Assignment 4 - AlgDat

Nicklas Warming Jacobsen
Christian Enevoldsen
Simon Warg
Robert Rasmussen

27. maj 2014

Task 1

Datastruktur

Datastrukturen for vores Online Incremental Connectivity graf gør brug af Path Compression metoden og Rank metoden, vi kan tillade os at bruge Path Compression da det ikke bliver fjernet kanter i grafen. Ved Path compression bliver parent'en, hos de elementer der indgår i en Find-Path (dvs. også i Query), substitueret med dis-join sættets repræsentant, hvilket betyder at den næste query bliver hurtigerer hvis nolge af de samme elementer indgår. Rank metoden er en metode som forgår under Link og Make-Set operationenerne. Hver gang der laves et nyt dis-join set med ét element via Make-Set bliver rank for elementet sat til 0. Under Link operationen kan der ske to forskellige udfald: 1; I tilfælde af at de to elementer der skal linkes har samme rank, bliver et arbitrært element valgt, det valgte elements rank bliver sat én op og det andet elements parent bliver sat til det valgte element. 2; I tilfælde af at de to elementer der skal linkes ikke har samme rank, bliver elementet med størst rank sat som det andet elements parent. Ved at bruge Rank og Path Compression metoderne bliver køretiden for en operation $O(lg^*(n))$ og $\Omega(1)$.

Algorithm 1 Online incremental connectivity

```
function MAKE-SET(x)
    x.parent = x
    x.rank = 0
end function

function FIND-SET(x)
    if x.parent ≠ x then
        x.parent = Find-Set(x.parent)
    end if
    return x.parent
end function

function QUERY(x, y)
    return Find-Set(x) == Find-Set(y)
end function

function LINK(x, y)
    if x.rank < y.rank then
        x.parent = y
    else
        y.parent = x
        if x.rank == y.rank then
            x.rank = x.rank + 1
        end if
    end if
end function
```

Task 2

Trods stor indsats har vi ikke været i stand til at løse Task 2, og derved heller ikke Task 3. Vi kommer derfor i steden med nogle af de tanker vi har gjort os.

Kørselstid

Antagelse at kørelsestiden afhænger af unlink operationerne

Algoritmen skulle køre i $O(m \cdot lg^*(n))$ tid, og m er antallet af pipes, hvilket antyder at tiden afhænger af antallet af unlink operationer (Da der ikke kan være flere unlink operationer end pipes). Hvis dette er korrekt vil bedst case være at der ingen unlink operationer er. Det betyder også at vi maks kan lave $k \cdot m$ dis-join operationer, da hver operation tager $O(lg^*(n))$ tid. Hvis algoritmen får en n liste af elementer, en tom liste af unlink operationer og en x liste af forbindelser, burde algoritmen retunerer de nuværende forbundede komponenter. Men da alle elementer ikke nødvendigvis er indholdt i listen af forbindelser, blive vi nød til at gennemgå listen af elementer. Hvis vi gør dette bliver tiden afhængig af n (hvilket er et problem).

Antagelse af kørelsestiden afhænger af antallet af pipes m

Hvis kørelsestiden strikt er bundet til antallet af pipes, vil best case være at der ingen forbindelser er hvilket giver en kørelsestid på $O(0 \cdot lg^*(n)) = O(1)$, hvilket også betyder at køresestiden er $\Theta(m \cdot lg^*(n))$ (hvilket ikke er i modstrid hvad opgaven siger, men ikke er præcis hvad opgaven siger). Vi vil også nemt kunne lave en algoritme der tager højde for dette lower bound, med logikken:

1. Hvis antallet af pipes m er $m = 0$, så retunerer længden af elementer
2. Hvis antallet af pipes m er $m > 0$, så beregn på normal vis

Exam outline - Nicklas W. Jacobsen

Et binært søgetræ er en datastruktur, som tillade at lave søgninger i $O(lg(n))$ tid. Det gælder at hvis x og y er elementer i samme binære søgetræ så er $x \leq y$ hvis x ligger til venstre for y, og det gælder at $x \geq y$ hvis x ligger til højre for y.

Indsættelse af element i et binært søgetræ

Ved indsættelse af et nyt element, sammenligner man det nye element z med elementerne i træet, hvis z er mindre så gå til venstre og hvis z er større gå til højre. Når man rammer bunden af træet indsættes det nye element z: Da

Algorithm 2 Tree-Insert

```
function TREE-INSERT(T, z)
    y = NULL
    x = T.root
    while x != NULL do
        y = x
        if z.key < x.key then
            x = x.left
        else
            x = x.right
        end if
    end while
    if y == NULL then T.root = z
    else if z.key < y.key then y.left = z
    elsey.right = z
    end if
end function
```

løkken køre én gang per højde i træet, tager funktion $O(lg(n))$ tid.

Exam outline - Robert Rasmussen

Points

- What is AVL trees
  - They're height balanced binary search trees
- Searching
  - To find the next of previous node in an AVL tree can be done in constant time, but when going from a node which a further apart than just next to it, it's then suddenly 2*log(n) time.
- Insertion
  - Whenever you insert a node in an AVL tree, you have to check it's consistency since it needs to be height balanced.
- Deletion
  - The same thing applies when nodes are deleted, you need to make sure the height is balanced, if not then make the right adjustments so it is.

Problem Instance

I'll show how an AVL tree is constructed by inserting a sequence of numbers and then do the necessary changes to maintain consistency in the tree.