Homework 6 Solutions

December 2, 2019

4.16

Section 4.5.2 describes a way of storing a complete binary tree of n nodes in an array indexed by $1, 2, \ldots, n$.

(a) Consider the node at position j of the array. Show that its parent is at position $\lfloor j/2 \rfloor$ and its children are at $2j$ and $2j+1$ (if these numbers are $\leq n$).

(b) What the corresponding indices when a complete d-ary tree is stored in an array?

Figure 4.16 shows pseudocode for a binary heap, modeled on an exposition by R.E. Tarjan. The heap is stored as an array h, which is assumed to support two constant-time operations:

- $|h|$, which returns the number of elements currently in the array;
- h^{-1} , which returns the position of an element within the array.

The latter can always be achieved by maintaining the values of h^{-1} as an auxiliary array.

(c) Show that the makeheap procedure takes $O(n)$ time when called on a set of n elements. What is the worst-case input? (Hint: Start by showing that the running time is at most $\sum_{i=1}^{n} \log(n/i)$.)

(d) What needs to be changed to adapt this pseudocode to d-ary heaps?

Figure 4.16 Operations on a binary heap.

```
procedure insert(h, x)
bubbleup(h, x, |h| + 1)

procedure decreasekey(h, x)
bubbleup(h, x, h^{-1}(x))

function deletemin(h)
if |h| = 0:
   return null
else:
   x = h(1)
   siftdown(h, h(|h|), 1)
   return x

function makeheap(S)
h = empty array of size |S|
for x ∈ S:
   h(|h| + 1) = x
for i = |S| downto 1:
   siftdown(h, h(i), i)
return h

procedure bubbleup(h, x, i)
(place element x in position i of h, and let it bubble up)
p = ⌈i/2⌉
while i ≠ 1 and key(h(p)) > key(x):
   h(i) = h(p);  i = p;  p = ⌈i/2⌉
h(i) = x

procedure siftdown(h, x, i)
(place element x in position i of h, and let it sift down)
c = minchild(h, i)
while c ≠ 0 and key(h(c)) < key(x):
   h(i) = h(c);  i = c;  c = minchild(h, i)
h(i) = x

function minchild(h, i)
(return the index of the smallest child of h(i))
if 2i > |h|:
   return 0 (no children)
else:
   return arg min{key(h(j)) : 2i ≤ j ≤ min{|h|, 2i + 1}}
```

Solution

(a) Let node n_j at position j of array. In the complete binary tree from top to bottom, n_j is at vertical depth: a (let root node be at depth 0). The number of nodes above n_j is $1 + 2^1 + 2^2 + \ldots 2^{a-1} = 2^a - 1$. From left to right, n_j is at horizontal order b (Number of nodes at left of n_j) and $0 \le b \le 2^a - 1$.

Now we have $j = 2^a - 1 + b + 1 = 2^a + b$.

Known the parent of n_j is at vertical depth $a - 1$ and horizontal order $\lfloor b/2 \rfloor$. So, n_j's parent is at position:

$2^{a-1} + \lfloor b/2 \rfloor = \lfloor j/2 \rfloor$

The children of n_j are at vertical depth $a + 1$, horizontal order $2b$ and $2b + 1$. So, the positions are

$$2^{a+1} + 2b = 2j$$

And

$$2^{a+1} + 2b + 1 = 2j + 1$$

(b) Its parent is at position $\lceil (j-1)/d \rceil$

Its children are at positions $jd - d + 2, jd - d + 1, \ldots, jd + 1$

(c) Based on **Figure** 4.16, minchild function takes O(1) time. Under the worst case, siftdown function would sift down all elements from position i (depth level $= \log i$) to position n (depth level $= \log n$). At most $\log n - \log i$ siftdown operations would be implemented, which take $O(\log n - \log i) = O(\log(n/i))$. Last, procedure makeheap calls siftdown function n times with input $i = n \ldots 1$. So, total running time of makeheap is

$$\begin{aligned} & O(\log(n/1) + \log(n/2) + \log(n/3) + \cdots + \log(n/i)) && (1)\\ =& O(\sum_{i=1}^{n} \log(n/i)) && (2)\\ =& O(\log \frac{n^n}{n!}) && (3) \end{aligned}$$

By Stirling's formula, $n! \approx \sqrt{2\pi n}(\frac{n}{e})^n$, get

$$\log \frac{n^n}{n!} \approx \log(\frac{e^n}{\sqrt{2\pi n}}) = n \log e - \log(\sqrt{2\pi n}) = O(n)$$

The worst-case input would be an array corresponding to a binary tree which all parents are larger than their children, such as an array sorted in decreasing order.

(d) Change all number 2 into suitable formula of d, and add d as an input argument for each function.

For example,change function minchild(h,i) into:

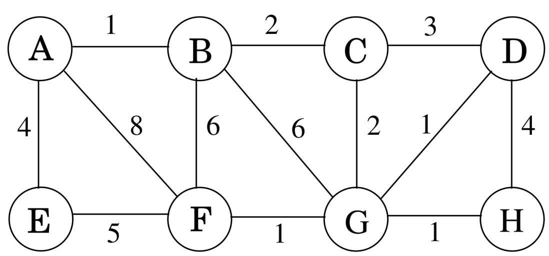
```
MINCHILD(h, i, d)
if id − d + 2 > |h|:
        return 0 (no children)
else:
        return argmin{key(h(j)) : id − d + 2 ≤ j ≤ min{|h|, di + 1}}
```

Change function bubbleup into

```
BUBBLEUP(h, x, i, d)
p = ⌈(i − 1)/d⌉
while i ≠ 1 and key(h(p)) > key(x):
        h(i) = h(p); i = p; p = ⌈(i − 1)/d⌉
h(i) = x
```

5.2

Suppose we want to find the minimum spanning tree of the following graph.



(a) Run Prims algorithm; whenever there is a choice of nodes, always use alphabetic ordering (e.g., start from node A). Draw a table showing the intermediate values of the cost array.

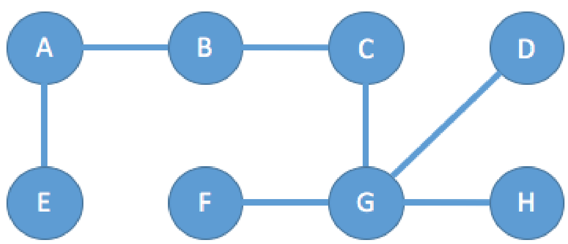
(b) Run Kruskal's algorithm on the same graph. Show how the disjoint-sets data structure looks at every intermediate stage (including the structure of the directed trees), assuming path compression is used.

Solution

(a)

Set S	A	B	C	D	E	F	G	H
{}	$0/nil$	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$1/A$	∞/nil	∞/nil	$4/A$	$8/A$	∞/nil	∞/nil
A,B			$2/B$	∞/nil	$4/A$	$6/B$	$6/B$	∞/nil
A,B,C				$3/C$	$4/A$	$6/B$	$2/C$	∞/nil
A,B,C,G				$1/G$	$4/A$	$1/G$		$1/G$
A,B,C,G,D					$4/A$	$1/G$		$1/G$
A,B,C,G,D,F					$4/A$			$1/G$
A,B,C,G,D,F,H					$4/A$			

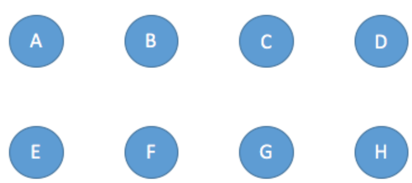
The tree looks like:



(b) – Sort the edges based on their weights:

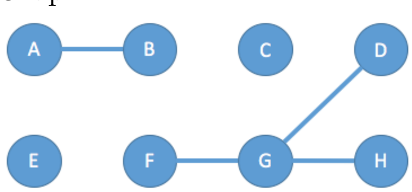
Edge	A-B	F-G	D-G	G-H	B-C	C-G	C-D	A-E	D-H	E-F	B-F	B-G	A-F
Weight	1	1	1	1	2	2	3	4	4	5	6	6	8

– Do makeset(A), makeset(B), ..., makeset(H).
Graph:

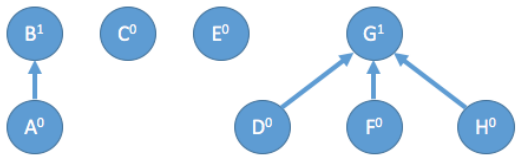


Path-compressed tree structure:

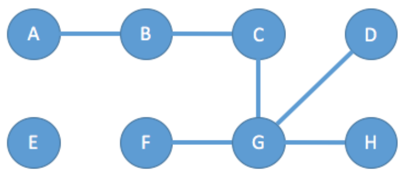
– Pick edges AB, DG, FG, GH. Do union(A, B), union(D, G), union(F, G),union(G, H).
Graph:



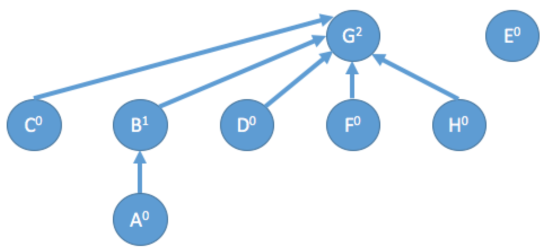
Path-compressed tree structure:



– Pick edges BC, CG. Do union(B, C), union(C, G). Graph:

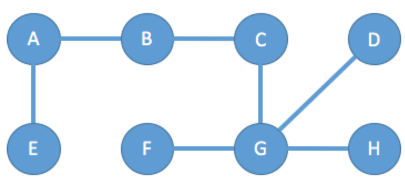


Path-compressed tree structure:

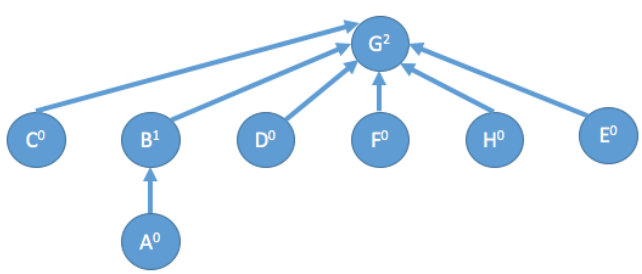


– Pick edge CD. find(C) = find(D). No update.

– Pick edges AE, DH. Only update AE. Do union(A, E). Graph:



Path-compressed tree structure:



5.3

Design a linear-time algorithm for the following task.

Input: A connected, undirected graph G.
Question: Is there an edge you can remove from G while still leaving G connected?

Can you reduce the running time of your algorithm to O($|V|$)?

Solution 1 in $O(|V| + |E|)$ time:

Perform DFS until a back edge is found. Then remove any edge on the cycle will leave G connected. If no such a back edge is found, then no such an edge that can be removed and still leave the graph connected. This takes $O(|E| + |V|)$ time due to DFS.

Solution 2 in $O(|V|)$ time:

Count the total number of edges on the adjacency list. If exactly $2(|V| - 1)$ edges are found, then the graph is a tree and removing any edge would disconnect the graph; If $2|V|$ edges have been counted, stop. Such an edge exists (though which edge is not clear). The constant 2 is due to each edge showing up twice in the adjacency list for an undirected graph. The total runtime is $O(|V|)$.