DD2458 popup09
Föreläsning 1
Algoritmkonstruktionsmetoder

Mikael Gerdin
Alexander Korling
Jonas Sundberg

2009-09-03

1 Algoritmkonstruktionsmetoder

Gemensamt för alla dessa tekniker är att vi löser det stora problemet genom att dela upp och betrakta problemet i mindre delar.

1.1 Giriga lösningar

Giriga lösningar innebär att man i en given situation med fler vägar att gå alltid väljer den väg som ger en lokalt optimerad lösning. För många problem ger denna strategi en globalt optimal lösning.

En egenskap hos girigia lösningar är lösningen ofta får en linjär körtid.

Det är viktigt att kontrollera att en girig lösning faktiskt ger en globalt optimal lösning. Det finns många problem då man kan missta sig för att en girig lösning är optimal när det egentligen finns bättre lösningar.

Exempel Ett exempel på en girig lösning är algoritmen för att konstruera ett minimalt spännande träd som går ut på att man hela tiden tar den kortaste kant som förbinder två noder som inte redan är sammanbundna i samma träd.

1.2 Dekomposition

Dekomposition handlar om att dela upp problemet i flera mindre disjunkta delar.

Man kan beskriva en dekompositionslösning i tre steg:

- Dela upp problemet i mindre delproblem.
- Lös delproblemen separat. Detta görs ofta genom att tillämpa dekomposition igen till delproblemen är så små att det finns en enkel lösning på delproblemet.
- Kombinera ihop dellösningarna för att få en global lösning.

Exempel Ett exempel på en dekompositionslösning är merge sort. I merge sort delar man upp en vektor i två lika stora delar, sorterar varje del var för sig och slår sedan samman de två dellistorna till en gemensam lista.

1.3 Dynamisk programmering

Dynamisk programmering innebär att man delar upp problemet i mindre delproblem där lösningen till ett visst delproblem kan behövas för att lösa flera olika större problem. Genom att återanvända dellösningar slipper man beräkna samma dellösning flera gånger.

Vanligt inom dynamisk programmering är att man konstruerar någon form av tabell och använder tidigare resultat för att fylla i ytterligare data i tabellen.

Exempel Ett exempel på en lösning med dynamisk programmering är när man beräknar binomialkoefficienter genom att konstruera Pascals triangel. Tabellen som konstrueras är i detta fall innehållet på en viss rad i Pascals triangel. Denna tabell kan sedan uppdateras med innehållet i raden nedanför till vi når den sökta lösningen.

1.3.1 Memoisering

Memoisering är en variant av dynamisk programmering där vi sparar resultat när vi har räknat ut dem första gången. Detta innebär att man måste hålla reda på vad man har beräknat, men med fördelen att man bara behöver beräkna de delresultat som behövs för att lösa det aktuella problemet.

Den stora föredelen är att vi utför färre beräkningar om vi bara behöver en del av alla dellösningar som skulle behöva beräknas vid vanlig dynamisk programmering. Om en stor andel av alla dellösningar behövs kan det dock kräva extra beräkningskraft då man hela tiden måste hålla koll på vilka dellösningar som är beräknade.

Valet mellan vanlig dynamisk programmering och memoisering är i många fall snarare en fråga om bekvämlighet än prestanda. Vissa problem lämpar sig bättre för memoisering medan vissa lämpar sig bättre för vanlig dynamisk programmering.

2 Exempel

2.1 Kontinuerlig kappsäck

Det kontinuerliga kappsäcksproblemet är en modifiering av kappsäcksproblemet. I det vanliga kappsäcksproblemet skall föremål av en diskret storlek packas i kappsäcken, vilket gör problemet NP-fullständigt. Om man istället ska packa föremål som kan väljas kontinuerligt blir problemet enklare att lösa.

Vi har en kappsäck och målet är att packa kappsäcken så att värdet på kappsäckens innehåll är så stort som möjligt. Vi har ett antal olika typer av föremål med givna värden och en viss mängd av varje föremål. I exemplet skall en kappsäck med en kapacitet på fem liter fyllas med guldsand, lego och bollar.

- Guldsanden är värd 100 kr/l.
- Legobitarna är värda 200 kr/l.
- Bollarna är värda 50 kr/l.

Observera att vi kan välja en kontinuerlig mängd av de olika föremålstyperna och inte är bundna att välja ett diskret antal saker av en viss typ.

2.1.1 Lösning

Vår strategi är att först undersöka om vi kan finna en girig lösning. Detta gör vi lätt då vi kan ta så mycket som möjligt av föremålstypen med högst värde till dess att föremålen av den typen är slut eller kappsäcken är full. När föremålen av en viss typ är slut fortsätter vi med den kvarvarande typ som är dyrast per volym till dess att kappsäcken är fylld.

2.2 Schemaläggning

I schemaläggningsproblemet finns det en föreläsningssal och ett antal aktiviteter. En aktivitet åt gången kan förekomma i föreläsningssalen och målet är att schemalägga så många aktiviteter som möjligt under en dag.

Given indata är ett antal aktiviteter som alla har en starttid och en sluttid. Vi ska välja ut ett så stort antal aktiviteter som möjligt som kan schemaläggas i föreläsningssalen utan att två aktiviteter krockar.

2.2.1 Lösning

Vi börjar med att undersöka om det finns en girig lösning. Ett föreslag på en lösning är att börja med att schemalägga den aktivitet som slutar först. Detta följs av att man hela tiden väljer den aktivitet som slutar tidigast utan att överlappa med någon av de redan valda aktiviteterna.

Om man väljer ut den aktivitet som slutar tidigast av alla aktiviteter kan man enkelt se att man inte hade kunnat välja två aktiviteter som slutar senast så sent som den valda aktiviteten. Vi har därmed valt ut den lokalt bästa aktiviteten fram till denna aktivitets slut.

Givet ett antal valda aktiviteter och att man ska schemalägga resten av dagen är det alltid säkert att välja bort alla aktiviteter som krockar med någon vald aktivitet. Efteråt kan vi välja ut den aktivitet som slutar tidigast av de kvarvarande aktiviteterna. Detta genererar en lokalt optimal lösning fram till slutet på den senast valda aktiviteten.

Om vi fortsätter med detta för hela dagen kommer vi till slut få en lösning som är optimal från dagens start till slutet av den sista föreläsningen, vilket ger en globalt optimal lösning.

2.3 Brevlådor

Bland brevlådekontruktörer så är det välkänt att den viktigaste egenskapen hos en brevlåda är hur många smällare som man kan smälla i brevlådan samtidigt utan att brevlådan exploderar. Man vet dessutom att en brevlåda alltid går sönder om man smäller 101 smällare i brevlådan.

Givet k likadana brevlådor ska vi utvärdera hur många smällare just den modellen klarar av. Vi vill också minimera antalet smällare som behövs för att bestämma detta i ett värsta fall.

Under testningen kan man stoppa in ett antal smällare i en brevlåda och smälla av dessa. När ett test görs går brevlådan antingen sönder eller håller för smällen. En brevlåda som håller för en smäll tar ingen skada från den och kan återanvändas vid senare tester.

2.3.1 Lösning

Vi börjar med att söka efter en girig lösning. En tanke på hur en girig lösning skulle kunna se ut är en binärsökning av någon form. Det går dock lätt att ta fram motexempel då andra strategier ger bättre resultat. De giriga strategier som kan tas fram är därmed inte globalt optimala och löser inte problemet.

Vi använder oss istället av memoisering. Vi definerar ett intervall som två punkter a och b på den tänkta linjen mellan 0 och 100, där vi vet att brevlådan garanterat överlever a smällare och att den garanterat går sönder av $b + 1$ st smällare. Ursprungsproblemet kan då formuleras om till att lösa problemet med k brevlådor och intervallet $(a = 0, b = 100)$.

Vi börjar med att söka efter basfall. Till att börja med så ser man lätt att om vi bara har en brevlåda kvar så måste vi göra en linjärsökning från nedre gränsen av intervallet till den punkt då brevlådan går sönder. Om vi hade hoppat över ett antal smällare och brevlådan sedan går sönder kan vi inte bestämma det exakta antalet smällare som krävs.

Om intervallet istället bara innehåller ett element så har vi svaret utan att behöva förbruka fler smällare.

Givet ett intervall (a, b) och ett antal brevlådor k kan vi välja en punkt i i intervallet som beskriver ett möjligt antal smällare att testa. Vi kan beräkna antalet smällare som kommer behövas i de två

fallen då brevlådan antingen går sönden eller håller. Genom att använda denna strategi rekursivt finner vi förr eller senare ett av våra basfall. För att minska antalet beräkningar som krävs kan vi använda memoisering för att spara tidigare uträknade delresultat. Att med dynamisk programmering beräkna alla tillåtna kombinationer av k, a och b är i det här fallet orimligt då antalet kombinationer är alltför stort.

Det som återstår är att före varje steg testa olika val av i för att finna det antal smällare i som ger minsta antal använda smällare i det värsta fallet. Detta antal i är det antal som vi kommer vilja smälla av om vi når den givna situationen.

För att beräkna antalet smällare kan vi definera funktionen $m(k, a, b)$ som antalet smällare som går åt i värsta fallet då vi har k brevlådor och intervallet (a, b) av möjliga maxantal smällare som brevlådan kan hålla för. Vi erhåller då förljande rekursionsekvation för våra beräkningar:

$$\begin{aligned} m(1, a, b) &= \sum_{i=a}^{b} i \\ m(k, a, a) &= 0 \\ m(k, a, b) &= min_i \left(max \left[m(k-1, a, i-1), m(k, i, b) \right] + i \right) \end{aligned}$$

2.4 Pianoflyttare

I pianoflyttarproblemet ska vi schemalägga en mängd pianoflyttare som ska flytta ett antal pianon under ett visst antal dagar. Det tar två pianoflyttare en hel dag att flytta ett piano. Varje piano har dessutom en första dag då pianot då pianot kan flyttas och en sista dag då pianot ska vara flyttat från platsen. Frågan är om det möjligt att flytta alla pianon i det givna tisintervallet och om det möjligt att flytta alla pianon utan att någon pianoflyttare behöver arbeta på helgen. Den första dagen är en måndag.

2.4.1 Lösning

Vi söker en girig lösning och ser att detta är en variant av schemaläggningsproblemet. Istället för att bara ha en lokal så har vi $\lfloor p/2 \rfloor$ par av pianoflyttare som kan flytta ett piano per par och dag.

Vi använder samma strategi som för schemaläggningsproblemet och prioriterar de pianon vars slutdatum är tidigast. Varje dag väljs upp till $\lfloor p/2 \rfloor$ pianon som får flyttas den givna dagen ut, med prioritering på de som har kortast tid till den sista dagen. Om det någon dag finns fler än $\lfloor p/2 \rfloor$ pianon som måste flyttas senast den givna dagen finns det ingen giltig lösning.

Vi börjar med att utföra beräkningen utan att betrakta helgdagar för se om det finns en lösning där ingen arbetar på helgen. Om det inte går att hitta någon lösning utan att använda helgdagar kan vi utföra en ny beräkning då vi även betraktar helgdagar.