Mästarprov 1 ADK14

Fredrik Berntsson, frebern@kth.se, 930910-6798

October 13, 2014

1 Jobbig Labyrint

Jag har valt att modifiera Djikstra's grafalgoritm för att hitta den kortaste vägen från två hörn i en graf. Detta fungerar att modifiera eftersom labyrinten (matrisen) går att tolka som en graf där varje ruta i matrisen är ett hörn och antalet hörn |V| = n*n, dvs. matrisens storlek och antalet kanter |E| = 2n(n-1). Varje hörn kan maximalt ha fyra kanter, dvs till dess närliggande rutor i matrisen. En prioritetskö (min-heap) används för att snabbt gå vidare från ett hörn till dess närmsta granne och istället för kantvikter används hörnvikter för att markera hur mycket det kostar att gå igenom en ruta. Dessutom tas det hänsyn diken mellan hörn som ibland omöjliggör att gå emellan dessa.

Algoritmen fungerar principiellt likadant som Dijsktra's. I början på algoritmen initieras en prioritetskö Queue, där de "billigaste" hörnen sedan kommer att hämtas ifrån, en matris Visited som sparar de hörnen vi redan har besökt och en matris Dist som sparar de kortaste avstånden vi hittat till hörnen. Först initialiseras allt genom att man för alla hörn (förutom starthörnet) i labyrinten sätter Dist till ∞ och Visited till false, samt lägger in hörnet i prioritetskön. Starthörnet sätts in i prioritetskön med prioriten lika med hörnvikten för startrutan i labyrinten. Det kommer alltså bli det första hörnet man tar ut ur prioritetskön.

Sedan körs huvudloopen, där man alltid hämtar ut det hörn med minst prioritet ur kön (dvs hörnet med minst avstånd). Här är algoritmen modifierad så att om hörnet vi hämtat har dist $= \infty$ vet vi att vi inte kan nå längre, eftersom det är hörnet med minst prioritet, och returnerar då "omöjligt". Om vi är på hörnet längst ner till höger i matrisen, dvs utgången, returnerar vi värdet i Dist för detta hörn och är då klara. För varje hörn kommer vi gå igenom alla grannar (som vi vet max kommer att vara fyra stycken), detta görs med en hjälpmetod visitNeighbor som körs fyra gånger. Det är egentligen här algoritmen är mest modifierad. Först måste vi kolla om grannen tillhör matrisen och om vi redan har besökt den. Gör den det måste vi också kolla om det finns ett dike till grannen och kolla om vi kan hoppa över det, eller om vi måste gå runt det. Om vi på detta vis hittar en ny kortare väg till grannen uppdaterar vi dess avstånd i Dist och ändrar dess prioritet i prioritetskön.

Algorithm: LabyrintPath

Input: Matris M[1..n][1..n] med hörnvikter, lista med diken TL[1..l]
Output: Minsta avståndet genom labyrinten

```
Queue(empty)
Visited[1..n][1..n]
Dist[1..n][1..n]

Dist[1][1] ← M[1][1]
Visited[1][1] ← true
Queue.insertWithPriority( (1,1), dist[1][1])

for i ← to n do
    for j ← 1 to n do
        if (i,j) ≠ (1,1) then
            Dist[i][j] ← ∞
            Visited[i][j] ← false
            Queue.insertWithPriority( (i,j), dist[i][j] )
        end
    end
end
while Queue is not empty do
    (i,j) ← Queue.extractMin
    Visited[i][j] ← false

    if Dist[i][j] = ∞ then
        return "impossible"
    end
    if i = n ∧ j = n then
        return dist[i][j]
    end
    visitNeighbor (i, j, i + 1, j )
    visitNeighbor (i, j, i - 1, j )
    visitNeighbor (i, j, i , j + 1)
    visitNeighbor (i, j, i , j - 1)

end
```

Algorithm: visitNeighbor

```
if i < 1 ∨ i ≥ n + 1 ∨ j < 1 ∨ j ≥ n + 1 ∨ Visited[i][j] = true then
    return
end
if ( (i,j), (oldi, oldj) ) in TL then
    if dist[oldi][oldj] < 1000 then
        cost ← M[i][j] + 100
    else
        cost ← ∞
    end
else
    cost ← M[i][j]
end
if cost ≠ ∞ then
    alt ← const + dist[oldi][oldj]
else
    alt ← cost
end
if alt < Dist[i][j] then
    Dist[i][j] ← alt
    Queue.decreasePriority( (i,j), alt )
end
```

Tidskomplexitet:

Initialisering: Vi går igenom hela matrisen, dvs. n*n gånger. I varje iteration görs två konstanta operationer och en insättning i prioritetskön (tar O(log n)). Komplexiteten för initialiseringen blir därför $O(n^2 \log n)$.

Huvudloop: Whileloopen körs i värsta n*n gånger, dvs genom alla hörn: $O(|V|) = O(n^2)$, den kan returnera tidigare om vi hittar en "snabb" väg. Vi kommer i varje iteration i while kommer hämta ut ett hörn ur kön. Eftersom kön kan innehålla |V| hörn, och om använder vi en min-heap (prioritetskö) kommer varje hämtning ur kön ta $O(\log |V|) = O(\log n^2) = O(2*\log n) = O(\log n)$ (eftersom vi måste "filtrera" om heapen sen) Vi kommer alltid att kolla om vi kan gå till fyra granner för hörnet vi står på. I värsta fall kommer vi kunna det och då körs hela visitNeighbor 4 gånger. I varje sådan iteration kommer vi i värsta fall leta igenom hela listan med diken: det tar O(l) tid och i värsta fall behöver vi minska prioriteten hos ett hörn i kön eftersom vi hittat en snabbare väg till ett hörn. Det tar $O(4*\log |V|) = O(\log n^2) = O(2*\log n) = O(\log n)$

Totalt: initialisering + huvudloop = $O(n^2 \log n + n^2 * \log n * l * \log n) = O(n^2 \log n + l * n^2 * \log n) = O(l*n^2*(\log n)^2)$

2 Lådor i lådor

Problemet är alltså att, givet en lista med lådor, returnera det maximala antalet lådor som kan "nästlas" i varandra.

Det kan först vara värt att notera att en låda b1 endast kan stoppas i en annan låda b2 om den minsta dimensionen för b1 är mindre än den minsta för b2, den mellersta dimensionen för b1 är mindre än den mellersta dimensionen för b2 och den största dimensionen för b1 är mindre än b2. Gäller detta betyder det att vi kan vrida på låda b1 på något sätt (om det behövs) så att den får plats i b2.

Algoritmen fungerar överblickande på följande sätt:

Vi vill leta efter de lådor som är störst, och som inte får plats i varandra. Om vi sedan har en låda kvar som går in i någon utav dessa stora lådor, vet vi att vi kan kan stoppa en låda i en annan en gång. Vi vill då ta bort de stora lådorna från listan och börja om på nytt, dvs. hitta nästa "nivå" av största lådor. Varje gång vi kan utföra denna iteration kommer resultera i att vi kan stoppa en låda inuti en annan. Därför kan vi bara räkna antalet iterationer detta utförs tills den ursprungliga listan med lådor är tom.

Mer ingående beskrivning:

Algoritmen itererar över listan med lådor (boxes) tills den är tom. Varje gång lägger vi första elementet i en lista kallad unNestables som kommer att hålla alla de största lådorna som inte går att stoppa i varandra (vi måste börja nån stans, därför stoppar vi in just den första, och antar att det kommer vara en stor låda). Sedan går vi igenom varje låda i boxes och jämför den med varje låda i unNestables.

Om lådan i boxes får plats i den i unNestables kan det inte vara en "stor" låda och vi kollar istället då direkt på nästa låda i boxes.

Om lådan i unNestables får plats i den i boxes tar vi bort lådan i unNestables och lägger till den från boxes istället, eftersom vi har hittat en större låda, och den gamla i unNestables är då inte längre "störst".

Om lådan i boxes inte går att stopppa in i någon av boxarn i unNestables på något sätt kan vi lägga till i unNestables, eftersom vi har hittat en ny "stor" låda som inte går att nästla.

När vi har itererat över alla lådor i boxes och alla i unNestables minskar vi boxes med innehållet i unNestables. För att detta ska göras effektivt används en temporär lista restBoxes som får alla lådor som inte hamnar i unNestables, och istället sätts lika med boxes, för att slippa utföra operationen $boxes \leftarrow (boxes - unNestables)$ (som skulle ta lång tid att utföra eftersom vi då måste gå ingeom alla element i boxes och jämföra dem med unNestables för att ta bort dem). Genom att räkna antalet iterationer som while-slingan körs vet vi hur många lådor vi som mest kan stoppa i varandra.

Algorithm: nestBoxes

```
Input: Lista med lådor boxes[1..n]
Output: Maximala antalet nästlade lådor

unNestables[]
amount ← 0

while boxes is not empty do
    unNestables.add(boxes[1])
    restBoxes[]
    for i ← 1 to length(boxes) do
        INV: Varje låda i unNestables passar inte boxes[1 .. i-1]

        big ← true
        for j ← 1 to length(unNestables) do
            INV: boxes[i] passar inte i unNestables[1 .. j-1] och unNestables[1 .. j-1] passar
            inte i boxes[i]

            if boxes[i] ≠ unNestables[j] then
                if boxes[i] fits unNestables[j] then
                    big ← false
                    break

                else if unNestables[j] fits boxes[i] then
                    unNestables.remove(unNestables[j])
                    restBoxes.add(unNestables[j])
                end
            end
        end
        if big = true then
            unNestables.add(boxes[i])
        else
            restBoxes.add(boxes[i])
        end
    end
    boxes ← restBoxes
    unNestables.clear
    amount ← amount + 1
end
return amount
```

Algorithm: fits

```
Input: Två lådor b1, b2

fits ← (b1.min < b2.min ∧ b1.med < b2.med ∧ b1.max < b2.max)
return fits
```

Detta fungerar eftersom i början av varje iteration av den yttre for-slingan, kommer unNestables att innehålla de största de största lådorna som inte går att stoppa i varandra och som inte kommer att få plats i någon utav lådorna i boxes[1 .. i-1], där i iterationen i slingan. Detta i sin tur fungerar eftersom i början av varje iteration av den inre for-slingan, kommer boxen man kollar på ur boxes, dvs. boxes[i], inte kunna stoppas i någon utav boxarna man tidigare jämfört med i unNestables, dvs. unNestables[0 .. j-1]. Detta är precis vad invarianterna säger och är det som måste stämma för att algoritmen ska vara korrekt. Eftersom dessa invarianter är sanna betyder det också att varje iteration av while-slingan kommer at gälla eftersom vi alltid tar bort det som finns i unNestables från boxes. Det spelar ingen roll vilken av lådorna i boxes vi kan stoppa i en annan ur unNestables. Eftersom vi endast är ute efter det maximala antalet lådor som går att nästa i varandra, räcker det att veta att någon kan stoppas i så vi kan öka antalet med ett och gå vidare till nästa "nivå".

Komplexitet: While-slingan kommer i värsta fall köras n iterationer, dvs om alla boxar kan nästlas inuti varandra. Den yttre for-slingan loopar över alla lådor i boxes, som i värsta fall körs n gånger. Den inre for-slingan kommer i värsta fall köras n gånger, om boxes innehåller boxar där ingen går att stoppa i varandra. Funktionen fits tar konstant tid eftersom vi alltid har en låda med tre dimensioner och alla andra operationer i algoritmen är konstanta (insättning och borttagning av element i listorna).

Vi har då tre stycken nästlade loopar som i värsta fall körs n iterationer var, därför blir tidskomplexiteten O(n*n*n) = $O(n^3)$