DD1352 Algoritmer, datastrukturer och komplexitet

Övning 8

Anton Grensjö
grensjo@csc.kth.se

5 november 2014

Dagordning

- Första timmen
  - Lösningar till Mästarprov 1
- Andra timmen
  - Oavgörbarhet

Jobbig labyrint

Algoritmen

- Grafproblem. Vi ska hitta den kortaste vägen från punkt A till punkt B genom en labyrint.
- Grafrepresentation:
  - Låt varje ruta i labyrinten motsvara en nod.
  - Bilda kanter mellan varje nod och dess närliggande rutor.
- Använd Dijkstras algoritm. Behövs modifikationer:
  - Vikterna är i noderna istället för på kanterna.
  - Måste ta hänsyn till diken. Dessa kostar 100 enheter extra, samt kan endast hoppas över om man har total ansträngning strikt mindre än 1000 *före* hoppet.
- Se Viggos pseudokod.

Jobbig labyrint

Tidskomplexitet

- Antalet diken är mindre än fyra gånger antalet hörn, så $|E| \in \mathcal{O}(|V|)$.
- Betrakta de olika delarna av koden
  - Konstruerar grafen på $\mathcal{O}(|V|)$.
  - Bygger upp en heap på tiden $|V| \log |V|$.
  - Kör while-slingan maximalt $|V|$ gånger, varje varv tar tid $\mathcal{O}(\log |V|)$.
- Sammantaget får vi tidskomplexiteten $\mathcal{O}(|V| \log |V|) = \mathcal{O}(n^2 \log n^2)$. Kan förenklas. Hur?
- Kom ihåg logaritmlagen

$$\log a^b = b \log a$$

Denna ger $n^2 \log n^2 = 2n^2 \log n$, så komplexiteten kan också skrivas $\mathcal{O}(n^2 \log n)$.

Lådor i lådor

Generellt

- En låda $L_1 = (l_1, h_1, d_1)$ får plats i en låda $L_2 = (l_2, h_2, d_2)$ om det finns en rotation (vältning) av L_1 så att L_1 är strikt mindre än L_2 på varje ledd.
- Hur tar vi reda på om L_1 får plats i L_2?
  - Ett sätt: Betrakta alla permutationer av (l_1, h_1, d_1) och jämför varje permutation komponentvis med den andra lådan (viktigt komma ihåg alla 6 fall).
  - Enklare: Sortera alla längd-tuplar inbördes och jämför.
- Vi ska givet en mängd lådor och deras mått ta reda på den längsta kedjan av lådor som får plats i varandra.
- Vi kommer betrakta flera olika sätt att lösa problemet på, och jämföra dem med varandra.

Lådor i lådor

Lösning 1

- Konstruera en riktad graf med lådorna som noder. Dra en kant från en låda L_1 till en låda L_2 omm låda L_1 får plats i låda L_2.
- Grafen måste vara acyklisk. Varför?
- Vi söker längsta vägen i grafen. Denna kan vi hitta t.ex. med hjälp av Floyd-Warshall. Hur?
- Vi låter alla kanter ha vikten -1 och söker istället den kortaste vägen. Floyd-Warshall klarar av grafer med negativa kantvikter, så länge negativa cykler saknas.
- Tidskomplexitet: $\mathcal{O}(n^3)$.

Lådor i lådor

Lösning 2

- Bilda på samma sätt som tidigare en DAG av lådorna, med kanter som visar vilken låda som får plats i vilken. Vi vill hitta längsta vägen i denna graf.
- Hur kan vi göra det snabbare än $\mathcal{O}(n^3)$?
- Idé: Kan vi inte bara traversera grafen rekursivt?
- Finns det något problem med följande algoritm?

```
1: Anropa denna funktion för varje nod, och välj det största värdet.
2: function LongestPath(node)
3:     best ← 1
4:     for v ∈ node.neighbours do
5:         best ← max(best, LongestPath(v) + 1)
6:     return best
```

- Betrakta till exempel fallet då alla lådor ingår i den längsta kedjan. Vi ser att tidskomplexiteten blir exponentiell. Hur löser vi detta?

Lådor i lådor

Lösning 2

- Vi beräknar samma sak väldigt många gånger.
- Problemet kan brytas ned i överlappande delproblem. Låter som ett jobb för dynamisk programmering!
- Lösning med memoisering:

```
1: dp ← Array(n)                                   ▷ Initialisera till 0
2: Anropa denna funktion för varje nod, och välj det största värdet.
3: function LongestPath(node)
4:     if dp[node] > 0 then return dp[node]
5:     best ← 1
6:     for v ∈ node.neighbours do
7:         best ← max(best, LongestPath(v) + 1)
8:     dp[node] ← best
9:     return best
```

- Tidskomplexitet: $\mathcal{O}(n^2)$

Lådor i lådor

Lösning 3

- Memoisering har dock dålig minneslokalitet och en del overhead. Hur kan vi lösa detta med vanlig bottom-up-DP?
- Vi behöver en beräkningsordning.
- Hitta en topologisk sortering! En ordning av lådorna sådan att om låda A får plats i låda B så ligger låda A före låda B i listan.
  - Vi kan konstruera grafen och använda en sedvanlig algoritm för topologisk sortering.
  - Sortera lådorna med avseende på volym.
  - Sortera lådorna med avseende på deras minsta mått.
- Se Viggos pseudokod.

Tung delmatris

- Låt $T[i, j]$ vara antalet tunga tal i matrisen $A[1..i, 1..j]$.
- Antalet tunga tal i $n \times n$-submatrisen med sitt nedre högra hörn i (p, q) är då (inklusion-exklusion)

$$T[p, q] - T[p - n, q] - T[p, q - n] + T[p - n, q - n]$$

- Vi kan beräkna $T[i, j]$ med hjälp av dynamisk programmering.

$$T[i, j] = \begin{cases} 0 & \text{om } i = 0 \text{ eller } j = 0 \\ T[i - 1, j] + \text{ antalet tunga element i } T[i, 1..j] & \text{annars.} \end{cases}$$

- Se Viggos pseudokod. Tidskomplexitet: $\mathcal{O}(n^2)$. Undre gräns: $\Omega(n^2)$, ty vi måste titta på alla tal.

Oavgörbarhet

Lite repetition

- Ett problem som har en algoritm som för alla instanser kan hitta en lösning i ändlig tid kallas *avgörbart*.
- Ett problem som inte kan lösas i ändlig tid av någon algoritm kallas *oavgörbart*.
- För beslutsproblem talar man om avgörbarhet. För övriga problem talar man istället om beräkningsbarhet (definieras analogt).
- Stopproblemet är oavgörbart. (Givet ett program P och indata X, kommer P någonsin att terminera om det startas med X?)
- Motsägelsebevis: För att bevisa ett påstående p, antag $\neg p$ och härled motsägelse. Då måste p vara sant.
  - Exempel. Vi vill visa P oavgörbart. Antag att P är avgörbart och reducera stopproblemet till P. Antagandet att P var avgörbart måste vara falskt.

Problem 1

Problemet *orm i kakel* tar som indata en mängd T av kakelplattstyper och två punkter p_1 och p_2 i planet. Frågan är om det går att användande endast kakeltyper i T kakla en orm som ringlar sig från p_1 till p_2, som inte bryter kakelmönstret någonstans och som hela tiden håller sig i det övre halvplanet.
Detta problem är oavgörbart.
Är följande varianter av ormproblemet avgörbara eller oavgörbara?

a) Indata utvidgas med en fjärde parameter, en kakeltyp $t \in T$. Första kakelplattan (den som täcker p_1) måste vara av typ t.

Antag att denna variant är avgörbar. Då kan vi lösa det ursprungliga ormproblemet genom att göra ett anrop till detta för varje kakeltyp. Motsägelse.
Svar: Oavgörbart.

Problem 1

b) Indata utvidgas med ett tal N och frågan utvidgas med bivillkoret att ormen måste bestå av *högst* N plattor.

Avgörbart, ty vi kan nu lösa problemet med totalsökning, då det bara finns ett ändligt antal möjliga ormar.

c) Indata utvidgas med ett tal N och frågan utvidgas med bivillkoret att ormen måste bestå av *åtminstone* N plattor.

Oavgörbart. *Bevis:*
Antag att denna variant är avgörbar. Då kan vi lösa det ursprungliga problemet genom att anropa denna variant med $N = 1$. Men det ursprungliga problemet är oavgörbart, så det är en motsägelse. Således kan inte denna variant vara avgörbar. □

Problem 2

Fråga: Finns det något explicit program P så att det givet y är avgörbart huruvida P stannar på indata y?

Betrakta följande program:

```
1: function P(y)
2:     return
```

Uppenbarligen så finns det massor av sådana program.

Problem 3

Fråga: Finns det något explicit program P så att det givet y är oavgörbart huruvida P stannar på indata y?

Låt P vara en interpretator och indata y vara ett program x följt av indata y' till det programmet.

$P(y)$ beter sig alltså precis som programmet x skulle göra på indata y', och det är ju oavgörbart huruvida ett program x stannar på ett visst indata y'. Svaret är alltså ja.

Problem 4

Fråga: Om programmet x stannar på tomt indata så låter vi $f(x)$ vara antalet steg innan det stannar. Annars sätter vi $f(x) = 0$. Definiera nu $MR(y)$, den maximala körtiden över alla program vars binära kodning är mindre än y.

$$MR(y) = \max_{x \leq y} f(x)$$

Är MR beräkningsbar?

Nej. Vi vet att det är oavgörbart huruvida ett program stannar på blankt indata. Reducera detta problem till $MR(y)$.

- Notera att om programmet y stannar, så gör det det på högst $MR(y)$ steg.
- Simulera därför y på blankt indata i $MR(y)$ steg och kolla om det stannar. Om det inte gör det så vet vi att det aldrig stannar.

Se Viggos pseudokod.

Problem 5

Problem: Visa att funktionen MR i föregående uppgift växer snabbare än varje rekursiv (beräkningsbar) funktion.

Visa närmare bestämt att det för varje rekursiv funktion g finns ett y så att $MR(y) > g(y)$.

Ledtråd: Tänk på föregående uppgift. Använd motsägelsebevis.

- Antag motsatsen, dvs att det finns en rekursiv funktion g så att $g(y) \geq MR(y)$ för alla y.
- Betrakta nu lösningen till föregående problem. Eftersom $g(y) \geq MR(y)$ kan vi lika gärna simulera programmet i $g(y)$ steg istället för $MR(y)$ steg.
- Detta betyder alltså att programmet med den förändringen är beräkningsbart, men det vet vi att det inte kan vara. Motsägelse.

Problem 6

Fråga: Anta att en turingmaskin M, indata X (som står på bandet från början) och en heltalskonstant K är givna. Är följande problem avgörbart eller oavgörbart?

Stannar M på indata X efter att ha använt högst K rutor på bandet (en använd ruta får skrivas och läsas flera gånger)?
Detta är faktiskt avgörbart! Varför?

- Turingmaskinen måste hålla sig inom K rutor på bandet $\implies$ Turingmaskinen har endast ett ändligt antal möjliga konfigurationer (över dessa rutor).
- Om maskinen har m tillstånd, så är antalet konfigurationer $m \cdot K \cdot 3^K$.
- Simulera maskinen $m \cdot K \cdot 3^K + 1$ steg. Kontroller att den inte rör sig över fler än K rutor.
- Om den stannar inom denna tid svarar vi ja. Om den inte har stannat måste den ha återkommit till en gammal konfiguration, och vara fast i en oändlig slinga. Svara nej.

Problem 7

En rekursivt uppräknelig mängd definierades på föreläsningen som ett språk som kan kännas igen av en funktion vars ja-lösningar kan verifieras ändligt.

En alternativ definition är en mängd som är uppräknelig (elementet kan numreras med naturliga tal) och kan produceras av en algoritm.

Använd den senare definitionen och visa att varje rekursiv mängd också är rekursivt uppräknelig.

Vi vill visa att om en mängd S är rekursiv (det finns en beräkningsbar algoritm som kan avgöra vilka element som tillhör S) så är S uppräknelig och kan produceras av en algoritm.

Så, om S är en rekursiv mängd, så finns det en algoritm $A(x)$ som returnerar true omm $x \in S$. Vi antar att elementen i S lagras som binära strängar.

1: **for** $i \leftarrow 0$ **to** ∞ **do**
2: **if** A(i) **then write** i

Problem 8

Den diagonaliserade stoppmängden består av alla program p som stannar på indata p. Visa att denna mängd är rekursivt uppräknelig.

Visualisering: oändlig tabell.

1: **for** $i \leftarrow 0$ **to** ∞ **do**
2: **for** $p \leftarrow 0$ **to** i **do**
3: Simulera beräkningen $p(p)$ under i steg
4: **if** $p(p)$ stannar inom i steg **then write** p

Notera: samma p kommer skrivas ut många gånger. Enkelt att fixa. Hur?

Nästa gång

- Redovisning av teoriuppgifter för labb 4
- NP-fullständighetsbevis