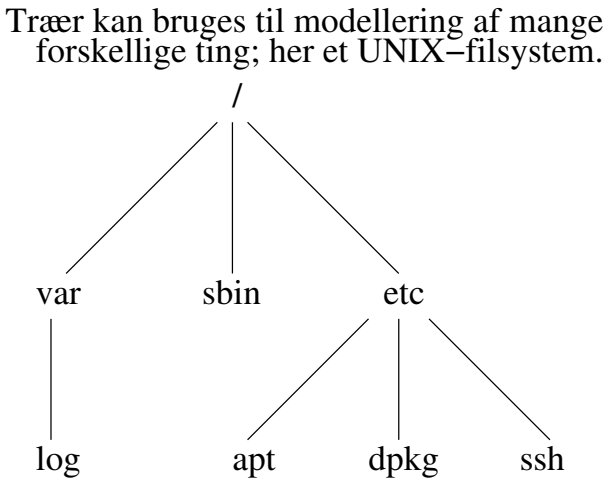
Figur 6.3: Eksempel på et komplet 3-ary træ og et 2-ary træ

m-ary træer

m-ary træer fortæller noget om søskendemængden i et givent rodtræ. Et 5-ary træ er således et rodtræ med højst fem søskende. Et *komplet* 5-ary træ er et rodtræ, hvor alle interne knuder har præcis fem børn.

Definition 41 *m-ary træ*

Et rodtræ kaldes et m-ary træ, hvis alle interne knuder har højest m antal børn. Træet benævnes et komplet m-ary træ, hvis alle interne knuder har netop m antal børn.



Figur 6.4: Eksempel på brugen af træer som modeller

Egenskaber ved et komplet m-ary træ

Sætning 8

Et komplet m-ary træ med i interne knuder indeholder $n = mi + 1$ knuder.

Bevis for sætning 8
Alle knuder i et træ undtagen rodknuden er børn til en intern knude. Idet alle interne knuder i har m børn, er der mi knuder i træet plus rodknuden. Således er der i alt $n = mi + 1$ knuder i træet.

◁

Sætning 9

Træet T er et komplet m-ary træ, hvor n er antallet af knuder i alt, i er antallet af interne knuder, og l er antallet af blade. Hvis den ene af de tre værdier er kendt, kan de to andre findes ved:

- **1a.** *n knuder har $i = \frac{n-1}{m}$ interne knuder*
- **1b.** *n knuder har $l = \frac{((m-1)n+1)}{m}$ blade*
- **2a.** *i interne knuder har $n = mi + 1$ knuder*
- **2b.** *i interne knuder har $l = (m-1)i + 1$ blade*
- **3a.** *l blade har $n = \frac{ml-1}{m-1}$ knuder*
- **3b.** *l blade har $i = (l-1)(m-1)$ interne knuder*

Bevis for sætning 9
Lad følgende gælde: n = antallet af knuder
i = antallet af interne knuder
l = antallet af blade

De seks punkter fra sætning 9 kan bevises ved at anvende ligningen $n = mi+1$ fra sætning 8 samt ligningen $n = l + i$, som er sand, da en knude n enten er en intern knude i eller et blad l.

Bevis for 1a og 1b:
1a. $n = mi + 1 \Leftrightarrow n - 1 = mi \Leftrightarrow i = (n-1)/m$
1b. Nu indsættes $i = (n-1)/m$ i ligningen $n = l + i$, hvilket giver $n = l + i \Leftrightarrow l = n - i \Leftrightarrow l = n - ((n-1)/m) \Leftrightarrow l = ((m-1)n + 1)/m$

$\triangleleft$

Sætning 10

Der er højest m^h blade i et m-ary træ med højden h.

Bevis for sætning 10

Basisskridt: Betragt først et m-ary træ med højden 1, som dermed består af rodknuden samt højest m børn, der alle er blade som følge af højden 1. Herved ses det, at der ikke er flere end $m^h \Rightarrow m^1 \Rightarrow m$ blade i et m-ary træ med højden 1. Således er basisskridtet sandt.

Induktionsskridt: Hypotesen er nu, at resultatet er sandt for alle m-ary træer med en højde mindre end h. Lad træet T være et m-ary træ med højden h.

Bladene i træet T forbliver blade i deltræerne af T, hvis vi antager, at der fjernes kanterne, der løber fra rodknuden til alle dens børn. Dette bevirker samtidig, at højden på hvert af deltræerne højest bliver $h - 1$.

Med den induktive hypotese ser vi således, at hvert rod-deltræ højest har $m^h - 1$ blade. Idet der højest er m deltræer med højden $m^h - 1$, kan vi udlede, at der må være $m \cdot m^h - 1 = m^h$ blade i det fulde rodtræ.

Dette slutter det induktive skridt, og sætningen er bevist.

◁

6.1 Udspændende træer

Udspændende træer er en særlig form for træer. Det er de træer, der findes som delgrafer i andre grafer og indeholder alle knuder fra den originale graf. Udspændende træer indeholder et minimum antal kanter. Følgende krav skal overholdes for, at et træ kan være et udspændende træ.

Definition 42 *Udspændende træ*

Lad G være en simpel graf. Et udspændende træ T er en delgraf af G, hvis T indeholder alle knuder fra G.

Ud fra denne definition kan følgende sætning udledes.

Sætning 11

En simpel graf er sammenhængende, hvis og kun hvis den har et udspændende træ.

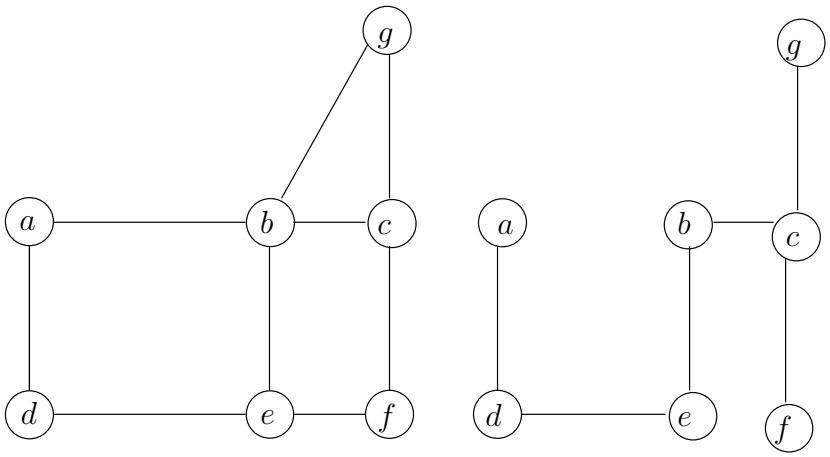
Bevis for sætning 11
Antag at en simpel graf G har et udspændende træ T, der indeholder alle knuderne fra G. Der er en sti imellem to vilkårlige knuder fra T. Jævnfør sætning 1. Idet T er en delgraf af G, er der også i G en sti imellem to vilkårlige knuder. Hermed er det bevist, at G er sammenhængende.

Antag nu, at G er sammenhængende og ikke er et træ. Derved indholder G en eller flere kredse. Fjern en kant fra en af disse kredse. Dette resulterer i en ny graf G'. Denne nye graf har en kant mindre end G. G' er sammenhængende, fordi når en kant fjernes imellem to knuder, der er sammenhængende i en kreds, er der stadigvæk en sti tilbage, som forbinder de to knuder. Denne fremgangsmåde bliver gentaget, indtil der ikke er flere kredse i G. Der er et træ, når sidste kant fjernes. Dette træ er et udspændende træ, idet det indeholder alle knuderne fra G.

Hermed er det bevist, at når grafen er sammenhængende, indeholder den et udspændende træ.

$\triangleleft$

Figur 6.5 viser en simpel sammenhængende graf G og et udspændende træ T. Som det ses indeholder G et udspændende træ T. T er blot et af mange forskellige udspændende træer.



Figur 6.5: Udspændende træ

Kapitel 7

Algoritmer

Dette afsnit bruges til at vise, hvorledes forskellige algoritmer i relation til de to foregående afsnit kan bruges. Den efterfølgende algoritme tager udgangspunkt i kapitlet grafteori. Den beregner den korteste vej fra en given knude til alle andre knuder i en sammenhængende ikke-orienteret vægtet graf.

Vi har i dette kapitel valgt at vise, hvad tidskompleksiteten for de forskellige algoritmer er. Dette skulle give muligheden for, at kunne differentiere imellem de forskellige algoritmer.

7.1 Dijkstra

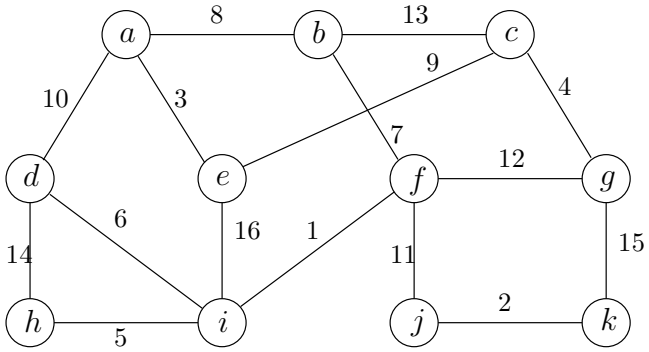
For at finde den korteste vej mellem en knude a og alle andre knuder i en graf har Edgar Wybe Dijkstra udarbejdet en algoritme med dette formål. Algoritmen tager udgangspunkt i knuden a og finder vægten af kanter til alle naboknuder. Da vægten til disse knuder er mindre end ∞, er den korteste vej til alle naboknuder givet. Dernæst undersøger algoritmen, om der er knuder, til hvilken afstanden er ∞. Er dette ikke tilfældet afsluttes algoritmen, og ellers fortsætter den, da der ikke er en vej fra a til alle andre knuder. Algoritmen finder den knude i grafen, hvor vægten af alle kanter fra a er mindst. Denne knude bliver, på samme måde som tidligere, brugt som udgangspunkt, når algoritmen efterfølgende finder dens naboknuder med mindst vægt fra a til den nye knude. På denne måde bliver der undersøgt, om vægten bliver mindre ved at "gå via en anden knude" fremfor at bruge den direkte vej. Dette kan opsummeres i følgende sætning:

Sætning 12 *Sætning*

Dijkstra's algoritime finder længden af den korteste vej mellem en given knude i grafen og alle andre knuder i en sammenhængende simple, ikke-orienteret vægtet graf.

En sidegevinst ved Dijkstras algoritme er, at den også finder et minimum udspændende rodfæstet træ i grafen.

For at illustrere hvorledes algoritmen fungerer, tages der udgangspunkt i figur 7.1. Der ønskes vejen fra a til alle andre knuder med den mindste samlede vægt.



Figur 7.1: En simpel vægtet graf

Eksempel
Algoritmen starter med at antage, at afstanden fra a til alle andre knuder er ∞. Derefter findes den korteste vej fra a til alle naboer. Dette betyder, at vægten af kanten mellem a og b bliver til 8. På samme måde bliver vægten af kanten mellem a og e til tre, og vægten af kanten mellem a og d bliver til 10. Da der stadigvæk er knuder, hvortil den samlede vægt af alle kanter fra a er ∞, bliver den knude med den mindste vægt valgt. I dette tilfælde e. Fra knuden e findes alle de kanter til naboer med den mindste samlede vægt fra a. Dette betyder, at den mindste samlede vægt af alle kanter fra a til c er 12, og mellem a og i er den 19. Algoritmen tager nu udgangspunkt i b, da dette er den knude, hvortil der er den mindste samlede vægt af kanter fra a. Fra knuden b findes alle de kanter til b's naboer med den mindste samlede

vægt fra a. Derfor er den mindste samlede vægt af alle kanter fra a til f lig med 15. I dette tilfælde vil vægten af alle kanter fra a til c **ikke** blive 21, da vægten af alle kanter fra a til c bliver mindre ved at bruge vejen a, e, c end ved at bruge vejen a, b, c. Denne fremgangsmåde fortsættes indtil vejen til alle kanter med den mindste vægt er fundet. Dette er opsummeret i tabel 7.1.

Mål	Vej	Vægt
b	$a - b$	8
c	$a - e - c$	12
d	$a - d$	10
e	$a - e$	3
f	$a - b - f$	15
g	$a - e - c - g$	16
h	$a - d - i - h$	21
i	$a - d - i$	16
j	$a - b - f - j$	26
k	$a - b - f - j - k$	28

Tabel 7.1: Sammenhæng mellem mål, vej og afstand fra knude a på figur 7.1.

◀

Algoritme 1 viser, hvorledes Dijkstra's algoritme kunne implementeres, koden er baseret på [2].

Algoritme 1 *Dijkstra's algoritme*

```
01  procedure Dijkstra (G)
02 // PRE: Proceduren modtager en vægtet forbundet simpel graf med positive vægter på kanterne
03 // POST: Den korteste vej fra a til z er fundet, hvor z er en vilkårlig knude i grafen G
04
05 //G har knuderne v_0, v_1, ..., v_n, hvor a = v_0 og z = v_n
06 //L(v_i) giver vægten af den samlede sti fra a til v_i
07
08 //Alle vægte på grafen sættes til ∞ undtagen L(a), som bliver sat til 0
09  for i := 1 to n
10  begin
11        L(v_i) := ∞
12  end
13  L(a) := 0
14 //S er en liste over de korteste veje til alle knuder fra a
15  S := ∅
16
17  while v ∉ S
18  begin
19        u := en knude med mindst L(u), u ∉ S
20        S := S ⋃ {u}
21        for alle knuder v ∉ S
22        begin
23              if L(u) + w(u,v) < L(v)
24                    then L(v) := L(u) + w(u,v)
25         end
26  end
```

Tidskompleksiteten af Dijkstra's algoritme

I gennemgangen af denne algoritme vil der udelukkende blive talt antal additionsoperationer.

I linie 25 bliver $L(u)$ adderet med $w(u, v)$. Dette sker $n - 1$ gange inde i for-løkken i linie 22 til 27, som så køres inde i for-løkken fra linie 18 til 28. Hermed skal $n - 1$ køres $n - 1$ gange. Summerer man så op fås, at Store-O af Dijkstra's algoritme bliver $O(n^2)$.

Bevis for sætning 12

Ved at bruge et induktionbevis vil vi bevise, at sætningen er sand. Dette gøres ved at bruge induktion i forbindelse med størrelsen af mængden af knuder S. Det antages, at der for hvert induktionsskridt gælder:

- $v \in S$, $dist(a, v)$ giver den korteste afstand fra a til v.
- $v \notin S$, $dist(a, v)$ giver den korteste afstand fra a til v via S.

Ved basisskridtet er $S = \emptyset$. Derved er afstanden fra a til alle andre knuder ∞, og basisskridtet er derved sandt.

Udpeg v fra mængden S med den korteste afstand. Hvis $dist(a, v)$ ikke er den korteste afstand, findes en kortere afstand ved at bruge en knude, som ikke er i S.

Ved induktionsskridtet n udpeges den knude v fra mængden S med den korteste afstand. Hvis der, ved at inkludere knuden u i S, skabes en kortere afstand, end hvis u ikke blev inkluderet, skal denne afstand medtages i induktionsskridtet for $n + 1$.

$\triangleleft$

7.2 Minimum udspændende træer

Vi præsenterer fire algoritmer, der opbygger et minimum udspændende træ udfra en simpel sammenhængende graf med n knuder.

Dybde-først-søgning

Dybde-først-søgning er en algoritme, der opbygger et minimum udspændende træ. Der vælges en vilkårlig knude i en simpel sammenhængende graf G som rod. Udfra roden opbygges et rodtræ. Der opbygges en sti i rodtræet ved at tilføje naboknuder og kanter, der ikke er indeholdt i rodtræt i forvejen. Algoritmen stopper, når alle knuderne i G er blevet tilføjet, og det udspændende træ er fundet.

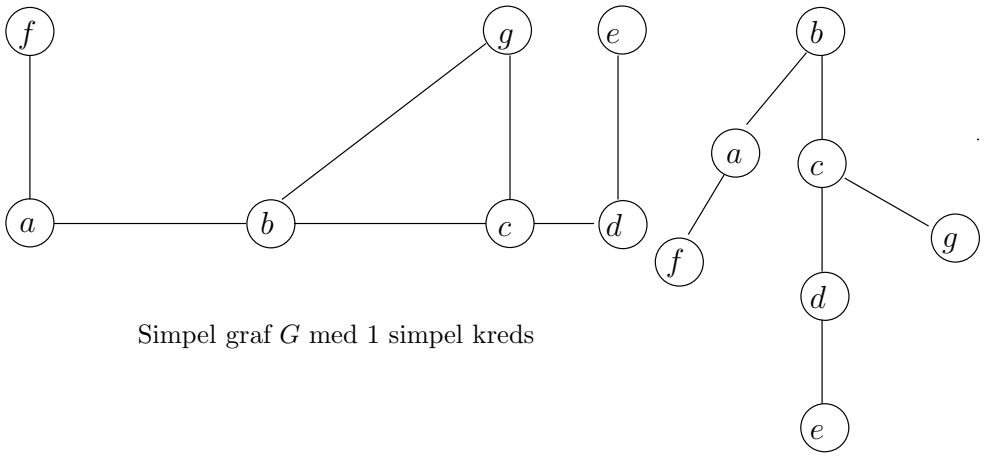
Algoritme 2 viser, hvordan dybde-først-søgningen kunne implementeres. Koden er baseret på [7].

Algoritme 2 *Dybde-først-søgning*

```
01 procedure DFS(G)
02 // PRE: Proceduren modtager en vægtet sammenhængende
   simpel graf G med n knuder
03 // POST: T er et minimum udspændende træ i G
04
05 T er et træ indeholdende rodknuden v1
06 visit(v1)
07
08 procedure visit(v:knude i G)
09 begin
10     for hver knude w ∉ T der er nabo til v
11     begin
12         tilføj knuden w og kanten {v,w} til T
13         visit(w)
14     end
15 end
```

Figur 7.2 viser en simpel graf G med en kreds. Figuren viser også det udspændende træ T, der er blevet opbygget via dybde-først-søgning i G.

Algoritmen dybde-først-søgning går så dybt, som det er muligt, inden algoritmen gør brug af teknikken *backtracking*. Backtracking er, når algoritmen er nået så dybt, at den ikke kan komme længere. Algoritmen går så en knude tilbage, for at danne en ny sti ud fra denne knude. Den forsætter med at gå



Figur 7.2: Dybde-først-søgning

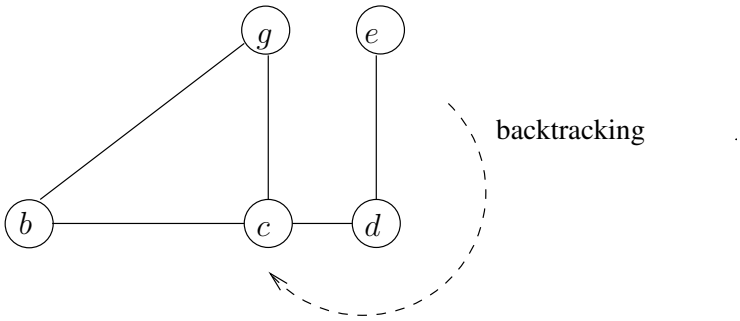
tilbage, indtil den finder en ny sti. Med ny sti menes der en sti med knuder, der ikke allerede er blevet tilføjet til det udspændende træ T. I figur 7.3 illustreres det, hvordan backtracking anvendes i grafen G. Til at starte med vælges knuden b som udgangspunkt. Derefter går algoritmen i dybden, dvs. at algoritmen når ned til knuden e. Fra e backtrackes der tilbage til knuden d og senere til knuden c, hvor der dannes en ny sti ud fra c.

Tidskompleksiteten af dybde-først-søgning algoritmen

Dybde-først-søgning algoritmen kan udføres i tiden $O(n^2)$, idet algoritmen ved hver kant i grafen G undersøges i værste faldt to gange for at kunne beslutte om kanten i grafen G kan tilføjes til det minimum udspændende træ.[7]

Bredde-først-søgning

Bredde-først-søgning er en algoritme, der opbygger et minimum udspændende træ. Algoritmen arbejder ved at opdele grafen i lag. Ligesom den forrige algoritme vælges en vilkårlige knude i en simpel sammenhængende graf G til at være rod b. Ud fra denne rod opbygges et rodtræ. Ud fra roden tilføjes samtlige kanter og naboknuder til roden. De tilføjede knuder danner lag et i det udspændende træ. For hver knuder i lag et tilføjes samtlige kanter



Figur 7.3: backtracking

og naboknuder, så længe der ikke dannes en kreds. Disse nye knuder danner lag to i det udspændende træ. Algoritmen gentages, indtil alle knuder er blevet tilføjet til træt T. Der er et endeligt antal kanter i grafen, så algoritmen slutter eventuelt.

Algoritme 3 viser, hvordan bredde-først-søgningen kan implementeres. Koden er baseret på [7].

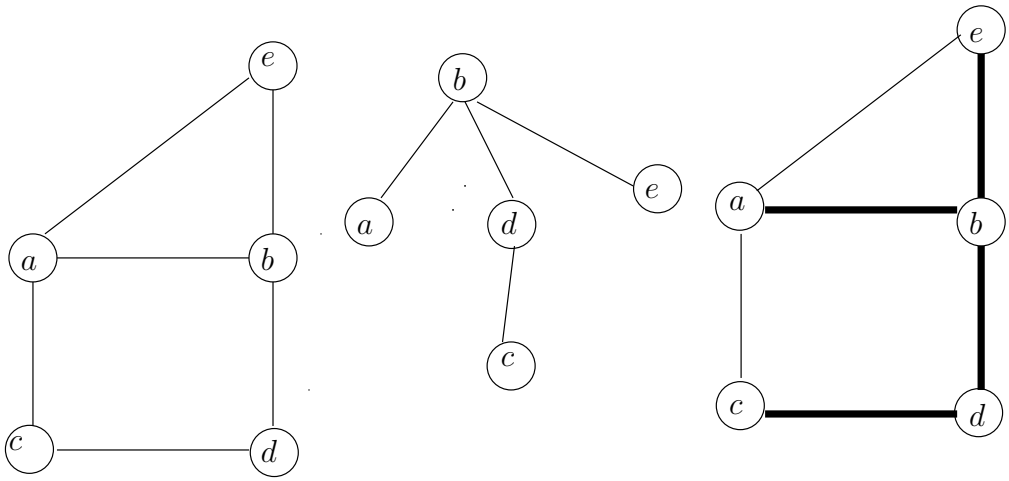
Algoritme 3 *Bredde-først-søgning*

```
01 procedure BFS(G)
02 // PRE: Proceduren modtager en vægtet sammenhængende
simpel graf G med n knuder som input
03 // POST: T er et minimum udspændende træ i G
04
05 T er et træ indeholdende rodknuden v1
06 L : = Tom liste der indeholder ikke-bearbejdede knude
07      tilføj v1 til listen L
08 while L ikke er tom
09 begin
10      fjerner den første knude v fra listen L
11      for hver naboknude w til v
12      begin
13          if w ikke er i L og ikke i T then
14          begin
15              tilføj knuden w til enden af listen L
16              tilføj w og kanten {v,w} til T
17          end
18      end
19 end
```

Figur 7.4 viser til venstre en simpel graf med kredse. Til højre ses to illustrationer af det udspændende træ T, der er blevet opbygget via algoritmen. Som rod i træet har vi valgt knuden b.

Tidskompleksiteten af bredde-først-søgning algoritmen

Bredde-først-søgning algoritmen kan udføres i tiden $O(n^2)$, idet algoritmen i værste fald undersøger hver kant to gang for at kunne beslutte, om kanten kan tilføjes til det minimum udspændende træ. [7]



Simpel graf G med 1 kreds udspændende træ T

Figur 7.4: Bredde-først-søgning

Definition 43 *Minimum udspændende træ*

Et minimum udspændende træ i en sammenhængende vægtet graf er et udspændende træ, der har den mindst mulige sum af vægtede kanter.

Prims algoritme

Prims algoritme tilføjer hele tiden kanter til træet, der har den mindst mulige vægt. Træet skal dog hele tiden være et sammenhængende træ. Algoritmens fremgangsmåde er følgende:

1. Der vælges en hvilken som helst kant med den midste mulige vægt. Hvis to kanter har den samme mindste mulige vægt, vælges en af de to kanter.
2. Denne kant indsættes som den første kant i det udspændende træ.
3. Herefter tilføjes kanten med den mindst mulige vægt, der er tilknyttet

en knude i træet, til det minimum udspændende træ. Den tilknyttende kant må ikke allerede være i træet, og tilføjelse af kanten må ikke danne en simpel kreds. Gruden til dette er, at hvis en kreds blev dannet, ville algoritmen ikke opbygge et minimum udspændende træ.

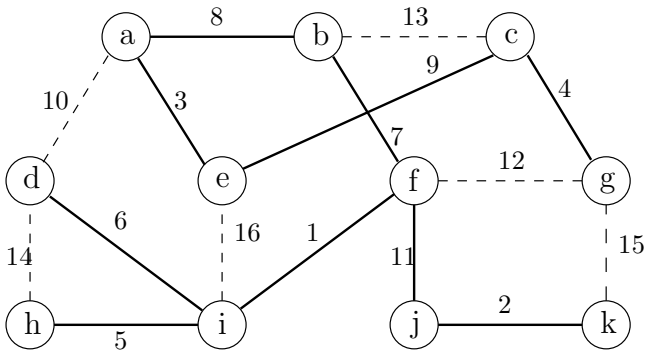
Hvis nogle af kanterne i algoritmen har den samme vægt, kan dette resultere i, at der er flere måder, hvorpå en kant kan tilføjes til træet. Dette kan resultere i, at der findes flere forskellige minimum udspændende træer i grafen. Ifølge sætning 7 har et træ altid $n-1$ kanter. Dette udnyttes af algoritmen, således den stopper, når $n-1$ kanter er blevet tilføjet.

Algoritme 4 viser, hvordan Prims algoritme kunne implementeres. Koden er baseret på [7].

Algoritme 4 *Prims' algoritme*

```
01  procedure Prim (G)
02 // PRE: Proceduren modtager en vægtet sammenhængende
simpel graf G med n knuder
03 // POST: T er et minimum udspændende træ i G
04
05  T er kanten med mindst vægt i det minimum udspændende
træ. 06
07  for i := 1 to n − 2
08  begin
09        e := kant med minimum vægt og nabo til en knude i T
10        //e må ikke danne en simpel kreds i T, hvis den tilføjes
11        T := T hvor e er tilføjet
12  end
```

Som det ses af ovenstående pseudokode, tilføjes der hele tiden en kant med minimum vægt til en allerede eksisterende kant med minimum vægt. I figur 7.1 vises først en sammenhængende simpel graf. Derefter vises det, hvordan man kommer frem til det udspændende træ ved brug af Prims algoritme. Dette ses i figur 7.5



Figur 7.5: Prims Algoritme

Tidskompleksitet af Prims algoritme

Lad m være antallet af knuder i grafen. Løkken i Prims algoritme 4 linie 06 vil altid gennemløbe algoritmen m antal gange. Man kan vise, at det kan lade sig gøre at finde kanten med den mindst mulig vægt blandt n elementer i tiden $O(\log n)$. Der kan højst være $n(n-1)/2$ kanter i en graf med n knuder. Da $\log n(n-1)/2 = \log n + \log(n-1) - \log 2 = \mathrm{O}(\log n)$ tager Prims algoritme højst $m \times \mathrm{O}(\log n) = \mathrm{O}(m \log n)$ skridt.

Kruskals algoritme

Kruskals algoritme tilføjer hele tiden, ligesom Prims algoritme, kanter med mindst mulig vægt til træet. Forskellen er den, at det ikke er nødvendigt, at en ny kant skal være nabo til kanter, der allerede er blevet valgt. Dvs, det er muligt at "springe" fra kant til kant. Algoritmens fremgangsmåde er følgende:

1. Der vælges en kant med den mindst mulige vægt.
2. Denne kant indsættes som den første kant i det udspændende træ.
3. Herefter tilføjes en kant med den mindst mulige vægt til det udpændende træ. Den nye kant må ikke allerede være i træet, og tilføjelse af kanten må ikke danne en simpel kreds.

Algoritme 5 viser, hvordan Kruskals algoritme kan implementeres. Koden er baseret på [7].

Algoritme 5 *Kruskals algoritme*

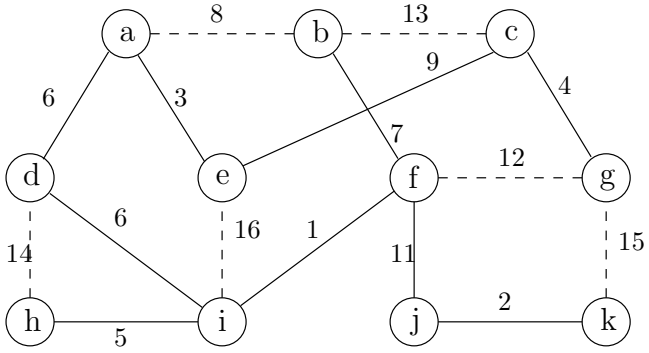
```
01  procedure Kruskal (G)
02 // PRE: Proceduren modtager en vægtet sammenhængende
simpel graf G med n knuder
03 // POST: T er et minimum udspændende træ i G
04
05  T : = en tom graf
06  Sorterer kanterne i grafen G
07
08  for i := 1 to n − 1
09  begin
10        e := En hvilken som helst kant i grafen G. 11
e må ikke danne en simpel kreds i T, hvis den tilføjes til T
12        T := T med e tilføjet
13  end
```

Ligesom i Prims algoritme er der flere måder, hvorpå man kan vælge en ny kant, hvis flere kanter har samme vægt.

Hvis kanterne er i kronologisk rækkefølge, opbygger Kruskal nøjagtig det samme minimum udspændende træ som Prims algoritme. Af denne grund har vi valgt ikke at vise, hvordan man laver et minimum udspændende træ ud fra figur 7.1, idet denne figurs kanters værdier er kronologiske. Vi har i stedet valgt at bruge den samme graf, men dog med andre vægte på kanterne. I figur 7.6 illustreres, hvordan et minimum udspændende træ findes. De fremhævede kanter er det udspændende træ.

Tidskompleksiteten af Kruskals algoritme

Tidskompleksiteten af Kruskals algoritme afhænger dels af, hvordan man sorterer kanterne i G og dels af, hvordan man sørger for ikke at vælge kanter i grafen, som resulterer i, at kredse opstår. Kruskals algoritme kan implementeres ved brug af forskellige datastrukturer, der giver forskellig tidskompleksitet. Ifølge [6] kan algoritmen f.eks. have tidskompleksiteten $O(m+n \log n)$, plus tiden det tager at sortere grafen G.



Figur 7.6: Kruskals Algoritme

7.3 Sammenligning af algoritmer

Prims og Kruskals algoritmer er grådige algoritmer, som vil sige, at algoritmerne laver et optimalt valg ved hvert skridt, de foretager. Selvom Prim og Kruskal er grådige algoritmer, garanterer de en optimal løsning. Prim og Kruskal starter altid med den kant, der har den mindst mulige vægt i grafen, hvorimod Dijkstra starter med en knude som udgangspunkt. Prim og Kruskal opbygger hele tiden et minimum udspændende træ. Dijkstra er beregnet til at finde den korteste vej imellem en knude og alle andre knuder i en graf. Den opbygger desuden også et minimum udspændende **rodfæstet** træ.

Prim og Kruskal finder ikke nødvendigvis den korteste vej imellem to vilkårlige punkter i en simpel graf, hvilket Dijkstra hele tiden gør. Prim og Kruskal er ikke tvunget til at opbygge et rodfæstet træ i modsætnig til Dijkstra og dybde-først-søgning og bredde-først-søgning. Dette betyder, at et træ, som er fundet via Prim eller Kruskal, **kan** gøres rodfæstet, hvis det ønskes. Da tidskompleksiteten af Dijkstra, dybde-først-søgning og bredde-f orst-søgning samtidig er værre end Prim og Kruskal, vil det ofte være bedre at benytte Prim og Kruskal, hvis målet er at opbygge et minimum udspændende træ. c

Kapitel 8

Kompleksitetsteori

Kompleksitetsteori er med i denne rapport for at kunne kategoriser hvilken type problem TSP er. En del af kompleksitetsteori er *beslutningsproblemer*. I forbindelse med beslutningsproblemer er det vigtigt at kunne kategorisere disse for derved at vide, hvordan man skal gribe det pågældende problem an. Skal man f.eks. forsøge at finde en effektiv/optimal løsning på problemet, eller skal man blot prøve at finde en tilnærmelsesvis hurtig løsning på problemet? I kapitlet om funktionsvækst definerede vi en algoritme til at kunne løses effektivt, hvis dens tidskompleksitet ikke overstiger $O(x^b)$.

Ydermere er det i forbindelse med beslutningsproblemer vigtigt at kunne afgøre, hvorvidt et problem overhovedet kan løses effektivt; dvs. inden for en polynomisk tidsramme.

8.1 Turing-maskiner

Inden vi begiver os i kast med beslutningsproblemer, er det hensigtsmæssigt med en introduktion til *Turing-maskiner* (*deterministisk* såvel som *nondeterministisk*), idet Turing-maskiner kan hjælpe os med at verificere eller løse beslutningsproblemer.

En Turing-maskine er en teoretisk, abstrakt modellering af en maskine. Turingmaskinen består af et uendeligt langt "bånd", som kan køre frem og tilbage. Båndet er opdelt i celler, så et "hoved"i Turing-maskinen kan læse/skrive sekventielt fra/til båndet.

Yderligere består Turing-maskinen af et såkaldt *tilstandsregister*, som holder styr på, hvilken tilstand Turing-maskinen er i. Antallet af mulige tilstande er

på forhånd givet, og der gives samtidig en starttilstand, som tilstandsregistret initialiseres ud fra. Hovedets startposition er i øvrigt altid ud for den første ikke-tomme celle længst til venstre på båndet.

Sidst men ikke mindst er der i Turing-maskinen en *handlingstabel*, hvor funktioner definerer, hvilken ny celle (og dermed hvilken ny tilstand), der skal hoppes videre til ud fra det givne symbol (inputet) samt, hvad den forrige tilstand var.

En Turing-maskine går fra tilstand til tilstand ud fra foruddefinerede femtupler af formen $(aktuel\_tilstand, læs\_celleværdi, ny\_tilstand, skriv\_celleværdi, flyt\_hoved)$. Dette kan eksemplificeres som $(s_0, 1, s_1, 0, R)$, hvor R betyder, at den skal flytte læsehovedet til højre (R for right).

Måden, hvorpå Turing-maskinen behandler et sådan eksempel på en femtupel, kan beskrives således:

1. Maskinen er i starttilstand s_0.
2. Maskinen læser værdien 1 fra cellen, som er ud for læsehovedets position.
3. Givet maskinens tilstand s_0 samt det læste input 1, går maskinen over i en ny tilstand s_1.
4. Maskinen skriver værdien 0 til cellen.
5. Maskinens læsehoved bevæger sig én celle til højre (R) og er klar til at læse næste celle.

En Turing-maskine kan laves som en deterministisk eller en nondeterministisk maskine. Den deterministiske maskine har altid kun én næste mulig tilstand, mens den nondeterministiske maskine i en given tilstand kan komme ud for, at den i næste tilstand kan være i en af flere mulige tilstande. I det følgende afsnit bliver den nondeterministiske maskine yderligere beskrevet.

TSP og en nondeterministisk maskine

Idet TSP er i mængden af NP problemer, kan man på en *nondeterministisk maskine* konstruere en løsning (nondeterministisk turing maskine).

En nondeterministisk maskine har den mulighed, at der efter hver tilstand er mulighed for, at maskinen går over i en ny tilstand, eller at maskinen går over i en sluttilstand. En nondeterministisk maskine kan repræsenteres ved, at man laver en tilstandstabel eller et tilstandsdiagram.

I en tilstandstabel listes alle mulige tilstande og input-værdier, samt hvilke tilstande det er muligt at gå over i.

I et tilstandsdiagram er alle mulige tilstande repræsenteret ved en cirkel. En

sluttilstand er markeret med en dobbelt cirkel, og der er en kant imellem alle tilstandene. Hver kant er markeret med input eller output, der fører til en ny tilstand.

Når et problem på en nondeterministisk maskine ender i en sluttilstand, er det pågældende problem løst.

Hvis der til det pågældende problem kan svares ja til, at der findes en optimal løsning er problemet løseligt. Den nondeterministiske maskine behøver ikke at teste samtlige muligheder. Den ved, hvilken mulighed der er den optimale.

Som tidligere nævnt består problemet i, at når man implementerer løsningen på en almindelig computer, vil løsningen ikke kunne løses inden for acceptabel tid, idét implementeringen ikke ved, hvilken mulighed der er den optimale. I implementeringen bliver man nødt til at afprøve samtlige muligheder, hvilket gør at implementeringen ikke kan løse problemet inden for polynomiel tid.

8.2 Kategorisering af beslutningsproblemer

De følgende afsnit omhandler teorien om problemklasserne, der på matematisk vis benævnes som *P, NP, NP-fuldstændig og NP-svær*. Disse problemer er alle beslutningsproblemer (ja/nej-problemer). Beslutningsproblemer er interessante, da det er netop denne problemtype, som udgør problematikken i TSP.

I de følgende afsnit vil vi kortlægge de forskellige klasser og deres egenskaber. Vi vil desuden også beskrive forskellene de forskellige klasser imellem.

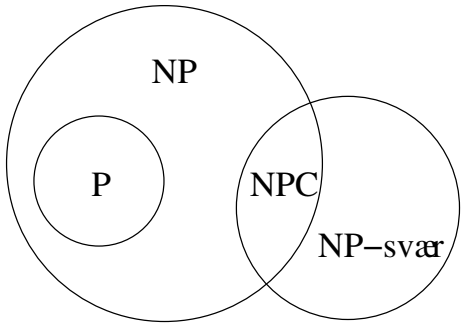
Vi har som nævnt ovenfor valgt at gøre brug af kategoriseringen *P, NP, NP-fuldstændig og NP-svær*, som begrænser sig til beslutningsproblemer.

Vi interesserer os især for de problemer, der ligger i NP, da vi i denne rapport skal forsøge at finde en løsning på problematikken omkring den handelsrejsende (TSP).

P-problemer

P-problemer definerer mængden af de beslutningsproblemer, som kan løses i *polynomiel tid* på en deterministisk maskine. Som tidligere nævnt betragter vi en sådan løsning som værende effektiv.

Funktionsvæksten for en given algoritme, som kan løse det pågældende P-problem, løses i $O(n^b)$, hvor b er et heltal, og $b \geq 1$. Et eksempel på en sådan



Figur 8.1: NP-fuldstændig (NPC) har karakteristika fra både NP og NP-svær

algoritme er *bubble sort*. Bubble sort's eksekveringstid er $O(n^2)$.

NP-problemer

NP-problemer er de beslutningsproblemer, der kan *verificeres* i polynomiel tid på en deterministisk maskine. Hvis der således for et givent problem kan svares ja, kan man verificere, hvorvidt der findes en løsning i polynomiel tid på det pågældende problem.

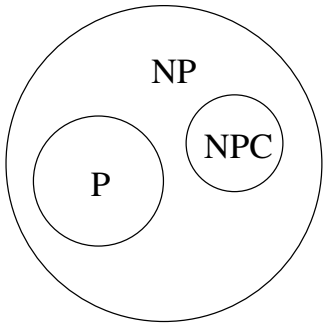
Samtidig kan NP-problemer *løses* i polynomiel tid ved hjælp af en nondeterministisk maskine. For at den nondeterministiske maskine kan løse et NP-problem i polynomiel tid, er det en forudsætning, at den altid gætter rigtigt.

NP-svær/hård

Et problem er *NP-svært*, hvis en metode for løsning af problemet kan videreføres over på et hvilket som helst andet NP-problem. Et NP-svært problem er således *mindst* lige så svært at løse som ethvert andet NP-problem. NP-svære problemer kan ligge i NP. Denne mængde af problemer kaldes NP-fuldstændig (jvf. næste afsnit).

NP-fuldstændig/komplet

NP-fuldstændige problemer betragtes som de sværeste NP-problemer at løse, hvilket hænger sammen med, at der til et NP-fuldstændigt problem knytter sig de karakteristika, som kendes fra *både* NP *og* NP-svær. Dette er illustreret i figur 8.1.



Figur 8.2: NP-fuldstændig (NPC) og P er begge delmængder af NP

I forbindelse med kompleksitetsteori undlader mange forfattere imidlertidig at illustrere mængden NP-svær. Uden NP-svær kan mængderne P, NP, og NPC illustreres som vist i figur 8.2.

De karakteristika, som kendetegner NP-fuldstændig, er:

(a) Som ethvert andet NP-problem, gælder der for et NP-fuldstændigt problem, at det kan løses i polynomiel tid på en nondeterministisk maskine, mens løsningen i polynomiel tid kan verificeres på en deterministisk maskine.

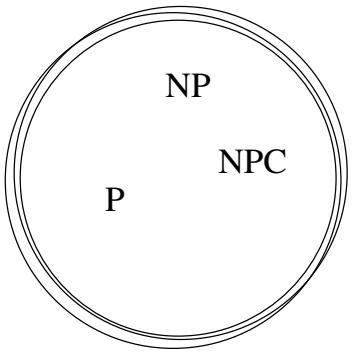
(b) Som med NP-svære problemer kan en metode (algoritme) for løsning af et NP-fuldstændigt problem videreføres over på et hvilket som helst andet NP-problem.

Af punkt **(b)** følger det således, at hvis en person en dag skulle gå hen og fremlægge et gyldigt bevis for en løsning i polynomiel tid af et NP-fuldstændigt problem, vil denne løsningsmodel kunne bruges til at løse et hvilket som helst problem i NP, inklusiv P. Herved bliver $P = NP = NPC$, og disse mængder kan nu illustreres som vist i figur 8.3.

TSP ligger som bekendt i NP-fuldstændig, hvilke betyder, at man tilstræber at finde en algoritme, der nærmer sig den optimale løsning. Denne algoritme skal dog stadigvæk være mest mulig effektiv. Dvs. den skal have den mindst mulige kørselstid (jf. store-O-notation) inden for polynomiel tid.

The Subset Sum Problem

Et andet eksempel på et NP-fuldstændigt problem (hvortil ingen optimal løsning således kendes) er "The Subset Sum Problem", hvor det skal påvises, hvorvidt der i en liste af heltal er en delmængde af disse heltal, hvor summen af delmængden giver nul.



Figur 8.3: Forholdet $P = NP = NPC$ gælder mellem mængderne, hvis det kan bevises, at et NP-fuldstændigt problem kan løses i polynomiel tid

Givet listen $\{-9, -5, -2, 1, 3, 4\}$ er svaret på problemet, at der faktisk findes sådan en liste, idét delmængden $\{-5, 1, 4\}$ netop giver nul.

Kun en nondeterministisk maskine kan i polynomiel tid producere en løsning. Hvis vi i øvrigt giver løsningen (delmængden $\{-5, 1, 4\}$) som input til en deterministisk maskine, vil denne i polynomiel tid kunne fastslå/verificere, at det givne løsningsforslag er korrekt.

Kapitel 9

Løsningsforslag til TSP

Udgangspunktet for TSP er en komplet graf, der beskriver en række byer og de veje, som forbinder disse. Ved at tage udgangspunkt i kapitel 5 kan fremstilles en sammenhængende, simpel vægtet graf. I denne graf repræsenteres byer af knuder, og en vej mellem to byer repræsenteres af en kant. Vægten på kanten viser, hvor langt der er mellem knuderne.

TSP kan illustreres grafisk vha. en graf $G = (V, E)$, hvor knuderne V i grafen repræsenterer byer, samtidig med at kanterne E i grafen repræsenterer veje mellem byerne.

Den vej gennem grafen, som besøger samtlige byer netop én gang og samtidig har samme start- og slutby, er en Hamilton-kreds (definition 5.3).

At finde denne vej er et NP-fuldstændigt problem jvf. afsnit 8.2. At finde en fornuftig vej baseres på, at der estimeres, hvor galt det kan gå med en given vej. Dette kunne f.eks. være, at det beviseligt kan garanteres, at en vej aldrig kommer til at overskride den optimale vej med mere end halvanden. Til at udforme denne bruges Store-O notation som beskrevet i afsnit 2.1. Endelig bruges der også beviser fra afsnit 3 til at påvise det fundne resultatet.

9.1 Løsningsforslag

Der findes i dag adskillige løsningsforslag, dvs. algoritmer, til TSP, hvor nogle er bedre end andre. Disse vægtes efter, hvad man vægter højst: kørselstid eller kvaliteten af den vej, som algoritmen producerer.

Som følge af den deadline, som dette projekt er underlagt, har vi valgt at begrænse os til følgende løsningsforslag: Brute force, 2 gange minimum, nær-

meste nabo, billigst/fjernest/nærmest indsættelse og Christofides. Disse beskrives alle i de følgende afsnit.

9.2 Brute force

Som nævnt i starten af rapporten, er det ikke ligetil at finde den optimale løsning til TSP. Det er dog ikke umuligt. En af de mulige løsninger er *brute force*. Det er måske en overfortolkning at kalde brute force for en algoritme. Alt hvad den gør, er at finde alle Hamilton-kredse (5.3) i grafen G, og herfra vælge den mest optimale løsning ud fra alle løsninger.

Algoritme 6 *Brute force algoritme*

```
01  procedure BruteForce (G)
02 // PRE: En sammenhængende graf G med flere end to knuder findes.
03 // POST: Den optimale vej igennem grafen fundet.
04
05  Find alle Hamilton-kredse og list dem
06
07  Udvælg og returner den Hamilton-kreds med mindst samlede vægt.
08  end
```

Som illustreret i algoritmen, er implementeringen forholdsvis let. Det svære og tidskrævende i algoritmen ligger i linie 5. Algoritmen er så tidskrævende, at køretiden bliver $O(n!)$, hvor n er antallet af knuder.

Bevis for at brute force er $O(n!)$
Idet der er n knuder i grafen G, må der være $n!$ forskellige måder at skabe Hamilton-kredse på. Udfra disse kredse, vil nogle af dem være ens. Da der er n knuder, vil der være n forskellige startknuder, og dermed vil der være $n!/n$ forskellige Hamilton-kredse. Med andre ord, hvis man tager udgangspunkt i een knude vil man unægteligt finde alle Hamilton-kreds, hvorefter man kan vælge en anden knude som startknude. Ved at regne lidt på dette,

fåes at $n!/n = (n-1)!$. Da kanterne i G er uordnet (def. 27). Hermed vil der være $(n-1)!/2$ kredse i grafen, da den samme kreds går igen to gange og derfor kan fjernes fra den samlede løsning (den kører bare den "anden" vej). Jf afsnit om Store-O (2.1) kan vi se bort fra konstanterne, og får derfor at $O(n!)$ er gældende for brute force algoritmen. Dette vil ligeledes være bedste gennemløb, da algoritmen skal gennemløbe alle Hamilton-kredse for at kunne udvælge den bedste.

$\triangleleft$

9.3 2 gange minimum algoritme

2 gange minimum er en algoritme, der finder en Hamilton-kreds i en sammenhængende vægtet graf. Dette gøres ved at finde grafens minimum udspændende træ. Herefter bliver alle kanter fordoblet, hvorved der dannes en multigraf, der indeholder en Euler-kreds. Derefter dannes der en Hamilton-kreds i grafen udfra Euler-kredsen.[**?**]

Det antages at vi har en vægtet sammenhængende graf G, hvor vægten af kanten e betegnes $vægt(e)$. Antag at vægtene opfylder trekantsuligheden, dvs. for alle punkter x, y, z er afstanden fra x til z mindre end eller lig med afstanden fra x til y + afstanden fra y til z.

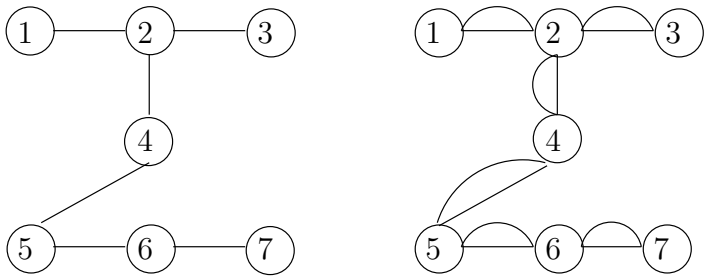
Algorimten har følgende faser:

1. I denne fase konstrueres der et minimum udspændende træ T. Dette kan gøres ud fra forskellige algoritmer (se sektion 6.1).

2. Her fordobles alle kanterne i grafen T. Lad T_2 være grafen der fremkommer ved, at fordoble alle kanterne i T. Dvs. for hver kant x, y i T er der to kanter i T_2 der forbinder x og y.

3. Ud fra grafen T_2 dannes der en Euler-kreds E. Figur 9.1 viser hvordan man finder en Euler-kreds ud fra et minimum udspændende træ. Euler-kredsen besøger følgende punkter i følgende rækkefølge. $3, 2, 1, 2, 4, 5, 6, 7, 6, 5, 4, 2, 3$. Denne graf har en Euler-kreds idet den har et lige antal grader (jf. sætning 3).

4. I denne fase konstrueres der en Hamilton-kreds H_1 ud fra E. Dette gøres ved, at tage punkterne i grafen E og tilføjer kanterne til Hamilton-kredsen. Der spinges over de punkter, der tidligere er besøgt i kredsen.



Et minimuns udspændende træ Euler kreds

Figur 9.1: Euler kreds

Grunden til at alroritmen kaldes "2 gange minimum" er, at den samlede vægt af den fundne Hamilton kreds maksimalt vil være to gange vægten af grafens minimum udspændende træ.

Sætning 13

2 gange minimum algoritmen danner en Hamilton-kreds L. Hvis L er den korteste kreds i G, så er vægt $(H_1) < 2 * vægt(L)$

L er den optimale løsning på TSP, altså den kreds med mindst samlet vægt.

Bevis for sætning 13
Lad e være en kant på L. Så er Le et minimum udspændende træ. Det minimum udpændende træ Le vil altid være kortere end løsningen på den mindste Hamilton-kreds L, fordi ved at fjerne en kant i en Hamilton-kreds fås et træ, som aldrig kan være mindre end det minimum udspændende træ. Med andre ord får vi at $vægt(T) \leq vægt(Le) < vægt(L)$, hvor $vægt(T)$ er vægten på det minimum udspændende træ.

Vægten af T_2 er den samme som vægten af to gange T. Dvs. at $vægt(T_2) = 2 * vægt(T)$.

Vægten af L er lig med eller mindre end vægten af T_2 Dette skyldes trekantsuligden (se sætning 2). Lad punkterne i T_2 være $x_0, x_1, \ldots, x_m$. Hvis f.eks. x_3 er det første punkt, der er besøgt tidligere på kredsen så erstattes x_2, x_3,

x_4 med x_2, x_4 (jf. definitionen på en Hamilton-kreds, def. 5.3). Derved fås en kreds, der ikke er længere end $vægt(T_2)$. Derfor er $vægt(L) \leq vægt(T_2) = 2 * vægt(T) < 2 * vægt(L)$.

Hermed er vores påstand bevist. Ved at bruge "2 gange minimum algoritmen" vil længende af den samlede kreds blive to gange længere end længende af det minimum udspændende træ.

$\triangleleft$

9.4 Heuristiske algoritmer

Med heuristiske algoritmer er man ikke garanteret en optimal løsning, men flere heuristiske algoritmer er gode til meget ofte at komme tæt på den optimale løsning.

De heuristiske algoritmer kan til forskel fra *worst-case* tilfælde af de funktionelle algoritmer tidsmæssigt altid køres i polynomiel tid.

Der er flere klasser af heuristiske algoritmer. bla. konstruktionsalgoritmer, optimeringsalgoritmer. Vi vil i det følgende nøjes med at koncentrere os om konstruktionsalgoritmer.

Nærmeste nabo

Nærmeste nabo indgår i klassen af heuristiske konstruktonsalgoritmer. Med Nærmeste nabo tages der udgangspunkt i en tilfældig knude. Herpå undersøges det, hvilken anden knude der ligger tættest på denne, og derpå tilføjes en kant til denne. Ruten udbygges efter dette mønster, indtil vi har en Hamilton-kreds i grafen.

Problemet med Nærmeste nabo er, at den hen mod slutningen ofte mangler enkelte knuder rundt omkring, som endnu ikke indgår i kredsen. Som resultat af dette kan der let forekomme meget lange kanter i kredsen for at nå rundt til disse resterende knuder.

Pre-betingelser til algoritmen er, at der findes en komplet graf G med V knuder og E kanter. Vægten på kanten fra a til b skrives med $w(a, b)$. Postbetingelsen er, at der returneres en TSP-løsning.

Algoritme 7 *Nærmeste nabo*

$i :=$*en hvilken som helst knude* G *startknuden*
$W := V - \{i\}$
$P := \emptyset$
$v := i$
while $W \neq \emptyset$
begin
Lad k *som er med i* W *være sådan at* $w(v,k) =$
mindstevægt$\{w(v,j)$ *hvor* $j \in W\}$
læg (v,k) *til mængden* P
$W := W - \{k\}$
$v := k$
end
læg (k,i) *til stien* P *for at producere kredsen*

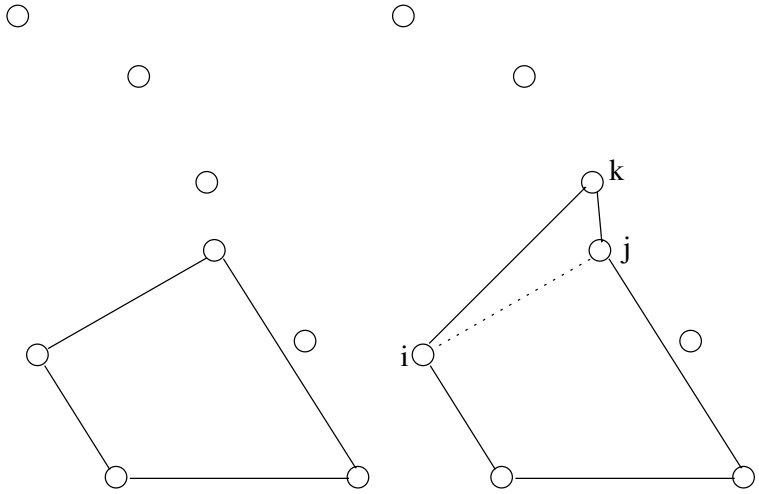
Gennem løbs tiden for denne algoritme vil være $O(n^2)$ i følge referencen[6]. hvilket vil opfylde vores betingelser for polynomiel tid i følge vores kapitel 2.1 .

Indsættelsesalgoritmer

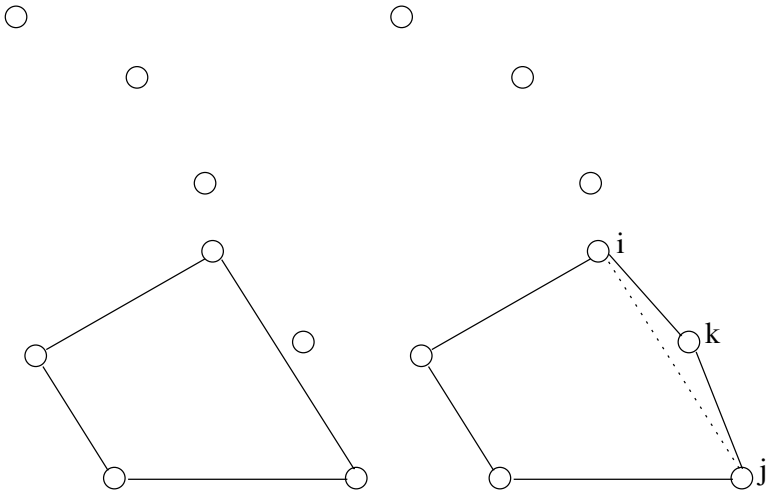
I kategorien indsættelsesalgoritmer vil vi i det følgende behandle *nærmeste*, *billigste* og *fjerneste*. Indsættelsesalgoritmer starter som en Hamilton-kreds bestående af normalt 3 knuder. Denne initielle kreds, som er kortest mulig ved nærmeste/billigste og længst mulig ved fjernest, udbygges knude for knude i henhold til den regel, man vælger at bruge som indsættelsesregel:

- **Nærmeste indsættelsesalgoritme**: Find en knude k ikke i kredsen Z og en knude j i kredsen Z, således at længden $w(k,j)$ er mindst mulig. Lad $\{i,j\}$ være kanten, som minimerer $w(i,k)+w(k,j)-w(i,j)$. Erstat $\{i,j\}$ med $\{i,k\}$ og $\{k,j\}$. Se figur 9.2.

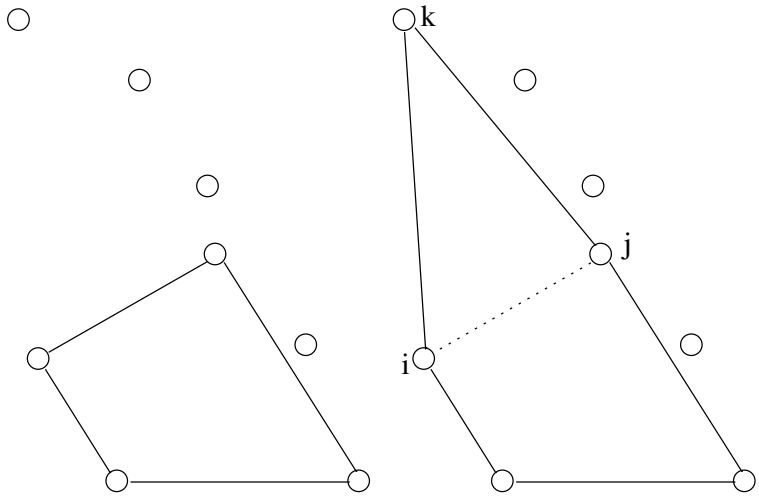
- **Billigste indsættelsesalgoritme**: Find for hver knude k ikke i kredsen Z kanten $\{i,j\}$, som minimerer $w(T,k)$ og dermed også minimerer $w(i,k) + w(k,j) - w(i,j)$. Erstat $\{i,j\}$ med $\{i,k\}$ og $\{k,j\}$. Se figur 9.3.



Figur 9.2: Nærmeste indsættelse



Figur 9.3: Billigste indsættelse



Figur 9.4: Fjerneste indsættelse

- **Fjerneste indsættelsesalgoritme**: Find en knude k ikke i kredsen Z og en knude j i kredsen Z, således at længden $w(k, j)$ er *længst* mulig. Lad $\{i, j\}$ være kanten, som minimerer $w(i, k)+w(k, j)-w(i, j)$. Erstat $\{i, j\}$ med $\{i, k\}$ og $\{k, j\}$. Se figur 9.4.

Selv om den fjerneste indsættelsesalgoritme måske umiddelbart ser ud til at benytte en fejlagtig strategi, ender den ofte op med en Hamilton-kreds, som er bedre end, hvad nærmeste og billigste typisk præsterer. Dette hænger sammen med, at fremgangsmåden for fjerneste indsættelsesalgoritme rimeligt hurtigt får produceret en kreds, som i grove træk formår at skitsere, hvorledes den endelige kreds kommer til at se ud.

Algoritme 8 er pseudokoden for nærmeste indsættelsesalgoritme.

Pre-betingelser til algoritmen er, at der findes en komplet graf G med V knuder og E kanter. Vægten på kanten fra a til b skrives med $w(a, b)$. Post-betingelsen er, at der returneres en TSP-løsning.

Algoritme 8 *Nærmeste indsættelsesalgoritme*

i :=*en hvilken som helst knude* G { *startknuden*}
knuden j :=*beregn sådan at* $w(i,j) =$
  mindstevægt$\{w(i,r)$ *hvor* $r \in V - \{i\}\}$
S :=*knuderne* i, j
C :=*mængden af kanterne* $\{(i,j),(j,i)\}$
while $S \neq V$
begin
  knuden k :=*den knude* r *hvor* $r \in V - S\}$
  $S := S \cup \{k\}$
  find en kant $(u,v) \in C$ *hvor* $w(u,k) + w(u,v) =$
    mindstevægt$\{w(x,k)+w(k,y)-w(x,y)$ *hvor* $(x,y) \in C\}$
  tilknyt kanterne (u,k) *og* (k,v) *til* C*, og fjern* (u,v) *fra* C
end

Gennem løbs tiden for denne algoritme vil være $O(n^2)$ i følge referencen[6]. hvilket vil opfylde vores betingelser for polynomiel tid i følge vores kapitel 2.1 .

Christofides' algoritme

Christofides' algoritme er den af de heuristiske konstruktionsalgoritmer, som har det bedste *worst-case* tilfælde. Christofides' algoritme producerer nemlig aldrig en Hamilton-kreds, som er længere end 1,5 gange den optimale Hamilton-kreds (se sætning 14).

Nærmeste nabo og indsættelsesalgoritmerne kan ganske vist i nogle tilfælde producere en bedre kreds end Christofides, men til gengæld kan man også nemt løbe ind i tilfælde, hvor disse to metoder giver et resultat, som er meget længere end 1,5 gange den optimale kreds.

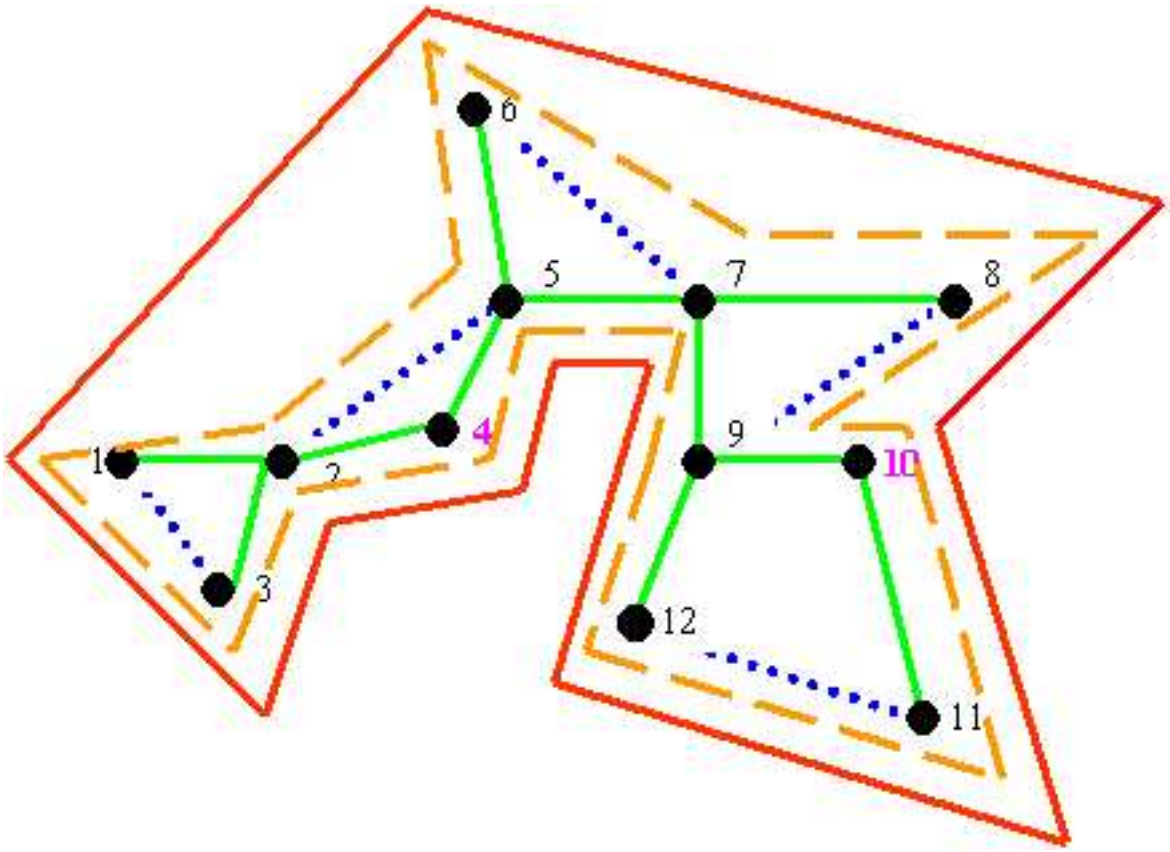
Definition 44 *Perfekt matchende mængde*

Lad $G = (V, E)$ være en komplet graf. M er en perfekt matchende kantmængde, hvor $M \subseteq E$. En kant i M må ikke være nabokant til andre kanter M. For alle knuder i V gælder det, at en knude har netop én tilstødende kant fra M.

Fremgangsmåden for Christofides' algoritme kan eksemplificeres således:

- Find et minimum udspændende træ T af en komplet graf G (grøn i figur 9.5).
- Lad knuderne i T med ulige grad udgøre mængden S.
- Find en perfekt matchende kantmængde M ud fra S
  (blå $\{1-3, 2-5, 6-7, 8-9, 11-12\}$ i figur 9.5).
- Lav en multigraf MG ud fra kanterne i T og M.
- Find en Euler-kreds i MG
  (orange $\{1, 3, 2, 4, 5, 7, 9, 12, 11, 10, 9, 8, 7, 6, 5, 2, 1\}$ i figur 9.5).
- Brug trekantsuligheden til at omforme Euler-kredsen til en Hamiltonkreds, således at knuder allerede besøgt én gang springes over
  (rød $\{1, 3, 2, 4, 5, 7, 9, 12, 11, 10, 8, 6, 1\}$ i figur 9.5).

Pre-betingelser til algoritmen er, at der findes en komplet graf G med V knuder og E kanter. Vægten på kanten fra a til b skrives med $w(a, b)$. Postbetingelsen er, at der returneres en TSP-løsning.



Figur 9.5: Christofides, støttetegning [3]

Algoritme 9 *Christofides' algoritme*

$T :=$ minimum udspændende træ af G
$S :=$ alle knuder med ulige grad i T
find en perfekt matchende mængde M ud fra S ved at sammenligne $w(i, j)$
$MG :=$ multigraf ud fra kanterne i T og M
find en Euler-kreds i MG se også definition 35
for *$i := 1$ til antallet af knuder i mængden S*
begin
    if *graden af knuden $S_i > 2$*
    begin
        Fjern alle kanter undtagen to fra knuden S_i
        Brug trekantsuligheden til at omforme Euler-kredsen til en Hamilton-kreds
    end
end

Gennem løbs tiden for denne algoritme vil være $O(n^3)$ i følge referencen[6]. hvilket vil opfylde vores betingelser for polynomiel tid i følge vores kapitel 2.1 .

Sætning 14

Christofides' algoritme er en approksimeringsalgoritme, som aldrig producerer en Hamilton-kreds længere end 1,5 gange den optimale Hamilton-kreds.

Vi vil bruge et direkte bevis for at bevise, at sætningen er korrekt.

Bevis for sætning 14

Lad V være en mængde af knuder og E være en mængde af kanter. Lad grafen $G = (V, E)$ være en komplet graf, og mængden Z_{opt} er den optimale

Hamilton-kreds i grafen G Lad mængden Z_{min} være et minimum udspændende træ af grafen G og mængden S være knuder i Z_{min} med ulige grad. Ud fra S laves en perfekt matchende kantmængde M. Mængden Z_{euler} er en Euler-kreds dannet ud fra Z_{min}. Mængden Z_c er den Hamilton-kreds dannet af christofides algoritme ud fra Z_{euler}.

- Med udgangspunkt i trekantsuligheden omformes en Euler-kreds Z_{euler} til en Hamilton-kreds Z_c. Dermed er $w(Z_c) \leq w(Z_{euler})$. Dette bevirker således også, at $w(Z_c) \leq w(Z_{min}) + w(M)$ **(1)**.

- Vi ved, at længden af et minimum udspændende træ Z_{min} fundet i G er kortere end den optimale kreds Z_c i G, da fjernelsen af en kant i Z_{opt} netop producerer et Z_{min}. Således får vi $w(Z_{min}) \leq w(Z_c)$ **(2)**.

- I det følgende bevises det, at $w(M) \leq \frac{1}{2} \cdot w(Z_{opt})$:

  Jvf. figur 9.6, som viser en Hamilton-kreds (cirklen), hvor knuderne S med ulige grad fra Z_{min} er fremhævet.

  Foruden Hamilton-kredsen er der ud fra knudemængden S to perfekt matchende kantmængder $M1$ og $M2$. Jvf. de tynde og de tykke kanter i figuren.

  Betragt figuren og se, at vi med udgangspunkt i trekantsuligheden får $w(M1) + w(M2) \leq w(Z_{opt})$.

  For de to kantmængder $M1$ og $M2$ gælder det, at de enten er lige lange, eller også er den ene kantmængde større end den anden. Hvis vi i det følgende antager, at $M1$ er mindre end eller lig med $M2$ får vi:

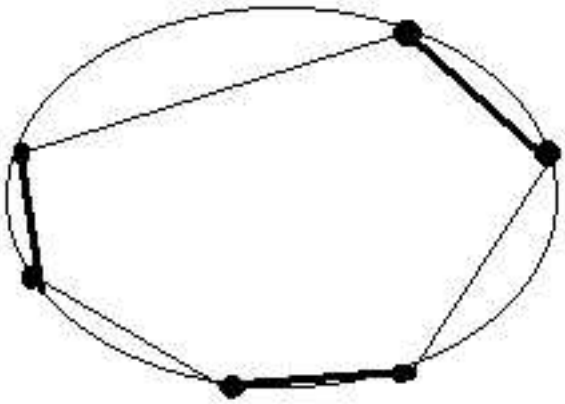
  $w(M1) \leq w(M2) \Rightarrow 2 \cdot w(M1) \leq w(Z_{opt}) \Rightarrow$
  $w(M1) \leq \frac{1}{2} \cdot w(Z_{opt})$ **(3)**.

- Ved at kombinere følgende tre uligheder

  (1) $w(Z_c) \leq w(Z_{min}) + w(M)$
  (2) $w(Z_{min}) \leq w(Z_{opt})$
  (3) $w(M1) \leq \frac{1}{2} \cdot w(Z_{opt})$

  får vi $w(Z_c) \leq \frac{3}{2} \cdot w(Z_{opt})$.

$\triangleleft$



Figur 9.6: Christofides, bevisfremførelse

Kapitel 10

Konklusion

For at kunne sammenligne forskellige løsninger skal man have et grundlag at gå ud fra, og forskellige løsninger kan være optimale på forskellige grundlag. Vi har derfor valgt at liste vores forskellige løsningsforslag og deres karakteristika. Hele formålet med denne rapport er at finde en optimal løsning på TSP problemet, som beskrevet i indledningen.

Her er brute force den optimale løsning, da denne finder den korteste Hamiltonkreds, men tidskompleksiteten for den er den værst tænkelige, da den er $O(n!)$. Dette opfylder ikke vores krav om, at TSP skal kunne løses i polynomiel tid.

Et andet løsningsforslag er to gange minimum algoritmen, der ikke finder den optimale løsning. Den udarbejder et minimum udspændende træ og fordobler kanterne af træet, så den kan lave en Euler-kreds. Herved bliver længden af TSP løsningen til to gange det minimum udspændende træ. Denne løsning er acceptabel, men ikke optimal, længden taget i betragtning.

Nærmeste nabo er måske umiddelbart den mest logiske algoritme at bruge, når man rent fysisk skal finde den korteste vej. Der er ingen kendt øvre grænse, for hvor lang en vej vil være i værste fald. Tidskompleksiteten vil være $O(n^2)$ [6], hvilket er hurtigere end brute force.

Nærmeste indsættelsesalgoritme har en tidskompleksitet på $O(n^2)$, mens billigste indsættelsesalgoritme har en tidskompleksitet på $O(n^2 \log n)$. De producerer begge en *worst-case* kreds på to gange den optimale kreds. Fjerneste indsættelsesalgoritme har en tidskompleksitet på $O(n^2)$, mens *worst-case* tilfældet af en produceret kreds kan ikke præcist fastslås [5].

Christofides' algoritme producerer, sammenlignet med de tre indsættelsesalgoritmer, aldrig en kreds længere end $\frac{3}{2}$ af den optimale kreds. Dette ga-

ranterer et bedre *worst-case* tilfælde, end de tre indsættelsesalgoritmer kan præstere. Til gengæld er dens tidskompleksitet dårligere, nemlig $0(n^3)$.

Ud fra ovenstående betragtninger og ud fra det faktum, at en algoritme, der løser problemet optimalt i polynomiel tid, endnu ikke er kendt, kan vi konkludere, at der endnu ikke kan findes én endegyldig bedste løsning på TSP. Valget af løsning (algoritme) afhænger af flere parametre, herunder krav til tidskompleksitet samt krav til kvaliteten af den genererede kreds.

Hvis de undersøgte algoritmer i denne opgave skal sættes i relation til erhvervslivet, skal kunders/brugeres behov tages i betragtning. Et sandsynligt krav fra en kunde vil formentlig være, at der skal kunne stilles en garanti for kørselstiden af den valgte algoritme. Med dette som grundlag ville vi i en sådan situation vælge Cristofides' algoritme, da en udvikler med den kan give kunden en rimelig estimering på kørselstid og kvaliteten af løsningen.

Litteratur

[1] The american heritage® dictionary of the english language. http://www.yourdictionary.com/ahd/h/h0179600.html.

[2] Frank M. Carrano. *Data Abstraction and Problem Solving with JAVA*. Addison Wesley, 1st edition, 2001. ISBN 0-201-70220-7.

[3] Hui Chen. Approximation algorithms for tsp. http://www.msci.memphis.edu/ giri/7713/f00/HuiChen/HuiChen2.htm.

[4] William J. Cook m.fl. *Combinatorial Optimization*. Wiley-Interscience, ? edition, 1998. ISBN 0-471-55894-X.

[5] Giovanni Righini. The largest insertion algorithm for the tsp. http://sansone.crema.unimi.it/ righini/Papers/larg_ins2.ps.

[6] Kenneth H. Rosen. *Handbook of Discrete and Combinatorial Mathematics*. Library of Congress Cataloging-in-Publication Data, ? edition, 2000. ISBN 0-8493-0149-1.

[7] Kenneth H. Rosen. *Discrete Mathematics and Its Applications*. McGraw-Hill, 5th edition, 2003. ISBN 0-07-119881-4.

[8] Steven S. Skiena. The algorithm design manual. http://www2.toki.or.id/book/AlgDesignManual/BOOK/BOOK/BOOK.HTM.