2D1458, Problemlösning och programmering under press

Föreläsning 4: Kombinatorisk sökning

Datum: 2006-09-25
Skribent(er): John Lindholm, Niclas Eklöw
Föreläsare: Mikael Goldmann

Den här föreläsningen handlade om kombinatoriska sökningar, en typ av problemlösning som ofta resulterar i en totalsökning. Föreläsaren berättade bland annat om spelen Pentatriss och 15-pusslet samt andra planeringsproblem och olika tillvägagångssätt för deras lösningar.

1 Innehåll

- Kombinatorisk Sökning
- Sökstrategier
- Heuristik/Approximationer
- Trick

2 Kombinatorisk sökning

Kombinatoriska sökproblem räknas allmänt till de svårare problemen att lösa beräkningsmässigt och detta beror på att lösningarna ofta resulterar i uttömmande sökningar. Typiskt för dessa problem är att man söker efter något kombinatoriskt eller matematisk objekt med vissa egenskaper.

- M-färgning av en graf
- Kappsäcksproblemet med önskad optimal lösning.
- Rubiks kub (en kub uppdelad i sektioner där man vrider sidorna för att få samma färg på var sida.)

Gemensamt för dessa och andra typer av kombinatoriska sökproblem är att det ofta inte finns någon effektiv lösning till problemen och då får man försöka så gott man kan.

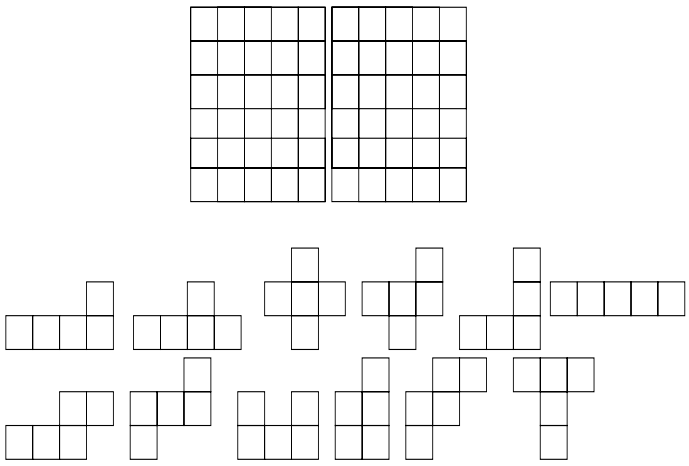
För små instanser går problemen snabbt att lösa med en totalsökning men när instanserna växer blir det svårare och för stora instanser kommer det att ta lång tid. Därför är det viktigt att vara noggrann med den implementation man väljer och det finns vissa saker man bör tänka på.

- Undvik att utforska fruktlösa dellösningar.
- Välj så bra algoritm som möjligt. Det är skillnad på $n!$, 2^n och $2^{\frac{n}{2}}$.
  - $n!$ går att lösa i rimlig tid med $n = 12$ då: $12! = 4.8 \cdot 10^8$.
  - 2^n ger oss bättre resultat, $n = 24$: $2^{29} = 5.4 \cdot 10^8$
  - $2^{\frac{n}{2}}$ kan vi lösa med $n = 58$: $5.4 \cdot 10^8$

3 PentaTriss

Pentatriss är ett spel där man skall pussla ihop bitar som tillsammans bildar ett bräde med två delar av storleken 5×6 rutor. Varje bit består av 5 rutor ihopsatta på olika sätt och för att pussla ihop spelet får man vrida och vända på bitarna hur man vill.

I spelet kan man avgränsa totalsökningen för probleminstansen genom att iaktaga att storleken av varje ny samling av lediga rutor som erhålles när man placerar in en pusselbit på brädet måste vara delbart med 5 efterssom alla pusselbitar täcker 5 rutor var. Dessutom kan man undvika att testa fall som redan testats genom att inse att vissa av pusselbitarna har symmetrisk form (Exempelvis pusselbiten som ser ut som ett kors, som varken behöver roteras eller speglas).



Figur 1: En bild föreställande alla olika bitar i ett Pentatriss.

För att kunna beräkna alla fall så snabbt som möjligt behövs ett sätt att snabbt kunna lägga till och ta bort pusselbitar och att kontrollera om en bit får plats på brädet på en viss position. Detta kan uppnås genom 'bitmasker' där de två 5×6 rutorna i brädet och även pusselbitarna kan representeras av 30 bitar expemelvis i en integer. Man låter då de första 6 bitarna representera varje ruta i översta raden och följande 6 bitar i andra raden osv. En etta betyder att rutan är upptagen och en nolla att den är ledig. Sedan kan man hitta snabba bitoperationer för att kontrollera när en pusselbit kan läggas till eller ej.

4 Sökstrategier

4.1 Planeringsproblem

För planeringsproblem finns olika kriterier som man ska tänka på när man bestämmer hur ens kombinatoriska problemet ska lösas.

- Det finns oftast en startposition från vilken man utgår.
- Vill nå en målkonfiguration, men det kan eventuellt finnas flera möjliga.

- Ska kunna påverka konfigurationen med enkla operationer.
- Ibland vill man optimera vägen dit, ibland bara hitta målet.

Som exempel på planeringproblem kan nämnas det klassiska spelet 15-pusslet som består av 15 siffror på brickor på ett bräde tillsammans med en tom ruta. Målet för spelet är att flytta dessa brickor så att en viss siffersekvens bildas på brädet. Men man får bara flytta brickor till den tomma platsen och dessa brickor måste angränsa denna för att få flyttas dit.

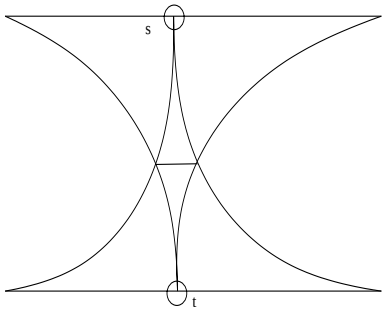
4.2 8-pusslet

En mindre version av 15-pusslet är 8-pusslet med 8 brickor och en tom ruta. Antalet möjliga konfigurationer för spelet är $\frac{9!}{2}$. Där är sökrymden tillräckligt liten för att den skall vara beräkningsbar i rimlig tid. Så hur går vi nu till väga för att lösa problemet?

Vad man kan göra är att använda sig av ett sökträd och genom att söka igenom detta försöka hitta lösningar till problemet. Det vi har är ett träd där varje nod är nåbar från rot-noden och representeras av ett tillstånd på brädet och där varje kant representerar den handling vi utförde för att komma dit från dess förälder. Sättet som trädet söks igenom beror på val av sökalgoritm som i sin tur beror på vilka fördelar man vill ha respektive nackdelar man får.

- BFS - Breadth First Search. BFS söker igenom hela trädet och expanderar alla noder på samma nivå. Nackdelen med BFS är att man riskerar att få en alltför stor kö och många noder att hålla reda på, vilket leder till stor minnesanvändning. BFS garanterar dock en optimal lösning.
- DFS - Depth First Search. Söker på djupet i sökträdet. Använder betydligt mindre minne men är inte optimal. Man kan således råka hitta en lösning långt ned i trädets grenar som är mycket sämre än den bästa.
- ID-DFS - Iterative Deepening Depth First Search. Är en variant av DFS som begränsar djupet i en vanlig DFS. Sedan ökas djupet i omgångar tills man nått tillräckligt djupt och en lösning kan hittas. Detta kan verka lite långsamt men faktum är att i ett sökträd finns ofta de flesta noderna långt ned i trädet. Genom detta kommer vi hitta en optimal lösning utan att behöva undersöka noder långt ned i grenar i trädet som kan innehålla dåliga lösningar. Komplexiteten blir då $\theta(b^d)$ där b är greningsfaktorn och d är djupet.

Det går att starta en sökning från starttillståndet, måltillståndet eller från båda hållen så att de möts på mitten, även kallat Bidirectional Search. Fördelen med att söka från båda hållen är att man minskar den totala mängden noder som måste genereras. Komplexiteten för dubbelriktad sökning, om greningsfaktorn är b och djupet är d, blir då $\theta(2 * b^{\frac{d}{2}}) = \theta(2 * b^{\frac{d}{2}})$ som med andra ord är snabbare än en sökning från bara ena hållet med kvadratroten. En nackdel kan vara att vi med en bidirektionell sökning måste kontrollera ifall vi har stött på någon nod från andra sidan så att man vet när man har krockat på mitten.



Figur 2: Dubbelriktad sökning, man söker från båda håll tills man möts. Den sökrymd man nu behöver kolla begränsar sig till att endast vara överlappningen mellan den över och undre sökrymden på bilden.

5 Heuristiker

Vid en heuristisk sökning använder man en skattning av kostnaden att nå målet i ett sökningsproblem från en viss plats där man befinner sig. För sökning i trädstrukturer kan heuristiker användas för att välja bra vägar längs grenar i sökträdet och på så sätt undvika grenar där det kan finnas dåliga lösningar. Vilken heuristik man väljer beror på problemet, enda kravet är att heuristiken **inte** får överskatta kostnaden för att nå en lösning. Det vill säga, heuristiken får tro att det finns en bättre lösning längs en viss stig än vad som sedan visar sig vara fallet. Det finns flera algoritmer som använder sig av heuristik för att hitta lämpliga vägar, exempelvis:

- Best First Search
- Branch-and-Bound

5.1 Best-First-Search

I Best First Search väljs varje gång den bästa vägen som heuristiken säger åt den att ta. Detta beskrivs genom evalueringsfunktionen f där det bästa valet för att nå målet bestäms av den heuristik som används på den plats x där man befinner sig.

5.1.1 A*

Ett exempel på en Best-First-Search algoritm är A* vars evalueringsfunktion räknar med antal steg man gått från starten till den plats där man för tillfället befinner sig g, plus det uppskattade antalet steg kvar till målet h. Evalueringsfunktionen får då utseeendet:

$f(x) = g(x) + h(x)$ // summan av tillryggalagd väg och väg kvar till målet.
$g(x) \geq dist(start, x)$ // g är en överskattning
$h(x) \leq dist(x, ml)$ // h är en underskattning

Algoritm 1: A* search algoritmen
Input: Ett sökträd bestående av noder och barn.
Output: Reultatet i form av success, failure.

```
A*(Ett träd)
(1)     Q ← {start}
(2)     while Q ≠ {}
(3)         finn x ∈ Q som har minst f(x)
(4)         if isGoal(x)
(5)             process(x)
(6)             return success
(7)         else
(8)             L ← getChildren(x)
(9)             foreach y ∈ L
(10)                if y obesökt
(11)                    markera y besökt
(12)                    lägg y till Q
(13)    return failure
```

5.1.2 Problematik

Ett problem som kan uppstå om man använder denna algoritm är att man riskerar att få en stor kö Q, exempelvis om vi har en stor sökrymd.

5.1.3 IDA*, Iterative deepening A*

Det kan vara rimligt att ha ett bestämt träddjup och att man sedan med en iterativt ökande algoritm ändrar djupet efter hur heuristiken ändras. Vad som skiljer algoritmen nedan med ID-DFS är att IDA* ibland ökar maxdjupet vi får besöka i trädet med mer än 1, och i en jämförelse med A* slipper vi nu också administrera en kö Q.

Algoritm 2: IDA* search algoritmen
Input: x roten till ett sökträd bestående av noder och barn, *depth* det djup i trädet vi gått hittils, *limit* en gräns satt för hur djupt vi max vill gå.
Output: Reultatet i form av success -1, eller den ökning *newlimit* vi måste göra för att nå målet enligt vår heuristik.
IDA*$(x, depth, limit)$
(1) $newlimit \leftarrow depth + h(x)$ // g(x) + h(x)
(2) **if** isGoal(x)
(3) process(x) **return** -1 // vi är klara och nöjda.
(4) **else if** $newlimit \leq limit$
(5) $L \leftarrow$ getChilderen(x)
(6) **foreach** $y \in L$
(7) $tmp \leftarrow$ IDA*$(y, depth + 1, limit)$
(8) $newlimit \leftarrow$ min$(tmp, newlimit)$
(9) **return** newlimit

Algoritm 3: Härifrån anropas IDA* algoritmen.
(1) $limit \leftarrow 0$
(2) **while** $limit \geq 0$
(3) $limit \leftarrow$ IDA*$(start, 0, limit)$
(4)

5.1.4 15-Spelet med IDA*

Vad skall man ha för heuristik $h(x)$ till 15-spelet? Man bör välja mellan snabb beräkning och god approximation.

- $h(x) = \sum_{i=1}^{15} manhattanavst.$ för bricka i från aktuell plats till rätt plats
- $h(x) = 0$ omm x är målet.

Om den algoritm man testade blev långsam kan man testa att:

- Ställa upp delmål, ex försök klara första raden i 15-spelet, sedan nästa osv. Här offrar man optimalitet men sparar mycket tid.

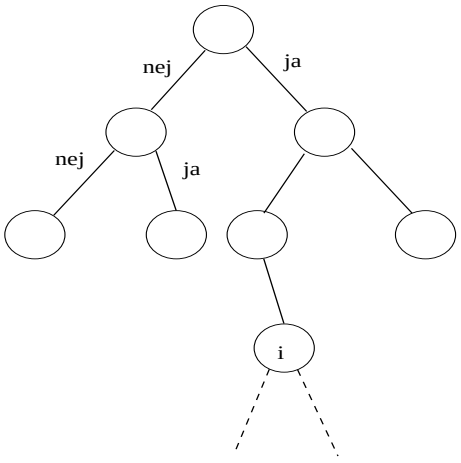
- Undvik samma konfiguration flera gånger längs samma stig upp till rotnoden. I det här fallet visar det sig vara mer effektivt att enbart se till att man inte går tillbaka till samma nod som man just kom ifrån istället för att kolla hela vägen upp till roten. Detta kan innebära att man får upprepade konfigurationer men man tjänar ändå tid eftersom det är mycket snabbare att testa.

- Profilera koden och försök se vad det är som tar mest/mycket tid. Om exempelvis heuristikberäkningen tar tid, försök byta ut denna till en annan som går snabbare. För 15-spelet gäller exempelvis att om x är föregående och x' är aktuell konfiguration så är $h(x')$ lätt att beräkna givet $h(x)$ och senaste draget.

5.2 Branch and Bound exempel

Tanken med Branch and Bound är att man kan upptäcka vid en nod i sökrymden, om dess barn i sökningen är värda att undersöka. Detta styrs av ett villkor som talar om utifall vägen kommer kunna ge en bättre lösning än den bästa hittills. Om villkoret ej uppfylls backtrackar man och testar andra vägar. Man kan använda Branch and bound för att beräkna en lösning till kappsäcksproblemet. K är är kappsäckens maximala kapacitet, C hittills packad vikt, V hittills packat värde och M är hittills bästa hittade packning.

Man börjar med att lägga alla saker i en lista och sortera dem så att $\frac{v_1}{w_1} \geq \frac{v_2}{w_2}$ osv. där v_i och w_i är värde respektive vikt för sak i



Figur 3: Sökträd vid branch and bound, där varje nivå i trädet motsvarar nästa sak i listan.

Vi börjar då plocka saker ur den sorterade kön, får första saken plats går vi till höger (säger ja), får nästa sak plats fortsätter vi till höger ända tills vi hittar en sak som ej får plats då går vi till vänster (säger nej). Om $\frac{vi}{wi}(K - C) + V \leq M$ kan vi strunta i noden eftersom vägen ej kan ge bättre resultat än M. När vi sökt igenom alla vägar så är M värdet av den bästa packningen.

Detta ger en totalsökning men man kommer ofta att kunna skära bort ganska många vägar eftersom villkoret vi jämför med ofta kommer bli falskt långt upp i trädet och på det sättet kan branch and bound effektivisera sökningen en hel del.