DD2458, Problemlösning och programmering under press

Föreläsning 4: Kombinatorisk sökning

Datum: 2009-09-25
Skribent(er): Kristina Nylander, Dennis Ekblom, Marcus Öman
Föreläsare: Fredrik Niemelä

1 Introduktion

Vissa problem är svåra att hitta lösningar till. Då kan man ibland ta till totalsökning för att hitta en optimal lösning. Denna sökning skulle då gå igenom alla möjliga kombinationer och se om det ger en lösning till problemet, och jämföra kostnaden för denna lösning med tidigare hittade lösningar. Som exempel går vi igenom totalsökning för sortering. Nu finns det i och för sig bättre lösningar än totalsökning för sortering men bara för att visa hur det kan gå till. I sorteringsproblemet har man som indata en array av objekt som man vill sortera. För att lösa detta med totalsökning skulle man kunna generera alla permutationer av objekten och se om den ordningen blir sorterad. När man till slut hittar en ordning som är sorterad så returnerar man denna. Tidskomplexiteten i detta fall blir $O(n!)$ där n är antalet objekt i indata. Detta är givetvis en tidskomplexitet man vill undvika och i detta fall har man hittat andra sätt att lösa problemet på lyckligtvis, exempelvis mergesort. Men för vissa problem så är det väldigt svårt att hitta bättre sätt än att just söka igenom alla potentiella lösningar. Vi tänkte här diskutera några metoder man kan använda sig av för att söka igenom möjliga lösningar och skippa de som ser hopplösa ut för att hitta den optimala lösningen.

2 Sökalgoritmer

2.1 Bredden-först-sökning

Bredden-först-sökning (BFS) är en totalsökningsalgoritm som löser problemet med den kortaste vägen mellan två olika noder i en oviktad graf.

2.1.1 Algoritm

Idén bakom BFS, är att man söker igenom alla noder i ordningen av hur långt bort de är ifrån start noden, med den närmaste först. Detta gör man genom att lägga till rotnoden i en kö, i en kö kan man bara ta elementet som ligger först och bara lägga till element sist. Därefter tar man bara den första noden i kön och letar igenom dess grannar. Man sparar vilken nod man kom till grannen med, för "backtracking" och fortsätter med nästa nod i kön. När man hittar noden man letar efter returnerar man bara vägen till noden.

Algoritm 1: En BFS algoritm.
Input: En Grannlistad graf $U = \{n_1, n_2, \ldots, n_n\}$ av noder, en sökt nod *soughtNode*
Output: Den kortaste vägen A till målnoden. (Om det inte finns någon väg till målnoden så, $A = \emptyset$).

```
BFS(U,soughtNode)
(1)   Queue queue
(2)   queue.add(U.root)
(3)   while endNotReached
(4)     node = queue.pop() if node == soughtNode
(5)       return (pathTo(node))
(6)     else
(7)       foreach node.neighbours
(8)         if !neighbour.visited
(9)           neighbour.visited = true
(10)          queue.add(neighbour)
(11)    if queue.size() == 0
(12)      endNotReached = false
(13)  return ∅
```

2.1.2 För- och nackdelar

Den stora fördelen med BFS är att man får den kortaste vägen till den sökta noden, detta gäller dock inte om man har kantkostnader. En annan stor fördel med BFS är att den är väldigt lätt att förstå och implementera. Den stora nackdelen är att den kräver väldigt mycket minne då man måste hålla reda på alla grannar.

2.1.3 BFS med kantvikter

Om man vill använda BFS på en graf med kantkostnad och ändå få den kortaste vägen finns det ett enkelt sätt man kan lösa detta.

Istället för en kö använder man en prioritetskö och sorterar den på avståndet från rotnoden. Notera att insättning i en sorterad lista inte är konstant utan i värsta fallet blir $O(log(n))$ [1] där n är antalet element i listan och i vårt fall är n i genomsnitt $|noder|/2$. Dessutom är poll på första elementet i en sorterad lista inte konstant utan $O(log(n))$ [2]. Denna metod kallas Uniform Cost Search, UCS[3].

2.2 Djupet-först-sökning

Djupet-först-sökning (DFS) är en totalsökningsalgoritm som löser problemet med en väg mellan två olika noder i en graf.

DFS letar upp en stig genom grafen till en målnod. Själva algoritmen är väldigt lik BFS men istället för att titta på grannnoderna i en speciell ordning så tittar man

[1] Tidskomplexiteten bygger på PriorityQueue från Javas API 1.5
[2] Tidskomplexiteten bygger på PriorityQueue från Javas API 1.5
[3] http://en.wikipedia.org/wiki/Uniform_cost_search

på den första grannen till den nod man tittar på för tillfället. Först när man har kommit så långt att man inte längre hittar några obesökta noder som är granne till den nuvarande noden går man tillbaka till den förra noden. Pseudo koden för DFS är också väldigt lik BFS men istället för en kö för noder man ska besöka, använder man sig utav en stack för de noder man besökt, exklusive förgreningar.

Algoritm 2: En DFS algoritm.
Input: En GrannListad graf $U = \{n_1, n_2, \ldots, n_n\}$ av noder, en sökt nod *soughtNode*
Output: Den kortaste vägen A till mål noden. (Om det inte finns någon väg till mål noden så, $A = \emptyset$).

```
DFS(U,soughtNode)
(1)     Stack pathTo
(2)     node = U.root
(3)     while endNotReached
(4)         if node == soughtNode
(5)             return pathTo
(6)         else
(7)             foreach node.neighbours
(8)                 if !neighbour.visited
(9)                     neighbour.visited = true
(10)                    path.add(neighbour)
(11)                    neighbourFound = true
(12)                    break
(13)            if neighbourFound != true
(14)                node = path.pop()
(15)        if node == root
(16)            endNotReached = false
(17)    return ∅
```

2.2.1 För- och nackdelar

Den stora fördelen med DFS är att man inte behöver hålla koll på alla noder utan bara de som man använt för att ta sig dit man är utan förgreningar. Även om man i värsta fallet måste spara alla noder i grafen. Den stora nackdelen med DFS är att man inte får den kortaste vägen utan vilken väg som helst. Den lämpar sig därför bäst om man är intresserad av att veta att det finns en väg till målet snarare än hur lång den är.

2.2.2 DFS med kantvikter

Ett enkelt sätt att göra så att man hittar den kortaste vägen med en DFS är att istället för att spara visited som en boolean spara den som den kostnad man tog för att komma dit och då istället skriva över kostnaden om den är högre. När man hittar mål noden så måste man dock fortsätta och leta eftersom det kan finnas någon

annan väg som kan vara kortare. Man sparar den kortaste vägen och jämför den med de andra vägarna man hittar. Notera att om man har negativa loopar kommer algoritmen gå in i en oändlig loop. Denna metod kallas Dijkstras algoritm[4].

2.3 Iterativ deepening depth first search

Iterative deepening depth first search, ID-DFS, är en totalsökningsalgoritm som letar upp den kortaste vägen till en nod i en graf.

ID-DFS är en kombination av fördelarna med DFS och BFS. Själva idéen är att man söker igenom grafen med DFS men man söker bara till ett givet djup i grafen. Man ökar sedan successivt djupet man söker på. Detta resulterar i att man besöker noderna för första gången precis som i BFS men man använder inte mer minne än DFS.

2.3.1 För- och nackdelar

Fördelarna är att man får den kortaste vägen i en oviktad graf samt att man inte använder mer minne än en DFS. Den stora nackdelen är ju dock att man måste köra en DFS flera gånger på samma noder. Tillägg: Det är dock inte så farligt att noder i början av sökträdet blir besökta flera gånger. Sökträdet är ju absolut bredast i slutet.

2.4 Branch-and-Bound

Branch-and-Bound är ett generellt sätt att hitta lösningar till kombinatoriska problem. Det går ut på att försöka hugga av grenar i sökträdet så tidigt som möjligt för att minska sökrymden. För att göra detta är det viktigt att ta beslut i rätt ordning. Man vill hitta motsägelser för att hugga av grenar så tidigt som möjligt. Därför bör stora beslut tas tidigt.

2.4.1 Exempel: Kappsäcksproblemet

I Kappsäcksproblemet har man en mängd med objekt som har värden $\{v_1 \dots v_i\}$ samt vikter $\{w_1 \dots w_i\}$. Med de objekten ska en kappsäck med kapacitet K fyllas så att innehållets värde maximeras.

Det första man bör göra är att fundera på om en DP-algoritm kan användas istället. Om värden och vikter ej är begränsade får man använda Branch-and-bound istället. Det första steget är att räkna ut kvoten $\frac{v}{w}$ för alla objekt och sortera dessa efter minskande storlek. Sedan börjar man försöka att packa objekt i kappsäcken. I figur 1 kan ett sökträd för problemet ses. I varje steg kan man göra två beslut. Endera tar man med objektet som noden representerar, eller så lämnar man det. Man skär av grenar i trädet genom att titta på nästa objekt. Om man skulle fylla resten av kappsäcken med det objektet, får man då ett bättre värde än det bästa hittills? Om man inte får det är det ingen idé att fortsätta nedför den vägen och det är istället dags att gå tillbaka. I algoritm 3 ses en skiss av algoritmen.

[4] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

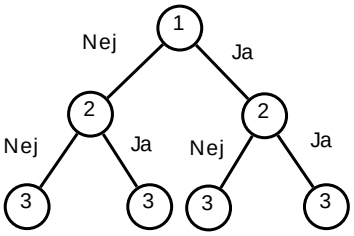

Figure 1: Sökträd för kappsäcksproblemet. Varje nod representerar ett objekt, och för varje objekt kan man endera välja att ta med det (följa Ja-kanten) eller inte ta med det (Nej-kanten).

Algoritm 3: Branch-and-bound-algoritm för kappsäcksproblemet
Input: En icke-tom mängd $V = \{v_1, v_2, \ldots, v_n\}$ representerande objektens värden, En icke-tom mängd $W = \{w_1, w_2, \ldots, w_n\}$ representerande objektens vikter. Ett tal C, kapaciteten på kappsäcken.
Output: En mängd S, innehållandes objekten som ingår i packningen med det maximala värdet.

KAPPSÄCK(V, W, C)
(1) **foreach** $v_i \in V, w_i \in W$
(2) $P = \{\frac{v_1}{w_1} \leq \frac{v_2}{w_2} \ldots \frac{v_i}{w_i}\}$
(3) $best = -1$ bästa lösningen hittills
(4) $curr_v = 0$ värde på packning hittills
(5) $curr_w = 0$ vikt på packning hittills
(6) **foreach** $\frac{v_i}{w_i} \in P$
(7) **if** $w_i > C$ **then** ta ej med objekt
(8) **if** $\frac{v_{i+1}}{w_{\lfloor}i+1}(C - curr_w) + curr_v < best$
(9) avbryt och ta ej med objekt
(10) **else**
(11) $curr_v += v_i, curr_w += w_i$
(12) **if** $curr_v > best$ **then** $best = \{curr_v, curr_w\}$
(13) **foreach** $obj \in best$
(14) $S \cup \{obj\}$
(15) **return** S

2.4.2 Exempel: Teckenavkodning

I teckenavkodningsproblemet har man ekvationer på formen $CA = AB + C6$. Alltså både siffror och bokstäver blandat. Problemet går ut på att man ska ge bokstäverna siffror så att ekvationen uppfylls och vänsterledet är minimalt. De begränsningar som finns är att första siffran i en term inte får vara 0 samt att det finns max 5 termer.

Den naiva metoden här skulle vara att generera alla kombinationer av siffror

som bokstäverna kan anta. Testa om en kombination fungerar, och om den fungerar jämföra den med den bästa lösningen hittills. En oerhörd mängd med kombinationer kommer att genereras och testas här. Går det inte att göra på något bättre sätt?

För att använda branch-and-bound behöver vi hitta regler som gör att vi kan avbryta sökningar. En regel har vi redan. Det är att den första siffran i en term inte får vara 0. Det betyder att vi aldrig behöver testa de kombinationer där 0 ersätter en bokstav som är första siffran i en term. Detta kanske räcker tillsammans med att vi vet att uttrycket ska minimeras?

Istället för en brute-force-algoritm representerar vi termerna i högerledet med $T_1 \dots T_4$, och vänsterledet med T_0. Vi markerar även de olika positionerna i en term med $0 \dots i$. Den mest signifikanta positionen i term T_1 är alltså T_{1i}. Den minst signifikanta är T_{1_0}. Då kan man göra följande uppställning:

	T_{1i}	$\dots$	T_{10}
	T_{2i}	$\dots$	T_{20}
	T_{3i}	$\dots$	T_{30}
	T_{4i}	$\dots$	T_{40}
$\sum$	T_{0i}	$\dots$	T_{00}

När vi nu vill minimera T_0 börjar vi tilldela T_{0i} och fortsätter sedan uppåt med $T_{4i} \dots T_{1i}$. Hela tiden försöker man använda så små siffror som möjligt. När raden längst till vänster är klar går vi ett steg åt höger. Genom hela lösningen måste man även hålla reda på minnessiffran som kan komma från kolumnen till höger. Eftersom vi har fyra termer kan den högst vara tiotalsdelen av $4 * 9 + 3 = 39$, alltså 3. Som minst kan den vara -3. Ligger minnessiffran utanför intervallet $[-3, 3]$ kan lösningen direkt förkastas. När man har kommit till positionen längst till höger har man hittat den optimala lösningen.

2.5 2-hålls-sökning (Bidirectional search)

Ett sökträd för en brute force-lösning förgrenar sig snabbt, och längst ner har man 2^d noder i sitt sökträd om varje nivå har två val och det finns d nivåer. Vet man redan om startnoden och slutnoden kan man istället för att endast söka uppifrån och ner, även söka nedifrån och upp. Det medför att komplexiteten går från 2^d till $2 \cdot 2^{d/2} = 2^{d/2+1}$. Detta kallas bidirectional search eller meet-in-the-middle. I figur 2 ses utbredningen för två sökningar. En från mål-noden och en för start-noden. Använder man 2-hålls-sökning är det endast sökrymden som är markerad i mitten som behöver beräknas. För att denna metod ska gå att utnytta till fullo krävs det dock att delresultaten går att kombinera på ett bra sätt.

2.5.1 Exempel: Dubbel DES

DES är en krypteringsalgoritm som använder en 56-bitars nyckel. Man krypterar sitt meddelande m genom $\text{DES}_{k_1}(m) = c$. I dubbel DES krypterar man två gånger med olika nycklar: $\text{DES}_{k_1}(\text{DES}_{k_2}(m)) = c$. Då får man istället för en nyckel på 56 bitar, en nyckel på 112 bitar att dekryptera. Detta är betydligt svårare att genomföra med brute force-lösningar.

Skulle man ändå försöka sig på en brute force-sökning kommer man att få ett sökträd med djup 2 och 2^{56} val i varje nod. Om man på något sätt skulle ha

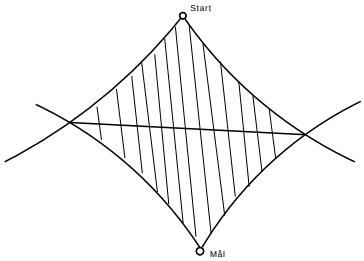

Figure 2: Utbredning för två sökningar. En sökning börjar i startnoden, den andra i målnoden. Endast markerat område behöver beräknas..

fått tag i en bit av meddelandet, och samma block av text i krypterat format, kan man istället använda sig av en 2-hålls-sökning. Det eftersom man har både startnod (meddelande) och slutnod (kryptotext). Då börjar man från startnoden och applicerar $DES_{k1}(m)$ på den och sparar resultatet i t.ex. en hashtabell. Sedan går man från slutnoden och applicerar $DES_{k2}^{-1}(c)$ på den. Dubbel-DES-nyckeln hittar man genom att söka igenom tabellen för startnoden och se om man hittar en match från resultatet från slutnoden.

2.6 Bäst-förstsökning

Bäst-förstsökning är en sorts trädsökning eller grafsökning där man expanderar en nod beroende på hur lovande lösningen från den noden ser ut, jämfört med de andra nodernas dellösningar. För att kunna evaluera hur bra en dellösning är använder man sig av en evalueringsfunktion. Om man lyckas göra en evalueringsfunktion som säger exakt hur nära lösningen man är skulle denna sökning inte gå ett steg fel över huvudtaget. I praktiken är detta omöjligt och man använder sig av en heuristisk funktion som uppskattar hur nära någon lösning man är, det vill säga kostnaden att komma till någon lösning.

2.7 A*

A* är en sorts bäst-förstsökning som inte bara utnyttjar den heuristiska funktionen, $h(x)$, utan den håller även reda på kostnaden för att komma till den punkten man evaluerar i, $g(x)$. Dessa två adderat ger oss funktionen $f(x)$. Det är väldigt viktigt att $h(x)$ är en exakt kostnad eller underskattning av kostnaden från den noden man är till målet. Detsamma gäller för $g(x)$ fast där ska det vara en exakt kostnad eller överskattning. Oftast är det ganska enkelt att hålla reda på exakt kostnad - eftersom man vet ju oftast hur mycket det kostade att komma dit man är - eftersom man har genomfört de stegen.
Alltså för en given dellösning x:
$f(x) = g(x) + h(x)$
$g(x) \geq \text{cost}(start, x)$
$h(x) \leq cost(x, ml)$

Algoritm 4: A*
Input: En graf, startnod, lösningskrav, funktionen $h(x)$ samt $g(x)$
Output: Den lägsta kostnaden från start till en lösning. Notera att Q är en prioritetskö, lägsta värdet är högst prioriterat.

```
A*(G, start, solution(x), h(x), g(x))
(1)     Q.ADD(start, h(start))
(2)     solution = ∞
(3)     while Q is not empty
(4)         node = Q.poll()
(5)         if solution(node) and g(node) < solution
(6)             solution = g(node)
(7)         else
(8)             foreach neighbor ∈ node.neighbors
(9)                 if not repeated state
(10)                    Q.add(neighbor, g(node)+
(11)                    cost(node, neighbor) + h(neighbor))
(12)    return solution
```

2.7.1 15-pusslet

15-pusslet är ett pussel där man har ett $4x4$-bräde med 15 olika 1x1-brickor med siffror på. Dessa siffror går från 1 till och med 15. Detta innebär att det blir en tom ruta över och denna ska man utnyttja genom att man kan flytta de icke diagonalt angränsande brickorna till denna ruta. Detta gör man tills man har kommit till målet, vilket är att alla brickorna är ordnade så att i den översta raden finns brickorna $1, 2, 3, 4$ i ordning, andra raden finns $5, 6, 7, 8$ och så vidare. 15-pusslet är ett problem man skulle kunna lösa med hjälp av A*-sökning. $g(x)$ är väldigt enkel att definiera, nämligen antalet förflyttningar man har gjort. Funktionen $h(x)$ är däremot lite svårare att definiera. Man skulle kunna sätta denna till antalet felplacerade brickor eller summan av manhattanavstånden för varje bricka till det läge den skall vara i slutet. Båda dessa fungerar eftersom de inte överskattar kostnaden till målet, fast den senare är att föredra eftersom den uppskattar kostnaden bättre än den tidigare.

2.8 Iterative deepening A*

Iterative deepening A* är precis vad det låter som, det är A*-sökning med en begränsning på hur bra lösningar man ska undersöka. När man ökar djupet så tillåter man potentiellt sämre lösningar. Detta gör man bara när man inte hittat en lösning på ett givet djup, och dessa potentiellt sämre lösningar är då egentligen de bättre lösningarna, men de kostar mer än vad man trodde behövdes från början.

Algoritm 5: ID-A*
Input: En graf, startnod, lösningskrav, funktionen $h(x)$ samt $g(x)$ och ett maxdjup
Output: Den lägsta kostnaden från start till en lösning. Notera att Q är en prioritetskö, lägsta värdet är högst prioriterat.

ID-A*$(G, start, solution(x), h(x), g(x), maxdepth)$
(1) Q.ADD$(start, h(start))$
(2) **if** $h(start) > maxdepth$
(3) **return** ∞
(4) $solution = \infty$
(5) **while** Q is not empty
(6) $node = Q.poll()$
(7) **if** $solution(node)$ and $g(node) < solution$
(8) $solution = g(node)$
(9) **else**
(10) **foreach** $neighbor \in node.neighbors$
(11) $fx = g(node) + cost(node, neighbor) + h(neighbor)$
(12) **if** not repeated state and $fx \leq maxdepth$
(13) $Q.add(neighbor, fx)$
(14) **return** $solution$