Mästareprov 1

Algoritmer, Datastrukturer och komplexitet

Hiba Qutbuddin Habib

School of Electrical Engineering and Computer Science (EECS)
Kungliga tekniska högskolan
Sverige
October 2023

Innehåll

1 Billig numrering av hörn i en graf

1.1 Val av algoritmen

För denna uppgiften söks en lämpligt algoritm som använder totalsökning för att lösa ett problem som kan beskrivas på detta sättet:

"En graf "G" med hörnmängd "V" och kantmängd "E" ska numeras sådan att summan$|f(u) - f(v)|, (u, v)E$ **där u och v är två olika hörn i grafen är så litet som möjligt"**

För att lösa ovanstående problem för en allmän graf så valde jag att använda "Backtrack algoritmen". Denna algoritmen kan vara en möjlig lösning för vårt problem då på ett systematiskt sätt testar den alla möjliga kandidater tills den får fram den eftersökt lösning. Det som gör att denna algoritmen kan vara lämplig vid numering av grafen är att den använder backtracking som det framgår från namnet. Alltså om vi utgår med att vi börjar med att numera varje hörn med ett nummer sen kolla vi summan i grafen och vi får att den är 5. Nu genom backtracking kan vi återgå till hörnen som har nummer ett och börjar vi kolla dess grannar och försöka nummera om de, om det är så att redan här vi får en summa som överstiger 5 då kommer denna algoritm inte fortsätta och den återgår igen och testar en annan värde tills den sökte lösningen hittas.

1.2 Komplexitet

För att få fram tidskomplexitet från denna algoritmen kan vi först utgå från att man i första steget bör alla möjliga numreringar för respektiv nod i grafen testas, alltså vi söker efter på hur många sätt kan dessa noder nummreras. OV represtenterar antal noden i grafen så kommer vi få $V!$ olika sätt att numrera dessa noder.

Då vi söker efter den minsta summan, så vid varje numrering beräknas differensen mellan noder som påverkas av hur dessa är sammankopplade. Om vi utgår från en värsta fall där vi har en fullständig graf där varje nod är kopplat till alla andra noder så får vi att varje nod är ansluten till $V - 1$ andra noder(alltså alla andra noder förutom sig själv). Det ger att vi har $V \times (V - 1)/2$ kanter.

Utifrån dess kan vi få fram att tids komplexitet för Backtrack algoritmen är $O(V! \times V^2)$.

1.3 Pseudokod

Algorithm 1

```
function NUMRERING(G)
    n = längd(G)
    min_kostnad = ∞
    effektiv_numrering = []
    function BACKTRACKING(nuvarandenumrering)
        if längd(nuvarandenumrering) = n then
            kostnad = kostnader(nuvarande_numrering, G)
            if kostnad < min_kostnad then
                min_kostnad = kostnad
                effektiv_numrering = nuvarande_numrering
            end if
        else
            for i = 0 to n − 1 do
                if i ∉ nuvarande_numrering then
                    BACKTRACKING(effektiv_numrering + [i])
                end if
            end for
        end if
    end function
    BACKTRACKING([])
    return effektiv_numrering
end function
function KOSTNADER(numrering, G)
    kostnad = 0
    for j = 0 to n − 1 do
        for k ∈ G[j] do
            kostnad += |numrering[j] − numrering[k]|
        end for
    end for
    return kostnad
end function
```

1.4 Korrekthetsbevis

För denna uppgiften korrekthetsbevis för pseudokoden behövs inte utföras dock vi söker efter vad skulle behövs visas i ett korrekthetsbevis.

För att kunna undersöka korrektheten av denna algoritmen så skulle vi behöva visa några saker, av de kan det undersöks om algoritmen verkligen utforskar alla möjliga numeringar av noder samt om den utför en sann differnes räkning. Vi skulle även behöva undersöka om summering utförs på ett rätt sätt och att algoritmen kan spara och returnera korrekt den eftersökta utdata. För backtrack funktione vi behöver även kontrollera att det rekrusiva anropet utförs på ett rätt sätt.

2 Pyramider i pyramider

2.1 Algoritmen

I denna uppgiften söks en effektiv algoritm för att lösa problemet av hur många pyramider av n stycken pyramider kan palceras på varandra maximalt så att den ovanstående pyramid täcker den nedre helt. Från grundläggande matematik vet man att volym av en pyramid kan beskrivas av $(s^2 \times h)/2$ där s är bottenytans sida och h är höjden. För att en pyramid ska kunna placeras ovanför en annan samt täcka den helt så måste den ena vara något större än den andra både i höjden och bredden.

För att kunna lösa den problemet effektiv så kan man tillämpa en algoritm som grundar sig i "Dynamisk programmerings metod (DP)". Dynmaisk programmering kan vara en bra alternativ för denna probelemt eftersom med tillämpning av denna metoden kan vi dela upp problemet i mindre delprobelm, vilket kan göra det effektivt att kombinera de olika dellösningar för att lösa den ursprungliga problemet. Dynamisk programmering är också ett effektivt metod eftersom det kan göra det möjligt att spara lösningar på de delproblem i en vald datastrukur vilket kan vara tidssparande vid lösningar av olika problem då respektiv dellösningar behövs inte lösas igen.

För denna probelemt kommer vi lösa det genom att först sorteras pyramiderna baserat på bottensidan s. Om två olika pyramider har samma storlek bottensida s då sorteras de utifrån höjden/(h, detta leder till att vi får en stigande lista av pyramiderna, där den största kommer att vara placerat i sista plats. Sortering algoritm som används i denna lösningen är Quicksort, algoritmen användas med hjälp av kursboken.[1].

När sortering är färdig så tillämpas dynmaisk programmering metoden för att räkna ut maximala antal pyramider baserat på en tidigare beräkningen av en midre pyramid. Det utföras genom att lista med storlek n där varje position i listan antar värdet 1, då man antar att varje pyramid kan vara den första placerade. Denna listan kommer att uppdateras konstant när en ny pyramid läggs till, och det är alltid max värdet som returners.

[1] Algorithm Design. Jon K, Eva T. Sid 732-734. 1st edition, 2006. Pearson Education, Inc

2.2 Tidskomplexitet

Tidskomplexitet för denna algortimen påverkas av olika faktorer bland de är val av den använda sorterings algortimen. Vid användning av snabba sortering algoritmer såsom mergesort eller Quicksort blir tidskomplexitet för sortering $O(nlogn)$ där n är antal pyramider. Vid användning av mindre effektiva sorterings algoritmen såsom Insertionsort eller Bubblesort kan tidskomplexitet för denna stegen blir $O(n^2$. Då vi söker lösningen som är effektivast och mindre tidskrävande så väljs Quicksort som det nämns ovan.

Tidskomplexitet för hela algoritmen påverkas av intiering av de olika n elementerna i det valde datastukturen. I denna lösningen väljs en lista där de olika n elementerna tilldelas ett startvärde som är 1, vilket kommer att kräva n operationer, alltså tidskomplexitet för denna delen blir $O(n)$.

Den sista operationen av denna algoritmen är att jämföra de olika pyramiderna efter sortering. Det kommer att ledan till en tidskomplexitet som är $O(n^2)$, då i denna funktionen kommer att den att innehålla två olika lopppar, där i värsta fall kommer vi behöva gå genom alla n elementer i varje loop.

Utifrån dessa operationen kommer vi att få följande tidskomplexitet $O(nlogn)+ O(n) + (O(n^2)$. Eftersom av alla dessa termer, vid stora värde kommer termen $O(n^2)$ att dominera så får vi tidskomplexitet som tillhör $O(n^2)$.

2.3 Pseudokod

Algorithm 2

```
Struktur Pyramid:
    s, h
function PYRAMIDSORTERING(pyramider)
    Sorterings algoritm(Quicksort) anropas, sortering sker baserat på s, sedan h
end function
function PYRAMIDPLACERING(pyramider)
    PYRAMIDSORTERING(pyramids)
    LISTA ← Lista med alla värden satta till 1
    for i = 1 to pyramids.length do
        for j = 0 to i − 1 do
            if pyramids[j].s > pyramids[i].s and pyramids[j].h > pyramids[i].h then
                DP[i] ← max(DP[i], DP[j] + 1)
            end if
        end for
    end for
    return max(DP)
end function
```

2.4 Korrekthetsbevis

Korrektheten för denna algoritmen kan diskuteras uttifrån pseudokoden ovan. För att en algoritm ska fungera så behöver den ha en korrekt initering samt terminering. Om vi observerar pesudokoden ovan så ser vi att den startar med en grundläggande defintiton för vad en pyramid är, i det fall beskrivas pyramid med höjden h och bassidan s. Terminering av algoritmen sker genom att max antal pyramider som kan placeras på varandra returneras, vilket gör att algoritmen returnerar svarapå den problemet som den är strukturerad för att lösa. Sortering av pyramider innan jämförelse samt placering gör det även lättare och mer tidssparande vid senare del.

Algoritmen ovan bygger på dynamisk problemlösning som gör det möjligt att lösa delprobelm samt spara lösningar för de delprobelm för att till slut komma fram till den eftersökte lösningen. Utifrån denna principen skapas en lista för att bevara antal pyramider som kan placeras på varandra och samtidigt uppdateras denna resultat ifall vi hittar en annan lösning. Till en början intieras denna lista till 1 då tanken att varje pyramid kan vara en botten pyramid. Genom for loopar som jämför pyramiderna med varandra baserat på h och s så uppdateras listan konstant samt genom att vi använder max som ser vi till att vi har den största värde från listan.

Att kontrollera dessa aspekter i algortimen kan vi se att den är konstruerade för att lösa den presenterade problemet, samt det samhangande ordning mellan de olika funktionen i pesudokoden gör den logisk och tyder på att med hjälp av den kan vi få fram en eftersökt lösning.