2D1458, Problemlösning och programmering under press

Föreläsning 12: Önskekonsert

Datum: 2006-12-04
Skribent(er): Henrik Sjögren, Jesper Hjertstedt
Föreläsare: Mikael Goldmann

Innan föreläsningen efterfrågades dels något praktiskt, dels något teoretiskt, därför kom föreläsningen att handla om Kruskals algoritm, en datorstruktur för operationer på disjunkta mängder som kan användas för att optimera denna algoritm samt amorterad analys, för analysera vår datastruktur.

1 Kruskals algoritm för MST

Under graf-föreläsningen nämndes Kruskals algoritm endast i förbigående. Kruskals algoritm bygger, givet en graf G, upp ett minimalt spännande träd för G genom att börja med en graf utan kanter och sedan succesivt lägga till kanter från G. I varje steg i algoritmen läggs den kant som, dels har minst vikt, dels inte skapar en cykel till i den nya grafen, till. (Se Algoritm 1).

Algoritm 1: Kruskals algoritm
Beskrivning: Vi börjar med $T = (V, 0)$ och lägger succesivt till kanter från G
Input: En graf $G = (V, E)$ med viktade kanter
Output: Minsta Spännande Träd för G

(1) $T \leftarrow 0$
(2) **for** $i = 1$ **to** $|V| - 1$
(3) Hitta billigast $e \in E$ sådan att $T \cup \{e\}$ ej har någon cykel
(4) $T = T \cup \{e\}$

Vi har två delar som är oklara hur vi bör hantera dem

- "Hitta billigaste $e \in E$": Om vi först sorterar kanterna efter vikt kan detta göras enkelt. Sortering tar tid $O(|E| \log |E|)$.
- "..sådan att $T \cup \{e\}$ ej har någon cykel": För att inte skapa en cykel måste noderna x och y för kanten $e = (x, y)$ ligga i olika komponenter i T. För att avgöra vilken komponent en nod ligger i kan vi, för varje nod x, lagra detta i säg $komponent[x]$, då kan vi göra uppslagningen här i konstant tid. Däremot kommer vi behöva slå ihop två komponenter i steget efter. Om vi lägger till en kant $e = (x, y)$ så måste vi då markera om alla hörn i antingen den komponent som x ligger i eller i den som y ligger i. Om vi alltid märker om den komponent som är minst får vi en total körtid $O(|V| \log |V|)$ för algoritmen. Efter ommärkning av en nod tillhör den en minst dubbelt så stor

komponent som innan. Vi kan alltså som mest märka om den $\log |V|$ gånger då komponenten noden tillhör efter $\log |V|$ ommärkningar måste innehåller hela V. Detta kan förbättras genom att använda den datatyp för disjunkta mängder som följer.

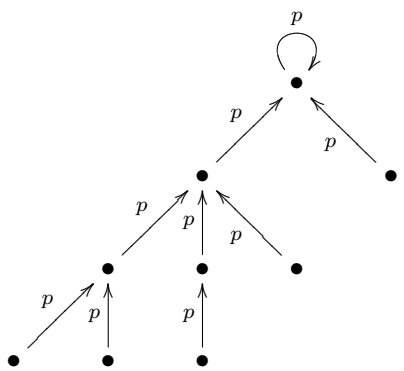
2 Datatyp för operation på disjunkta mängder

Låt $X = \{x_1, \ldots, x_n\}$ vara alla element som skall behandlas och låt $S_1, \ldots, S_m$ vara ett antal mängder av dessa element. Vi definierar följande operationer:

- $MakeSet(x_i) = \{x_i\}$
- $Find(x_i) = j \in \mathbb{N}$ sådant att $x_i \in S_j$
- $Union(x_i, x_j) =$ Slår ihop $S_{Find(x_i)}$ och $S_{Find(x_j)}$ till en mängd. De två ursprungliga mängderna förstörs.

2.1 Implementation

Vi representerar en mängd som ett träd med noder i X och riktade kanter mot trädets rot. Vi låter trädets rot representera hela trädet (Se Figur 1).



Figur 1: Trädrepresentation av en mängd

2.1.1 Operationer

Vi kan nu implementera våra operationer enligt Algoritm 2 och 3. Våra träd kan dock i värsta fall degenerera till listor (om argumentet b i link alltid är ensamt i sin mängd) och då tar Find-operationen i värsta fall tid $O(k)$ där k är antal element i mängden.

För att undvika detta kan vi hålla reda på storleken av varje delträd: Vi låter $rank[x]$ vara en (över-) skattning av djupet i det träd där x är rot och skriver om våra operationer som i Algoritm 4 och 5 (där Union är som tidigare) så att vi hela tiden försöker hålla träden så grunda som möjligt. Find-operationen kommer nu max att ta tid $O(s)$ där s är djupet i trädet som representerar den sökta mängden,

Algoritm 2: Operationer på disjunkta mängder
Beskrivning: Ett första försök till implementation av tidigare definierade operationer på disjunkta mängder (MakeSet och Find)

```
MakeSet(x)
    p[x] ← x #gör x till rot

Find(x)
    if x = p[x]
        return x
    else
        return Find(p[x])
```

Algoritm 3: Operationer på disjunkta mängder
Beskrivning: Ett första försök till implementation av tidigare definierade operationer på disjunkta mängder (Union)

```
Union(x, y)
    a ← Find(x)
    b ← Find(y)
    if a ≠ b
        Link(a, b)

Link(a, b)
    p[a] ← b
```

operationen kommer dessutom hela tiden försöka hålla s så litet som möjligt genom att uppdatera förälderpekarna (Se Figur 2).

2.2 Analys

Våra operationer har nu komplexitet som följer för (n element total):

- MakeSet: $O(1)$
- Find: $O(\log n)$ men väldigt få Find-anrop kommer ta så lång tid.
- Union: $O(1)$
- Link: $O(1)$

Vi vill visa att Find sällan tar så lång tid som $O(\log n)$ stämmer i påståendet ovan och att $O(m \cdot \alpha(n))$ faktiskt är en övre gräns för körtiden om vi gör m st operationer (av någon av ovanstående typer) på n st element (där $\alpha(n)$ är en funktion som växer mycket långsamt). Vi vill då också visa att funktionen $\alpha()$ ovan växer väldigt långsamt. Låt A_k vara som följer:

$$A_k(j) = \begin{cases} j+1 & \text{om } k = 0 \\ A_{k-1}^{(j+1)}(j) & \text{annars} \end{cases}$$

Algoritm 4: Operationer på disjunkta mängder
Beskrivning: Ett andra försök till implementation av tidigare definierade operationer på disjunkta mängder där vi försöker att minska trädens djup (MakeSet och Find)

```
MakeSet(x)
    p[x] ← x # gör x till rot
    rank[x] ← 0 # initiera rank[x]


Find(x)
    if x ≠ p[x]
        p[x] =Find(p[x]) # uppdatera förälderpekare
        enligt Figur 2.
    return p[x]
```

Algoritm 5: Operationer på disjunkta mängder
Beskrivning: Ett andra försök till implementation av tidigare definierade operationer på disjunkta mängder där vi försöker att minska trädens djup (Link)

```
Link(a, b)
    if rank[a] < rank[b]
        p[a] ← b
    else
        p[b] ← a
        if rank[a] = rank[b]
            rank[a] ← rank[a] + 1 # om träden är lika
            djupa kommer det totala djupet att öka med
            ett
```

där $A_{k-1}^{(j+1)}(j) = A_{k-1}(A_{k-1}(\ldots(A_{k-1}(j))\ldots))$ så att A_{k-1} appliceras $j+1$ gånger (Se Figur 3).

Vi kan nu definiera α som följer (Se även Figur 4):

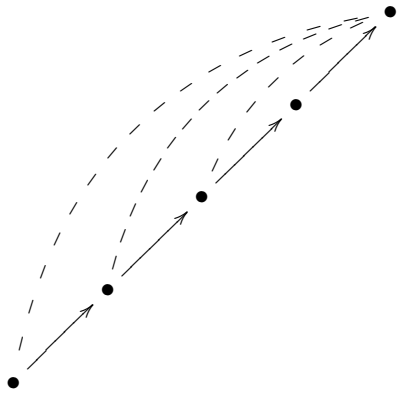
$$\alpha(n) = min\{k|A_k(1) \geq n\}$$

För att visa att vårt α faktiskt stämmer gör vi en amorterad analys av vår datastruktur.

3 Amorterad analys, datastrukturer

Det finns tre vanliga typer av amorterad analys:

- Totalanalys
- Bokföringsmetoden
  Man tilldelar här varje operation en amorterad kostnad $\hat{c}_i$ sådan att $\hat{c}_i$ inte



Figur 2: Uppdatering av förälderpekare i Find-operationen

nödvändigtvis är större än den verkliga kostnaden c_i men så att $\sum_{i=1}^{m} \hat{c}_i \geq \sum_{i=1}^{m} c_i$ där vi summerar över alla operationer i en körning.

- Potentialmetoden
  Man skapar en potentialfunktion $\Phi(D_i) = \Phi_i$, där D_i är datastrukturen efter i operationer, sådan att $\Phi_0 = 0$ och $\Phi_i \geq 0$. Man tilldelar sen operationerna amorterade konstnader enligt $\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$. Vi får då att

  $$\sum_{i=1}^{m} \hat{c}_i = \sum_{i=1}^{m} c_i + \Phi_i - \Phi_{i-1} = \Big(\sum_{i=1}^{m} c_i\Big) + \Phi_m - \Phi_0 \geq \sum_{i=1}^{m} c_i \ .$$

  (Potentialfunktionen utformas sådan att "billiga" operationer ökar potentialen något medan dyra operationer sänker den.)

Vi kommer att använda Potentialmetoden.

Ex 3.1 *Låt oss analysera en stack, vi har operationerna push(x), pop() och multipop(k) där push och pop fungerar som vanligt och multipop poppar k element från stacken. Låt potentialfunktionen $\Phi(D_i)$ vara stackdjupet efter i operationer. Varje operation ges amorterad kostnad $\hat{c}_i = c_i +$ <ökning av Φ>, d.v.s:*

- *push*(x) : $\hat{c}_i = 1 + 1 = 2$
- *pop*() : $\hat{c}_i = 1 - 1 = 0$
- *multipop*(k) : $\hat{c}_i = k - k = 0$

4 Amorterad analys av vår tidigare datastruktur

Vi är nu redo att analysera vår datastruktur för disjunkta mängder. Notera följande:

$$
\begin{aligned}
A_1(j) &= A_0^{(j+1)}(j) = A_0^{(j)}(j+1) = 2j+1 \\
A_2(j) &= 2^{j+1}(j+1) - 1
\end{aligned}
$$

$$
\begin{aligned}
A_0(1) &= 2 \\
A_1(1) &= 3 \\
A_2(1) &= 7 \\
A_3(1) &= A_2^{(2)}(1) = A_2(7) = 2047 \\
A_4(1) &= A_3^{(2)}(1) = A_3(2047) = A_2^{(2048)}(2047) \gg A_2(2047) \gg 2^{2048}
\end{aligned}
$$

Figur 3: Värden för $A_k(j)$ för olika k, j

$$
\begin{aligned}
&\alpha(n) = 0 && \text{för } n \leq 2 \\
&\alpha(n) = 1 && \text{för } n = 3 \\
&\alpha(n) = 2 && \text{för } 4 \leq n \leq 7 \\
&\alpha(n) = 3 && \text{för } 8 \leq n \leq 2047 \\
&\alpha(n) = 4 && \text{för } 2048 \leq n \leq A_4(1) \gg 2^{2048}
\end{aligned}
$$

Figur 4: Värden på $\alpha(n)$ för olika n

- Om $x = p[x]$ och $rank[x] = r$ har trädet med rot i x minst 2^r noder.
- $rank[x] \leq n - 1$ (faktiskt $\leq \log n$)
- $rank[x] = rank[p[x]]$ omm x är en rot
  $rank[x] < rank[p[x]]$ annars
- $rank[x]$ kan aldrig minska
- om $x \neq p[x]$ så ändras inte $rank[x]$ mer

4.1 Potentialfunktion

Vi vill definiera en funktion $\varphi_i(x)$ för varje x sådan att $\Phi_i = \sum_x \varphi_i(x)$. För att göra detta behöver vi två hjälpfunktioner:

4.1.1 $level(x)$

Låt $level(x) = \max\{k | rank[p[x]] \geq A_k(rank[x])\}$. Notera att $level(x)$ endast är definierad om $x \neq p[x]$ och $rank[x] \geq 1$. Notera också att att $0 \leq level(x) < \alpha(n)$.

4.1.2 $iter(x)$

Notera att

$$A_{level(x)}(rank[x]) \leq rank[p[x]] < A_{level(x)}^{(rank[x]+1)}(rank[x]) = A_{level(x)+1}(rank[x])$$

Låt $iter(x) = \max\{i | A^{(i)}_{level(x)}(rank[x]) \leq rank[p[x]]\}$. Notera att $1 \leq iter(x) \leq rank[x]$.

4.1.3 $\varphi(x)$

Vi har nu två mått på hur mycket större *rank* är för föräldern till en nod x jämfört med *rank* för x och kan med dessa definiera vårt φ enligt följande:

$$\varphi(x) = \begin{cases} \alpha(n) \cdot rank[x] & \text{om } x = p[x] \text{ eller } rank[x] = 0 \\ (\alpha(n) - level(x))rank[x] - iter(x) & \text{om } x \text{ inte är en rot och } rank[x] \geq 1 \end{cases}$$

Notera att $0 \leq \varphi(x) \leq \alpha(n) \cdot rank[x]$ då $iter(x)$ både kan ökas eller minskas, om den minskar gör den då det med som mest $rank[x] - 1$ varvid $level(x)$ ökar. $level(x)$ kan dessutom aldrig minska. Detta medför att om någon av dem ändras i steg i gäller

$$\varphi_i(x) \leq \varphi_{i-1}(x) - 1$$

4.2 Analys av datastrukturen

I analysen räknar vi Union-operationen som två Find-operationer och en Link-operation och antar att vi kör alla MakeSet-operationer i början innan någon annan typ av operation. Då får vi amorterade konstnader för de olika operationerna enligt följande.

4.2.1 MakeSet

Både före och efter anropet till MakeSet kommer $rank[k]$ att vara noll, alltså kommer $\varphi_i(x) = \varphi_{i-1}(x) = 0$ om MakeSet körs i steg i. För $y \neq x$ kommer även $\varphi_i(y) = \varphi_{i-1}(y)$ att gälla, vi har alltså ingen förändring i potential och

$$\hat{c}_i = c_i = 1$$

4.2.2 Link

Vi antar att $rank[x] \geq rank[y]$. I så fall kommer x bli ny förälder till y och $\varphi(y)$ kan minska, liksom $\varphi(z)$ där z ligger i den mängd som har x som rot. Skulle båda träden vara lika djupa ökar $rank[x]$ med 1 och då ökar $\varphi(x)$ med $\alpha(n)$. Vi får då

$$\hat{c}_i \leq c_i + \alpha(n) = 1 + \alpha(n)$$

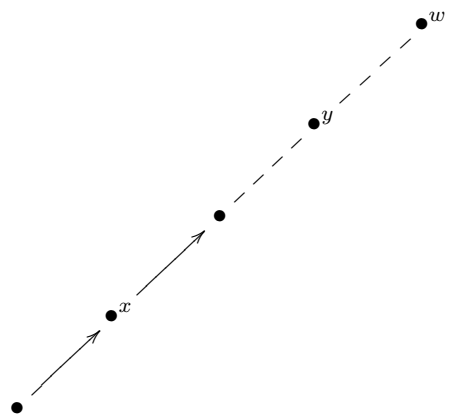
4.2.3 Find

Vi betraktar $Find(x)$ där x ligger s steg från roten i sin mängd, c_i är då precis s. Vi vill visa att för minst $s - (\alpha(n) + 2)$ noder längs stigen så minskar potentialen. Vi har då

$$\Phi_i \leq \Phi_{i-1} - s + \alpha(n) + 2$$

och som en direkt följd

$$\hat{c}_i \leq s - s + \alpha(n) + 2 \in O(\alpha(n))$$



Figur 5: Situation innan en körning av Find-operationen

Betrakta situationen i Figur 5, låt situationen vara sådan att vi före operationen har $level(x) = k$, $iter(x) = i$, $level(y) = k$. Låt x vara en nod så att $rank[x] > 0$ och x har en annan nod y efter sig som inte är en rot. $level(y) = level(x)$ precis innan operationen. Alla förutom högst $\alpha(n) + 2$ noder uppfyller dessa krav på x. De som inte uppfyller dem är första noden(om den har rank 0), sista noden(roten) och sista noden ω for ilken $level(\omega) = k$, för varje $k = 0, 1, 2, ..., \alpha(n) - 1$. Låt oss finna en sådan nod x och visa att x's potitial minskar med minst 1.

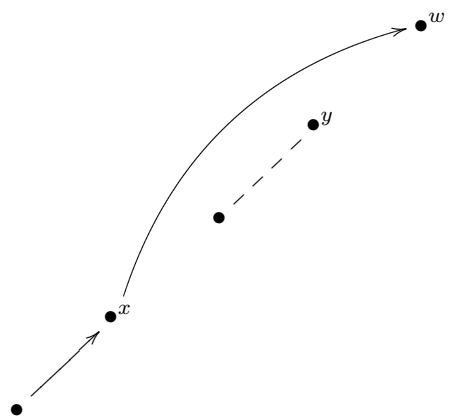
Låt $r = rank[p[y]]$. Vi har då även att:

$$\begin{aligned} rank[y] &\geq rank[p[x]] \geq A_k^i(rank[x]) \\ r = rank[p[y]] &\geq A_k(rank[y]) \\ r &\geq A_k(rank[y]) \geq A_k(A_k^{(i)}(rank[x])) = A_k^{(i+1)}(rank[x]) \end{aligned}$$

Efter operationen får vi (Se Figur 6):

$$rank[p[x]] \quad \geq \quad r \geq A_k^{(i+1)}(rank[x])$$

Eftersom operationen gör att både x och y får samma förälder, vet vi att efter operationen gäller $rank[p[x]] = rank[p[y]]$ och att operationen inte minskar $rank[p[y]]$. Då $rank[x]$ inte ändras får vi $rank[p[x]] \geq A_k^{(i+1)}$. Operationen kommer antingen att få $iter(x)$ att öka (till minst $i+1$) eller $level(x)$ att öka (vilket inträffar om $iter(x)$ ökar till minst $rank[x] + 1$). Oavsett vilket så kommer $\varphi(x)$ att minska med minst 1 med hjälp av $\varphi_i(x) \leq \varphi_{i-1}(x) - 1$. Den amorterade kostnaden av operationen är den aktuella kostnaden plus skillnaden i potential. Den aktuella kostnaden är s och vi har visat att den totala potentialen minskar med $s - (\alpha(n)) + 2)$. Den amorterade kostnaden är därför som mest $s - (s - (\alpha(n) + 2)) = s - s + \alpha(n) + 2 = \alpha(n)$.



Figur 6: Situation under körning av Find-operationen, w blir ny förälder till x