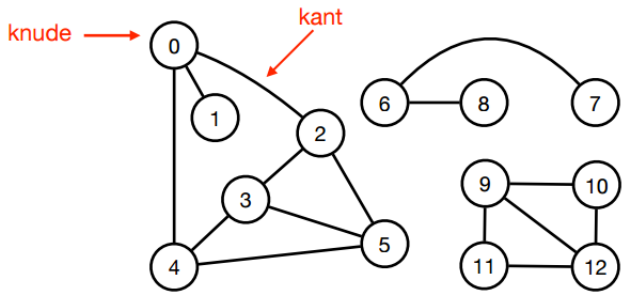
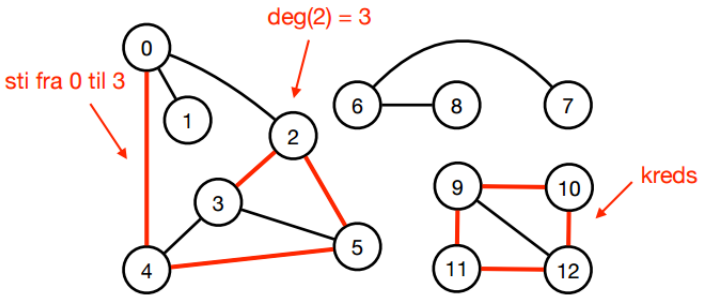
DMA 2016
- Noter* til uge 15 -

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.

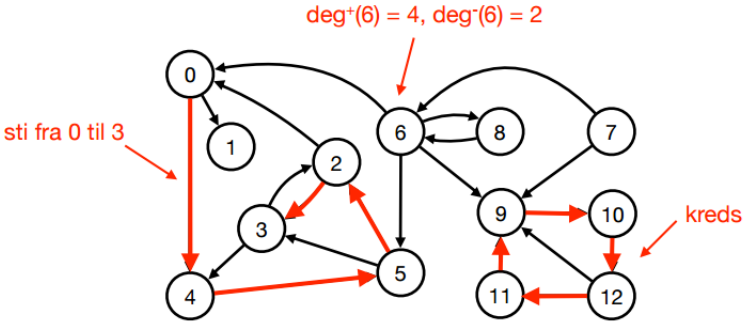


Vi skriver $|V|$ for antallet af knuder i grafen, og $|E|$ for antallet af kanter. Der et par vigtige begreber vi bruger om grafer. En **sti (eng: path)** er en sekvens af knuder forbundet af kanter i grafen. En **kreds (eng: cycle)** er en sti, som starter og slutter i samme knude. Hvis $\{u, v\} \in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



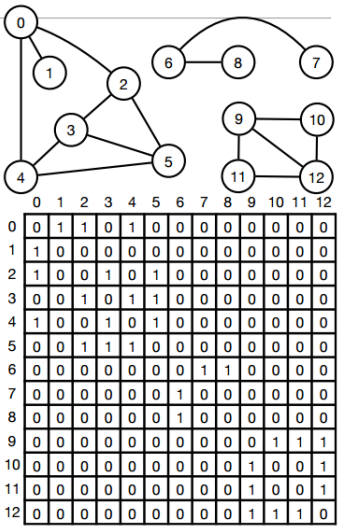
En **orienteret** graf G er en mængde af knuder V og en mængde E af **ordnede par** (u, v) af kanter. Så E er en binær relation på V, $E \subseteq V^2$. Der er en kant fra u til v, hvis $(u, v) \in E$, omvendt fra v til u hvis $(v, u) \in E$. Man kan også betragte en uorienteret graf, som en orienteret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

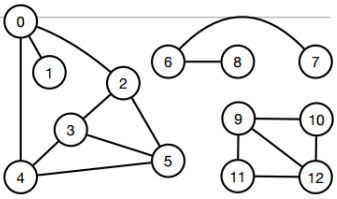
Repræsentation For at kunne skrive algoritmer på grafer, skal vi have en konkret måde at repræsentere dem på. Der er et par forskellige strategier. Vores datastruktur skal understøtte føglende operationer.

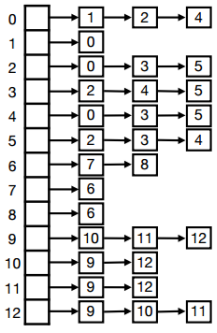


1. Adjacent(u,v): Afgør om u og v er naboer.
2. Neighbours(u): Returnér alle naboer til u.
3. Insert(u,v): Indsæt kanten $\{u, v\}$, hvis den ikke allerede findes.

Incidensmatrix I har allerede set at vi kan repræsentere grafer med en incidens matrix 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.





Incidensliste En anden repræsentation som undgår dette problem bruger en såkaldt incidensliste. 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,

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

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

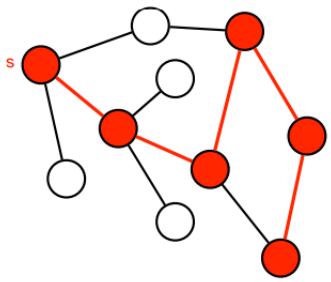
Adjacent(u,v) og Neighbours(u) kan nu løses med en løkke som gennemløber u's naboliste i adjacency listen. De har dermed køretid $O(\deg(u))$.

Dybde først søgning En meget basal algoritme på grafer er dybe 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 udfra 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++
```

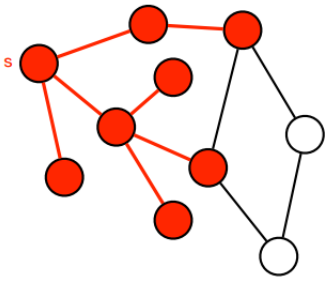


For at få interessant information om grafen, vil vi sætte to labels på hver knude. Værdien **v.d** (eng: discovery time) angiver hvornår vi første gang besøger og markerer en knude. Værdien **v.f** (eng: finish time) angiver hvornår det rekursive kald på en knude er færdigt, dvs. når vi har besøgt alle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

Vi besøger hver knude højst én gang, og for hver knude v looper vi igennem incidenslisten, 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 udfra $s \in V$ 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 incidenslisten. Så køretiden er identisk med DFS: $O(|V|+|E|)$.