CS345A

Algorithms-II

Deepanshu Bansal(150219)
Mukul Chaturvedi(150430)

Q-1 Binary search under insertions

Let k = [log(n + 1)], and let the binary representation of n be $<n_{k-1}, n_{k-2}, \ldots, n_0>$. We have k sorted arrays $A_0, \ldots, A_{k-2}, A_{k-1}$ where for i = 0, 1, ..., k – 1, the length of A_i is 2^i. Each array is either full or empty, depending upon whether $n_i = 1$ or $n_i = 0$, respectively. The total number of elements held in all k arrays is, therefore, $\sum_{i=0}^{k-1} 2^i =$ n.

Algorithms

Search

In the search algorithm we search for the value in the sorted arrays one by one until we do not find it.

Insert

We create a new sorted array of size 1 containing the new element to be inserted. If array A_0 (which has size 1) is empty, then we replace A_0 with the new sorted array. Otherwise, we merge sort the two arrays into another sorted array of size 2 empty the A_0 array. If A_1 (which has size 2) is empty, then we replace A_1 with the new array otherwise we merge sort the arrays as before and empty A_1 and continue doing this process until we find an empty array. Since array A_i is of size 2^i , if we merge sort two arrays of size 2^i each, we obtain one of size 2^{i+1} , which is the size of A_{i+1} . Thus, this method will result in another list of arrays in the same structure that we had before with just updating different arrays.

Analysis

Insert

We know merging takes $O(m+n)$ time to merge two sorted arrays of size m and n. Now let's compute the time of insertion of new element let's say the empty array is at position m that is all other A_0 to A_{m-1} are full. Cost of merging two arrays of size 2^j is $2*2^j$ .Thus adding this new element takes $\sum_{j=0}^{m-1} 2*2^j = O(2^m)$ time.

Now from the binary counter example of class we know that bit at position m flips atmost $n/2^m$ times. So total time for m insertions will be

$$\sum_{m=0}^{k-1} 2^m * n/2^m = O(nk) = O(n\log n) \text{ as } k = [\log(n+1)]$$

Hence any sequence of n insertions takes **O(nlogn)** time.

Search

We will simply search each of the filled arrays till find the answer. Now, the worst in a searching a x size array is $O(\log(x))$. So, the worst time in finding a value will be when we do not find it any of the filled array and all the k arrays (A_0, A_1, A_2,...., A_{k-1}) are full,where $k = [\log(n+1)]$.

$$T(n) = O(\log(2^0) + \log(2^1) + \dots\dots\dots\dots + \log(2^{k-2}) + \log(2^{k-1}))$$

$$= O((0) + (1) + \dots\dots + (k-2) + (k-1))$$

$$= O(k(k-1)/2) = O(k^2)$$

$$= O((\log(n+1))^2) = O(\log^2 n)$$

Thus, the worst case running time is **$O(\log^2 n)$**.

Q-2 Dynamic Reachability

Algorithm for function

```
**ProcedureUpdate** – R(i, j)
if (R[i] == true) and (R[ j] == false) then
        R[ j] ← true;
        for each neighbour q of j do
                ProcedureUpdate – R[ j, q];
        end
end
```

Part-1 $\phi(.)$

Our potential function is the sum of degrees of all vertices which are not reachable from s.

$$\phi(G') = c \sum_{R[v]==false} deg(v)$$

Here deg(v) denotes the degree of vertex v and G' is the graph at any point of time. We have $\phi(G')$ >= 0 at all points as deg(v) >= 0. This potential function is zero at beginning as at starting point no vertex is reachable from s and no edge is present in graph ie. deg(v) = 0 for all vertices hence $\phi(G^0) = 0$ where G^0 denotes the starting graph. Thus this is valid potential function.

Part-2 Actual Cost

Actual cost of edge insertion. Let edge inserted be (i, j)

Case-1

if R[i] == false, Cost = c (some constant)

We are not entering the ProcedureUpdate.

Case-2

if R[i] == true && R[j] == true, Cost = c (some constant)

We are not entering the ProcedureUpdate.

Case-3

if R[i] == true && R[j] == false, Cost = $c + \sum_{R[v]==false \,\&\&\, v \text{ is reachable from } j} deg(v)$

We are entering the ProcedureUpdate and going to visit all those vertices which are reachable from j and who are not reachable from s.

Part-3 Amortized Cost

We have $\Delta(\phi)$ as

Case-1

if R[i] == false, $\Delta(\phi) = c$ (some constant)

Degree of vertex i increases by 1 due to added edge.

Case-2

if R[i] == true && R[j] == true, $\Delta(\phi) = 0$

Degree of i increases by one due to added edge but won't change potential because vertex i is reachable from s.

Case-3

if R[i] == true && R[j] == false, $\Delta(\phi) = \phi(G^{new}) - \phi(G^{old})$

$\Delta(\phi)$ = Σdeg(v) st. v is not reachable from s after insertion of edge (i, j) -Σdeg(v) st. v is not reachable from s before adding this edge

$\Delta(\phi)$ = -(Σdeg(v) st. v is not reachable from s but reachable from j)

$$\Delta(\phi) = -\sum_{R[v]==false\ \&\&\ v\ is\ reachable\ from\ j} deg(v)$$

	Actual Cost	Δφ	Amortized Cost
Case-1	c	c	2c
Case-2	c	0	c
Case-3	$c + \sum_{R[v]==false\ \&\&\ v\ is\ reachable\ from\ j} deg(v)$	$-\sum_{R[v]==false\ \&\&\ v\ is\ reachable\ from\ j} deg(v)$	c

Hence the amortized cost of handling an edge insertion is indeed **O(1)**.

Q-3 Tinkering with the marked nodes

All the bounds on the various operations of fibonacci heap will still hold. In this case we attach an extra value to every node representing number of children which has been cut of the given node for example if a node has lost one child it's additional value will be 1 and if it loses second child value becomes 2. We say that the node is marked only when 2 of it's child have been cut. So when a marked node loses it's child we cut it, unmark it and add it to the root list.

PROOF

Let us define the potential function for fibonacci heap **H** be

$$\phi(H) = ct(H) + 2cm(H)$$

Where t(H) = number of trees in the root list of fibonacci heap

m(H) = number of marked nodes in fibonacci heap

Amortized Analysis of various operations

Operations	Actual Cost	Change in potential function	Amortized Cost	O(operation)
Decrease Key(H,x,Δ)	c + cm(H_{old}) – cm(H_{new})	cm(H_{new})–cm(H_{old})	c	O(1)
Extract_min(H)	ct(H_{old})	ct(H_{new}-H_{old})	ct(H_{new})	O(logn)
Find_min(H)	c	0	c	O(1)
Merge(H_1,H_2)	c	0	c	O(1)

Explanation for decrease key :

Change in potential function = ct(H_{new})-ct(H_{old})+2m(H_{new}-H_{old}). Now we know that number of nodes getting unmarked is of the same order as increase in the trees in root list so

$\Delta(\phi) =$cm(H_{new})–cm(H_{old}). Also, actual cost of decrease key will include steps of nodes getting unmarked and moving to root list so it is equal to c + cm(H_{old}) – cm(H_{new}).

Thus, amortized cost is equal to constant.

Explanation for extract min:

For extract min operation to be of O(logn) we have to prove that maximum degree of a tree in a fibonacci heap is $O(\log_2 n)$ or we can say that minimum number of nodes in a subtree of root having degree k is a^k where a is some constant.

As discussed in class, let there be a node x of degree k and we store it's children according to increasing order of time of becoming children of x.Degree of v_1 is positive or zero and for $i \geq 2$ $v_i \geq (i-1)$ at the time of it's becoming the child of x, now at any given time $v_i \geq (i-3)$ as it cannot lose more than 2 of it's child and still remain in subtree of x.

Now, let s_k be minimum number of nodes in subtree rooted at node of degree k. Now, $s_0=1, s_1 \geq 2, s_2 \geq 3$ and $s_k \geq 1 + 1 + 1 + \sum_{3 \leq i \leq k} s_{i-3}$. We have seen in class that $s_k \geq F(k)$ and as $F(k) \geq a^k$ where $a = (1+\sqrt{5})/2 \approx 1.618$ is the golden ratio therefore $s^k \geq a^k$.

Thus, we can say that extract min can be done in **O(logn)** amortized time.