Algoritmer och Komplexitet ht 08. Övning 5

Flöden. Reduktioner

Förändrat flöde

a) Beskriv en effektiv algoritm som hittar ett nytt maximalt flöde om kapaciteten längs en viss kant *ökar* med en enhet.

   Algoritmens tidskomplexitet ska vara linjär, alltså $O(|V| + |E|)$.

b) Beskriv en effektiv algoritm som hittar ett nytt maximalt flöde om kapaciteten längs en viss kant *minskar* med en enhet.

   Algoritmens tidskomplexitet ska vara linjär, alltså $O(|V| + |E|)$.

Snabb lådpackning *Lådpackningsproblemet* (*the bin packing problem* på engelska) är följande problem. Du får n stycken metallprylar som var och en väger mellan 0 och 1 kg. Du får också ett antal stora men sköra lådor. Målet är att hitta det minsta antal lådor som behövs för att förvara alla n metallprylarna utan att någon låda innehåller mer än 1 kg.

Detta är ett känt problem som är svårt att lösa exakt (det är ett så kallat *NP-fullständigt* problem). Därför ska vi nu nöja oss med att hitta en lösning som inte är optimal med hjälp av följande enkla algoritm:

Anta att både metallobjekten och lådorna är numrerade från 1 till n. Ta en metallpryl i taget (i tur och ordning) och stoppa den i den första låda som rymmer den (dvs den låda med lägst nummer som inte blir för tung om man stoppar i prylen).

Din uppgift är att beskriva hur denna algoritm kan implementeras så att den går i tid $O(n \log n)$ (i värsta fallet och med enhetskostnad). För att uppnå detta kommer du att behöva konstruera en heapliknande datastruktur i vilken du snabbt kan söka upp den första låda som rymmer den aktuella prylen.

Reduktion som ger negativt resultat I förra uppgiften beskrevs en algoritm som hittar en approximativ lösning till lådpackningsproblemet. Algoritmen fungerar så att den placerar varje pryl i den första låda som rymmer den. Uppgiften på övning 4 var att implementera algoritmen i tid $O(n \log n)$. Visa att $\Omega(n \log n)$ är en undre gräns för algoritmens tidskomplexitet.

Reduktion som ger positivt resultat Ett ofta användbart sätt att lösa problem på är att hitta en reduktion till ett problem som man redan vet hur man ska lösa. Du ska nu använda denna metod för att lösa kantsammanhängandegradsproblemet som definieras på följande sätt.

> INMATNING: En sammanhängande oriktad graf $G = (V, E)$ och ett positivt heltal K mellan 1 och $|V|$.
>
> PROBLEM: Är det tillräckligt att ta bort K kanter från grafen G för att göra den osammanhängande (dvs uppdelad i flera sammanhängande komponenter)?

Reduktioner mellan besluts-, optimerings- och konstruktionsproblem Anta att algoritmen GraphColouring(G,k) på tid $T(n)$ (där n är antalet hörn i G) svarar 1 omm hörnen i G kan färgas med k färger utan att någon kant har likfärgade ändpunkter.

a) Konstruera en algoritm som givet en graf G med n hörn bestämmer det minimala antalet färger som behövs för att färga G. Tiden ska vara $O(\log n \cdot T(n))$.

b) Konstruera en algoritm som givet en graf G med n hörn färgar hörnen med minimalt antal färger i tid $O(P(n)T(n))$ där $P(n)$ är ett polynom.

Lösningar

Lösning till Förändrat flöde

a) Anta att kanten från u till v får sin kapacitet ökad med ett. Utgå från det tidigare maximala flödet Φ och gör bara en ny iteration i Ford-Fulkersons algoritm med den ändrade grafen: I restflödesgrafen ökas kapaciteten i kanten (u, v) med ett. Gör en grafsökning (i tid $O(|V| + |E|)$) för att se om det nu finns någon stig i restflödesgrafen längs vilken flödet kan öka. Om det finns det måste det vara ett flöde av storleken 1 (eftersom alla flöden är heltalsflöden). Om det inte finns något flöde i restflödesgrafen är Φ fortfarande det maximala flödet.

b) Anta att kanten från u till v får sin kapacitet minskad med ett. Om det tidigare maximala flödet Φ inte utnyttjade hela kapaciteten i (u, v) så förändras inte flödet alls. I annat fall måste vi uppdatera flödet på följande sätt:

   Eftersom det kommer in en enhet större flöde till u än som går ut och det kommer in en enhet mindre flöde till v än det går ut så måste vi försöka leda en enhets flöde från u till v någon annan väg. Sök alltså i restflödesgrafen efter en stig från u till v längs vilken flödet kan öka med ett. Detta görs med en grafsökning i tid $O(|V| + |E|)$. Om det finns en sån stig uppdaterar vi Φ med det flödet.

   Om det inte finns en sån stig måste vi minska flödet från s till u och från v till t med en enhet. Det gör vi genom att hitta en stig från u till s i restflödesgrafen längs vilken flödet kan öka med ett och en stig från t till v i restflödesgrafen längs vilken flödet kan öka med ett. (Det måste finnas såna stigar eftersom vi hade ett flöde från s till t via (u, v).) Uppdatera sedan Φ med dessa två flöden.

□

Lösning till Snabb lådpackning

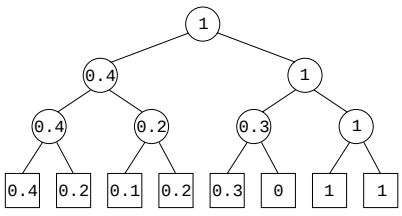
Eftersom n prylar ska stoppas ner i lådorna i tid $O(n \log n)$ så söker vi en metod som letar reda på i vilken låda en pryl ska stoppas ner i tid $O(\log n)$. Eftersom antalet lådor kan vara upp till n så måste sökalgoritmen i varje steg förkasta ungefär hälften av lådorna. I så fall har man efter $\log n$ steg förkastat alla lådor utom en, och då vet man i vilken låda prylen ska ligga.

Det finns bara två kriterier efter vilka man kan förkasta lådor:

1. om en låda inte rymmer prylen,
2. om en låda har högre ordningsnummer än den första låda som rymmer prylen.

För att kunna förkasta hälften av lådorna med en sökning så måste vi hålla reda på hur tung den tyngsta prylen får vara som kan stoppas ner i den första hälften av lådorna respektive i den andra hälften av lådorna. Detta måste vi hålla reda på rekursivt för varje hälft.

Datastrukturen blir alltså ett fullständigt binärt träd i $\log n$ nivåer, där trädets löv utgörs av lådorna. I varje löv lagrar vi hur mycket motsvarande låda rymmer (inledningsvis 1). I varje inre nod i trädet lagrar vi det största av sönernas värden. Datastrukturen ser nu ut som en heap med det största värdet överst. Exempel med åtta lådor som är fyllda med 0.6, 0.8, 0.9, 0.8, 0.6, 1, 0 och 0 kg:



Algoritmen som stoppar ner en pryl med vikt x i den första låda som rymmer den blir nu:

```
void FindBin(double x, int i)
{ if (i >= n)                    /* Är detta ett löv (dvs en låda)? */
    H[i] = H[i] - x;
  else {
    if (H[2*i] >= x)             /* Rymmer vänstra sonen x? */
      FindBin(x, 2*i);           /* Ja, fortsätt i vänster delträd. */
    else
      FindBin(x, 2*i+1);         /* Nej, fortsätt i höger delträd. */
    H[i] = max(H[2*i],H[2*i+1]); /* Uppdatera aktuell nod. */
  }
}
```

Man anropar proceduren med `FindBin(x,1)`. □

Lösning till Reduktion som ger negativt resultat

Som vanligt för undre gränser som är $\Omega(n \log n)$ så konstruerar vi en reduktion av problemet att sortera n tal till det aktuella problemet. Vi vet att det är omöjligt att med hjälp av jämförelser sortera n tal snabbare än $\Omega(n \log n)$. Detta gäller även om dom n talen är heltalen 1 till n permuterade, och även om vi tillåter oss att göra en inledande linjär omskalning av talen. Låt oss visa hur vi med hjälp av lådpackningsalgoritmen kan sortera dessa tal.

Idé: Vi skalar om talen som ska sorteras med faktorn $1/(2n)$ så att dom ligger mellan $1/(2n)$ och $1/2$. Sedan konstruerar vi hjälpprylar som ska fylla lådorna så att man sedan får plats med exakt talen $1/(2n)$ till $1/2$ (i ordning från låda 1 till låda n). Om man först stoppar i dessa hjälpprylar och därefter dom omskalade talen enligt algoritmen så kommer talen att bli sorterade.

Anta att $v[1..n]$ är posterna som ska sorteras och att nyckelfältet heter *key*. Anta vidare att algoritmen för varje låda returnerar en lista med indexen för dom prylar som den innehåller.

Sort($v[1..n]$) =
  $p \leftarrow 1/(2n)$
  for $i \leftarrow 1$ **to** n **do** $x[i] \leftarrow 1 - i \cdot p$
  for $i \leftarrow n+1$ **to** $2n$ **do** $x[i] \leftarrow v[i-n] \cdot p$
  $L[1..n] \leftarrow$FirstFit($x[1..2n]$)
  for $i \leftarrow 1$ **to** n **do** $res[i] \leftarrow v[L[i][2]-n]$ // ta andra prylen ur varje låda
  return $res[1..n]$

Funktionen Sort reducerar alltså sorteringsproblemet till FirstFit. Reduktionen (oräknat anropet till FirstFit) tar tid $O(n)$, så om man kunde implementera FirstFit så den tar tid mindre än $\Omega(n \log n)$ så skulle det gå att sortera n tal snabbare än $\Omega(n \log n)$, vilket är omöjligt. □

Lösning till Reduktion som ger positivt resultat

Först bör vi försöka förstå problemet. Anta att X är en minimal kantmängd vars borttagande gör G osammanhängande. (Detta ska tolkas som att ingen strikt delmängd av X uppfyller detsamma.) Då gäller att G uppdelas i två komponenter, och alla kanterna i X går mellan dessa två. (Annars skulle ju X inte vara minimalt.) X svarar alltså mot ett snitt i grafen (en uppdelning av hörnmängden i två delar). Antalet kanter i X kan sägas vara snittets storlek. Detta innebär att det minsta antalet kanter som vi måste ta bort för att G ska bli osammanhängande — kalla detta $\lambda(G)$ — är lika med storleken på ett minsta snitt (V_1, V_2) i G.

Minimalt snitt är ju samma som maximalt flöde. Vi ska därför försöka reducera kantsammanhängandegradsproblemet till maximalt flöde mellan två hörn s och t i en graf. Om vi utökar G till en flödesgraf G' genom att ge varje kant dubbelriktad kapacitet 1, så kan vi alltså finna $\lambda(G)$ genom att beräkna maxflödet från s till t för olika s och t. Vi behöver dock inte variera båda dessa. Om vi väljer s godtyckligt så måste det höra till någon av mängderna V_1 och V_2 ovan. Genom att variera t över alla hörn utöver s kommer så vi garanterat att träffa något hörn i den andra

mängden. Slutsatsen blir alltså att för ett godtyckligt hörn s gäller

$$\lambda(G) = \min_{t \in V - \{s\}} \{\text{MaxFlow}(G', s, t)\}.$$

Om vi ska vara noggranna så var nu uppgiften att avgöra om $K \geq \lambda(G)$. Vi kan alltså svara NEJ om $K < \text{MaxFlow}(G', s, t)$ för alla $t \in V - \{s\}$ och JA annars.

Flödesalgoritmen anropas $|V| - 1$ gånger. Varje flödesberäkning kan göras i tid $O(|V|^3)$ (vilket man inte behöver kunna utantill). Alltså blir komplexiteten för vår algoritm $O(|V|^4)$. □

Lösning till Reduktioner mellan besluts-, optimerings- och konstruktionsproblem

a) Vi vet att antalet färger är mellan 1 och n. Använd binärsökning i detta intervall för att med hjälp av algoritmen GraphColouring hitta ett k så att GraphColouring$(G, k) = 1$ och GraphColouring$(G, k-1) = 0$. Detta innebär nämligen att minimala färgningen har k färger. Det behövs $\log n$ halveringar av n för att komma ner till 1. Tidskomplexiteten blir därför $O(\log n \cdot T(n))$.

b) Hitta först det minimala antalet färger k med metoden ovan. Vi vill färga hörnen i G med färger med nummer 1 till k. Följande algoritm gör det:

```
CreateColouring(G = (V, E), k)=
u ← första hörnet i V
C ← {u}; u.colour ← k
foreach v ∈ V − {u} do
    if (u, v) ∉ E then
        if GraphColouring((V, E ∪ {(u, v)}), k) = 1 then E ← E ∪ {(u, v)}
        else C ← C ∪ {v}; v.colour ← k
if k > 0 then CreateColouring((V − C, E), k − 1)
```

GraphColouring anropas högst en gång för varje par av hörn i grafen. Tidskomplexiteten för hela algoritmen blir därför $O(\log n \cdot T(n) + n^2 \cdot T(n)) = O(n^2 \cdot T(n))$. □
