Grafer

Definition

En *graf* er pr. definition et par $G = (V, E)$. Grafen består af en mængde *knuder* V (eng: vertices) og en mængde *kanter* E (eng: edges), som forbinder knuderne.

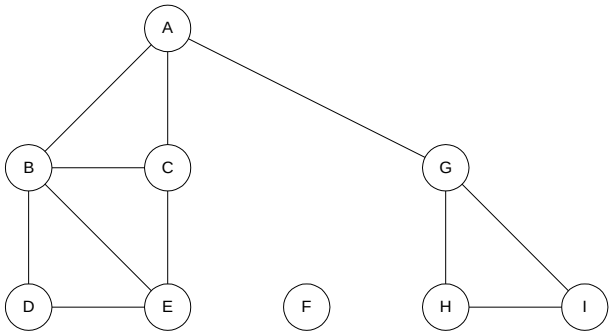


Fig: Ikke-orienteret graf

- I *ikke-orienterede grafer* benævnes kanterne (u, v), $u, v \in V$ og $(u, v) = (v, u)$.

- For *orienterede grafer* spiller retningen derimod en rolle, og vi anvender derfor ofte en anden notation: $\langle u, v \rangle$. Orienterede grafer kaldes også for *digrafer* (Directed Graphs).

- Ofte tilknyttes der en vægt eller omkostning til kanterne, og grafen benævnes da en *vægtet graf*. Vægten kan beskrive enhver metrik:

    - Omkostning
    - Afstand
    - Tid
    - Straf
    - Tab

    Eller en anden kvantitativ størrelse, som akkumulerer lineært langs stien, som vi ønsker at optimere.

Anvendelse

Grafer anvendes til at modellere problemer, hvor nogle objekter er indbyrdes forbundne. Fx kan et jernbanenet beskrives vha. en ikke-orienteret graf, hvor knuderne er stationer, og kanterne er banestrækninger. Afstanden mellem to stationer kan beskrives vha. vægten på den tilhørende kant.

Grafer, hvor objekterne afhænger af hinanden, modelleres vha. orienterede grafer. Fx kan et projekt med dets indbyrdes forbundne aktiviteter med fordel beskrives som en orienteret graf. Her kan vægten på kanten mellem to aktiviteter fx beskrive udstrækningen i tid af den første aktivitet.

Begreber

Grafteorien indeholder en righoldig nomenklatur, og herunder listes de væsentligste begreber. Definitionerne gives – hvor andet ikke er nævnt – ud fra ikke-orienterede grafer; men

definitionerne kan umiddelbart overføres til orienterede grafer. Bemærk tillige, at en del begreber kan være defineret forskelligt i forskellig litteratur.

- En *sti* (path) er en følge af knuder $v_1, v_2, ..., v_n$, hvor $(v_i, v_{i+1}) \in E, 1 \le i \le n$
  Stiens længde er antallet af kanter = $n - 1$; medens *stiens vægt* er summen af kanternes vægte.

- En *simpel sti* er en sti, hvor alle knuder – evt. på nær den første og den sidste – er forskellige.

- En *kreds* eller *cykel* er i en orienteret graf er en sti med længde mindst én, hvor $v_1 = v_n$
  Kredsen er *simpel*, hvis stien er simpel

- For ikke-orienterede grafer vil vi kræve, at kanterne i en kreds skal være forskellige.

- Hvis vi tillader en sti fra en knude til sig selv, så er denne stis længde lig 0 (nul)
  En sådan kant (v, v) kaldes også en *løkke* (eng: loop)

- En graf uden kredse kaldes en *acyklisk graf*

- En graf kaldes en *fuldstændig graf*, hvis ethvert par af knuder er forbundet med en kant

- En graf kaldes en *sammenhængende graf*, hvis ethvert par af knuder er forbundet med en sti. En orienteret graf kaldes *stærkt sammenhængende*, hvis ethvert knudepar er forbundet med en (orienteret) sti. Bemærk tillige, at der således må være en sti begge veje!

- *Graden* af en knude er dens antal forbindelser til kanter.

- I orienterede grafer taler man tillige om *udgraden* (antallet af kanter ud fra knuden) og *indgraden* (antallet af kanter ind i knuden). Graden af en knude i en orienteret graf er da summen af indgraden og udgraden.
  Det er oplagt, at summen af indgrader må være lig summen af udgrader i en orienteret graf, altså: $\sum_{v \in V} \text{indgrad}(v) = \sum_{v \in V} \text{udgrad}(v) = |E|$, hvor $|E|$ er antallet af kanter i grafen. Da hver kant således bidrager med 2 grader, vil summen af alle grader i grafen være et lige tal, udtrykt ved:

$$\sum_{v \in V} \text{grad}(v) = 2|E|$$

- Knuder med grad 0 (nul) kaldes *isolerede knuder*

- En sammenhængende acyklisk graf kaldes også for et *frit træ*

- Hvis vi fjerner betingelsen om sammenhæng får vi i stedet for en *skov* (af træer)

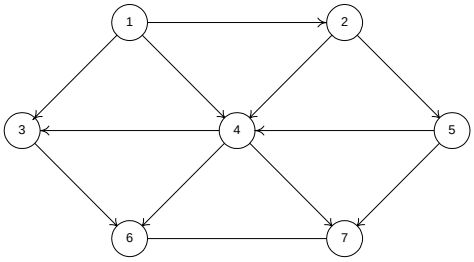
- De træer, vi kender fra tidligere, kan betragtes som orienterede grafer, idet kanterne går fra forælder til børn. Man bruger betegnelsen *orienterede træer* om disse for at skelne dem fra de frie træer defineret ovenfor. Orienterede træer er tillige kendetegnet ved at have en *rod* (en "forælderløs" knude).

Repræsentation af grafer

Grafer kan repræsenteres på flere forskellige måder. Herunder vil vi se på to klassiske måder: *Adjacency Matrix*- og *Adjacency List*-repræsentationen.

Adjacency Matrix

Betragt følgende ikke-orienterede graf, hvis knuder er nummererede, og grafens tilhørende adjacency matrix-repræsentation



Orienteret graf

	1	2	3	4	5	6	7
1	0	1	1	1	0	0	0
2	0	0	0	1	1	0	0
3	0	0	0	0	0	1	1
4	0	0	1	0	0	1	1
5	0	0	0	1	0	0	0
6	0	0	0	0	0	0	0
7	0	0	0	0	0	1	0

Adjacency Matrix

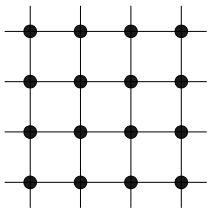
Matricens celler indeholder 1 (sand), hvis kanten findes og 0 (falsk), hvis kanten ikke findes. Hvis det drejer sig om vægtede grafer, så indsættes vægtene på 1'ernes plads i matricen. 0'erne vil blive erstattet af $\pm\infty$ som indikation på, at kanten ikke findes. Kanten bliver med andre ord "for dyr" at passere.

Hvis grafen er fuldstændig haves:

$$|E| = \Theta(|V|^2)$$

hvor $|E|$ er antallet af kanter og $|V|$ er antallet af knuder i grafen. Den tilhørende adjacency matrix bliver således *tæt* (eng: dense).

Hvis grafen derimod ikke er fuldstændig og fx indeholder knuder med en konstant grad, så bliver den tilhørende adjacency matrix *tynd* (eng: sparse), hvilket vil sige, at den får et stort indhold af 0'er. Dette er ikke hensigtmæssigt. Betragt følgende vejnet a la Manhattan:



"Manhattan"-vejnet

Lad knuder agere vejkryds og lad kanterne være vejstrækninger. I en sådan graf vil antallet af kanter være:

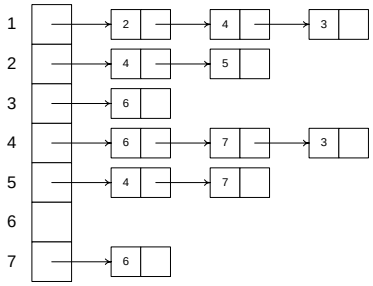
$$|E| \cong 4|V|$$

Hvis en sådan graf består af 4 knuder (vejkryds), så vil den tilhørende adjacency matrix indeholde 50% 1'ere. Består grafen af 8 knuder, så er antallet reduceret til 25%, osv. Pointen er, at hvis antallet af kanter i grafen vokser lineært med antallet af knuder, så vil den tilhørende

adjacency matrix indeholder en tiltagende andel af 0'er og dermed blive mere og mere tynd. For at afhjælpe denne problemstilling vil vi vende os mod en anden grafrepræsentation, nemlig adjacency-listen:

Adjacency List

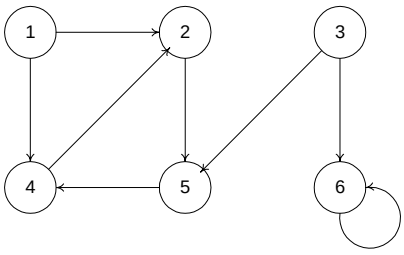
I adjacency listen opretholdes til hver knude en lænket liste bestående af naboknuder. Bemærk, at hvis grafen er ikke-orienteret, så vil alle kanter (u,v) forekomme i to lister. Herunder er adjacency-listen for grafen tidligere i afsnittet vist:



Adjacency list

Pladskompleksiteten for denne type grafrepræsentation er $O(|E|+|V|)$, og den er således lineær i forhold til grafens størrelse. Dette gør den anvendelig til grafer, der ikke er fuldstændige, hvorimod matrix-repræsentationen er god, når det drejer sig om fuldstændige eller næsten-fuldstændige grafer.

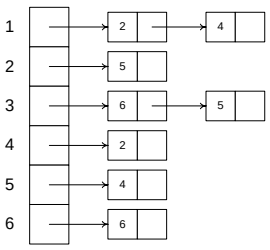
Eksempel – Orienteret graf



Orienteret graf

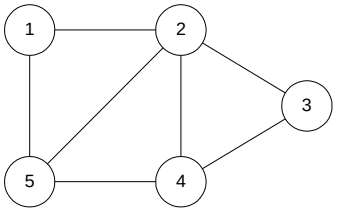
	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Adjacency Matrix



Adjacency List

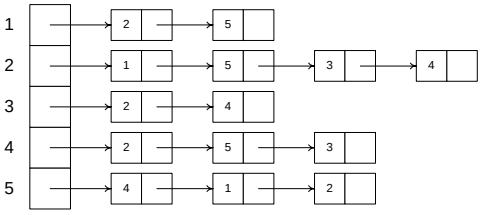
Eksempel – Ikke-orienteret graf



Ikke-orienteret graf

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Adjacency Matrix



Adjacency List

Bemærk, at i adjacency matrix-repræsentationen er det reelt kun den øvre (eller nedre) matrix, der anvendes, idet celleværdierne spejles i diagonalen. Man kan betragte en ikke-orienteret kant som to parallelle og modsatrettede orienterede kanter. Dette forhold betyder desuden, at knuderne optræder to gange i adjacency list –repræsentationen.

Grafgennemløb

Da grafer er meget mere komplicerede objekter, skulle man tro, at der var et væld af gennemløbsmetoder. Litteraturen indskrænker sig dog sædvanligvis til to typer: Bredde-først-søgning (BFS) og Dybde-først-søgning (DFS).

BFS

Med udgangspunkt i en given knude besøger bredde-først-søgningen først sine naboer, (dvs. de knuder, hvortil den aktuelle knude er forbundet), og derefter sine naboers naboer. På denne vis breder søgningen sig som ringe i vandet, indtil alle grafens knuder er besøgt. Om grafen skal naturligvis gælde, at den er sammenhængende. Den resulterende graf er et træ, der udspænder alle de knuder, det er muligt at nå fra udgangspunktet.

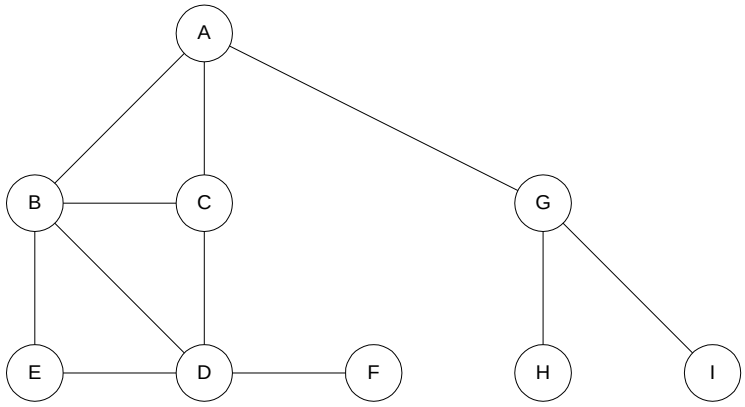


Fig: Ikke-orienteret graf

Når algoritmen bevæger sig gennem en graf (V, E), så udgør $E' \subseteq E$ mængden af kanter, som er besøgt. Vi vil for illustrationens skyld udstyre knuderne med en attribut *farve*, hvor *hvid* angiver, at knuden er ubesøgt; *grå* at knuden er opdaget, men ikke undersøgt; og endelig *sort*, som angiver, at knuden er færdigbehandlet.

```
Præ: G=(V, E) er en sammenhængende graf, s er startknuden

BFS(s)
    Q ← ∅
    for alle knuder u ∈ V\{s}
        u.farve ← hvid
    s.farve ← grå
    Q.enqueue(s)
    sålænge Q ≠ ∅
        u ← Q.dequeue()
        for alle knuder v ∈ Adj(u) // alle naboer til u
            hvis v.farve = hvid
                v.farve ← grå
                Q.enqueue(v)
        u.farve ← sort
```

Tidskompleksiteten for løkken, hvor nabolisten til hver knude inspiceres, er $O(|E|)$, da den samlede længde af disse lister er $\Theta(|E|)$. Med initialiseringen bliver algoritmens tidskompleksitet i alt $O(|V|+|E|)$

Dybde-først-søgning

Når dybde-først-algoritmen bevæger sig gennem en graf (V,E), så udgør $E' \subseteq E$ mængden af kanter, som er besøgt. Vi vil også her udstyre knuderne med en attribut *farve*, som angiver, hvorvidt knuden er hhv. ubesøgt, opdaget eller færdigbehandlet.

Bemærk, at dybde-først-algoritmen "skynder sig" at komme så langt væk fra sit udgangspunkt som muligt, idet den gemmer de nære knuder til sidst.

DFS

Dybde-først-søgning formuleres bedst rekursivt. Se følgende pseudokode:

```
Præ: (V, E) er en ikke-orienteret, sammenhængende graf

DFS(u)
    u.farve ← grå
    for alle knuder v ∈ Adj(u) // alle naboer til u
        hvis v.farve = hvid
            DFS(v)
    u.farve ← sort
```

Traverseringen startes med følgende initialiseringer:

```
For alle knuder u ∈ V
    u.farve ← hvid
u ← vilkårlig knude
DFS(u)
```

Bemærk, at algoritmen besøger samtlige knuder i grafen, da den er sammenhængende.

Tidskompleksiteten for DFS-algoritmen, er $O(|V|+|E|)$. Ligesom ved BFS er tidskompleksiteten for løkken, hvor nabolisten til hver knude inspiceres $\Theta(|E|)$, medens initialiseringen tager $O(|V|)$.

Algoritmen udført på grafen fra tidligere kunne fx beskrives ved følgende sekvens, hvor indrykningen beskriver rekursionsdybden.

```
DFS(A)
    DFS(B)
        DFS(C)
            DFS(D)
                DFS(E)
                DFS(F)
    DFS(G)
        DFS(H)
        DFS(I)
```

Algoritmen besøger samtlige knuder i grafen, hvis den er sammenhængende.

Såfremt vi ikke ved, om grafen er sammenhængende, så vil vi i stedet for foretage følgende initialiseringer:

```
For alle knuder u ∈ V
    u.farve ← hvid
For alle knuder u ∈ V
    Hvis u.farve = hvid
        DFS(u)
```

Eulergraf

En Euler-graf er en graf, der indeholder en Euler-tur, som defineres ved følgende:

Eulertur

En Euler-tur er en kreds i en sammenhængende orienteret graf $G = (V, E)$, som gennemløber hver kant netop én gang. Knuderne må gerne passeres flere gange.

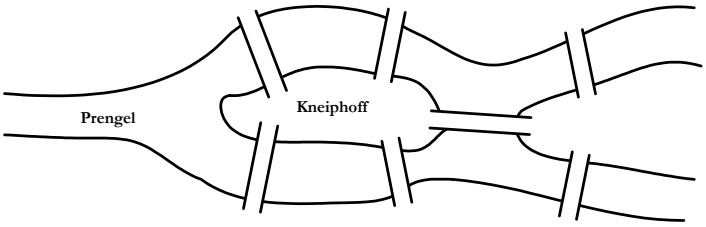
Det kan vises, at G har en Euler-tur, hvis og kun hvis der for alle knuder i grafen gælder, at:

$$\text{indgrad}(v) = \text{udgrad}(v)$$

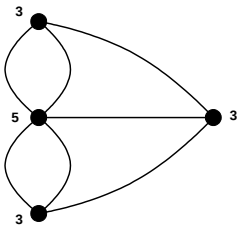
Det følger endvidere af ovenstående, at en Euler-tur kun kan starte og slutte i samme knude, hvis samtlige knuder har lige grad. Hvis der findes knuder af ulige grad i grafen, så må der maksimalt være to – (Der vil altid være et lige antal!) – og turen må så nødvendigvis starte i den ene og ende i den anden.

Baggrund

Gennem byen *Königsberg* i Østprøjsen, (nu Kaliningrad, Rusland), løber floden *Prengel*. I 1700-tallet var der syv broer, som forbandt byen, og det var blandt det bedre borgerskab en yndet tradition at lægge sin spadseretur rundt i byen, så man krydsede alle syv broer. Herved opstod spørgsmålet, om det var muligt at tilrettelægge sin tur, så man passerede hver bro netop én gang. Dette problem var i mange år uløst, indtil man rådspurgte matematikeren *Leonhard Euler*, der på det tidspunkt (1736) arbejdede i Skt. Petersborg. Han viste ovenstående, og er siden blevet betragtet som den moderne grafteoris fader.



Floden Prengels løb gennem Königsberg, som det så ud på Eulers tid (1707-83)



Graf, hvor knuderne illustrerer de fire landmasser og kanterne de syv broer. Tallene angiver knudernes grader

Topologisk sortering

For orienterede grafer er kravet om (stærk) sammenhæng meget restriktivt, idet en stærkt sammenhængende orienteret graf nødvendigvis indeholder kredse.

Topologisk Sortering – Algoritme nr. 1

Vi skal herunder se på en modificeret DFS, hvor vi vil sørge for, at skoven kommer til at bestå af orienterede træer. Dette opnås ved at lade de nye "skud" være orienterede.

```
Præ: (V, E) er en orienteret, acyklisk graf
Post: Stak af knuder, der er topologisk ordnet

DFS(u)
    u.farve ← grå
    for alle v ∈ Adj(u) // alle naboer til u
        hvis v.farve = hvid
            DFS(v)
    S.push(u)
    u.farve ← sort
```

Traverseringen startes med følgende initialiseringer:

```
S ← ∅
for alle knuder u ∈ V
    u.farve ← hvid
for alle knuder u ∈ V
    Hvis u.farve = hvid
        DFS(u)
```

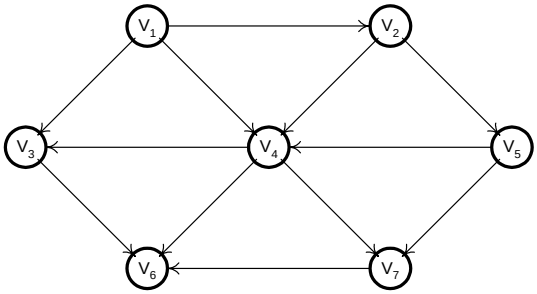
Tidskompleksiteten for denne algoritme, er ligesom for dybde først-søgningen på ikke-orienterede grafer $O(|V| + |E|)$.

Stakken indeholder nu en ordning af grafens elementer. Denne ordning eller sortering kaldes, (uvist af hvilken grund), for *topologisk sortering*. Om topologisk sortering kan følgende siges:

Givet en orienteret graf, der fx beskriver afhængighederne mellem grafens elementer udtrykt i orienteringen af kanterne mellem knuderne.

- En topologisk sortering er en følge af knuder, der ikke bryder med disses afhængigheder.
- En topologisk sortering er ikke mulig, hvis grafen indeholder kredse, da der for to knuder *u* og *v*, således må gælde, at *u* kommer før *v*, og *v* samtidig kommer før *u*.

En topologisk sortering er ikke nødvendigvis entydig, se følgende eksempel:



Begge følger:

$v_1, v_2, v_5, v_4, v_3, v_7, v_6$ og
$v_1, v_2, v_5, v_4, v_7, v_3, v_6$

udgør en topologisk ordning af knuderne i ovenstående graf.

Topologisk Sortering – Algoritme nr. 2

Hvis den forrige algoritme synes kompliceret, så gives herunder en særdeles simpel algoritme:

1. Find en knude med indgrad 0 (nul)
2. Udskriv knuden
3. Fjern knuden sammen med dens kanter
4. Gentag 1 – 3, indtil alle knuder er fjernet

Knuderne er nu udskrevet i en topologisk ordning. Omkostningen forbundet med punkt 1 – 3 er $O(|V|)$, da alle knuder nødvendigvis må gennemløbes for at finde en knude med indgrad = 0. Punkterne 1 – 3 gennemføres i alt $|V|$ gange, hvorfor algoritmens samlede tidskompleksitet må være $O(|V|^2)$.

Topologisk Sortering – Algoritme nr. 3

Vi ved, det kan gøres bedre, så lad os se på følgende forbedrede version af sidste algoritme:

1. Saml alle knuder med indgrad = 0 i en kø
2. Fjern én knude ad gangen, og opdatér samtidig indgraden på dens naboer
3. Hvis en knudes indgrad falder til nul, så placeres den i køen

Hvis det antages, at knuderne allerede findes i en adjacency-liste, og at indgraderne på forhånd er beregnet, så fås en tidskompleksitet på $O(|V| + |E|)$.

Korteste sti

Problemet med at finde den korteste sti mellem to vilkårlige knuder i en graf er et af de klassiske problemer inden for grafalgoritmer.

Problemet kan beskrives på følgende måde:

Givet en vægtet orienteret graf $G = (V, E)$ med en vægtfunktion w, som afbilder kantmængden over i de reelle tal. Vægten af en sti $p = \langle v_0, v_1, ..., v_k \rangle$ er lig med summen af kanternes vægte:

$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Hvis grafen ikke er vægtet er summen lig med $k - 1$, (antallet af kanter mellem knuderne).

Korteste sti fra knude u til v defineres ved:

$$\delta(u,v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightarrow} v\} \\ \infty \end{cases}$$

hvor $u \overset{p}{\rightarrow} v$ angiver stien mellem u og v, hvis denne findes. Korteste sti fra u til v defineres således som enhver sti p med $w(p) = \delta(u, v)$.

Der findes adskillige algoritmer til at løse korteste sti-problemet, alt efter hvorledes grafen er defineret. Problemet findes i flere variationer:

Variationer af korteste sti-problemet

Single source shortest path

Givet en graf $G = (V, E)$ og en knude s (source). Find korteste sti fra s til enhver anden knude $v \in V$ i grafen.

Single destination shortest path

Givet en graf $G = (V, E)$ og en knude t (sink). Find korteste sti til t fra enhver anden knude $v \in V$ i grafen. Problemet løses ved at vende retningen på alle kanter og anvende single source shortest path-algoritmen med t som argument.

Single pair shortest path

Givet en graf $G = (V, E)$. Find korteste sti fra u til v for et givent knudepar (u, v). Dette problem løses også vha. single source shortest path-algoritmen. Der kendes ingen algoritmer, der kører asymptotisk bedre end single source shortest path-algoritmen

All pairs shortest path

Givet en graf $G = (V, E)$. Find korteste sti fra u til v for alle par (u, v). Problem løses ved at løse single source shortest path-problemet på hver knude. Problemet kan også løses som et selvstændigt problem.

Negative kredse

Hvis grafen indeholder kredse med negativ vægt, så er korteste sti-problemet udefineret.

Korteste sti-træ

Et korteste sti-træ er en orienteret delgraf $G' = (V', E')$, hvor $V' \subseteq V$ og $E' \subseteq E$, hvor følgende gælder:

- G' er et træ med rod i s
- V' er mængden af knuder, som kan nås fra s
- For enhver knude $v \in V$ gælder, at den entydige simple sti fra s til v i G' er en korteste sti fra s til v i G.

Korteste stier er ikke nødvendigvis entydige, ej heller korteste sti-træer.

Algoritmerne

Bredde først søgning (Moore)

BFS (Moore's algoritme) er en korteste sti-algoritme, som virker på ikke-vægtede sammenhængende grafer eller grafer, hvor hver kant er vægtet med én.

Dijkstra

Dijkstra's algoritme forudsætter, at grafen ikke indeholder negative vægte.

Bellman-Ford

Bellman-Ford's algoritme kan godt håndtere negative vægte; men den tillader ikke negative kredse. Ydermere kan algoritmen detektere og rapportere negative kredse, hvis disse forefindes.

Floyd-Warshall

Floyd-Warchall's algoritme kan også håndtere negative vægte, og kan i lighed med Bellman-Ford's algoritme detektere negative kredse.

Moore's algoritme

Den første algoritme, som vi skal kigge på går ofte under navnet Moore's algorime. Algoritmen finder den korteste sti i en ikke-orienteret og ikke-vægtet graf og er blot en modifikation af BFS-algoritmen. Modifikationen går ud på, at man tillige vedligeholder en afstands- og forgænger-attribut på knuden, hvor værdien af afstandsattributten angiver knudens afstand fra udgangsknuden; medens den korteste sti kan findes vha. backtracking gennem forgænger-attributterne:

```
Præ: G=(V, E) er en ikke-vægtet graf, s er startknuden
Post: Alle knuder er mærket med afstanden fra s

Moore(G, s)
    Q ← ∅
    for alle knuder u ∈ V\{s}
        u.farve ← hvid
        u.afstand ← ∞
        u.forgænger ← null
    s.farve ← grå
    s. afstand ← 0
    s.forgænger ← null
    Q.enqueue(s)
    sålænge Q ≠ ∅
        u ← Q.dequeue()
        for alle v ∈ Adj(u):
            hvis v.farve = hvid
                v.farve ← grå
                v.afstand ← u.afstand + 1
                v.forgænger ← u
                Q.enqueue(v)
        u.farve ← sort
```

Tidskompleksiteten for Moore's algoritme er med samme argumenter som for BFS lig med $O(|V| + |E|)$.

Det vil sandsynligvis være sværere at finde den korteste sti i en vægtet graf, så derfor må vi være forberedte på, at den algoritme, vi finder frem til, har en større tidskompleksitet.

Dijkstra

Dijkstra's algoritme anvender en teknik, der kaldes relaxering. Relaxeringsteknikken går ud på, at man løbende på hver knude nedskriver den øvre grænse på værdien af den korteste sti fra udgangsknuden *s* til den givne knude. Denne proces fortsætter indtil den øvre grænse er lig med den korteste sti.

Relaxering

```
Relaxér(u, v)
    hvis v.afstand > u.afstand + w(u, v)
        v.afstand ← u.afstand + w(u, v)
        v.forgænger ← u
```

hvor w betegner kantvægten. Teknikken går som sagt ud på, at man vedligeholder en attribut afstand, som er et *estimat* – en øvre grænse – på værdien af den korteste sti fra *s* til *v*.

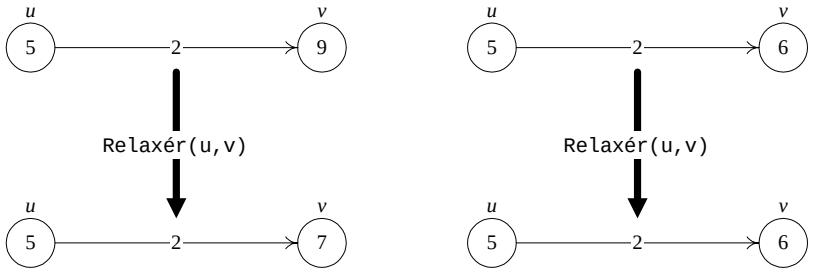


Fig: Relaxering

På ovenstående figur er relaxeringen på kanten (*u*, *v*) vist. Tallet inde i knuden angiver det aktuelle korteste sti-estimat, medens tallet på kanten angiver vægten *w*.

Da `v.afstand > u.afstand + w(u, v)` i figuren til venstre, så relaxeres estimatet på knude v fra 9 til 7. I figuren til højre er `v.afstand ≤ u.afstand + w(u, v)` og estimatet forbliver uændret.

Initialisering

```
Initialisér(G, s)
    for alle knuder u ∈ V
        u.afstand ← ∞
        u.forgænger ← null
    s.afstand ← 0
```

Initialiseringen sætter afstands-attributten på alle knuder til ∞, på nær startknuden selv, som har afstands-attributten 0. Forgænger-attributten anvendes også her til slutteligt at afrapportere den korteste sti vha. backtracking.

Dijkstra

```
Præ: G=(V, E) er en vægtet graf uden negative vægte.
     Vægtfunktionen er givet ved w, s er startknuden
Post: Alle knuder er mærket med afstanden fra s

Dijkstra(G, s)
    Initialisér(G, s)
    Q ← V // liste indeholdende alle grafens knuder
    sålænge Q ≠ ∅
        u ← Q.deleteMin() // udtræk knuden med mindste afstand
        for alle v ∈ Adj(u) // alle naboer til u
            Relaxér(u, v)
```

Algoritmen arbejder sig gennem grafens knuder ved altid"grådigt" at vælge den forhåndenværende bedste knude, dvs. knuden med det mindste estimat i afstands-attributten. Derefter relaxeres kanterne til knudens naboer. På denne vis relaxeres hver kant i grafen netop én gang. Dette grådige valg tvinger algoritmen mod en hurtig terminering, men det er på den anden side også dette valg, der fratager algoritmen evnen til at håndtere negative vægte

Tidskompleksiteten for algoritmen er $O(|E|+|V|^2) = O(|V|^2)$, da `Q.deleteMin()` koster $O(|V|)$, og denne operation foretages i alt $|V|$ gange; dernæst gennemløbes samtlige kanter i for-løkken. Hvis Q implementeres som en prioritetskø, kan tidskomplesiteten mindskes til $O(|E|+|V|\log|V|)$.

Bellman-Ford.

I lighed med Dijkstra's algoritme, så anvender Bellman-Ford's algoritme også relaxeringsteknikken, hvor man løbende estimerer den øvre grænse på værdien af den korteste sti mellem knuderne. Proceduren fortsætter, indtil grænsen er lig med den korteste sti, og dette er opnået efter $|V|-1$ iterationer, under forudsætning af, at grafen ikke indeholder negative kredse. Algoritmen kan desuden udvides, så den detekterer og rapporterer, hvis grafen indeholder negative kredse.

Bellman-Ford

```
Præ: G=(V, E) er en vægtet graf med vægtfkt. w, s er startknuden
Post: Alle kanter er mærket med afstanden fra s.

Bellman-Ford(G, s)
   Initialisér(G, s)
   for i ← 1 til |V|-1
      for alle (u, v) ∈ E
         Relaxér(u, v)
```

Algoritmen kan udvides med følgende linjer, således at den returnerer den boolske værdi FALSK, hvis grafen indeholder negative kredse:

```
   for alle (u, v) ∈ E
      hvis v.afstand > u.afstand + w(u, v)
         returnér FALSK
   returnér SAND
```

Algoritmens tidskompleksitet er $O(|V||E|)$, hvilket er dårligere end Dijkstra's.

Floyd-Warshall (All pair shortest path)

Herunder gives en algoritme til beregning af den korteste afstand mellem alle par af knuder i en graf.

Lad $V = \{v_1, v_2, ..., v_n\}$. Vi vil definere $d_{st}^{(k)}$, k>0, som vægten af den korteste sti mellem *s* og *t* blandt de stier mellem *s* og *t*, der kun indeholder kanter fra $\{v_1, v_2, ..., v_k\}$, men ikke $\{v_{k+1}, v_{k+2}, ..., v_n\}$. For $k = 0$ er $d_{st}^{(k)}$ vægten af kanten mellem *s* og *t*, hvis en sådan eksisterer, ellers ∞.

Floyd-Warshall

```
Præ: G=(V, E) er en vægtet graf med vægtfunktion w
     d_ij^(0) = w(v_i, v_j) - vægten af kanten mellem v_i og v_j: ellers ∞
Post: d_ij^(k) er vægten af korteste sti mellem v_i og v_j

Floyd-Warshall(G)
   for i ← 1 til |V|
      for j ← 1 til |V|
         for k ← 1 til |V|
            d_ij^(k) ← min{d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1)}
```

Der er negative kredse i grafen, hvis $d_{ii}^{(k)} < 0$. Algoritmens tidskompleksitet er ikke overraskende $O(|V|^3)$.

Minimum udspændende træ

Et ofte stillet spørgsmål i forbindelse med grafer er: ”Hvorledes kan man forbinde samtlige grafens knuder, med den mindste omkostning til følge?”.

For at svare på spørgsmålet, skal vi først have et par definitioner på banen:

Udspændende træ (definition)

Et træ, der indeholder alle knuder *V* fra en graf *G*=(*V, E*), og hvis kanter *E*’ tilhører *E* kaldes et *udspændende træ* for *G*.

Minimum udspændende træ

Som det måske antydes i definitionen, så kan der findes flere sådanne træer i en given graf. Det træ blandt disse udspændende træer, der har den mindste længde (vægt), kaldes for et *minimum udspændende træ*, *MST* – fra engelsk: *minimum spanning tree*. Dette træ er heller ikke entydigt; der kan godt være flere udspændende træer med minimal længde.

Et MST er netop det, vi søgte efter, da vi stillede vores oprindelige spørgsmål. Vi skal herunder se på to af de mest prominente algoritmer til at finde et MST i en graf. Inden da vil vi definere et MST mere formelt:

Definition på et MST

- Givet en sammenhængende og ikke-orienteret graf *G*=(*V, E*)
- En acyklisk delgraf $T \subseteq E$, som forbinder alle knuder i grafen G, med en samlet vægt på $w(T) = \sum_{(u,v)\in T} w(u,v)$, som er minimal, kaldes et MST.

Bemærk, at et MST indeholder $|V|-1$ kanter.

En generel metode til at finde et MST kan gives ved følgende pseudo-kode, der så at sige ”gror” et træ:

```
MST()
    A ← ∅ // A er en delmængde af et udspændende træ
    Sålænge A ikke er et udspændende træ
        find (u, v) ∈ E, som er en ”sikker” kant til A
            A ← A ∪ {(u, v)}
    returnér A
```

Mængden A er på ethvert tidspunkt i løbet af algoritmens afvikling en delmængde af et udspændende træ. Et sådant udsagn, der er statisk kaldes for en invariant.

```
Invariant: A er en delmængde af et udspændende træ
```

Ovenstående pseudokode fordrer nok en forklaring på begrebet en ”sikker” kant. En sikker kant er en kant, med minimum vægt, der kan føjes til mængden A, således at A stadig udgør en delmængde af et udspændende træ. Se figur.

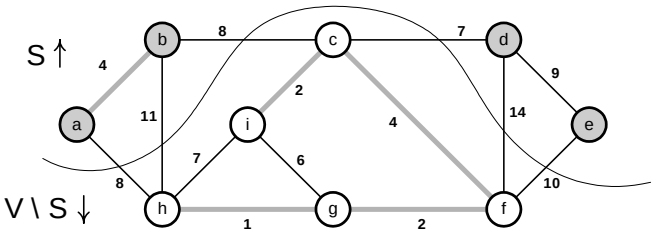


Fig: Mængden A er de tykke grå streger. En kant med minimum vægt, som krydser adskillelsen mellem S *og* S\V *kaldes en "sikker kant".*

En metode til at finde et MST skyldes Kruskal. Han anvender en grådig strategi, idet han hele tiden udvælger en kant med minimal vægt, og lader den indgå i løsningen, såfremt den ikke skaber en kreds.

Kruskal

```
Kruskal()
    A ← ∅ // A er en delmængde af et udspændende træ
    for alle knuder u ∈ V
        MakeDisjointSet(u)
    sortér grafens kanter efter ikke-faldende orden
    for alle kanter (u, v) ∈ E – efter ikke-faldende orden
        hvis find(u) ≠ find(v)
            A ← A ∪ {(u, v)}
            union(u, v)
    returnér A
```

Med henvisning til ovenstående pseudokode, så kan følgende siges om Kruskal:

- Invarianten A er en skov
- En sikker kant er en kant med mindst vægt, der forbinder to uafhængige dele af grafen

Kruskal vedligeholder således en skov at træer. Til at starte med er der $|V|$ træer, der hver især består af én knude. Når en kant tilføjes, så samles to træer til ét, og algoritmen terminerer, når der kun er ét træ i skoven.

Prim

```
Prim()
    Q ← V // Placér alle knuder i en prioritetskø
    for alle knuder u ∈ Q
        u.nøgle ← ∅
    rod.nøgle ← 0
    rod.forgænger ← null
    sålænge Q ≠ ∅
        u ← Q.deleteMin()
        for alle knuder v ∈ Adj(u)
            hvis v ∈ Q AND w(u, v) < v.nøgle
                v.nøgle ← w(u, v)
                v.forgænger ← u
```

Med henvisning til ovenstående pseudokode, så kan følgende siges om Prim:

- Invarianten A er et træ
- En sikker kant er en kant med mindst vægt, der forbinder træet med en knude, som ikke i forvejen er i træet

Prim gror således et træ – én kant ad gangen med udgangspunkt i en vilkårlig knude. For hvert skridt vælges en kant med minimal vægt fra en knude i træet til en knude, der ikke indgår i træet. Algoritmen terminerer, når alle knuder i grafen er repræsenteret i træet.