2D1458, Problemlösning och programmering under press

Föreläsning 5: Grafer Del 1

Datum: 2006-10-02
Skribent(er): Henrik Sjögren, Patrik Glas
Föreläsare: Gunnar Kreitz

Den här föreläsningen var den första av två om grafer och grafalgoritmer. Det som togs upp var olika sätt att representera grafer samt algoritmer för att finna minimala spännande träd, starkt sammanhängande komponenter, och minsta avstånd i en graf.

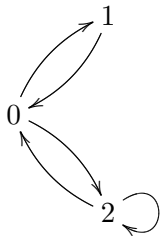
1 Grafer

En graf G är en tupel (V, E), där V är en mängd av noder och E en mängd av par $\{(u, v) : u, v \in V\}$ som betecknar kanter mellan noderna.

2 Representation

2.1 Grannmatris

En grannmatris är en matris $A = \{a_{ij}\}$ där $a_{ij} = 1$ om $(i, j) \in E$, annars 0.



$$\begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 1 \end{bmatrix}$$

0	–	$[(0,1),(0,2)]$
1	–	$[(1,0)]$
2	–	$[(2,0),(2,2)]$

Figur 1: En riktad graf och dess representation som grannmatris respektive kantlistor.

2.2 Kantlistor

För varje hörn $v \in V$ finns en lista $adj[v]$ av alla kanter ut från v. D.v.s.

$$(u, v) \in E \Leftrightarrow (u, v) \in adj[u]$$

2.3 Jämförelse

Grannmatriser har fördelen att de är marginellt lättare att implementera. Kantlistor blir å andra sidan mindre för glesa grafer vilket gör att de tar mindre minne och kan vara snabbare att söka igenom. Det finns fall när man kan vilja använda båda representationer, t.ex. om man vill filtrera bort multipla kanter i indata, vilket lätt görs med en grannmatris, men sedan vill använda en algoritm som fungerar bättre med kantlistor.

3 Oriktade grafer

Oftast kan vi omvandla en oriktad kant (u, v) till två riktade kanter (u, v) och (v, u). Detta fungerar dock inte för alla algoritmer (t.ex. Eulercykel, där vi går längs alla kanter).

4 Vanliga fallgropar

Vanliga fallgropar när man löser grafproblem är man missar att tänka på självloopar, multipla kanter och osammanhängande grafer. Skriver man i C++ bör man tänka på att maps av typen

```
map<pair<int,int>,int>
```

inte är lämpliga att använda för grafer med multipla kanter då dessa kanter kommer att skriva över varanda i map:en.

5 Grafalgoritmer

5.1 Minsta Spännande Träd

Ex 5.1. *Vi har ett antal namngivna städer, ett antal existerande vägar och ett antal offerter på nya vägar. Vårt problem är att koppla ihop alla städer (indirekt) så billigt som möjligt.*

Modell 5.1.1. *Vi skapar en graf G med en nod för varje stad, en kant med vikt noll för varje existerande väg och en kant med priset som vikt för varje offert. Vi vill gärna jobba med numrerade noder sådana att $V = \{0, \ldots, n-1\}$. Vi kan uppnå detta genom att tilldela varje stadsnamn ett nummer med en index map*[1].

[1] Vi översätter index till stadsnamn genom att ha en vektor med alla stadsnamn i, en översättning går i konstant tid. För att översätta stadsnamn till index kan vi använda en hashtabell, för vilken översättningar går i förväntad konstant tid, eller en map (binärt sökträd), för vilken översättningar går i $O(log|V|)$. För varje ny stad vi läser in lägger vi till den sist i vektorn.

Lösning 5.1.1. *Problemet är nu att hitta ett Minsta Spännande Träd (MST) i vår graf G och vi kan lösa det med Prims eller Kruskals algoritm, dock går vi bara igenom Prims.*

Algoritm 1: Prims algoritm
Beskrivning: Vi börjar i godtycklig nod, markerar den som besökt och utökar sen trädet med den billigaste kanten som går från en besökt nod till en obesökt.
Input: En graf G med viktade kanter
Output: Minsta Spännande Träd för G

```
(1)   V = {0, ..., n − 1}
(2)   foreach u ∈ V
(3)       P[u] ← −1 /* Förälder till u */
(4)       d[u] ← ∞ /* Kostnad för att lägga till u */
(5)       vis[u] ← false /* Sann om u är besökt */
(6)   d[0] ← 0
(7)   for i = 1 to |V|
(8)       Hitta en nod u s.a. vis[u] = false, d[u] minimal
          och d[u] ≠ ∞
(9)       vis[u] ← true
(10)      foreach (u, w) ∈ E
(11)          if not vis[w] and d[w] > wt(u, w)
(12)              d[w] ← wt(u, w)
(13)              p[w] ← u
```

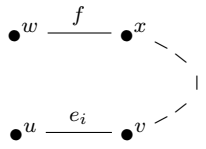
Det är inte självklart hur vi bör implementera rad (8).

- *(I) Antingen linjärsöker vi genom alla hörn vid (8). Algoritmen går då i tid* $\Theta(|V|^2)$.
- *(II) Smartare är då att i (8) välja första elementet i en prioritetskö med alla obesökta noder u (för vilka* $d[u] \neq \infty$*), sorterade efter ökande kostnad. I (12) måste vi då uppdatera prioritetskön, vilket går att göra i tid* $log|V|$*. Vi får då en tidskomplexitet på* $O(|E|log|V|)$ *för hela algoritmen om vi använder kantlistor*[2]*.*

Bevis. Vi vill visa att det träd som väljs i varje steg kan utökas till ett optimalt träd. Vi visar detta med induktion.
Basfall: Från början finns inga kanter så vi har ett delträd till det optimala.
Induktionsantagande: Mängden $E_{i-1} = \{e_1, \ldots, e_{i-1}\}$ av kanter i G kan utökas till ett MST.
Induktionssteg: Vi har valt E_{i-1}, enligt induktionsantagandet finns ett optimalt träd T s.a. $E_{i-1} \subseteq E(T)$, där $E(T)$ är mängden av kanter i T.

- Om $e_i \in T$ är vi klara.

[2]Notera att om grafen är tät (med $|E| \approx |V|^2$) så är altertaiv (I) snabbare.



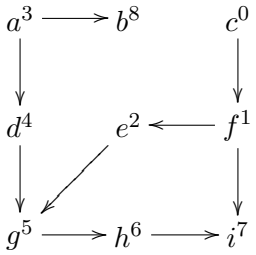
Figur 2: Noderna u, w och alla noder till vänster om dessa är besökta, noderna v, x och alla noder till höger om dessa är obesökta.

- Annars så innehåller $\{e_i\} \cup T$ en cykel som i sin tur innehåller en kant f som går mellan en besökt och en obesökt nod (se Figur 2). Vi har att $wt(f) \geq w(e_i)$, annars skulle algorimen valt f istället för e_i. Då följer att $T' = (T \cup \{e_i\}) \backslash f$ också är ett träd och att $wt(T') \leq wr(T)$ men då T är optimalt har vi att $wt(T') = wt(T)$. Alltså går $E_i = \{e_1, \ldots, e_i\}$ att expandera till ett optimalt träd.

□

5.2 Topologisk sortering av Riktad acyklisk graf (DAG)

Vi vill tilldela alla noder u ett nummer, $topnr[u]$, som beskriver i vilken ordning vi kan besöka dem om vi utgår från en specifik startnod. Mer specifikt har vi för $u, v \in V$ att $topnr[u], topnr[v] \in \{0, \ldots, n-1\}$ s.a. om det existerar en stig från u till v så gäller $topnr[u] < topnr[v]$. Vi löser detta enligt Algoritm 2.



Figur 3: Topologisk sortering av noderna i en graf

Bevisskiss. För att visa att algoritmen är korrekt måste vi visa att för varje kant (u, v) gäller att $topnr[u] < topnr[v]$. Detta kan vi göra genom att notera följande:

- Använder vi kanten (u, v) i den DFS-sökning som besöker nod v följer det att att $topnr[u] = topnr[v] - 1$.
- Annars så är nod u inte besökt (för då skulle även nod v vara det) och då kommer DFS-sökningen som besöker nod v startas med $nr < topnr[v]$.

Algoritmen har tidskomplexitet $O(|V| + |E|)$ då vi i loopen i (5) fortsätter tills alla noder är besökta, aldrig besöker en redan besökt nod, och för varje nod vi

Algoritm 2: Topologisk sortering
Beskrivning:
Input: En DAG, G
Output: En vektor med nummer för alla noder i G

```
(1)   for u = 0 to n − 1
(2)     vis[u] ← false /* Sant om u är besökt */
(3)     topnr[u] ← ∞ /* Topologisk nummer för u */
(4)   nr ← n − 1
(5)   for u = 0 to n − 1
(6)     if not vis[u]
(7)       nr ← DFS(u, nr, vis, topnr)
(8)
(9)
(10)  DFS(u, nr, vis, topnr)
(11)    vis[u] ← true
(12)    foreach (u, v) ∈ E
(13)      if not vis[v]
(14)        nr ← DFS(v, nr, vis, topnr)
(15)      else
(16)        if topnr[v] = ∞
(17)          CYKEL!!
(18)    topnr[u] ← nr
(19)    return nr − 1
```

besöker undersöker alla utgående kanter.

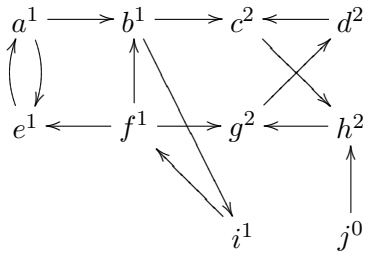
5.3 Starkt Sammanhängande Komponenter

En starkt sammanhängade komponent (strongly connected component, SCC) C är en maximal delmängd noder s.a. om $u, v \in C$ så finns det en stig $u \rightsquigarrow v$ och en stig $v \rightsquigarrow u$. Att hitta alla SCC:er i en graf $G = (V, E)$ går att göra i tid $\Theta(|V| + |E|)$ enligt följande: Först kör vi en toplogisk sorting på grafen, men ignorerar eventuella cykler. Sedan skapar vi grafen G^T som är identisk med G fast med alla kanter vända. D.v.s.

$$\begin{aligned} V(G^T) &= V(G) \\ E(G^T) &= \{(v, u) : (u, v) \in E(G)\} \end{aligned}$$

Till sist loopar vi över alla noder i G^T enligt den tidigare sorteringen och gör, för varje obesökt nod, en DFS-sökning. De noder som besöks i en DFS-sökning kommer vara en unik SCC.
Bevisskiss. Vi visar inte att algoritmen är korrekt men ger ett Lemma som hjälper en bit.

Lemma 5.1. *Låt C, C' vara olika SCC i $G = (V, E)$. Antag att $(u, v) \in E^T = E(G^T)$ där $u \in C$, $v \in C'$. Då kommer första noden i C' före alla noder i C i den topologiska sorteringen.*



Figur 4: En graf med markerade starkt sammanhängande komponenter (markerade 0, 1 och 2)

Detta medför att C' räknas ut före C, och DFS-sökningen som startas på en nod i C kommer då inte besöka någon nod i C' då de redan är besökta.

5.4 Hitta avstånd i en graf

Vi vill hitta det kortaste avståndet i en graf G från en given nod s till alla andra noder. Vi börjar med att betrakta grafer där kanterna har ickenegativa vikter.

5.4.1 Dijkstra

Dijkstras algoritm (Algoritm 3) för att hitta minsta avstånd i en graf med ickenegativa vikter bygger på Prims algoritm för MST men istället för att i innersta loopen uppdatera kostnader för att lägga till en nod i ett MST uppdaterar vi avståndet till noden.
Algoritmen har samma tidskomplexitet som Prims algoritm (d.v.s $O(|V|^2)$ eller $O(|E|log|V|)$ beroende på implementation), då funktionen RELAX, precis som rad (12) i Prim kan kräva en uppdatering av en prioritetskö vilket tar tid $O(log|V|)$.
Vi kan visa att algoritmen fungerar med induktion på de besökta noderna.

Bevis. För varje nod v kommer avståndet till noden fixeras först när noden är besökt, vi måste alltså visa att ingen nod kan besökas medan den fortfarande kan få ett lägre avstånd tilldelat.
Detta kan visas med induktion enligt följande:
Basfall: Noden s ligger alltid på avstånd 0 från sig själv och besöks alltid först, alltså är dess avstånd fixerat när den besöks.
Induktionsantagande: Låt $l(u \rightsquigarrow v)$ beteckna längden av kortaste stigen från nod u till nod v. Låt u vara den obesökta nod som har minimalt $d[u]$. För alla noder v för vilka $l(s \rightsquigarrow v)$ är mindre än $l(s \rightsquigarrow u)$, gäller att v är besökt och att $d[v]$ därmed är fixerat till just $l(s \rightsquigarrow v)$. För alla noder w, som har en ingående kant från en besökt nod, har vi dessutom att $d[w]$ är satt till det minsta avstådet från s via besökta noder.
Induktionssteg: För att $d[u]$ för den nod u som vi besöker skall uppdateras med ett lägre värde efter att den besökts måste det finnas en obesökt nod $w \neq u$, med en kant (v, w) från en besökt nod v sådan att stigen $s \rightsquigarrow v \rightarrow w$ är optimal bland de stigar från s till w som endast går via besökta noder. Vi har då enligt induktionsantagandet

att $d[w] = d[v] + wt(v,w) = l(s \rightsquigarrow v) + wt(v,w)$. Dessutom måste det finnas en stig $w \rightsquigarrow u$ och det måste gälla att $d[u] > l(s \rightsquigarrow v) + wt(v,w) + l(w \rightsquigarrow u)$ men eftersom $d[w] = l(s \rightsquigarrow v) + wt(v,w)$ och $l(w \rightsquigarrow u)$ är positiv har vi att $d[u] > d[w]$ vilket innebär att algoritmen skulle valt w och inte u. Alltså är avståndet till u optimalt.
Fixerar vi $d[u]$ och markerar den som besökt måste induktionsantagandet gälla även för den nod u' som nu har minimalt $d[u']$ bland de obesökta noderna. Det vill säga:

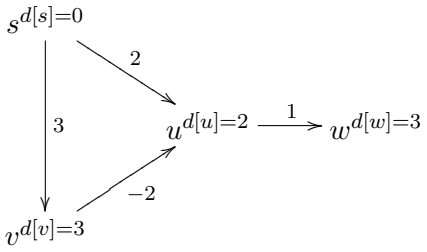
- För alla noder v för vilka $l(s \rightsquigarrow v) < l(s \rightsquigarrow u')$ är v besökt och $d[v] = l(s \rightsquigarrow v)$, detta gäller sen tidigare för alla $v \neq u$ och nu även för u då $d[u]$ är optimalt enligt ovan.

- För alla noder w, där $(v,w) \in E$ är $d[w] = min(\{l(s \rightsquigarrow v) + wt(v,w) : v \text{ besökt}\})$ vilket gäller för alla w, för vilka $(u,w) \notin E$, sen tidigare. För alla w, för vilka $(u,w) \in E$, gäller nu att $d[w] = min(d[u] + wt(u,w), d'[w])$ där $d'[w]$ är det förra värdet på $d[w]$. Om sista noden i den kortaste stigen via besökta noder till w är u så är $d[u] + wt(u,w) < d[w]$ då $d[u]$ är optimal och $d[w]$ är optimal bland de stigar som bara går via $\{v : v \text{ besökt}, v \neq u\}$, $d[w]$ blir då $d[u] + wt(u,w)$ vilket är optimalt då $d[u]$ är det. Annars behåller $d[w]$ sitt värde vilket fortfarande är optimalt.

□

Algoritm 3: Dijkstra
Input: En graf G med viktade kanter och en startnod s
Output: Minsta avståndet från s till alla andra noder i G

```
(1)   V = {0,...,n-1}
(2)   foreach u ∈ V
(3)     p[u] ← -1 /* Förälder till u */
(4)     d[u] ← ∞ /* Avståndet från s till u */
(5)     vis[u] ← false /* Sann om u är besökt */
(6)   d[s] ← 0
(7)   for i = 1 to |V|
(8)     Hitta en nod u s.a. vis[u] = false, d[u] minimal
        och d[u] ≠ ∞
(9)     vis[u] ← true
(10)    foreach (u,w) ∈ E
(11)      if not vis[w]
(12)        RELAX(u,w)
(13)
(14)
(15)  RELAX(u,w)
(16)    if d[w] > d[u] + wt(u,w)
(17)      d[w] ← d[u] + wt(u,w)
(18)      p[w] ← u
```



Figur 5: Ett exempel på när Dijkstra inte fungerar, $d[u]$ sätts till 2 när den egentligen borde bli 1 och $d[w]$ sätts till 3 när den borde vara 2. Att uppdatera även intilliggande besökta noder när en nod besökts skulle ge rätt värde på $d[u]$ men $d[w]$ skulle behålla sitt felaktiga värde.

5.4.2 Bellman-Ford

Om vi däremot har negativa vikter på vissa kanter fungerar inte Dijkstra (se Figur 5). Då kan vi istället använda Bellman-Ford (Algoritm 4) som fungerar så länge det inte finns cykler med negativ vikt. Om det finns negativa cykler kan den upptäcka detta. Då vi här inte behöver en prioritetskö (till skillnad från Dijkstra) går RELAX i konstant tid. Vi anropar RELAX $O(|V||E|)$ gånger vilket då också är komplexiteten för hela algoritmen.

Vi vill visa att algoritmen är korrekt.
Bevisskiss. Om $s \to u_1 \to u_2 \to \ldots \to u_r$ är den kortaste stigen till u_r så har vi efter i varv den kortaste stigen till u_i klar. Det högsta antalet kanter i den kortaste stigen till någon nod (utan negativa cykler) är $|V| - 1$.
Har vi en negativ cykel som kan nås från startnoden kan vi alltid uppdatera avstånden längs den cykeln i all oändlighet så om vi, när vi har räknat ut avstånd för alla noder (utan att bry oss om dessa cykler), fortfarande kan minska något avstånd måste vi ha en negativ cykel (under antagandet att algoritmen annars är korrekt). Kan cykeln inte nås från startnoden och alltså ligger på oändligt avstånd, kommer alla uppdateringar att återigen ge samma avstånd[3] och den negativa cykeln kan då alltså inte hittas.

[3]Väljer vi att representera oändlighet som ett stort tal kan vi även detektera onåbara negativa cykler men vi har då istället svårare att skilja oändlighet från ett giltigt värde.

Algoritm 4: Bellman-Ford
Input: En graf G med viktade kanter och en startnod s
Output: Minsta avståndet från en nod s till alla andra noder i G

```
(1)     for u = 0 to |V| − 1
(2)         d[u] ← ∞
(3)         p[u] ← −1
(4)     d[s] ← 0
(5)     for i = 1 to |V| − 1
(6)         foreach (u, w) ∈ E
(7)             RELAX(u,w)
(8)
(9)     /* Leta efter negativa cykler */
(10)    foreach (u, v) ∈ E
(11)        if d[w] > d[u] + wt(u, w)
(12)            NEGATIV CYKEL!!
```