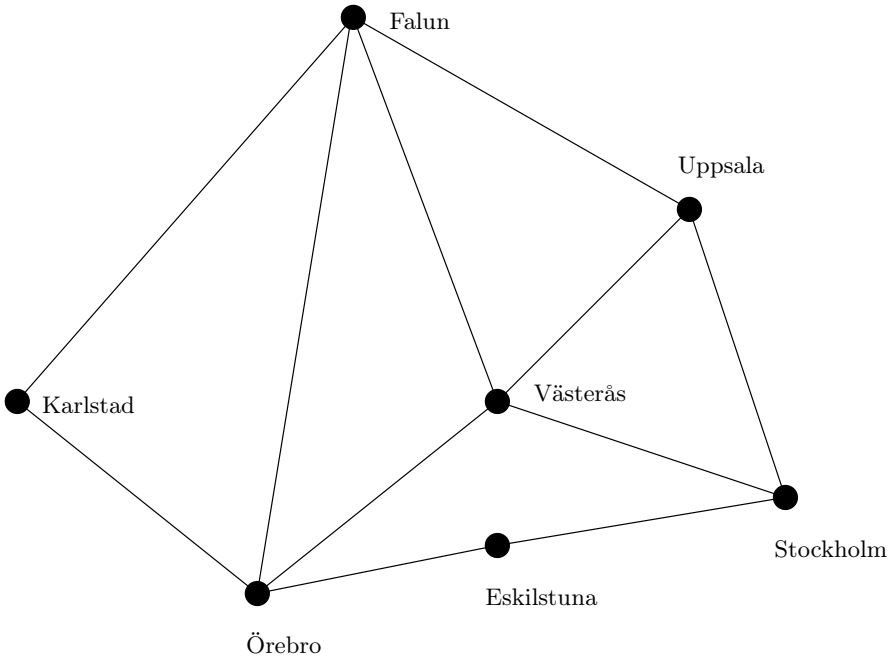
Föreläsning 2: Grafer

- Vad är en graf?
- Terminologi
- Representationer
- Genomgång av hörnen i en graf
- Kortaste väg-problemet

Exempel på graf



En graf G består av en mängd *hörn*, V, och en mängd *kanter*, E. Notation: $G = (V, E)$
I en riktad graf, skrivs ofta $G = \langle V, E \rangle$, är alla kanter riktade.

Exempel på grafproblem

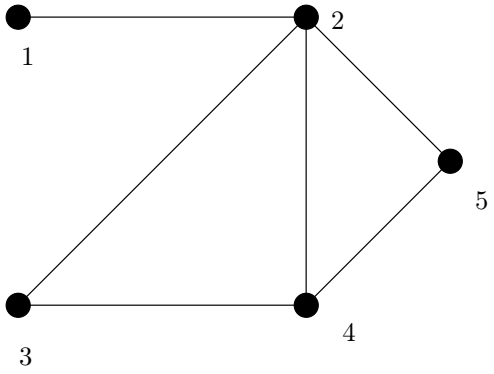
- Vilken är den kortaste vägen mellan Stockholm och Karlstad?
- Jag vill åka på en rundtur som börjar och slutar i Stockholm och passerar alla städerna. Vilken är den kortaste turen?
- Man vill bygga nya motorvägar längs en del av sträckorna så att varje par av städer har motorvägsförbindelse. Vilka vägsträckor ska väljas ut så att så lite motorväg som möjligt behöver byggas?

Alla dessa frågor rör *viktade grafer*, där varje kant har en (vanligen positiv) vikt. Här svarar vikterna mot avstånden mellan städerna.

Terminologi

En *stig* är en följd av hörn där konsekutiva hörn är förbundna av en kant. Inget hörn förekommer mer än en gång.
I en *sammanhängande* graf finns det en stig mellan varje par av hörn.
En *cykel* är en stig som börjar och slutar i samma hörn.
Ett *träd* är en sammanhängande graf utan cykler.
I en *multigraf* tillåts flera kanter mellan samma par av hörn.

Olika representationer av grafer



Grannmatris:

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

Grannlista:

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

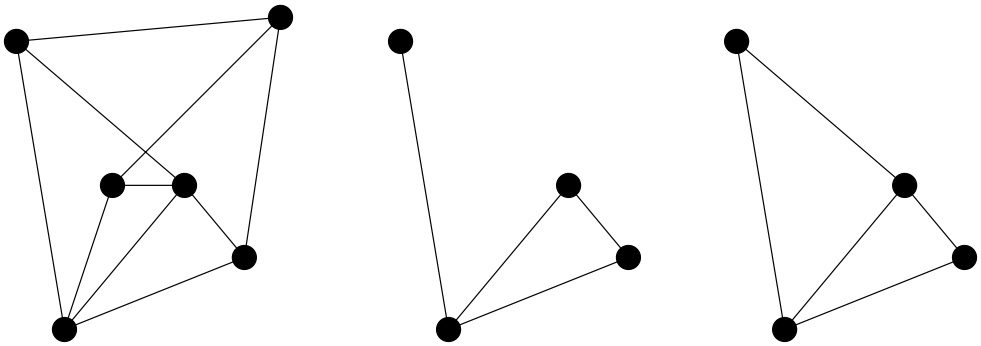
Representationen *kantmatris* förekommer också: Det är en $|V| \times |E|$-matris där element (i, j) är 1 om $v_i \in e_j$, 0 annars.

Täta och glesa grafer

En *tät* graf har $\Theta(|V|^2)$ kanter och representeras med fördel med en grannmatris.
En *gles* graf har $\Theta(|V|)$ kanter och representeras lämpligen med någon form av grannlistor.
Glesa grafer är vanliga i tillämpningar och prestanda för en algoritm på sådana kan i praktiken vara lika viktigt som tidskomplexiteten i värsta fallet.
Graden hos ett hörn är antalet kanter som innehåller hörnet medan gradtalet hos en graf är det högsta gradtalet för något hörn i grafen.

Delgrafer

En *delgraf* till grafen $G = (V, E)$ består av en delmängd E_1 av kanterna och en delmängd V_1 av hörnen (som måste innehålla alla hörn som ingår i någon av kanterna i E_1).
En *inducerad delgraf* till G består av en delmängd av hörnen samt de kanter som förbinder två hörn i denna mängd.
Ex.



G H_1 H_2

H_1 är en delgraf och H_2 en inducerad delgraf till G.

Breddenförstsökning

BFS tar en graf och går igenom hörnen ett i taget. Algoritmen börjar i starthörnet s och tar sedan grannarna till s, grannarna till grannarna till s etc. En kö används för att hålla reda på ordningen i vilken hörnen besöks. Det oviktade avståndet från s beräknas samtidigt.

```
BFS(V, E, s)
(1)     d[u] ← ∞ för alla u ∈ V
(2)     d[s] ← 0
(3)     Q ← {s}
(4)     while Q ≠ ∅
(5)         u ← Dequeue(Q)
(6)         foreach granne v till u
(7)             if d[v] = ∞
(8)                 d[v] ← d[u] + 1
(9)                 p[v] = u
(10)                Enqueue(Q, v)
(11)        Bearbeta u här
```

Tidskomplexitet: $O(|V| + |E|)$

Djupetförstsökning

DFS går igenom de hörn som kan nås från starthörnet s. Detta görs genom att gå så långt det går i grafen utan att återkomma till något hörn och sedan stega tillbaka tills något nytt hörn går att besöka.

```
DFS(V, E, s)
(1)     foreach u ∈ V
(2)         vis(u) ← 0
(3)         p[u] ← NULL
(4)     DFS − Visit(s)
```

```
DFS-Visit(u)
(1)     vis(u) ← 1
(2)     Bearbeta u här
(3)     foreach granne v till u
(4)         if vis(v) = 0
(5)             p[v] = u
(6)             DFS-Visit(v)
```

Tidskomplexitet: $O(|V| + |E|)$

Tillämpning av DFS

DFS kan användas för att avgöra om en graf är en DAG (Directed Acyclic Graph). Idén är att markera de ”halvfärdiga” hörn för vilka anropet till DFS-Visit ännu inte är avslutat.

```
DFS-DAG(V, E)
(1)     foreach u ∈ V
(2)         vis(u) ← 0
(3)     foreach u ∈ V
(4)         if vis(u) = 0
(5)             DFS-Visit(u)
(6)     return Grafen är en DAG
```

```
DFS-Visit(u)
(1)     vis(u) ← 0.5
(2)     foreach granne v till u
(3)         if vis(v) = 0.5
(4)             return Grafen innehåller en cykel
(5)         if vis(v) = 0
(6)             DFS-Visit(v)
(7)     vis(u) ← 1
```

Topologisk numrering

En topologisk numrering av en DAG är en numrering $topnum[u]$ av noderna så att om (u, v) är en riktad kant så är $topnum[u] < topnum[v]$.
En sådan numrering kan genomföras med en modifierad variant av DFS:

```
DFS-TOPNUM(V, E)
(1)     foreach u ∈ V
(2)         vis(u) ← 0
(3)     t ← |V|
(4)     foreach u ∈ V
(5)         if vis(u) = 0
(6)             DFS-Visit(u)
(7)     return Grafen är en DAG
```

```
DFS-Visit(u)
(1)     vis(u) ← 0.5
(2)     foreach granne v till u
(3)         if vis(v) = 0.5
(4)             return Grafen innehåller en cykel
(5)         if vis(v) = 0
(6)             DFS-Visit(v)
(7)     vis(u) ← 1
(8)     topnum[u] ← t
(9)     t ← t − 1
```

Ytterligare tillämpning: Bipartita grafer

För att kontrollera om en graf G är bipartit kör vi BFS. Sätt sedan alla noder u med $d[u]$ jämn till Blå och de med $d[u]$ udda till Röda. Kontrollera alla kanter. Om någon går mellan en Blå och en Röd nod är grafen **inte** bipartit. Annars är den bipartit.

Ytterligare en tillämpning: Stark sammanhängande grafer

För att avgöra om en riktad graf G är starkt sammanhängande väljer vi en startnod s och gör en DFS-sökning från s. Om $T = V$ fortsätter vi på följande sätt: Låt G_{rev} vara G med kantriktningarna omkastade. Gör DFS-sökning på G_{rev} med s som startnod. Om vi också nu får $T = V$ så är grafen starkt sammanhängande.

Kortaste väg-problemet

Givet två hörn s och t i en riktad graf $G = (V, E)$, hur hittar man den kortaste vägen mellan s och t?
Kanten e har vikten $l(e)$ som svarar mot längden hos vägsträckan. Vi söker alltså den stig i G från s till t som minimerar summan av de ingående kanternas vikter.

Vi tillåter negativa kantvikter. Men det kan uppstå svårigheter. En cykel i grafen vars kantsumma är negativ kallas för en **negativ cykel**. Negativa cykler ställer till problem eftersom det inte är klart vad man skall mena med kortaste vägar då negativa cykler finns.

En väg mellan två noder får innehålla en nod flera gånger. En stig mellan två noder får inte innehålla någon nod mer än en gång. Vad är skillnaden mellan kortaste vägar och kortaste stigar?

Varje kortaste väg i en graf G är en stig $\Leftrightarrow$ Det finns ingen negativ cykel i G.

Om negativa cykler finns så finns det inte kortaste vägar mellan alla noder (eftersom man kan hitta vägar med vikt mindre än $-N$ för varje positivt N). Däremot finns det alltid kortaste stigar.

De effektiva algoritmer som finns för att hitta kortaste vägar klarar inte av att hitta kortaste stigar i allmänhet. Men om grafen saknar negativa cykler är varje kortaste väg en stig och problemet uppstår inte.

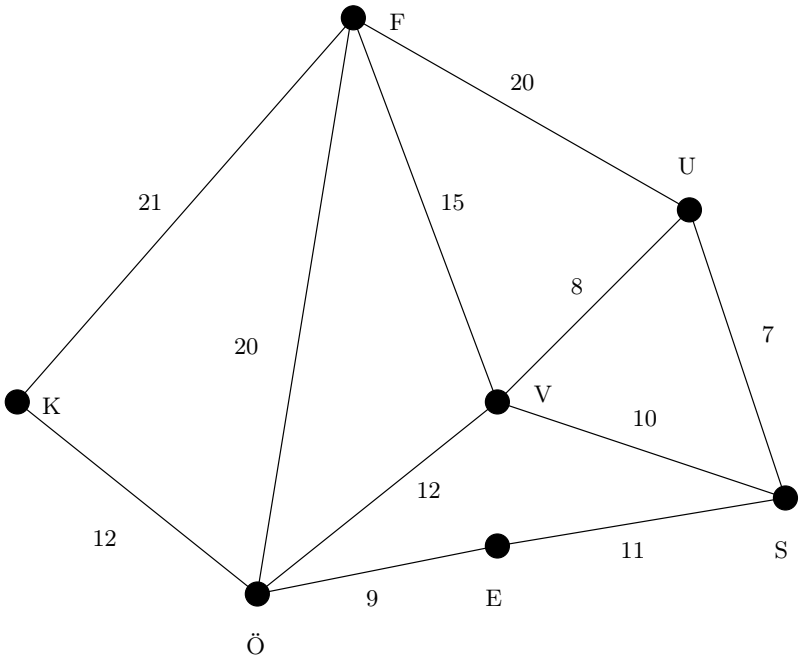
Det enklaste är att anta att alla kantvikter är positiva. Då kan Dijkstras algoritm användas.

Dijkstras algoritm

DIJKSTRA($G = \langle V, E \rangle$,$s, t \in V$, $l : E \to \boldsymbol{R}$)
(1) **foreach** $u \in V$
(2) Märk u med $d[u] = \infty$
(3) **foreach** u så att $(s, u) \in E$
(4) Märk u med $d[u] = w(s, u)$
(5) Märk s med $d[s] = 0$
(6) $S \leftarrow \{s\}$
(7) **while** $t \notin S$
(8) Välj $u \notin S$ med $d[u]$ minimal.
(9) $S \leftarrow S \cup \{u\}$
(10) **foreach** $v \notin S$ så att $(u, v) \in E$
(11) **if** $d[v] > d[u] + w(u, v)$
(12) $d[v] \leftarrow d[u] + w(u, v)$
(13) $p[v] \leftarrow u$
(14) **return** Märkningen hos t

G antas representerad i form av grannlistor.
När algoritmen är klar markerar $d[v]$ kortaste avståndet från s till v. Kortaste vägen fås baklänges som $p[v], p[p[v]], ...$

Dijkstra forts.: Exempel och analys



Tidskomplexitet:
Den inledande märkningen tar $O(|V|)$ tid. Att hitta hörnet som S ska utvidgas med tar $O(|V|)$ tid, och detta utförs högst $|V|$ gånger. För varje kant i grafen kommer högst en uppdatering ske.
Totalt: $O(|V| + |V|^2 + |E|) = O(|V|^2)$
(eftersom $|E| \in O(|V|^2)$)

Korrekthetsbevis

Vi antar för enkelhets skull att alla noder kan nås från s. Låt $\delta[s, v]$ betyda kortaste vägen från s till v och $\delta_S[s, v]$ betyda kortaste vägen som går genom noder i S, förutom v.
Vi använder följande invariant för den yttersta slingan:

Alla noder u i S är märkta med $d[u] = \delta[s, u]$. Dessutom är alla $v \notin S$ med $d[v] < \infty$ märkta med $d[v] = \delta_S[s, v]$. Noderna märkta med $d[v] = \infty$ går inte att nå med en S-stig.

Invarianten är sann innan slingan börjar köras. Antag att den är sann i ett steg i algoritmen. I nästa steg läggs u till. Vi måste visa att $d[u] = \delta[s, u]$. Antag att $d[u] \neq \delta[s, u]$. I den korstaste vägen från s till u finns en första nod x som ligger utanför S. Då gäller $\delta[s, u] = \delta[s, x] + \delta[x, u]$. Men $\delta[s, x] = \delta_S[s, x] = d[x]$ enligt antagandet. Då gäller $\delta[s, u] = d[x] + \delta[x, u] \geq d[x] \geq d[u]$ **eftersom negativa vägar inte finns**.
Eftersom $d[u] < \infty$ måste det finnas $y \in S$ så att $d[u] = d[y] + w(y, u)$. Eftersom $d[y] = \delta_S[s, y]$ så är $d[y] + w(y, u)$ längden på en möjlig väg från s till u. Därför är $\delta[s, u] \leq d[u]$. Det ger $d[u] = \delta[s, u]$.

Vi lägger nu till u till S. Kalla den nya mängden S'. Vi märker om noderna utanför S'. Vi måste visa att $d[v] = \delta_{S'}[s, v]$ för de ommärkta noderna. Vi antar att $v \notin S'$ är granne till u och att v har märkningen $d[v]$ före en eventuell ommärkning. Om $d[v] = \delta_{S'}[s, v] = \delta_S[s, v]$ så är $\delta_{S'}[s, v] \leq \delta[s, u] + w(u, v) = d[u] + w(u, v)$. Noden kommer inte att märkas om utan behåller märkningen $d[v] = \delta_S[s, v] = \delta_{S'}[s, v]$.

Om $\delta_{S'}[s, v] < \delta_S[s, v]$ så måste kortaste S'-vägen gå över u. Låt z vara noden som kommer efter u på en sådan väg. Om $z \neq v$ måste $z \in S$ och $\delta_{S'}[s, v] = \delta_S[s, u] + w(u, z) + \delta_S[z, v] \geq \delta_S[s, z] + \delta_S[z, y] \geq \delta_S[s, v]$. Men detta är omöjligt. Alltså måste $z = v$ och $\delta_{S'}[s, v] = \delta_S[s, u] + w(u, v) = d[u] + w(u, v)$. Noden v kommer då att märkas om så att den får $d[v] = d[u] + w(u, v) = \delta_{S'}[s, v]$.

När man går ur slingan har man beräknat korrekt kortaste avstånd $d[t]$ från s till t.

Negativa kantvikter tillåtna

Om en graf har negativa kantvikter men **inga negativa cykler** så finns kortaste vägar. Men de kan inte hittas med Dijkstras algoritm. Mer avancerad algoritm krävs.

Om en graf har negativa cykler finns följande val.
1. Acceptera att kortaste avstånd inte kan beräknas och förkasta grafen.
2. Leta efter kortaste stigar istället för vägar. Ingen effektiv algoritm som gör detta är känd.
3. Leta efter kortaste vägar med begränsning att de får innehålla högst k kanter. Detta går att göra relativt enkelt. (Men är det det man är ute efter?)

Bellman-Fords algoritm

Följande algoritm löser problemet enligt fall 1 ovan. Vi antar att varje nod kan nås från s. Målet är då att hitta kortaste vägar från s till varje annan nod.

```
BELLMAN-FORD(G = ⟨V, E⟩, s ∈ V, w : E → R)
(1)   foreach u ∈ V
(2)       d[u] ← ∞
(3)   d[s] ← 0
(4)   for i = 1 to |V| − 1
(5)       foreach (u, v) ∈ E
(6)           if d[v] > d[u] + w(u, v)
(7)               d[v] ← d[u] + w(u, v)
(8)               p[v] ← u
(9)   foreach (u, v) ∈ E
(10)      if d[v] > d[u] + w(u, v)
(11)          return Negativ cykel!
(12)  return d, p
```

Om en negativ cykel finns talar algoritmen om det. Annars svarar den med d och p.

Komplexitet

Inledande märkningen tar $O(|V|)$ steg. Yttre slingan tar $O(|V|)$ steg. Inre slingan tar $O(|E|)$ steg. Det ger tillsammans $O(|V||E|)$ steg. Den sista kontrollen tar $O(|E|)$ steg. Totalt får vi $O(|V||E|) = O(|V|^3)$. Det är sämre än Dijkstras algoritm.

Korrekthetsanalys

Vi antar först att grafen inte har negativa cykler. Då existerar kortaste vägar. Vi använder följande invariant för yttre slingan:

Efter iteration k gäller för alla noder u med $d[u] < \infty$ att $d[u] \leq \delta_k[u]$ där $\delta_k[u]$ = längden på den kortaste väg som har högst k kanter. Pekarna $p[u]$ som har tilldelats ett värde genererar ett träd som är rotat i s och om $p[v] = u$ så gäller $d[v] \geq d[u] + w(u, v)$.

Påståendet är sant för $k = 0$. Vi antar att det är sant för k och visar att det då är sant för $k + 1$ också. Om u kan nås med en väg som innehåller högst k kanter finns det en kortaste sådan väg. Kalla längden på den för $\delta_k[u]$. Antag att noden före u på en sådan väg är x. Då måste $\delta_k[u] = \delta_{k-1}[x] + w(x, u)$ och enligt antagandet är då $\delta_k[u] \geq d[x] + w(x, u)$. Om $d[u] > d[x] + w(x, u)$ kommer $d[u]$ att märkas om till $d[x] + w(x, u)$. I senare steg i samma iteration kan värdet på $d[u]$ minska. I vilket falls som helst kommer $d[u] \leq \delta_k[u]$ när iterationen är klar.

För en kant (u, v) med $p[v] = u$ gäller $d[v] = d[u] + w(u, v)$ precis efter märkningen. Om inte v märks om (och $p[v]$ ändras) kan ändå $d[u]$ sänkas. Men $d[v] \geq d[u] + w(u, v)$ kommer att fortsätta gälla.

Antag att algoritmen i något steg skulle införa en pekare $p[y] = x$ som genererar en pekarcykel. Då skulle det **före** införandet av pekaren finnas en cykel $(x, y), (y, z_1), ..., (z_r, x)$ där $d[y] > d[x] + w(x, y)$ (annars ingen ny pekare) och med $d[z_1] \geq d[y] + w(y, z_1)$ o.s.v. för de övriga kanterna. Beräkna summan $(d[y] - d[x]) + (d[z_1] - d[y]) + ... + (d[x] - d[z_r])$. Summan är 0 och samtidigt $> w(x, y) + w(y, z_1) + ... + w(z_r, x)$. Cykeln är då negativ vilket är omöjligt. Pekarna genererar inga cykler. Man kan då lätt se att de bildar ett träd som är rotat i s.

Antag nu att yttersta slingan i första delen av algoritmen är klar. Då är $d[u] \leq \delta_{n-1}[u]$ för alla u. Men eftersom det finns en väg $u, p[u], p[p[u]], ...$ med $d[u] = d[p[u]] + w(p[u], u)$ o.s.v. som leder till s så ser vi att det finns en väg med värde $d[u]$ från s till u. Då måste $d[u] = \delta_{n-1}[u]$. Eftersom varje kortaste väg innehåller som mest $n - 1$ kanter får vi $d[u]$ = längden på kortaste vägen från s till u.

Återstår att se om sista steget i algoritmen godkänner grafen. Men $d[v] > d[u] + w(u, v)$ är omöjligt eftersom d värdena nu anger korrekta avstånd. Så grafen måste bli godkänd.

Antag nu att grafen innehåller en negativ cykel med noderna $u_1, u_2, ..., u_r$. Första steget i algoritmen kommer att generera d- värden (som inte kommer att vara korrekta avstånd). Om $d[u_{i+1}] \leq d[u_i] + w(u_i, u_{i+1})$ för alla i skulle summan $(d[u_2] - d[u_1]) + (d[u_3] - d[u_2]) + ... + (d[u_1] - d[u_r])$ vara både 0 och $w(u_1, u_2) + ... + w(u_r, u_1)$. Eftersom cykeln är negativ är det omöjligt och det måste finnas i med $d[u_{i+1}] > d[u_i] + w(u_i, u_{i+1})$ Den olikheten kommer sista delen av algoritmen att upptäcka.