Mästarprov 1 i Algoritmer, Datastrukturer och Komplexitet

Martin Hwasser

6 mars 2009

1 Problem 1: SL-Upplysningen

1.1 Problemanalys

För att lösa problemet kommer vi använda oss av en grafstruktur med hållplatser som hörn, och deras tid i minuter från t som kantvikt. Funktionen $time(s, h)$ ger är tidsavståndet från s till h, och $min(Q)$ returnerar det hörn i Q som är märkt med den kortaste tiden, och Q implementeras då lämpligen med en prioritetskö. Funktionen $databas(h, t)$ tar en station h och tiden t och returnerar alla platser vi kan åka till ifrån h samt tiden efter t som vi kommer fram. Vi använder oss av Dijkstras algoritm för att hitta den kortaste stigen.

1.2 Algoritmbeskrivning

Databas:

Funktion: *database(h,t)*

Indata:

En hållplats h, och en tidpunkt t.

Utdata:

En lista med alla hållplatser vi kan åka till ifrån h och tiden efter t som vi kommer fram.

SL-Algoritm:

Indata:

Starthållplats s.

Sökt hållplats d.

Tiden t.

En tom mängd M.

En tom prioritetskö Q.

Utdata:

Antal minuter efter t som man kommer fram till destinationshållplatsen d.

Algorithm 1 : SL-algoritmen

```
s ← 0
M ← {s}
t ← tiden
Q.add(databas(s, t))
while d ∉ M do {Så länge vi inte har hittat d}
   h ← min(Q) {Låt h vara minsta elementet i Q}
   Q.remove(h) {Ta bort h från Q}
   M.add(h) {Lägg till h i M}
   for all hållplatser h ∈ databas(h, t) do
      if h redan har blivit märkt then
         h ← ∞ {Ty, det finns en kortare väg hit}
      else
         h ← time(s, h) + 1 {Vi har gjort transportbyte, märk h med tiden efter t + 1}
         Q.add(h)
      end if
   end for
end while
return time(s, d) − 1
```

1.3 Algoritmanalys

SL-algoritmen kommer alltid hitta den kortaste vägen från s till d, ty varje stig vi väljer är alltid den kortaste från startnoden, och således kommer vi ha valt den kortaste totala stigen när vi hittar d. I värsta fall kommer $for\ all$-loopen köras $n-1$ gånger, dvs när en nod är granne med alla andra noder. Vi behöver också räkna med sorteringen i vår prioritetskö när vi lägger till element i varje $for\ all$-loop, något som sker på $log(n)$ operationer. Den yttre $while$-loopen körs i värsta fall $n-1$ gånger, vilket sker då d är den sista noden vi hittar. Tidskomplexiteten blir sålunda $O(n \cdot n \cdot log(n)) = O(n^2 \cdot log(n))$.

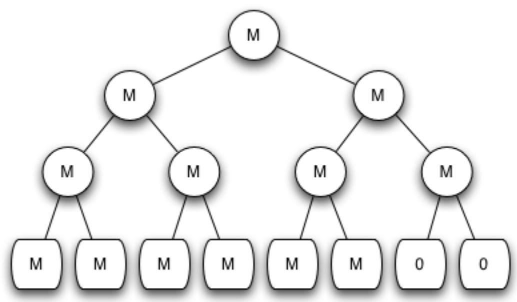
2 Problem 2: Energisnålt garage för tåg

2.1 Problemanalys

Vi börjar med att konstatera att för varje tåg i med tåglängd t_i, gäller att $1 \leq t_i \leq n$. Således krävs för varje tåg i att garagesidan $max(t_i) \leq M \leq n$, ty M måste vara minst lika stort som det längsta tåget ($max(t_i)$), och vi får alltid plats med alla tåg så länge M är lika stort som n. För att hitta det optimala M kan vi utföra intervallhalvering, och ansätter först $M = ((n - max(t_i))/2) + max(t_i)$.

2.2 Algoritmanalys

Intervallhalvering kommer ha tidskomplexiteten $O(log(n))$ eftersom vi hela tiden förkastar hälften av alla möjliga M. Nu behöver vi alltså en algoritm som letar reda på vilken perrong tåget i ska köras in på i $O(n \cdot log(n))$ för att uppfylla kravet på den totala tidskomplexiteten $O(n \cdot log^2(n))$. Vi gör detta med en heapliknande trädstruktur. Vår datastruktur kommer bli ett binärt träd med $log(n)$ nivåer, där löven representerar utrymmet som är kvar i de tillgängliga perrongerna i M (överflödiga perronger nollas enligt bild nedan). Noderna i vårt träd är inledningsvis längden på M. Således har vi alltså följande träd om t.ex. $n = 6$:



Poängen är sedan att parkera tåg i i ett löv så långt till vänster som möjligt där tåget fortfarande får plats, och subtrahera lövets värde med tåglängden t_i. Vi vill låta de vänstra noderna vara mindre än de högra, för på så vis förkastar vi vid varje operation hälften av alla perronger. Vi ska placera alla n tåg i trädet, vilket för varje tåg nu går på $log(n)$, och med intervallhalveringen får vi alltså tidskomplexiteten $O(log(n) \cdot n \cdot log(n)) = O(n \cdot log^2(n))$ som vi eftersträvat.

2.3 Algoritmbeskrivning

Vi låter $min \leftarrow max(t_i)$ (längsta tåget) och $max \leftarrow n$ (antalet tåg). Funktionen $generateTree(M)$ genererar ett nytt träd med uppdaterat M. Att parkera tågen i trädet görs sedan enligt Algorithm (2), som anropas med $findMinM(root, t_1, (max - min)/2 + min)$.

Algorithm 2 : **findMinM(**$node, t_i, M$**):**

if $node < t_i$ **then** {Är tåg i är längre än M?}
  if $(max - M) <= 1$ **then** {Vi har hittat minsta M}
    return max
  end if
  $min \leftarrow M$ {Uppdatera min}
  $M \leftarrow ((max - min)/2) + min$. {Öka M med intervallhalvering}
  $generateTree(M)$ {Generera nytt träd med större M}
  $findMinM(root, t_1, M)$ {Upprepa processen i nya trädet}
  break
else
  if $node$ is a leaf **then**
    $node \leftarrow node - t_i$ {Dra av tåglängden från perrongen}
    if $i = n$ **then** {Har det sista tåget parkerat?}
      $max \leftarrow M$ {Uppdatera max}
      $M \leftarrow (max - min)/2 + min$ {Minska M med intervallhalvering}
      $generateTree(M)$ {Generera nytt träd med mindre M}
      $findMinM(root, t_1, M)$ {Upprepa processen i nya trädet}
      break
    end if
    $findMinM(root, t_{i+1}, M)$
  else
    if $t_i \leq node.left$ **then**
      $findMinM(node.left, t_i, M)$
    else
      $findMinM(node.right, t_i, M)$
    end if
  end if
  $node = max(node.left, node.right)$ {Uppdatera så vi vet hur mycket barnen rymmer}
  if $node.left > node.right$ **then** {Balansera så att vänstra noden alltid är minst}
    $tmp \leftarrow node.left$
    $node.left \leftarrow node.right$
    $node.right \leftarrow tmp$
  end if
end if
return max {Vi har hittat rätt eftersom max var senast fungerande M}

3 Problem 3: Hållbart fiske

3.1 Problemanalys

Vi börjar med att konstatera att om $n \leq k$, så använder vi en biljett på varje fiskeplats. Mer intressant är problemet då $n > k$, i vilket fall vi konstruerar en matris k_{ij}, där $1 \leq i \leq n$, och $1 \leq j \leq k$, och löser problemet med dynamisk programmering.

Vi angriper problemet bakifrån, och adderar de olika fiskekvoterna med hjälp av Algorithm (3). Tanken är att vi summerar fiskekvoter i matrisen utifrån hur vi får stega i den. Hur vi får stega bestäms av att vi åker mellan fiskeplatserna 1 till n i tur och ordning, och vi skall också använda fiskebiljetterna i tur och ordning samt ej mer än en gång på varje fiskeplats. Dvs, har vi fiskat på $k_{i,j}$, får vi inte fiska vid något fiskeplats $\leq i$ och inte använda en fiskebiljett $\leq j$. Vi måste alltså stega snett nedåt i matrisen varje gång vi använt en fiskebiljett på en fiskeplats.

Algorithm 3 : Fiskalgoritmen

for $j = k - 1$ to 1 **do**
  for $i = n - 1$ to j **do**
    if $k_{i,j} + k_{i+1,j+1} > k_{i+1,j}$ **then**
      $k_{i,j} \leftarrow k_{i,j} + k_{i+1,j+1}$
    else
      $k_{i,j} \leftarrow k_{i+1,j}$
    end if
  end for
end for

3.2 Algoritmanalys

Efter detta blir det trivialt att hitta den optimala rutten, eftersom vi summerat fiskekvoterna. Allt vi behöver göra är att leta upp det största värdet i första kolumnen. Om max finns på t.ex. $k_{i,j}$ lägger vi till den fiskeplatsen till en mängd M, och letar sedan reda på max i $k_{i+1 \to n, j+1}$ osv, så länge $i + 1 \leq n$ och $j + 1 \leq k$, och lägger till M. Tidskomplexiteten blir $O(k \cdot n) + O(k \cdot n) = O(kn)$, ty för varje fiskebiljett $1 \leq j \leq k$ går vi igenom $j \leq i \leq n$ fiskeplatser (det blir en vänstertriangulär matris eftersom vi inte kan använda en biljett j som är större än fiskeplats i). Operationerna utförs både vid adderingen samt vid sökandet efter max.