Algoritmer og Datastrukturer 2

Aflevering 2

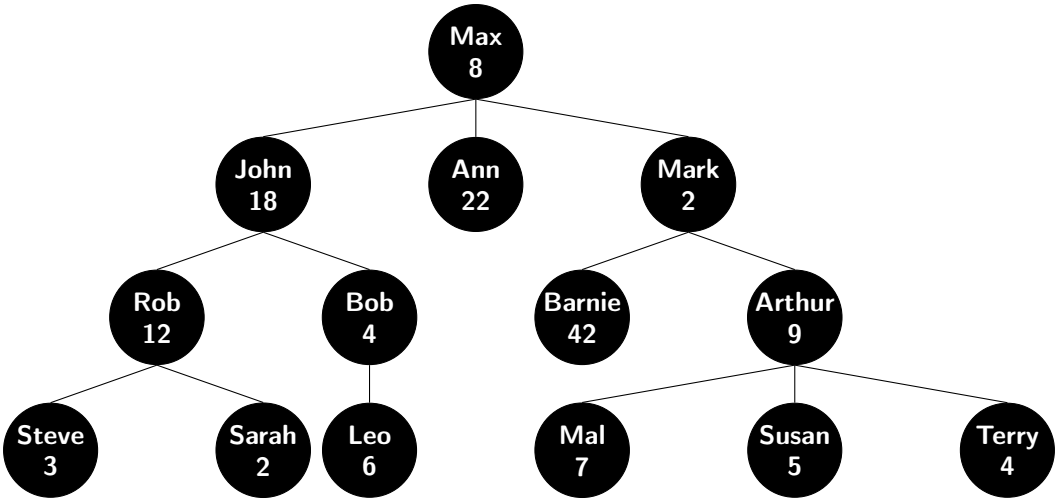
Kristian Gausel[1], Lasse Alm[2], and Steffan Sølvsten[3]

[1]201509079@post.au.dk
[2]201507435@post.au.dk
[3]201505832@post.au.dk

25. april 2016

1 Problem 15-6

Der er givet en træstruktur, der beskriver strukturen af ansatte i et firma. Dette træ benytter sig af "*venstre barn, højre søskende*"repræsentationen. Hver knude, repræsenterende en ansat, indeholder den ansattes navn (*name*) og en vurdering af deres munterhed (*conviviality*).



Figur 1: Eksempel på træstruktur. Her er "*venstre barn, højre søskende*"strukturen ikke vist.

Der gives til opgave, at danne en gæsteliste til en fest, hvori conviviality maksimeres, men under den restriktion, at en ansat ikke bør besøge festen, hvis dens nærmeste overordnede. I træstrukturen betyder dette, at ikke en knude og knudens børn kan tilføjes til listen samtidig.

1.1 Algoritme

Idet det ikke er lovligt at inkludere roden og dens børn, så bør den optimale løsning for et subtræ findes blandt de to muligheder. Hertil sammenlignes summen af conviviality i den optimale løsning i alle børn med conviviality af roden og de optimale løsninger i alle dens børnebørn. Dette giver anledning til følgende rekursionsligning.

$$r(n) = \begin{cases} max\left(0, n.conviviality\right) & \text{hvis } n \text{ er et blad} \\ max\left(\sum_{c \in n.children} r(c), \left(\sum_{g \in n.grandchildren} r(g)\right) + n.conviviality\right) & \text{else} \end{cases}$$

For et kald til en knude n skal først beregnes løsningen indeholdende de optimale løsninger af børnenes undertræer og løsningen indeholdende knuden og de optimale løsninger af børnebørnenes undertræ. Beregningen af disse løsninger indeholder både værdien for conveviality og listen af ansatte.

Hertil summeres først den optimale løsning for alle børnene ved brug af et rekursivt kald til samme algoritme. Herefter gøres samme for børnebørnene. Den summerede løsning for børnene sammenlignes med den summerede løsning for børnebørnene inklusive knuden. Løsningen med den største værdi af conveviality returneres som den optimale løsning.

Algoritmen benytter sig af *memoization*, hvilket undgår genberegningen af subproblemer. Hertil gemmes værdien af conviviality og listen af ansatte for den optimale løsning til et undertræ i dets rod.

Pseudokode for algoritmen kan findes i Appendix A.

1.2 Korrekthed

Løsning for et blad, som giver den maksimale conviviality værdi i dets undertræ, indeholder endten den selv eller ingen knuder. Såfremt at dens convivality rating er negativ indeholder løsningen ingen knuder.

I løsningen for en knude antages det, at løsningen udregnet for og gemt i dets børn og børnebørn er optimal. For at bestemme den optimale løsning af hele det nye undertræ, bør det bestemmes, hvorvidt knuden skal inkluderes i løsningen. Såfremt, at knuden inkluderes, så ekskluderer den alle dens direkte børn fra at være del af løsningen. Derfor må resten af den optimale løsning kunne findes i alle knudens børnebørn. Modsat, hvis knuden bør ekskluderes fra den optimale løsning, så kan den optimale løsning bestemmes som samlingen af optimale løsninger i knudens direkte børn.

Den optimale løsning til hele undertræet er den, hvori den samlede conviviality er højest. Derfor kan den optimale løsning bestemmes ved følgende

$$\sum_{c \in n.children} r(c) \geq \sum_{g \in grandchildren} r(g) + n.conviviality$$

$\rightarrow$ Vælg løsning indeholdende børnenes løsning

$$\sum_{c \in n.children} r(c) < \sum_{g \in grandchildren} r(g) + n.conviviality$$

$\rightarrow$ Vælg løsning indeholdende børnebørn og knuden selv

Idet algoritmen ved rekursion først får beregnet alle børnenes værdier optimalt, så kan det konkluderes, at algoritmen korrekt bestemmer den optimale løsning for hele problemet.

1.3 Tidskompleksitet

Uden brug af *memoization* skal løsningen for et undertræ genberegnes flere gange gennem rekursionen. Derimod ved at gemme den bedste løsning for et subproblem er det muligt at nedsætte dette til en tidskompleksitet på $O(n)$.

Efter første løsning af et undertræ vil genbereningen heraf kun tage $O(1)$ tid. Første beregning af den optimale løsning af et undertræ bruger et gennemløb af alle optimale løsninger i børnenes undertræ og børnenbørnenes undertræ. Dette betyder, at en knude kun bliver kaldt under et gennemløb to gange, først som barnebarn og derefter som barn. Med garanteret kun to kald vil hver knude kun udføre præcis $2O(1) = O(1)$ tid at bearbejde. Idet n knuder skal gennemgås til at opbygge den optimale løsning til hele problemet, så er tidskompleksiteten for algoritmen være $O(n)$.

Litteratur

[1] Cormen, Thomas H. mfl.: *Introduction to Algorithms*, 3rd ed.

[2] Kristian, Gausel mfl.: *Python implementation with test cases* `https://github.com/yurippe/School/tree/master/dADS/15-6%20Planning%20a%20company%20party`

A FindMaxRating Pseudokode

Kode 1: FindMaxRating

```
0  FindMaxRating(node)
1      //The current subtree is already solved
2      if node.childSum ≠ NIL ∧ node.childrenInvited ≠ NIL
3         //Returns a tuple (node.childSum, node.childrenInvited)
4         return node.childSum and node.childrenInvited
5
6      //The current node is a leaf, and is thus solved
7      if node.left == NIL
8          if 0 ≤ node.conviviality
9              node.childSum = node.conviviality
10             node.childrenInvited = [node.name]
11         else
12             node.childSum = 0
13             node.childrenInvited = []
14         //Returns a tuple (node.childSum, node.childrenInvited)
15         return node.childSum and node.childrenInvited
16
17     //Sum of all subtrees with root in children
18     currentChild = node.left
19     do
20         maxofcurrent = FindMaxRating(currentChild)
21         maxChildrenSum += maxofcurrent[0]
22         childrenInvited += maxofcurrent[1]
23         currentChild = currentChild.right
24     while currentChild ≠ NIL
25
26     //Sum of all subtrees with root in grandchildren
27     currentChild = node.left
28     do
29         currentGrandChild = currentChild.left
30         if currentGrandchild ≠ NIL
31             do
32                 maxofcurrent = FindMaxRating(currentGrandChild)
33                 maxGrandchildrenSum += maxofcurrent[0]
34                 grandchildrenInvited += maxofcurrent[1]
35                 currentGrandchild = currentGrandChild.right
36             while currentGrandchild ≠ NIL
37         currentChild = currentChild.right
38     while currentGrandChild ≠ NIL
39
40     //The best subtree solution is saved and returned
41     if maxChildrenSum < maxGrandchildrenSum + node.conviviality
42         node.childSum = maxGrandchildrenSum + node.conviviality
43         node.childrenInvited = grandchildrenInvited.append(node.name)
44     else
45         node.childsum = maxChildrenSum
46         node.childreninvited = childrenInvited
47
48     //Returns a tuple (node.childSum, node.childrenInvited)
49     return node.childSum and node.childrenInvited
```