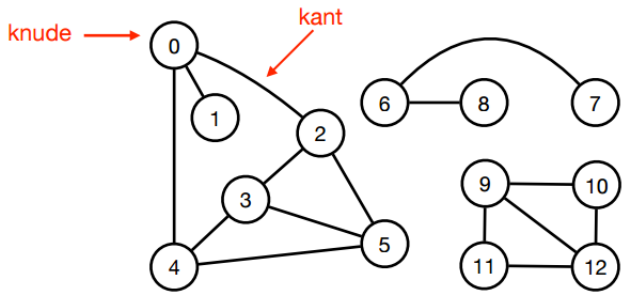
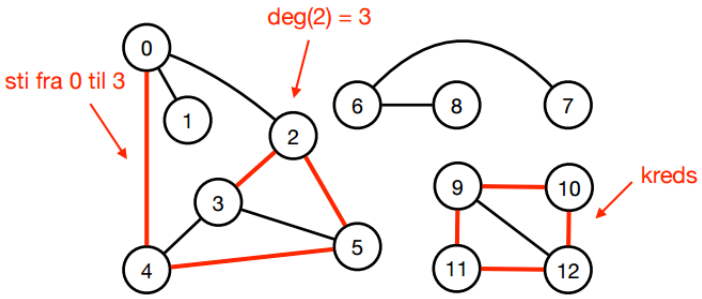
DMA 2021
– Noter om grafer*–

Grafer

En **uorienteret** graf G er en mængde V af **knuder** og en mængde E af **uordnede** par $\{u, v\}$ hvor $u, v \in V$. Ofte skriver vi $G = (V, E)$ for at angive en graf. Denne matematiske definition søger blot at præcisere den intuition I allerede har om grafer; at de er en mængde af punkter forbundet parvist af kanter.

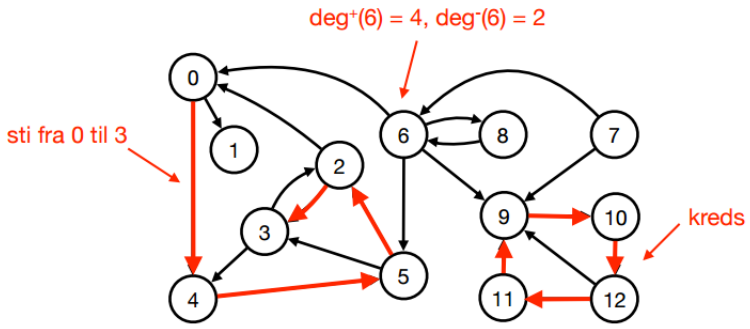


Vi skriver $|V|$ for antallet af knuder i grafen, og $|E|$ for antallet af kanter. Der et par vigtige begreber vi bruger om grafer. En **sti (eng: path)** er en sekvens af knuder forbundet af kanter i grafen. En **kreds (eng: cycle)** er en sti, som starter og slutter i samme knude. Hvis $\{u, v\} \in E$ er en kant i grafen, siger vi at u og v er **naboer**. For en knude $v \in V$ kaldes antallet af naboer for **graden (eng: degree)** af knuden og skrives $\deg(v)$. To knuder $u, v \in V$ er **forbundne (eng: connected)** hvis der er en sti fra u til v.



En **orienteret** graf G er en mængde af knuder V og en mængde E af **ordnede par** (u, v) af kanter. Så E er en binær relation på V, $E \subseteq V^2$. Der er en kant fra u til v, hvis $(u, v) \in E$, omvendt fra v til u hvis $(v, u) \in E$. Man kan også betragte en uorienteret graf, som en orienteret graf hvor relationen E er symmetrisk.

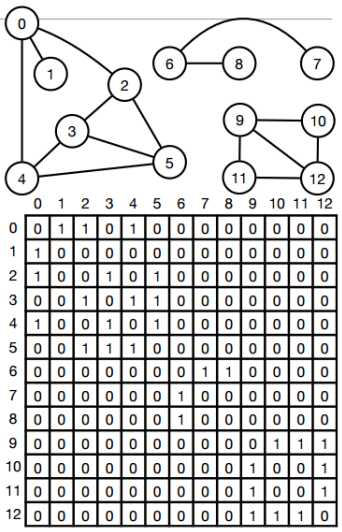
*Disse noter er stærkt inspireret af noter af Philip Bille og Inge Li Gørtz til kurset Algoritmer og Datastrukturer, på DTU, http://www2.compute.dtu.dk/courses/02105+02326/2015/#generelinfo



I det nedenstående vil vi se på uorienterede grafer, men uden større ændringer kan det anvendes på orienterede grafer.

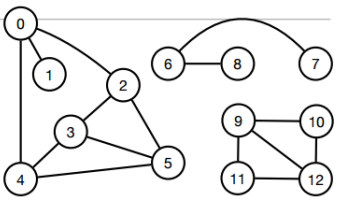
Repræsentation For at kunne skrive algoritmer på grafer, skal vi have en konkret måde at repræsentere dem på. Der er et par forskellige strategier. Vores datastruktur skal understøtte følgende operationer.

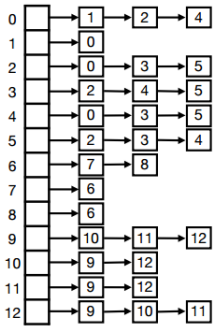
1. Adjacent(u,v): Afgør om u og v er naboer.
2. Neighbours(u): Returnér alle naboer til u.
3. Insert(u,v): Indsæt kanten $\{u, v\}$, hvis den ikke allerede findes.



Nabomatrix I har allerede set at vi kan repræsentere grafer med en nabomatrix[1] A (eng: **adjacency matrix**), hvor indgangen $A[i][j]$ er 1 hvis der er en kant $\{i, j\}$ og 0 ellers. Denne repræsentation vil bruge $O(|V|^2)$ plads, og Adjacent(u,v) samt Insert(u,v) kan løses i $O(1)$. Derimod er Neighbours(u) $O(|V|)$.

Da en knude kan have $|V|-1$ kanter, er dette asymptotisk optimalt. I praksis vil grafer dog ofte være **tynde** (eng: **sparse**), hvilket betyder at graden af knuderne er langt fra $|V| - 1$ (der er mange 0'er i matricen). Derfor kan det være spild af plads at gemme alle disse 0'er.





Naboliste En anden repræsentation som undgår dette problem bruger en såkaldt naboliste. Hvis G er en graf med n knuder så $|V| = n$ bruger vi en tabel $A[0..n-1]$ med en indgang for hver knude. Indgangen $A[i]$ indeholder så en liste over alle naboer til i.

Nu gemmer vi altså kun information, når der rent faktisk er en kant mellem to knuder. Pladsforbruget bliver dermed n for vores tabel A, plus summen af længden af alle disse lister. Længden af listen gemt i $A[i]$ er præcis $\deg(i)$ (antallet af naboer). Vi kan finde summen af graderne af alle knuder i grafen,

$$\sum_{i \in V} \deg(i) = 2|E|$$

[1]Somme tider bruges "incidens" på dansk fejlagtigt i stedet for "nabo", men det betyder noget andet.

da en hver kant optræder på præcis 2 lister. Dermed bliver pladsforbruget $O(|V|+|E|)$, hvilket er en forbedring når $|E|$ er tilpas lille.

Adjacent(u,v) og Neighbours(u) kan nu løses med en løkke som gennemløber u's naboliste. De har dermed køretid $O(\deg(u))$.

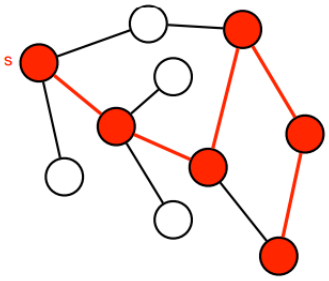
Traversering

Dybde først søgning En meget basal algoritme på grafer er dybde først søgning eller DFS. Målet er at besøge hver eneste knude i grafen, og eventuelt annotere hver knude med noget brugbar information, f.eks. hvornår den besøges ifht. andre knuder. Vi markerer knuderne som hhv. besøgt og ubesøgt for at holde styr på hvor vi har været i grafen.

DFS starter med en knude $s \in V$ og lader alle knuder være umarkede. For hver umarkeret nabo til s, kører vi DFS algoritmen rekursivt på denne nabo. Det betyder at vi udforsker grafen ud fra s, og når vi kommer til en blindgyde, går vi tilbage til sidst besøgte knude som stadig har uudforskede naboer.

```
DFS(s)
    time = 0
    DFS-VISIT(s)

DFS-VISIT(v)
    v.d = time++
    marker v
    for alle umarkerede naboer u
        DFS-VISIT(u)
        u.π = v
    v.f = time++
```

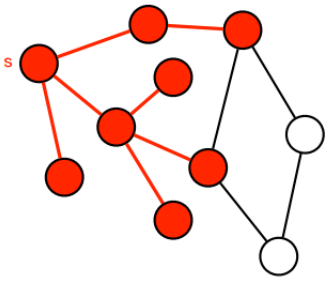


For at få interessant information om grafen, vil vi sætte to labels på hver knude. Værdien **v.d** (eng: discovery time) angiver hvornår vi første gang besøger og markerer en knude. Værdien **v.f** (eng: finish time) angiver hvornår det rekursive kald på en knude er færdigt, dvs. når vi har besøgt alle knudens børn og alle deres efterkommere. Værdien **u.**π er en peger til en knude v, hvor v er knuden vi gik igennem for at komme til u.

Vi besøger hver knude højst én gang, og for hver knude v looper vi igennem nabolisten, som har længde $\deg(v)$. Så køretiden bliver $O(|V|+\sum_{v\in V}\deg(v)) = O(|V|+|E|)$. Algoritmen vil kun besøge knuder som er forbundet til s, så det kræver en lille ændring for at besøge alle knuder i grafen hvis grafen ikke er sammenhængende.

Bredde først søgning Bredde først søgning er en anden måde at besøge alle knuder i en graf. Den eneste forskel fra DFS er rækkefølgen hvori vi besøger naboer til en knude. En god måde at få en fornemmelse for forskellen på BFS og DFS er at bruge visualiseringer af algoritmerne. Se f.eks. her, eller find selv andre på nettet.

```
BFS(s)
    marker s
    s.d = 0
    K.ENQUEUE(s)
    gentag indtil K er tom
        v = K.DEQUEUE()
        for alle umarkede naboer u
            marker u
            u.d = v.d + 1
            u.π = v
            K.ENQUEUE(u)
```



Intuitionen for BFS er at udforske grafen ud fra $s \in V$ ved først at besøge alle knuder i afstand 1 fra s, så alle knuder i afstand 2 osv. Ovenstående algoritme besøger også hver

knude højst én gang, og for hvert besøg loopes nabolisten. Så køretiden er identisk med DFS: $O(|V| + |E|)$.

Korteste veje

En **vægtet orienteret graf** G er en orienteret graf $G = (V, E)$ hvor hver kant har en vægt. Dette er udtrykt ved en vægtfunktion $w : E \to \mathbb{R}$. Konkret kan $w(e)$ fortælle hvor lang en kant $e \in E$ er hvis grafen modellerer f.eks. et vejnet. Lad P være en sti/vej i grafen, en ordnet sekvens af knuder $P = (v_0, \cdots, v_n)$ som er forbundet af kanter i grafen. Det vil sige der gælder $e_i = (v_{i-1}, v_i) \in E$ for alle $1 \leq i \leq n$. Vi vil ofte brugte notationen $P : u \rightsquigarrow v$ som en forkortelse for: P er en vej fra u til v. Eftersom vi har en vægt funktion er det ganske naturligt at definere vægten af en sti P, til at være summen af vægtene af alle kanterne.

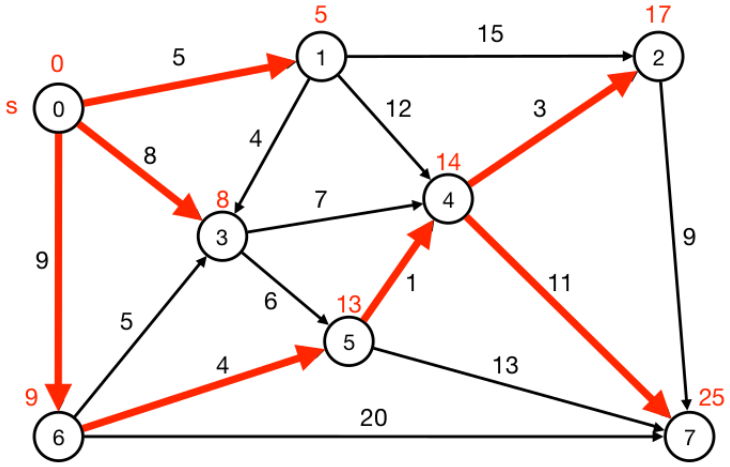
$$w(P) = w(v_0, v_1) + w(v_1, v_2) + \cdots + w(v_{n-1}, v_n) = \sum_{i=1}^{n} w(v_{i-1}, v_i).$$

Givet to knuder $u, v \in V$ definerer vi **vægten af en korteste vej** fra u til v som,

$$\delta(u, v) = \begin{cases} \min\{w(P) \mid P : u \rightsquigarrow v\} & \text{, hvis der findes en sti fra } u \text{ til } v \\ \infty & \text{, hvis der ikke findes en sti fra } u \text{ til } v. \end{cases}$$

Vi siger at en vej P er **en korteste vej** fra u til v, hvis $w(P) = \delta(u, v)$. Bemærk der kan være flere forskellige korteste veje, og hvis der ikke findes en sti fra u til v er der ingen korteste vej.

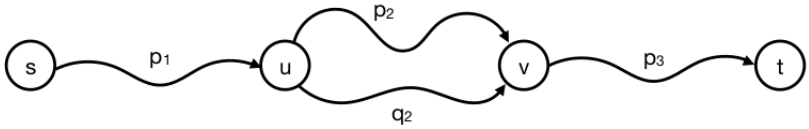
Givet en bestemt startknude $s \in V$ er vi interesserede i at finde korteste veje fra s til alle andre knuder i G. Vi vil finde både vægten af en given korteste vej, og en sti fra s som har denne denne mindste vægt. Disse stier kan vi repræsentere som et træ med rod i s.



Hvis vi antager at grafen G er sammenhængende ved vi at der altid er mindst én vej fra s til alle andre knuder i G. Derfor findes der altid en korteste vej fra s til u, for alle $u \in G$. Denne antagelse er ikke nødvendig, men gør situationen lidt simplere. En vigtig egenskab er følgende.

Lemma 1.1. *Enhver delvej af en korteste vej, er en korteste vej.*

Proof. Lad P være en korteste vej fra s til t, som indeholder en delvej P_2 fra u til v. Kald første stykke af P, fra s til u, for P_1 og sidste stykke, fra v til t, for P_3. Betragt situationen på billedet.

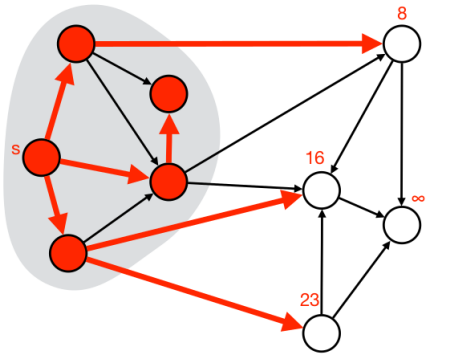


Vi vil vise at P_2 er en korteste vej fra u til v, dvs. $w(P_2) \leq w(Q_2)$ for en hvilken som helst anden vej Q_2 fra u til v.

Der gælder $w(P) = w(P_1) + w(P_2) + w(P_3)$. Antag for modstrid at Q_2 er en en anden vej fra u til v, som er kortere end P_2, så $w(Q_2) < w(P_2)$. Da vil P_1, Q_2 og P_3 tilsammen udgøre en vej fra s til t med samlet vægt $w(P_1) + w(Q_2) + w(P_3) < w(P_1) + w(P_2) + w(P_3) = w(P)$. Dette er i modstrid med at P var en korteste vej, så Q_2 kan ikke være kortere end P_2, og derfor er P_2 en korteste vej fra u til v. □

Djikstras Algoritme Vi vil nu løse et lidt simplere problem end det vi stillede tidligere. Givet en orienteret vægtet graf G, med **ikke negative vægte**, dvs. $w(e) \geq 0$ for alle kanter $e \in E$, og en knude $s \in V$, beregn korteste vej fra s til alle knuder i G.

```
RELAX(u,v)
    if (v.d > u.d + w(u,v))
        v.d = u.d + w(u,v)
```



Bemærk at Djikstra kun virker under antagelse af at alle vægte er ikke negative. For hver knude $v \in G$ vil vi vedligeholde et afstandsestimat $v.d$, som er længden af den korteste vej fra s til v vi har fundet indtil videre. Så $v.d$ er en øvre grænse for $\delta(s, v)$.

Algoritmen vil fungere ved at opdatere disse estimater $v.d$ på en bestemt måde indtil at $v.d = \delta(s, v)$. Opdateringen vil foregå gennem en procedure Relax(u, v), der givet en kant (u, v) forsøger at opdatere estimatet $v.d$ ved at tage en vej gennem u.

Selve træet som angiver de korteste veje vi finder, repræsenterer vi som tidligere ved en peger $v.\pi$ for hver knude v. Vi har $v.\pi = u$, præcis når den korteste vej fra s til v vi finder bruger en kant $(u, v) \in E$. Husk at notationen $v \in V \setminus T$, betyder at v er en knude i V, som *ikke* er i T. Djikstras algoritme fungerer da således:

1. Sæt $s.d = 0$ for $v.d = \infty$ for alle $v \in V \setminus \{s\}$.
2. Opbyg et træ T med rod s.
3. Find den knude $u \in V \setminus T$ med *mindste* afstandstandsestimat $u.d$, og tilføj den til T. Hvis $T = V$ er vi færdige.
4. Kald Relax(u, v) på alle kanter, der udgår fra u, og hop til skridt 3.

Bemærk at vi kun tilføjer en knude til T én gang, og til slut indeholder T alle knuderne i grafen, altså $T = V$. Følgende lemma lyder måske simpelt, men er ikke desto mindre vigtigt.

Lemma 1.2. *Hvis afstandsestimatet $v.d$ kun opdateres af kald til Relax gælder $v.d \geq \delta(s, v)$ for alle $v \in V$.*

Proof. Vi benytter induktion over antallet af kald til Relax. Inden første kald har vi $v.d = \infty \geq \delta(s, v)$ for alle $v \in V$ (husk definitionen af $\delta(s, v)$). Antag nu at $v.d \geq \delta(s, v)$ gælder efter $n - 1$ kald til Relax og lad Relax(u,v) være det n'te kald til Relax.

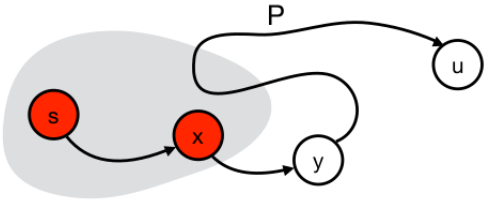
$$\begin{aligned} v.d = u.d + w(u, v) &\geq \delta(s, u) + w(u, v) && (u.d \geq \delta(s, u) \text{ fra induktions antagelsen}) \\ &\geq \delta(s, v) && (\text{Definition af } \delta(s, v)) \end{aligned}$$

□

Vi er selvfølgelig interesseret i at Djikstras algoritme rent faktisk virker. Det næste sætning viser netop dette når Djikstras bliver brug på en graf med ikke negative vægte.

Theorem 1.3. *Djikstras algoritme beregner $v.d = \delta(s, v)$ for all $v \in V$.*

Proof. Vi benytter induktion over antallet af knuder tilføjet til T. Den første knude vi tilføjer er s, og da har vi $s.d = 0 = \delta(s, s)$, så induktionsskridtet er gyldigt. Induktionsantagelse: Antag nu at $v.d = \delta(s, v)$ for alle $v \in T$, som er tilføjet til T. Lad u være næste knude, der til tilføjes til T. Lad $P : s \rightsquigarrow u$ være en korteste vej. Eftersom s er i T og u endnu ikke er i T, findes en kant (x, y) på vejen P, hvor $x \in T$ og $y \in V \setminus T$.



Da y ikke er tilføjet til T, og fordi vi altid tilføjer den knude med mindste afstandsestimat, gælder $u.d \leq y.d$. Knuden x er tilføjet til T, så der gælder $x.d = \delta(s, x)$ per vores induktionsantagelse. Faktisk gælder $y.d = \delta(s, y)$: Når x tilføjes til T, kalder vi Relax(x, y), så der må gælde

$$\begin{aligned} y.d &\leq x.d + w(x, y) && \text{(Fordi vi kalder Relax}(x, y)) \\ &= \delta(s, x) + w(x, y) && (x.d = \delta(s, x)) \\ &= \delta(s, y) && \text{(Delveje af korteste veje, er korteste veje)} \end{aligned}$$

Da der altid gælder $\delta(s, y) \leq y.d$ har vi $y.d = \delta(s, y)$ når u til føjes. Derfor har vi $u.d \leq y.d = \delta(s, y)$. Men alle kanter har ikke negativ vægt, og y ligger på den korteste vej fra s til u, så $\delta(s, y) \leq \delta(s, u)$. Dermed gælder $\delta(s, u) = u.d$ når u tilføjes til T. □

Implementation Efter vi har forsikret os om at metoden virker, må vi finde ud af hvorledes vi bedst muligt implementerer Djikstras algoritme. Det centrale for at opnå en god køretid, er hvordan vi finder knuden med det mindste afstandsestimat. Hertil benytter vi en prioritetskø P, hvor nøglen (eng: key) i køen er afstandestimaterne $v.d$. For at finde knuden med mindste afstandsestimat, kalder vi Extract-min. Når vi kalder Relax(u, v) og opdaterer afstandsestimater må vi dermed også kalde Decrease-key$(v, v.d)$.

```
DIJKSTRA(G, s)
    for alle knuder v∈V
        v.d = ∞
        v.π = null
        INSERT(P,v)
    DECREASE-KEY(P,s,0)
    while (P ≠ ∅)
        u = EXTRACT-MIN(P)
        for alle v som u peger på
            RELAX(u,v)
```

```
RELAX(u,v)
    if (v.d > u.d + w(u,v))
        v.d = u.d + w(u,v)
        DECREASE-KEY(P,v,v.d)
        v.π = u
```

Bemærk at vi nu også opdaterer $v.\pi$ værdierne i Relax(u, v). Vi tilføjer kun knuder til køen P én gang, og fjerner én knude per loop af while løkken. Derfor kører while løkken $|V|$ gange. For hver knude kører vi dens naboliste (såfremt vi bruger naboliste repræsentation) igennem, og kalder Relax. Derfor har vi $|E|$ kald til Relax i alt. Dermed foretager vi $|V|$ indsættelser i P, $|V|$ Extract-min, og $|E|$ Decrease-key operationer.

Prioritetskø	INSERT	EXTRACT-MIN	DECREASE-KEY	Total
tabel	O(1)	O(n)	O(1)	$O(n^2)$
binær hob	O(log n)	O(log n)	O(log n)	O(m log n)
Fibonacci hob	O(1)†	O(log n)†	O(1)†	O(m + n log n)

† = amortiseret køretid

Køretiden afhængder derfor af hvorledes vi implementerer prioritetetskøen. Hvis vi bruger en min-hob, er køretiden af Insert $O(\log|V|)$, køretiden af Extract-min er $O(\log|V|)$ og køretiden af Decrease-key er $O(\log|V|)$. Derfor bliver køretiden $O(|E|\log|V|)$. Der findes andre måder at implementere en prioritetskø, som er mere effektive. En såkaldt Fibonacci-hob har amortiseretkøretid, $O(1), O(\log|V|)$ og $O(1)$ for hhv. Insert, Extract-min og Decrease-key. Med denne implementation bliver køretiden $O(|E| + |V|\log|V|)$.