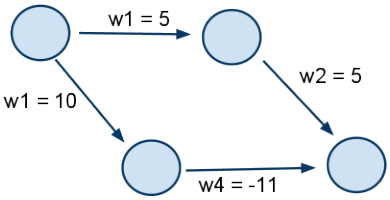
Föreläsningsanteckningar F6

Martin Andersson & Patrik Falkman

Kortaste vägen mellan en nod och alla andra noder

Detta problem innebär att givet en graf G = (E,V) hitta den kortaste vägen över E från en nod u i V till alla andra noder i V.

Dijkstras algoritm löser effektivt fallet när man har en riktad/oriktad graf med bara positiva kanter. Dock fungerar den inte när man har en eller flera kanter som är negativa, då algoritmen hela tiden utökar det uppspännande trädet via den nod vars avstånd från startpunkten är minst. Den antar på så sätt att om du kommer till en nod *a* från nod *b* kommer det aldrig vara kortare att gå till *a* via en annan nod *c* istället.



Bellman-Ford algoritmen kan dock hantera negativa kanter, och kan även användas för att hitta negativa cykler. Den fungerar på ett liknande sätt som Dijkstra, fast den går igenom alla kanter och minskar om möjligt avståndet mellan två noder *u* och *v* för varje kant i grafen:

Algorithm 1: Bellman-Ford
Input: En graf G(E, V) där E är en mängd kanter och V en mängd noder.
Output: minsta avståndet som en funktion p

```
for varje nod v i G(E, V)
     for varje kant (u, v) i E
         if (dist[u] + w(u, v) < dist[v] AND dist[u] is not infinite)
             dist[v] = dist[u] + w(u, v)
             p[v] = u
```

Tidskomplexitet: O(|V||E|)
Något som kan vara bra att fundera på när man implementerar Bellman-Ford är hur man ska hantera oändligheten: Ifall man strikt följer definitionen av oändligheten, dvs att ∞ - w(u,v) = ∞ kommer man inte kunna upptäcka negativa cykler!
Det man istället borde göra är att låta alla tal över en viss gräns E representera oändligheten. Då gäller det dock att välja E tillräckligt lågt så att man undviker overflow men samtidigt tillräckligt högt så att dist(v) aldrig råkar bli större än eller lika med E utan att dist(v) egentligen ska vara oändligt.

För att upptäcka negativa cykler kan man köra den inre for-loopen en extra gång efter att

algoritmen är klar; vi har redan kört den |V| gånger, vilket är det största antalet kanter som kan ligga på vägen mellan två noder. Ifall avståndet till en nod kan minskas med ännu en iteration betyder det att vi kan utöka en stig med en kant som redan finns i stigen, vars vikt är negativ så att summan av alla vikter över stigen minskas. Vi har då en negativ cykel!

Vi observerar att tidskomplexiteten till Bellman-Ford är större än tidskomplexiteten till Dijkstra, som är O(log|V||E|), och att det skulle vara snabbare att köra Dijkstras algoritm om alla negativa vikter kunde tas bort på ett snabbt sätt, förutsatt att grafen från början inte har några negativa cykler.

Kortaste vägen mellan alla par av noder

Detta problem innebär att givet en graf G = (E,V) hitta den kortaste vägen över E från varje nod u i V till alla andra noder i V.

Ett naivt sätt att räkna ut detta på är att köra Bellman-Ford algoritmen mellan alla par av noder vilket kommer ta $O(|V|^2|E|)$ tid.

Ifall man kunde köra Dijkstras algoritm skulle detta kunna göras på O(|V||E| log|V|) tid istället, något som kräver att vi först måste ta göra om alla negativa kanter till positiva! Johnsons algoritm gör detta på ett effektivt sätt:
Man definierar en ny funktion h(u) = w'(u,v) som definierar den nya vikten på kanten mellan nod u och v. Sedan skapar man en ny nod *s* och lägger till kanter med vikt 0 mellan s och alla andra noder. Man beräknar sedan avståndet d(s, u) med Bellman-Ford för alla *u* i *V* och observerar att h(w) <= h(u) + wt(u,w) på grund av triangelolikheten. Detta ger 0 <= w(u,w) + h(u) − h(w) = w'(u,w), alltså kommer alla nya vikter w' vara positiva. Tidskomplexiteten blir då O(|V||E| + |V||E| log|V|) = O(|V||E| log|V|).

Ett annat sätt att lösa problemet är med Floyd-Warshalls algoritm. Den definierar $d_{k,u,v}$ som det minsta avståndet från nod u till nod v, där bara k noder får användas utöver u och v. Definitionen görs rekursivt:

$d_{0,u,v} = w(u,v)$
$d_{k,u,v} = \min(d_{k-1,u,v}, d_{k-1,u,k} + d_{k-1,k,v})$ för alla k = 1, ..., n

det vill säga att $d_{k,u,v}$ bestäms genom att kolla om man kan få en kortare väg från u till v ifall man går genom noden k. Observera att w(u,v) har ett oändligt stort värde ifall det inte finns en kant från u till v. Samma gäller för $d_{|V|,u,v}$ ifall det inte finns någon stig från u till v.

Det ursprungliga sättet att implementera Floyd-Warshall på är att använda en 3-dimensionell vektor för att representera d. Detta kommer uppenbarligen ta $|V|^3$ minne, och det är därför av intresse att man hittar ett smartare sätt att implementera algoritmen. Vi observerar att det enda index av k som används är k och k-1 (för olika värden av k), och att vi kan ersätta den 3-dimensionella vektorn med två stycken matriser istället. Den ena matrisen sparar värden från det förra värdet på k medans den andra sparar det nuvarande. Ännu en observation är att algoritmen hela tiden kommer läsa det gamla värdet och sedan skriva det nya värdet, i den ordningen. Vi kan därför använda bara 1 matris, eftersom vi aldrig kommer behöva det gamla värdet igen efter att vi eventuellt har skrivit över det!

Algorithm 2: Floyd-Warshall

Input: En graf G(E, V) med kantfunktion w(u, v)
Output: Kortaste vägen mellan alla par av noder och vägen i sig

```
/*Initialize*/
foreach u ∈ V
    D[u, u] = 0
    foreach v ∈ V
        D[u, v] = ∞
        P[u, v] = null
foreach (u, v) ∈ E
    D[u, v] = w(u, v)
    P[u, v] = u

/*create d_{k,u,v}*/
for k = 1 to n
    foreach u ∈ V
        if D[u, k] < ∞
            foreach v ∈ V
                if D[u, k] + D[k, v] ≥ D[u, v]
                    D[u, v] = D[u, v]
                    P[u, v] = P[u, v]
                else
                    D[u, v] = D[u, k] + D[k, v]
                    D[u, v] = P[k, v]
return (D, P)
```

Tidskomplexiteten är $O(|V|^3)$.

När ska man använda viken algoritm?
Ifall vi har en gles graf kan det vara värt att använda Johnons algoritm istället för Floyd-Warshalls. Mer specifikt, ifall $|E| < |V|^2/\log|V|$.

Eulercykler och stigar

En Eulercykel är en cykel i en graf som passerar varje kant en gång. En eulerstig är som en Eulercykel, fast start- och slutnoden ej behöver vara den samma.
Observera skillnaden på en Eulercykel och en Hamiltoncykel: En Hamiltoncykel är en cykel i en graf där varje *nod* finns med en gång. Att hitta en sådan cykel är ett NP-fullständigt problem.

För att en Eulercykel ska finnas i en graf måste följande villkor uppfyllas:

- Alla kanter i grafen måste vara sammanhängande
- I en oriktad graf måste varje nod ha ett jämnt antal kanter
- I en riktad graf måste varje nod ha lika många inkommande kanter som utgående dito.

Följande villkor är tillräckliga för att det ska finnas en Eulerstig. Observera att det kan finnas en Eulerstig i en graf där det inte kan finnas en Eulercykel, i så fall gäller:

- Alla kanter i grafen måste vara sammanhängande

- I en oriktad graf får max två noder ha udda valens, resten av noderna måste ha jämn valens.
- I en riktad graf måste antal inkommande kanter vara lika med antal utgående kanter för alla noder, förutom max en nod där antal inkommande kanter är en mer än antal utgående kanter, och max en nod där antal utgående kanter är en mer än antal ingående dito.

Det går därför att på O(|E|) tid kolla om en graf innehåller en Eulercykel/stig.
Algoritmen för att hitta en Eulercykel/Eulerstig delas upp i två steg, en stigfinnare och en kontrollant:

- Stigfinnaren börjar på en godtycklig nod (om vi vill hitta en Eulercykel) eller på startnoden (ifall man vill hitta en Eulerstig). Sedan väljer man en godtycklig kant att traversera och markerar denna kanten som använd. Detta görs om ända tills det inte längre är möjligt att fortsätta. Enligt villkoren ovan befinner sig då algoritmen i samma nod som startnoden (för Eulercykler) eller i slutnoden (för Eulerstigar).
  Vi får då en serie kanter S = $\{e_1, e_2, ... , e_k\}$

- Kontrollanten traverserar över S. Ifall den upptäcker att någon kant e från någon nod v inte är markerad som använd startar den Stigfinnaren igen från nod v, över e, och "klistrar in" den returnerade stigen S' = $\{f_1, f_2, ..., f_m\}$ på rätt ställe i S, så att man får
  S = $\{e_1, e_2, ..., e_i, f_1, f_2, ..., f_m, e_{i+1}, ...e_k\}$.

Flöde i grafer

Att hitta det största möjliga flödet i en riktad och viktad graf går att göra med Ford-Fulkerssons algoritm ifall grafen är bipartit. Ofta använder man Ford-Fulkerssons algoritm för att hitta minsta bipartita matchningen i en viktad bipartit graf.

Ford Fulkerssons algoritm görs i två steg: Man försöker hitta en stig som ökar flödet genom grafen, sedan uppdaterar man grafen och flödet. När man inte längre kan hitta en sådan stig har man hittat ett maximalt flöde!

Algorithm 3: Ford-Fulkersson
Input: En graf G(E, V) där E är en mängd kanter och V en mängd noder.
Output: Minimala flödet genom grafen och om så önskas kanterna och deras respektive flöde.

```
foreach kant (u,v) ∈ E
    r[w][v] = w(u, v)
while C = getPath()
    V = min(w((u, v) ∈ C))
    for (u, v) = C
        r[u][v]- = V
        r[v][u]+ = V
```

Tidskomplexiteten till algoritmen ifall alla vikter är heltal är O(|E|*f), där f är det maximala flödet i grafen. Ifall vikterna är irrationella tal är tyvärr inte Ford-Fulkersson ens garanterad

att avslutas.

Ford-Fulkerssons algoritm använder sig oftast av en djupet-först sökning för att hitta utökande stigen i getPath(). Detta kan dock innebära problem i vissa specialfall, särskilt när bredden-först sökningen av någon orsak ständigt hittar stigar i grafen med låg kapacitet innan de stigarna med högre kapacitet.

Edmonds-Karp är en variant av Ford-Fulkerssons algoritm som istället använder en bredden-först först sökning, och har en tidskomplexitet på $O(|V||E|^2)$.