2D1458, Problemlösning och programmering under press

Föreläsning 6: Grafer del 2

Datum: 2006-10-09
Skribent: Per Wennersten
Föreläsare: Gunnar Kreitz

Den här föreläsningen var den andra av två om grafer och grafalgoritmer. Det som togs upp var Floyd-Warshall och Johnson's algoritmer för att hitta kortaste stigen mellan alla par av hörn i en graf, en algortim för att hitta Eulercykler, Ford-Fulkerssons metod för maxflöde i en graf, och några exempel på tillämpningar av maxflöde för problemlösning.

1 Floyd-Warshall

Floyd-Warshall används för att hitta kortaste stigar mellan alla nodpar (x,y) i en graf.

En naiv lösning är att köra $|V|$ st Dijkstra eller Bellman-Ford. Det ger komplexitet $O(|V||E|log(|V|))$ respektive $O(|V|^2|E|)$, men notera att Dijkstra endast fungerar för icke-negativa kantvikter.

Floyd-Warshall är en bättre lösning som bygger på dynamisk programmering.

Definition 1.1 *Låt $D^k_{x,y}$ beteckna minimiavståndet från nod x till nod y där stigen endast får innehålla noderna 0...k-1 förutom start- och slutnoden.*

Låt på samma sätt $P^k_{x,y}$ beteckna den sista noden som besöktes innan slutnoden i en stig med minimalt avstånd.

Initialisering

(1) $D^k_{x,y} \leftarrow \infty$
(2) $D^k_{x,x} \leftarrow 0$
(3) $P^k_{x,y} \leftarrow -1$
(4) **foreach** $(x, y) \in E$
(5) $\quad D^0_{x,y} \leftarrow wt(x, y)$
(6) $\quad P^0_{x,y} \leftarrow x$

Huvudalgoritmen

(1) **for** $k = 1$ **to** $|V|$
(2) **for** $x = 0$ **to** $|V| - 1$
(3) **if** $D^{k-1}_{x,k-1} < \infty$
(4) **for** $y = 0$ **to** $|V| - 1$
(5) **if** $D^{k-1}_{x,k-1} + D^{k-1}_{k-1,y} < D^{k-1}_{x,y}$
(6) $D^{k}_{x,y} \leftarrow D^{k-1}_{x,k-1} + D^{k-1}_{k-1,y}$
(7) $P^{k}_{x,y} \leftarrow P^{k-1}_{k-1,y}$
(8) **else**
(9) $D^{k}_{x,y} \leftarrow D^{k-1}_{x,y}$
(10) $P^{k}_{x,y} \leftarrow P^{k-1}_{x,y}$

Analys

Vid första anblick tycks algoritmen kräva tid och minne $\Theta(|V|^3)$, men om vi studerar loopen noggrannare ser vi att endast föregående värde på k används, så om vi bara lagrar nuvarande och föregående värden klarar vi oss med minne $\Theta(|V|^2)$. Faktum är att i just Floyd-Warshalls algoritm kan vi klara oss med att bara lagra ett $D_{x,y}$ och skippa k-indexeringen helt och hållet. Denna typ av lösning är vanlig inom dynamisk programmering, det lönar sig ofta att se hur mycket av information man faktiskt behöver ha kvar i varje steg.

En annan iakttagelse är att vi enkelt kan upptäcka negativa cykler med Floyd-Warshall, om $D^k_{x,x}$ är negativ så ingår x i en negativ cykel.

2 Johnsons algoritm

Vi gör en observation: upprepad Dijkstra går i $O(|V||E|log|V|)$, vilket är snabbare än Floyd-Warshalls $O(|V|^3)$ i glesa grafer. En idé är att patcha Dijkstra så att den fungerar för negativa vikter.

Potentialfunktionen

Vi inför en potentialfunktion $\pi : V \to \mathbf{R}$. Sedan viktar vi om alla kanter i grafen enligt $wt'(u, w) = wt(u, w) + \pi(u) - \pi(w)$. För att kontrollera denna lösning kan vi utreda vikten $wt'(P)$ för en stig P.

$$P = u_0, u_1, \ldots, u_k$$

$$wt'(P) = \sum_{i=1}^{k} wt(u_{i-1}, u_i) + \pi(u_{i-1}) - \pi(u_i)$$

Vi ser att detta är en teleskopsumma: alla potentialer utom den första och sista förekommer både som positiv och negativ term, så uttrycket kan förenklas till

$$wt'(P) = wt(P) + \pi(u_0) - \pi(u_k)$$

Eftersom $\pi(u_0)$ och $\pi(u_k)$ är konstanta för alla stigar mellan u_0 och u_k så kommer den nya viktningen ge samma kortaste stig som den gamla.

Så, det kvarstår att hitta en potentialfunktion sådan att $wt(u,w)+\pi(u)-\pi(w)$ är ickenegativ för alla tänkbara kanter. Detta kan åstadkommas genom att välja $\pi(u)$ som avståndet från ett fixt hörn till u. Triangelolikheten ger då

$$\pi(w) \leq \pi(u) + wt(u,w)$$

omskrivning ger

$$wt(u,w) + \pi(u) - \pi(w) \geq 0$$

vilket var precis det som söktes. Så, vilket fixt hörn ska vi använda som utgångspunkt? Det visar sig vara bra att lägga till ett nytt hörn med kanter av längd 0 till alla andra hörn, och använda avståndet till detta nya hörn som π.

Färdig algoritm

(1) $G' \leftarrow$ lägg till hörn s med kanter med avstånd 0 till alla hörn i G.
(2) Räkna ut π m.h.a Bellman-Ford
(3) **if** $\exists$ negativ cykel
(4)   avbryt
(5) **foreach** $v \in V$
(6)   DIJKSTRA(G, v, wt')

3 Parentes om Bellman-Ford

Bellman-Ford detekterar som vi vet negativa cykler som kan nås från startnoden, men hur är det om de inte kan nås från startnoden? Då beror det plötsligt på hur oändligheten implementeras. Om vi sätter oändligheten till ett stort tal, så kommer cyklerna ändå att upptäckas när vi gör den vanliga sökningen efter negativa cykler. Om vi däremot specialbehandlar oändligheten så att $\infty - wt(u,w) = \infty$, går cyklerna inte att upptäcka. Det är viktigt att tänka efter när man använder stora tal för att representera oändligheten. De måste vara större än alla riktiga värden vi kan få, men också tillräckligt små för att vi aldrig ska göra någon addition där summan orsakar overflow - det kan leda till elaka buggar.

4 Eulercykler

En Eulercykel är en cykel som besöker alla kanter i en graf exakt en gång. Det finns tillräckliga och nödvändiga villkor för existens av Eulercykler i riktade och oriktade grafer. En oriktad graf har en Eulercykel omm $deg(u)$ är jämnt för alla u och grafen är sammanhängande. En riktad graf har en Eulercykel omm $indeg(u) = utdeg(u)$ för alla u och grafen är sammanhängande. Notera här att en graf kan ha en Eulercykel trots att den har isolerade hörn, det är bara kanterna som måste besökas. Alltså, när vi kräver att grafen är sammanhängande, bortser vi från isolerade hörn.

Så, hur hittar vi en Eulercykel? En enkel algoritm består av två delar, en stigfinnare och en kontrollant.

Stigfinnare

Starta i godtyckligt hörn u. Traversera grafen så långt det går, utan att passera samma kant flera gånger. När det tar stopp är vi tillbaka i u, men det kan finnas obesökta kanter.

Kontrollant

Följ stigen från u som stigfinnaren har hittat tills vi kommer till ett hörn w med en obesökt kant. Starta stigfinnaren från w, för att hitta en stig av obesökta kanter som leder tillbaka till w. Skjut in hela den nya stigen där w besöktes i den gamla. Upprepa sökningen på samma sätt från w, längs den nya stigen. När vi kommer tillbaka till u är vi klara.

5 Flöde och Ford-Fulkerson

Vi tänker oss en uppgift som går ut på att ställa upp vägspärrar mellan Stockholm och Göteborg för att stoppa all biltrafik. Vi vill veta minimalt antal vägspärrar som behöver placeras ut, och var de ska ställas.

För att ta reda på hur många vägspärrar som krävs kan man representera vägnätet med en graf och låta alla kanter ha kapacitet 1. Vi vill sedan ta reda på vad det maximala flödet genom grafen från Stockholm till Göteborg är, och hur man uppnår det. Att lösa maxflödesproblemet kan man göra med hjälp av en metod som kallas Ford-Fulkerson.

Ford-Fulkerson

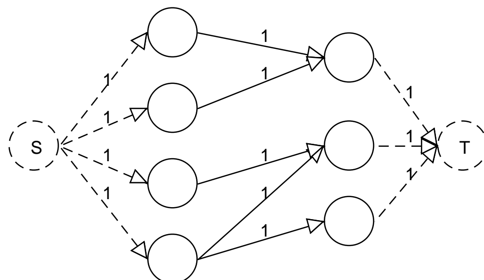
Grundidén i Ford-Fulkersson är ganska enkel

```
(1)   Sätt flödet till 0
(2)   while ∃ utökande stig
(3)       öka flödet längs stigen
```

En utökande stig är en stig från starthörnet till sluthörnet där vi kan få igenom ett flöde. När vi letar efter stigen använder vi oss av en residualflödesgraf, där residualkapaciteten är ursprungliga kapaciteten − nuvarande flödet. Vi ser också till att $f(u, w) = -f(w, u)$, alltså att vi har negativa flöden åt motsatt riktning, för att representera att vi kan “ångra” det flödet som tidigare gått genom kanten. För att åstadkomma detta inför vi för varje kant (u, w) en kant (w, u) med kapacitet 0, om det inte redan fanns en kant (w, u). När vi letar efter en utökande stig måste alltså alla ingående kanter ha en positiv residualkapacitet. Ett bra sätt att hitta utökande stigar är med en vanlig breddenförst-sökning, det kallas Edmonds-Karps algoritm. Det ger komplexitet $O(|V||E|^2)$. Om man istället använder en djupetförst-sökning riskerar man att få en väldigt dålig körtid på elaka exempel.

Var sätter vi vägspärrarna?

Vi börjar med att göra en sökning från startnoden i residualflödesgrafen, och markerar alla hörn vi når. Startnoden är nu markerad, medan slutnoden inte är det,



Figur 1: Illustration av bipartit matchning med hjälp av flöde.

eftersom vi då skulle ha hittat en utvidgande stig. Sedan sätter vi vägspärrar på alla vägar som går från markerade noder till omarkerade. Flödet genom vägarna vi nu har spärrat måste ha varit lika med kapaciteten på vägarna, annars hade vi kunnat ta oss genom kanten i residualflödesgrafen och markerat båda ändarna. Flödet längs de spärrade vägarnna kan inte heller vara större än flödet från s till t, eftersom varje flöde från s till t går över från markerade till omarkerade noder högst en gång (om det går flöde från u till w och w är markerad så är även u markerad, eftersom vi kan gå baklänges längst flödet i residualgrafen).

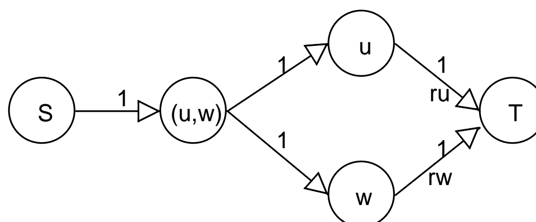
6 Flödesxempel: Bipartit matchning

En bipartit graf är en oriktad graf sådan att hörnen kan delas in i två disjunkta mängder så att alla kanter i grafen har ett hörn i varje mängd. En matchning är ett val av kanter sådant att max en utvald kant går till ett givet hörn. Vi kan hitta en maximal bipartit matchning i en graf genom att ge alla kanter kapacitet 1 och rikta dem från ena mängden till den andra. Sedan lägger vi till ett hörn som är start och ett hörn som är mål, och sätta kanter från starten till alla hörn i ena mängden, och kanter från alla hörn i andra mängden till målet. De ursprunliga kanterna som har ett positivt flöde ingår i en maximal matchning.

Om kanterna dessutom har en kostnad kan vi hitta en maximal matchning av minimal kostnad om vi hela tiden hittar den billigaste utökande stigen i Ford-Fulkersson. Eftersom kantvikterna kan vara negativa görs detta lämpligen med Bellman-Fords algoritm.

7 Flödesxempel: Bus Tour

Det här är ett exempel från NWERC 2003. En buss åker runt i en stad där några gator är enkelriktade, och frågan är om det existerar en Eulercykel. När det finns både oriktade och riktade kanter i en graf har vi inget enkelt sätt att avgöra om en Eulercykel existerar, men vi kan istället se det som att vi ska rikta alla oriktade



Figur 2: Illustration av lösning av Bustour med hjälp av flöde

kanter på ett sådant sätt att det sedan finns lika många in- och utkanter från varje hörn.

Betrakta ett hörn u som har deg_+ kanter ut, deg_- kanter in och deg_0 oriktade kanter. Vi vill rikta r_u kanter in mot hörnet så att

$$r_u + deg_- = (deg_+ + deg_- + deg_0)/2$$

När vi funnit r_u för alla hörn kan vi konstruera en graf för att lösa problemet med hjälp av flöde. Skapa ett hörn i den nya grafen för varje hörn och oriktad kant i den gamla, och låt kanter med kapacitet ett gå från hörnen som representerar kanter till båda hörnen som kan nås från kanten. Låt sedan kanter med kapacitet 1 gå från ett starthörn till alla hörn som representerar kanter. Slutligen, skapa kanter med kapacitet r_u från alla hörn som representerar ett hörn u i ursprungsgrafen till ett sluthörn.

En enhet av flöde från hörnet (u, w) till hörnet u representerar att kanten (u, w) i ursprungsgrafen riktas mot hörnet u. Om maxflödets storlek är lika med $\sum r_u$ så existerar en Eulercykel i den ursprungliga grafen.