9/28/07 6.854: Optional Lecture on Link-Cut Trees

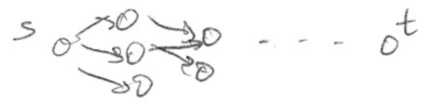
Lecturer: Katherine Lai

Today

- Link-cut Paths
- Extend to Link-cut Trees
- Running Time Analysis

Application

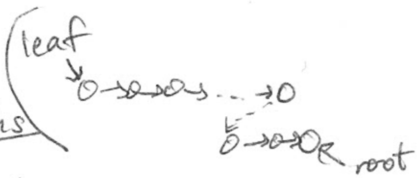
Finding a blocking flow in capacitated graphs



advance
retreat
augment

Two tasks: ① Don't waste work
② Query and update capacities quickly

Link-cut Paths



Assume degree 1
⇒ concatenate and split paths
draw here ←

Link(v,w): make w parent of v
v must be a root
w must be a leaf

[These ops take care of advance and retreat.]

cut(v): cut v off from its parent

find-root(v): returns root of v's path

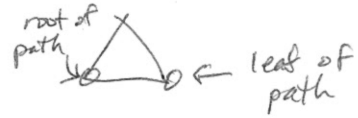
[We will delay discussion of storing capacities until later]

Goal: $O(\lg n)$ time amort. per op.

[Ask: what's a good data structure that has $O(\lg n)$ amort. per op. and can store paths?]

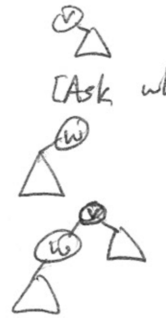
Use splay trees!

Order nodes by depth



link(v,w):
splay(v)
splay(w)
left(v)=w

[Ask what v's splay tree looks like]



cut(v):
splay(v)
left(v)=null

find-root: ≡ find-min
splay(v)
walk left to min
splay the min

Supporting Augment

represent capacities at a node instead of an edge
each node only has at most 1 parent in the path

Let $u(v) = u(v,w)$ where w is v's parent in the path

find-min(v): returns min capacity node in v's path

update(v,x): adds real number x to all capacities in v's path

[Want to take advantage of BST structure.]

Too costly to store node capacities just by themselves

Store differences between capacities for a node and its parent (in the splay tree)

$$\Delta u(v) = \begin{cases} u(v) & \text{if } v \text{ is the root of the splay tree} \\ u(v)-u(y) & \text{if } y \text{ is the parent of } v \text{ in the splay tree} \end{cases}$$

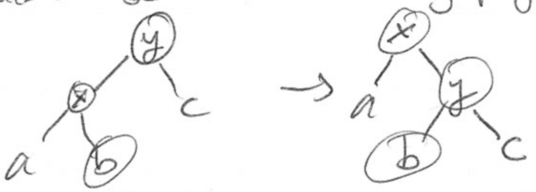
[Intuition: if splaying something, you can walk down from the root to it and calculate its capacity just by adding up the Δu's on the nodes in that path]

On root-to-v path in splay tree,
$u(v) = \sum_{w \in \text{path}} \Delta u(w)$

store min values as well
Let $\min(v)$ = value of min capacity node in subtree rooted at node v
- also too costly by itself

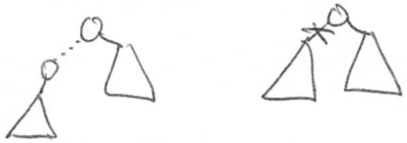
Store $\Delta\min(v) = u(v) - \min(v)$ instead
$\geq 0 \ \forall v$
$\Rightarrow$ can calculate $\min(v)$ with $\Delta\min(v)$ and $u(v)$
$\Delta\min(v)$: stored at node; $u(v)$: calculated from traversal

Δ values can be maintained during splays



do simple arithmetic using known values at involved nodes
O(1) per splay

also true for link, cut



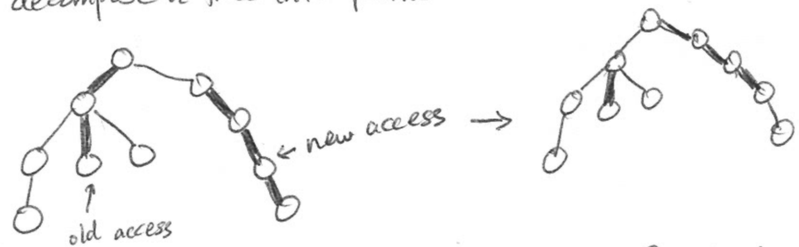
find-min(v):
```
splay(v)
min(v)=u(v)-Δmin(v)
while u(v) ≠ min(v):
    v ← child w of v where min(w)=min(v)
return v
(splay v is optional?)
```

update(v,x):
```
splay(v)
Δu(v) += x
```

[So to support augment, we call find-min to find the min, update using the value, and then call find-min and cut for each node that now has capacity Ø.]

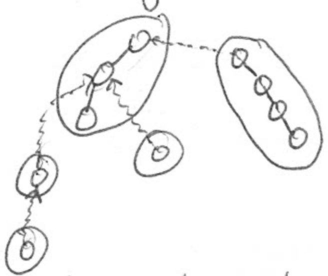
Link-cut Trees

reuse ideas:
decompose a tree into paths



"preferred path": paths connected by preferred edges
a node has ≤ 1 preferred child
still store paths as splay trees; must maintain structure of tree

Auxiliary tree:



 ← splay tree on its contents

path-parent pointers connect roots of the splay trees to the actual parent of the path represented

"super splay" needed to bring a node into the root splay tree for link, cut, etc. ops.
[see example at top of page]

prefered path from root always ends at last item accessed

Access (v)

splay(v) within its splay tree

cut off right subtree of v
and give it a path-parent ptr to v

loop until we get to the root:

w ← path-parent(v)

splay(w)

cut off w's right subtree and have it point to w

right(w) = v

path-parent(v) = null

v = w

splay v

v now root of the aux. tree's root splay tree

link(v,w): v must still be a root of the actual tree
w can be anything

access(v)

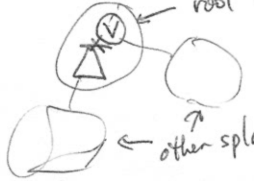
access(w)

left(v) = w

cut(v):

access(v)

left(v) = null



find-~~root~~(v): finds ~~[illegible]~~ root on root-to-v path

access(v)

do find-min in v's splay tree (splay tree find-min)

find-min and update similar

Δ values can still be updated in O(1) more time per op

Running Time

All ops have O(1) calls to access and O(1) calls to splays (outside access)

$O(\text{access}) + O(\text{splay})$

$O(\text{splay}) = O(\lg n)$

$O(\text{access})$: (# splay trees or path changes to the root × (splay)

Technique: Heavy-light decomposition

useful for trees of any degree

Let size(v) = # nodes in v's subtree

Call the edge between v and parent(v) heavy if $\text{size}(v) > \frac{1}{2}\,\text{size}(\text{parent}(v))$ and light otherwise.

Notes:

≤ 1 heavy child per node

light edges on root-to-v path [Ask here] ≤ lg n

Bound # preferred child changes

O(lg n) light edges become preferred

heavy edges harder [Ask for ideas]

Amortized analysis:

sequence of m operations

after all heavy edges become preferred, a heavy edge only gets preferred again if it is unpreferred first

O(lg n) heavy edges become unpreferred per access (light edges would have to get preferred)

at most n−1 heavy edges (chain)

⇒ O(m lg n) + n − 1 heavy edges preferred over m accesses

for m big enough, get O(lg n)

⇒ $O(\lg^2 n)$ bound for access [But I promised you O(log n)!]

More Amortization

Intuition: all those splays in the loop are not that bad

whole auxiliary tree is like one big compartmentalized splay tree, and accessing a node costs the same

size(v) = # nodes under v in auxiliary tree

$\Phi = \sum_v \lg \text{size}(v)$

Each splay only affects v's splay tree; other parts of aux. tree is untouched.

Intermediary splay costs $O(\lg \text{size}(r) - \lg \text{size}(v))$

Total cost is a telescoping sum, and we get

$$O(\lg n - \lg \text{size}(v)) = O(\lg n)$$

link:  only one change + $O(\lg n)$

cut: only decreases potential

$\Rightarrow O(\lg n)$ per link-cut operation

$\Rightarrow O(m \lg n)$ per blocking flow

$\Rightarrow O(mn \lg n)$ for max flow