DD2458, Problemlösning och programmering under press

Föreläsning 4: Kombinatorisk sökning

Datum: 2007-09-25
Skribent(er): Martin Boij, Erik Gustafsson
Föreläsare: Mikael Goldmann

1 Bakgrund

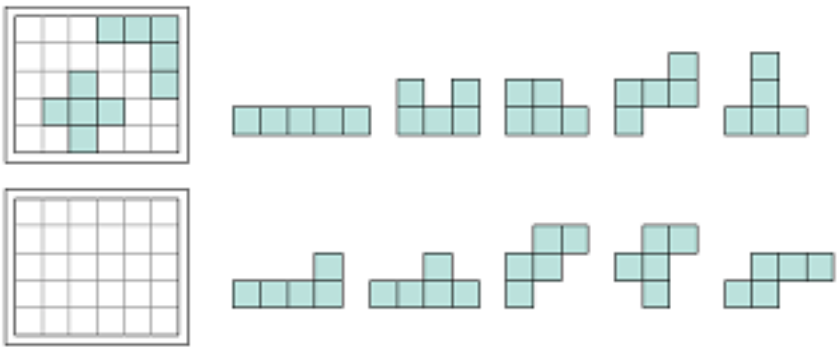
Totalsökning är en metod för kombinatoriska optimeringsproblem med vilken man alltid får en optimal lösning. Det går att representera en totalsökning grafiskt med ett så kallat sökträd. Låt en specifik följd av val kallas för ett tillstånd i totalsökningen och låt varje sådant tillstånd representeras av en nod i sökträdet. Starttillståndet, roten i sökträdet, motsvaras av den tomma följden. Låt dessutom övergångar mellan olika tillstånd, alltså de olika valmöjligheterna, motsvaras av kanter mellan sökträdets noder. Tyvärr tar det ofta orimligt lång tid att undersöka alla lösningsförslag, eftersom sökträdet lätt blir mycket stort. Lyckligtvis kan man ibland ta till kombinatorisk sökning, vilket är ett antal metoder för att beskära och förändra sökträd på olika sätt.

Värt att tänka på när man arbetar med kombinatorisk sökning är att det ofta är implementationen som avgör om ens lösning är effektiv nog, till skillnad från lösningar baserade på till exempel dynamisk programmering, där det är den valda algoritmen som spelar störst roll.

Har man verkligen någon användning av kombinatorisk sökning? Dynamisk programmering, dekomposition och giriga algoritmer är att föredra framför kombinatorisk sökning, men tyvärr går inte dessa metoder att applicera på alla problem.

1.1 Exempel: Pentatris

I pusslet Pentatris ska två rektangulära ramar fyllas i med hjälp av tolv unika pusselbitar. Pusselbitarna, som får roteras och vändas fritt, är vardera fem rutor stora och kan tillsammans precis fylla ut de 60 rutorna som rektanglarna består av. Vid en första anblick kan pusslet se trivialt ut, men efter ett par misslyckade försök inser man att pusslet är klurigt. Då den första rektangeln fyllts i, vilket ofta går smärtfritt, upptäcks i de flesta fall att resterande bitar inte går att placera ut så att den andra rektangeln blir fylld.



Rektanglarna och pusselbitarna i Pentatris

För att lösa pusslet totalsöker man genom att testa alla möjliga positioner och rotationer för varje pusselbit. Alla bitar har åtta rotationer eftersom det är tillåtet att vända på pusselbitar, men för vissa bitar kommer några av dessa att vara likadana. Det är till exempel ingen idé att rotera eller vända pusselbiten som ser ut som ett kors.

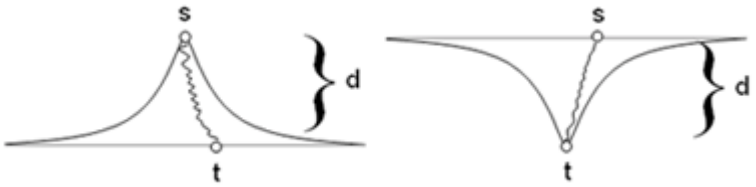
Börja med att dela upp pusselbitarna i två högar, vilket kan göras på $\binom{12}{6}$ sätt. Fixera en av pusselbitarna till en viss rektangel för att undvika att försöka lösa samma uppdelning två gånger, där enda skillnaden är att rektanglarna bytt plats. Det gäller även att en startbiten i varje rektangel bara har högst två olika relevanta rotationer, oavsett bit, då resterande kan fås genom att vrida och spegla rektangeln.

För att effektivt kunna placera ut pusselbitar kan varje rektangel representeras med en följd av 30 bitar. Låt de första sex bitarna motsvara rektangelns översta rad, nästa sex bitar rektangelns näst översta rad och så vidare. En bit satt till 1 betyder att motsvarande position i rektangeln är täckt av någon pusselbit. Låt även varje tänkbar position och rotation av en pusselbit representeras på detta sätt. Med bitoperationen & kan man då kontrollera om en pusselbit kan placeras på en position, med addition kan man lägga en pusselbit och med subtraktion kan man ta bort en pusselbit.

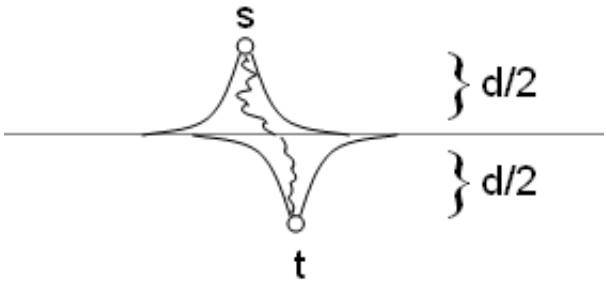
En tänkbar optimering är att i samband med utplaceringen av en pusselbit undersöka om pusselbiten gav upphov till en isolerad region vars storlek inte är delbar med fem. Om en sådan region uppstod vet man att det inte är lönt att fortsätta med försöket, eftersom det aldrig kommer gå att fylla en sådan region med enbart pusselbitar av storlek fem.

2 Meet-in-the-middle

Ett sökträd med två alternativ på varje nivå och d nivåer kommer ha 2^d tänkbara slutnoder. Utgår man från slutnoden kommer det istället att finnas 2^d möjliga startnoder. Ifall man på förhand har kännedom om startnod och slutnod, är det tänkbart att arbeta sig igenom de två sökträden till hälften, för att sedan sätta ihop delresultaten till en global lösning. Detta medför att exponenten d halveras, det vill säga sökträdets storlek reduceras till kvadratroten av den ursprungliga storleken. För att detta ska löna sig markant krävs dock att det går att kombinera delresultaten på ett effektivt sätt. Denna metod kallas, intuitivt nog, för *meet-in-the-middle*.



Sökträd för sökning från startnod samt sökträd för sökning från slutnod



Sökträd för meet-in-the-middle

2.1 Exempel: Maximalt stöldgods

En tjuv vill maximera värdet av sakerna han stjäl, givet att han inte kan bära hur mycket som helst. Vilka saker ska han stjäla? Varje sak i har ett värde v_i och en vikt c_i, och tjuven kan bara bära saker med sammanlagd vikt C eller mindre. Dessutom gäller olikheten $n + 2 \cdot \log_2 V \leq 75$, där n är antalet saker och V deras sammanlagda värde.

Detta är inget annat än kappsäcksproblemet, vilket är ett NP-fullständigt problem. Om det finns många saker att stjäla blir totalsökning för långsam, och om sakerna har höga värden kommer dynamisk programmering inte att vara effektivt, eftersom det inte finns någon övre gräns på vad kapaciteten C kan vara i värsta fall. Det extra villkoret säger dock att om det finns många saker kommer dessa ha låga värden, medan om det enbart finns ett fåtal saker kommer dessa i sin tur ha höga värden. Tidskomplexiteten för en lösning baserad på dynamisk programmering är $O(n \cdot V)$. Används istället totalsökning blir tidskomplexiteten $O(n \cdot 2^n)$. Värsta tänkbara indata kommer uppfylla $V = 2^n$, vilket i sin tur innebär att $n \leq 25$. Då $n = 25$ tar det alltså lite väl lång tid för en totalsökning eller en lösning baserad på dynamisk programmering (det blir i storleksordningen 10^9 operationer). Men som tur är kan meet-in-the-middle tillämpas för att snabba upp totalsökningen.

Problemet löses i två etapper. Först bestämmer man vad man ska göra med de $n/2$ första elementen, vilket ger $2^{n/2}$ möjligheter. Därefter bestäms för resterande hälften saker huruvida de ska ingå i stöldgodset eller inte, vilket också ger $2^{n/2}$ möjligheter. Sedan gäller det att para ihop de två delresultaten till ett globalt maximum, men det gäller att se upp! Försöker man para ihop alla möjligheter från första etappen med alla möjligheter från andra etappen fås åter igen 2^n möjligheter.

En bättre plan är att sortera delresultaten i den första etappen efter stigande vikt. Alla delresultat (v_i, w_i) som väger mer men är värt mindre än något annat delresultat (v_j, w_j), alltså $w_i \geq w_j, v_i \leq v_j$, tas bort eftersom dessa aldrig kan ingå i en optimal lösning. För varje möjlighet i den andra etappen binärsöker man sedan bland de kvarvarande möjligheterna från första etappen och sparar undan den hittills bästa kombinationen. Till slut fås en optimal lösning. Tidskomplexiteten för denna metod blir $O(n \cdot 2^{n/2})$.

Med en ny tidskomplexitet för totalsökningen, kommer värsta tänkbara indata istället att uppfylla $V = 2^{n/2}$. Detta leder i sin tur till att $n \leq 37,5$, vilket avrundat blir 38. Detta är acceptabelt eftersom vi har halverat exponentens storlek (storleksordningen 10^7 operationer).

3 Branch-and-bound

Då ett sökträd traverseras kan det vara lönsamt att stanna upp och undersöka om det verkligen är värt att fortsätta söka längre ner i det aktuella delträdet. Ifall det långt upp i sökträdet går att upptäcka ofruktsamma grenar, kan söktiden förkortas markant genom att man helt enkelt struntar i dessa grenar. Denna metod går under namnet *branch-and-bound*.

För att kunna tillämpa branch-and-bound måste man hålla reda på bästa lösningen hittills, samt ha möjlighet att på ett lätt sätt kontrollera om en gren kan innehålla en bättre lösning. Användandet av Branch-and-bound kan anpassas till olika problem. Till exempel kan det finnas val som har hög sannolikhet att leda till fruktlösa delträd. För att tidigt beskära sökträdet bör dessa val placeras högt upp.

3.1 Exempel: Teckenavkodning

Hur räknar man ut ett matematiskt uttryck där en del siffror ersatts av bokstäver? I ett teckenavkodningsproblem skall man givet ett sådant uttryck, till exempel $CA = AB + 6C$, tilldela varje bokstav en siffra så att uttrycket stämmer och vänsterledet så litet som möjligt. Varje sådant uttryck kommer innehålla högst fem tal, varav exakt ett står i vänsterledet. Inget tal kan börja på siffran 0. I exemplet ovan utgör $C \leftarrow 8,\ A \leftarrow 1$ och $B \leftarrow 3$ en optimal lösning.

Ett naivt sätt att lösa problemet är att prova alla olika kombinationer av tilldelningar av siffror, och sedan välja den kombination som gav lägst resultat. Detta kan dock ta orimligt lång tid. Ett bättre sätt att angripa problemet är att först skriva om uttrycket på formen $0 = T_1 \pm T_2 \pm T_3 \pm T_4 - T_0$ där tecknen framför T_2 till T_4 är samma som i det ursprungliga uttrycket. Eftersom det är T_0 som ska minimeras är det naturligt att prioritera att bokstäverna som ingår däri får låga värden. Ju mer signifikant en bokstav är, desto viktigare är det att den minimeras.

För att lösa problemet adderar man alla tecken i varje position för sig. Låt position i vara den mest signifikanta positionen, position 1 den minst signifikanta, och låt $c_i = c_0 = 0$. Summan av alla tecken på position k samt ett tal c_{k-1} ska bli

$10 \cdot c_k$. Talet c_k är den så kallade carryn (minnessiffran) som fås vid additionen av de tecken på position k, och alla dessa ligger mellan -3 och 3.

Börja på position i och tilldela alla tecken som förekommer på denna position värden så att kraven är uppfyllda. Räkna ut vad c_{i-1} måste vara; ligger c_{i-1} inom det tillåtna intervallet så upprepa proceduren för nästa position, annars testar man en annan tilldelning. De möjliga följderna av tilldelningar bygger upp ett sökträd och i de flesta fall upptäcker man tidigt in i processen att det inte går att tilldela de aktuella tecknen sådana värden att kraven uppfylls. Man utnyttjar i dessa fall branch-and-bound och konstaterar att det är hopplöst att fortsätta med den aktuella tilldelningen.

Alla de tilldelningar som uppfyller kraven i samtliga steg ger korrekta uttryck. Vill man hitta den tilldelning som minimerar vänsterledet i ursprungsuttrycket måste man istället börja med att fixera det mest signifikanta tecknet i vänsterledet till lägsta möjliga värdet, och därefter kolla om uttrycket går att lösa. När lägsta möjliga värdet hittats för detta tecken, går man vidare och fixerar det näst mest signifikanta tecknet, och så vidare. Vänsterledet blir då garanterat så litet som möjligt.

Teckenavkodningsproblemet finns på KATTIS under titeln *decoding*, för den som är intresserad av att implementera och testa en lösning.

4 Sökning med hjälp av heuristiker

Heuristisk sökning innebär att man traverserar ett sökträd med hjälp av en heuristisk funktion. Att *expandera* ett av sökträdets löv innebär att man utgår från den följd av val lövet motsvarar och utökar den följden med ytterligare ett val. Vid traversering av sökträd vill man gärna undvika att expandera en massa löv förgäves, och den heuristiska funktionens uppgift är just att hjälpa till med att välja det löv som bör expanderas i nästa steg. För att klara av den uppgiften skattar den heuristiska funktionen i varje söksteg avståndet kvar till något mål.

Exakt vilken heuristik som används beror på vilken typ av problem det gäller. Gemensamt är dock att det är viktigt att ingen heuristik överskattar avståndet kvar till något mål, eftersom man då kan missa tänkbara optimala lösningar. För att det ska löna sig att använda heuristisk sökning gäller det allmänt att den heuristiska funktionen bör vara lättberäknad och att skattningen är någorlunda nära det sanna värdet.

4.1 Best-first-search

Ett sätt att utnyttja heuristisk sökning är att hela tiden expandera det löv som är bäst enligt den använda heuristiken. Ett exempel på en algoritm som utnyttjar best-first-search är *A**, som använder sig av följande heuristiska funktion: $f(u) = g(u) + h(u)$, där $g(u)$ är en överskattning av avståndet från startnoden till den aktuella noden u, och där $h(u)$ är en underskattning av kortaste avståndet från u till något slutmål. I A* väljer vi i varje steg att expandera den nod u med lägst $f(u)$.

Algoritm 1: A*
Input: En startnod.
Output: Huruvida det finns en lösning eller inte.

```
A*(startnod)
(1)   Q ← {startnod}
(2)   while Q ≠ ∅
(3)     hitta den nod u ∈ Q som har lägst f(u)
(4)     Q ← Q \ {u}
(5)     if isGoal(u)
(6)       process(u)
(7)       return success
(8)     else
(9)       L ← getChildren(u)
(10)      foreach y ∈ L
(11)        if y obesökt
(12)          markera y besökt
(13)          Q ← Q ∪ {y}
(14)  return failure
```

4.2 IDA*

IDA* är en vidareutveckling av A*, men som i varje steg bara söker på ett bestämt sökdjup. IDA* är en variant av IDS (Iterative Deepening Search) där man istället för att söka iterativt på djupen d ($d = 0, 1, 2, \ldots$) söker på de djup där det kan finnas en slutnod. Ifall man vet att slutnoden i den kortaste lösningen ligger på nivå minst tre, finns ingen anledning att söka på de tidigare nivåerna. Samma heuristiska funktion $h(u)$ som används i A*, en underskattning av avståndet kvar till något slutmål från u, utnyttjas här för att bestämma nästa nivå att söka på. Väldigt önskvärt är att $h(u) = 0 \iff \text{isGoal}(u)$.

Algoritm 2: IDA*
Input: Startnod, startdjup samt maximalt sökdjup.
Output: Om det finns en lösning mellan startdjupet och maxdjupet, returneras det djup denna lösning finns på. I annat fall returneras nytt maxdjup.

```
IDA*(u, level, bound)
(1)   if h(u) = 0
(2)      process(u)
(3)      return (true, level)
(4)   else if level + h(u) > bound
(5)      return (false, level + h(u))
(6)   else
(7)      m ← ∞
(8)      OK ← false
(9)      W ← getChildren(u)
(10)     foreach w ∈ W
(11)        (OK, b) ← IDA*(w, level + 1, bound)
(12)        if OK then return (true, b)
(13)              else m ← min(m, b)
(14)     return (false, m)
```

För att anropa IDA* används följande bit pseudokod:

```
(1)   b ← h(start)
(2)   found ← false
(3)   while !found
(4)      (found, b) ← IDA*(start, 0, b)
```

4.3 Exempel: Femtonspelet

Femtonspelet består av en sexton rutor stor kvadratisk ram innehållandes femton brickor, vardera en ruta stor, numrerade från 1 till 15. En ruta i ramen är således alltid tom. Målet med spelet är att genom brickförflyttningar få brickorna i sorterad ordning, det vill säga så att översta raden i ramen består av brickorna med siffrorna 1, 2, 3 respektive 4 (i den ordningen), och så vidare. Brickorna får inte lyftas bort från ramen, utan den enda tillåtna förflyttningen i varje steg är att skjuta en av de brickor som angränsar till den tomma rutan.

Vill man inte lösa problemet för hand så går det till exempel att använda IDA* för att, givet ett startläge, få reda på förflyttningarna som leder till den sökta brickordningen (om det finns en sådan följd förflyttningar). En tänkbar heuristisk funktion är då $h(\text{bräde}) = \sum_{i=1}^{15} \text{dist}(\text{bricka } i, \text{position } i)$, där avståndsfunktionen är manhattanavståndet mellan två positioner. Denna funktion uppfyller att $h = 0$ om och endast om alla brickor är på rätt plats. Funktionen är dessutom väldigt

snabberäknad om värdet i föregående steg sparas undan, eftersom förändringen av värdet enbart beror på den flyttade brickans föregående och nuvarande position (om brickan flyttades ett steg närmare sin rätta position är det nya värdet det gamla värdet *minus* ett och i annat fall det gamla värdet *plus* ett). IDA* kommer då hitta den kortaste lösningen, och process(u) står för utskriften.

Brädet kan lämpligtvis representeras med hjälp av en vanlig matris. Håller man reda på positionen för den tomma rutan går det snabbt att uppdatera tillståndet för brädet efter en förflyttning genom att byta plats på värdena i två matriselement. Ifall man håller reda på de brädtillstånd man redan besökt går det dessutom att undvika att flytta brickorna tillbaka till något av dessa tillstånd. Det är dock inte garanterat att detta lönar sig effektivitetsmässigt, men det kan vara smart att se till man åtminstone inte hoppar tillbaka till det närmast föregående tillståndet, vilket snabbt går att kontrollera och knappt kräver något extra minne.