Amortized Analysis (and Shortest Path Algos)

Last time we saw Kruskal's MST (Min Spanning Tree) Algo.

And a data structure for maintaining disjoint sets over some universe to support the operations

- makeset(x) ← make a singleton set $\{x\}$
- find(x) ← return the "name" of set containing x
- union(x, y) ← merge the sets containing x & y.

Theorem (we proved): (starting from an empty system)

For any sequence of n makesets, F finds and U unions the list-based data structure takes time $O(n + F + U \log n)$.

This is an example of amortized analysis. On average each union takes $O(\log n)$ time. Some unions may take more time, of course, but there can be only few of these expensive unions.

The proof was via the "banker's method".

We gave each element $O(\log n)$ dollars when it ~~was added via makeset.~~ first participates in a union.

$\Rightarrow$ $\boxed{\text{total \# of dollars given } \le 2(\log n) \cdot U}$

because when an element first participates in a union, it lies in a singleton set

$\Rightarrow$ if both sets being unioned together are singletons, we pay $2(\log n)$.

Moreover: each makeset and find takes $O(1)$ time: makeset is trivial

find: we just look up the root pointer for the element.

Finally the unions:

When we union two lists A & B
we have runtime ~~we pay~~ $O(\min(|A|, |B|))$ ← the min of their lengths.

Make the shorter list elements pay \$1 each to pay for this runtime

~~Moreover~~ ⇒ so all runtime is paid for by elements.

~~Hence~~ **Moreover** each element pays at most its $(\log_2 n)$ dollars
because its list size can double at most $(\lg_2 n)$ times

⇒ no element runs out of money.

Hence: we put in $O(u \log_2 n)$ dollars.
And paid for the runtime of the unions using these dollars

⇒ total time for union $\leq O(u \log_2 n)$. ☺

Today we'll see more amortized analysis,
where we average the cost of expensive operations (few)
over the (many) cheap operations that must have preceded them.

Another example of Amortized Analysis

Binary counter.

0000
0001 ← cost = 1.
0010 = 2
0011 = 1
0100 ← cost = 3 because 3 bits changed.

Claim: total cost of n increments (starting from 0) is $\leq 2n$.

Pf: each time you write 1, put \$1 on it.
each time you change $1 \to 0$, use that \$1 to pay for it.

Each increment changes some 1s to 0s, pay using their \$1.
finally converts $0 \to 1$, use \$1 to pay for it.
use \$1 to keep on this new 1.

$\Rightarrow$ \$2 per increment. $2n$ overall. ∎

Another way: define potential $\Phi(k) = \#$ of 1s in binary repr. of k.

Initially $\Phi(0) = 0$. $\Phi(n) \geq 0 \cdot \forall n$.

change in Φ

At each increment, convert some i 1s to 0s. $\Delta\Phi = -i$

convert a single $0 \to 1$ $\Delta\Phi = +1$

$\Delta\Phi = -i+1$.

Actual cost $= (i+1)$.

$\Rightarrow$ Amortized cost $\underset{\text{definition}}{=}$ Actual cost $+ \Delta\Phi = (i+1) + (-i+1) = 2$.

"Pay \$2 from pocket each time. Bank account never goes negative.
All operations paid for, either from pocket or bank.
$\Rightarrow$ if n operations, total cost $\leq 2n$."

Q: What if you do ~~~~ decrements as well? Is amortized cost $O(1)$ still?

Assume no underflow, always remain non-negative

Ans: No. (Exercise: show sequence of incs & decs of length n s.t. cost $= \Omega(\log n)$.)

Amortized

Or total cost is Omega(n log n).

Q: Suppose you do n_1 increments, n_2 decrements starting from all zeros.

Show that amortized cost of incs $= O(1)$ (in fact 2)

decs $= O(\log n_1)$

i.e. total cost $\leq O(n_1) + O(n_2 \log n_1)$

Q: Show that $\sum_{i=1}^{k} i \cdot 2^i = O(1)$.

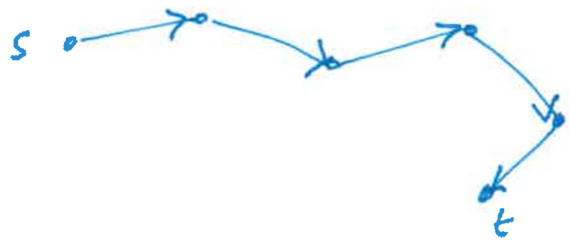
Hint: write the total cost of counter for n increments ($n = 2^k$)

how many times do you pay i?

Shortest Path Algorithms

Given a graph $G=(V,E)$ with lengths $\ell(e)$ on the edges, source s, find shortest paths from s to all other vertices of G.

Let's clarify: given a path P from $\underline{s}$ to $\underline{t} \in V$, $\ell(P) := \sum_{e \in P}' \ell(e)$.



So we want to find, among all these s-t paths, one with smallest length.

<u>Assume</u>: $\forall$ vertex $v \neq s$, $\exists$ a path from s to v, so that things are well-defined.

<u>Some questions</u>:

(1) Is the graph directed or undirected?

(Will not matter, let's say directed. So going from $s \to t$ may have different length than $t \to s$).

(2) Are the edge lengths non-negative, or can they be negative?

For some algos, we assume non-negative, other algos will work even with negative edge lengths, as long as there are no negative length cycles in G.

If negative cycle, then going around this cycle multiple times reduces length of path, so shortest paths can become of length $-\infty$. So we don't allow this possibility.

A Classical algo for Single Source Shortest Paths (SSSP)

(Dijkstra 1959)

(Di) Graph $G=(V,E)$ non negative edge lengths $\ell(e) \geq 0$. $\forall e \in E$

single source vertex $s \in V$. All nodes reachable from s.

Else unreachable nodes have distance $= \infty$

Algorithm

Set $d_s = 0$, $d_v = \infty$ $\forall v \neq s$. ← "estimates" of distance from s.

$A = \emptyset$

while $A \neq V$:

Let $u =$ vertex in $V \setminus A$ with smallest d_u value

$A \leftarrow A \cup \{u\}$.

$\forall v \in V \setminus A$, set $d_v \leftarrow \min\{d_v, d_u + \ell((u,v))\}$

return $\{d_u\}_{u \in V}$. ← can do this only for $v \in$ Neighbors(u)

As always: ① Correctness

② Runtime Analysis.

Fact 1: Let $\delta(u)$ be the shortest path length from $s \to u$.

then $d_u \geq \delta(u)$. ← estimates are always overestimates

Pf: by induction.

Base case: $d_s = 0$, $d_v = \infty \geq \delta(v)$
$= \delta(s)$

overestimates by Induction hypoth.

Induction: new $d_v = \min\{d_v, d_u + \ell((u,v))\}$

but $\delta_v \leq \delta_u + \ell((u,v))$

$\leq d_u + \ell((u,v))$.

☺

Claim 2: $\forall$ vertices in A, $d_u = \delta_u$.

Pf: Again base case is true because distances are non-negative.

For induction: Consider vertex u being added to A.

~~And consider a shortest path P from s to u.~~

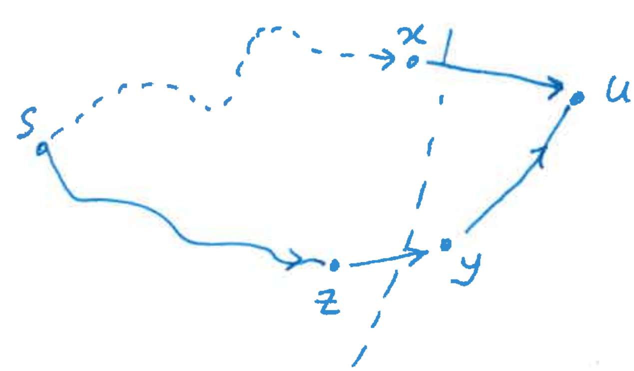
By construction, currently $d_u = d_x + \ell((x,u))$ for some $x \in A$.

By I.H, $d_x = \delta_x$ and so $\exists$ a shortest $s \to x$ path of length d_x. $\Rightarrow \exists$ a $s \to x \to u$ path of length d_u.

We claim this is a shortest $s \to u$ path.

Sps not. Sps $\exists$ a path P shorter, from $s \to u$.

And let y be the first vertex on P not in A (~~when~~ just before u was added).



Then $\delta_y \le \delta_u$ (because all edges are nonnegative) $< d_u$

and $\delta_y = \delta_z + \ell((z,y))$ ~~so at this time~~

$\Rightarrow d_y = d_z + \ell((z,y)) = \delta_y < d_u$

for sake of contradiction

$\Rightarrow y$ would have been chosen by the algo, not u.

A contradiction

Hence the $s \to x \to u$ path is indeed shortest.

And so $d_u = \delta_u$.

This means the d. values for vertices in A <u>are</u> the shortest-path lengths.

Can use this to find paths as well.

Just keep "previous" pointers.

$prev(s) = s$.

And when doing update, if $d_u + \ell((u,v)) < d_v$ then

$$\text{set}\begin{cases} d_v \leftarrow d_u + \ell((u,v)) \\ prev(v) \leftarrow u. \end{cases}$$

And finally $A = V$, so all d values are the shortest path distances!

<u>Runtime?</u>

Need a $\boxed{\text{priority queue}}$ data structure.

At each time have a set of ~~keys~~ elements, each with a ~~key~~. value.

Want the operations:

- $h \leftarrow$ <u>makePQ</u>() make an empty priority queue.
- <u>insert</u>(h, u, i) insert element u with value i
- (u, i) ← <u>min</u>(h) return element with smallest value in h.
- (u, i) ← deletemin(h) return <u>and</u> delete elt with " " " "
- <u>delete</u>(h, u) delete the element u from priority queue h.

Actually: can do deletemin by first finding min(h), and then deleting the returned element u.

So suffices to implement the 4 marked operations.

Using a priority queue data structure, Dijkstra runs as follows:-

Make PQ.
Insert $(s, value = 0)$
$\forall v \neq s$, Insert $(v, value = \infty)$.
while $(A \neq V)$:
. $(u, d_u) \leftarrow$ deletemin
. $\forall v \in Neighbor(u)$, decreasekey$(v, d_u + \ell((u,v)))$
return (u, d_u) for all u.

Assume decreasekey will leave old d_v value if $d_u + \ell(u,v) > d_v$

$\Rightarrow$
1 makePQ
n inserts
n deletemins
m decreasekeys.

Standard Binary heap gives: $O(\log n)$ time for all $\Rightarrow O(m \log n)$ if $m \geq n$ (say).

[See slides from Kevin Wayne]
[See slightly better heap: Binomial heaps. — follow slides]

Can do better using Fibonacci heaps: $O(\log n)$ time inserts, deletemins
$O(1)$ time decreasekeys.

[Won't see in this course, see links on webpage] $\Rightarrow O(m + n \log n)$ time for Dijkstra.

Can we do $O(1)$ for all ops? (~~saying~~ in general, for arbitrary values)

No! then can sort n numbers in $O(n)$ time. (Insert all, do n deletemins)

But if numbers are $\boxed{\text{"small" integers}}$ can do better.

(van Emde Boas heaps give $O(\log \log U)$ time where all values $\in \{1, 2 .., U\}$

~~Next lecture~~.