COP 3503 4/11/24

(1) Bellman-Ford

Single Source Shortest Path w/ neg edges

Cycle Detection

(2) Rolling Hash Function

- Probabilistic

Bellman-Ford

edge relaxation

d[v] = estimate to vertex v from source

d[v] = estimate to vertex v from source

W[u][v] = edge weight from u to v

d[v] = min(d[v], d[u] + w[u][v]);

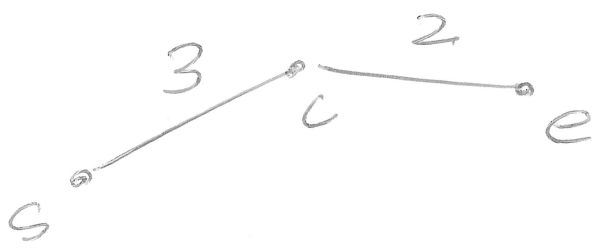
Initialize d[source] = 0, d[rest] = ∞.

```
for (edge e : edges)
    d[e.v] = min(d[e.v], d[e.u] + e.w)
```

(e.v: to; e.u: from; e.w: edge weight)

After 1 iteration, you'll have all shortest paths from source that use a single edge.

What happens if we run that loop a second time?



If shortest path from s to e uses 2 edges, then rerunning edge relaxation will discover this route.

After k iterations of Bellman-Ford (edge relaxation) all shortest paths of k or fewer edges have been discovered.

If a graph w/ no neg weight cycles, all shortest paths are $|V|-1$ edges or ~~fewer~~ fewer edges.

Bellman-Ford Alg

n vertices

```
for (i=0; i<n-1; i++)
    for (edge e: edges)
        d[e.v] = min(d[e.v], d[e.u]+e.w)
```

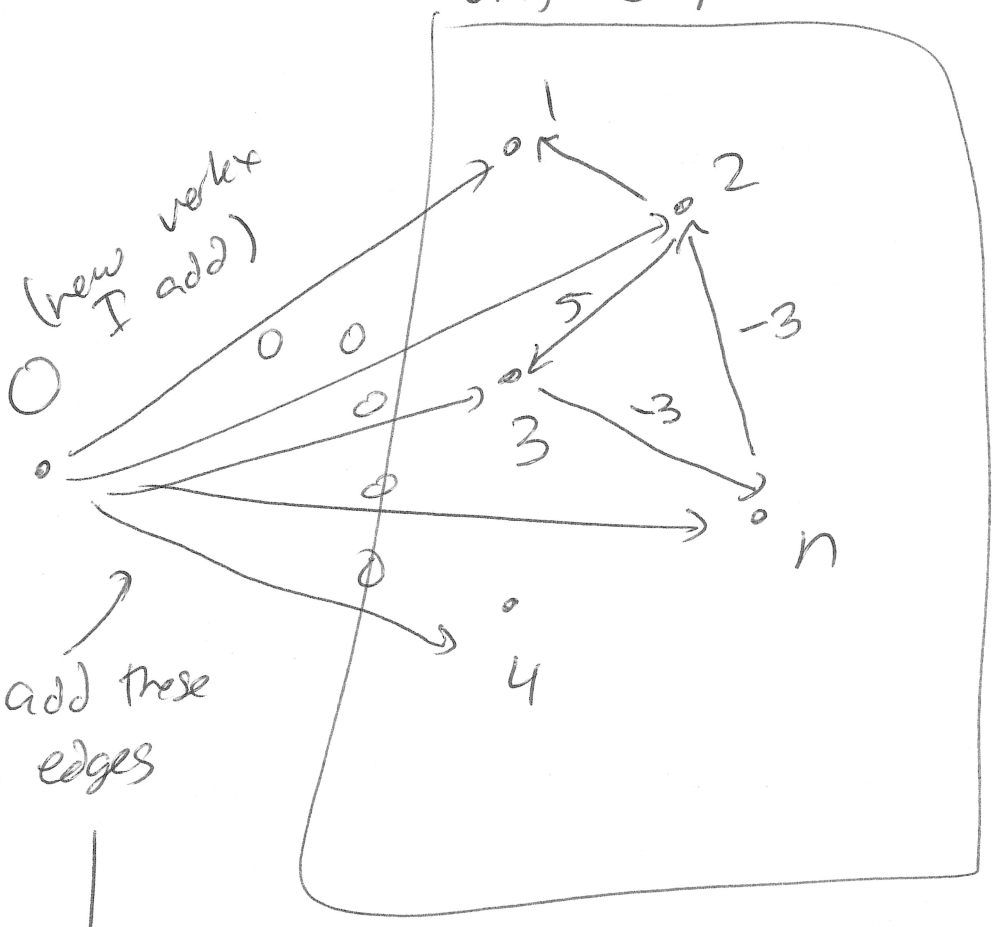
e.w = weight edge from u to v

Run-Time $O(V \cdot E)$

- if $E \sim V \rightarrow O(V^2)$
- if $E \sim V^2 \rightarrow O(V^3)$

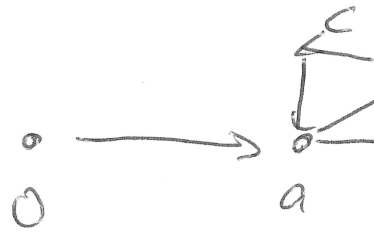
} depending on # edges performance varies greatly

Cycle Detection Corrections



→ this allows us to reach all nodes in a graph where a negative cycle exists but is unreachable from vertex 1.

path could be

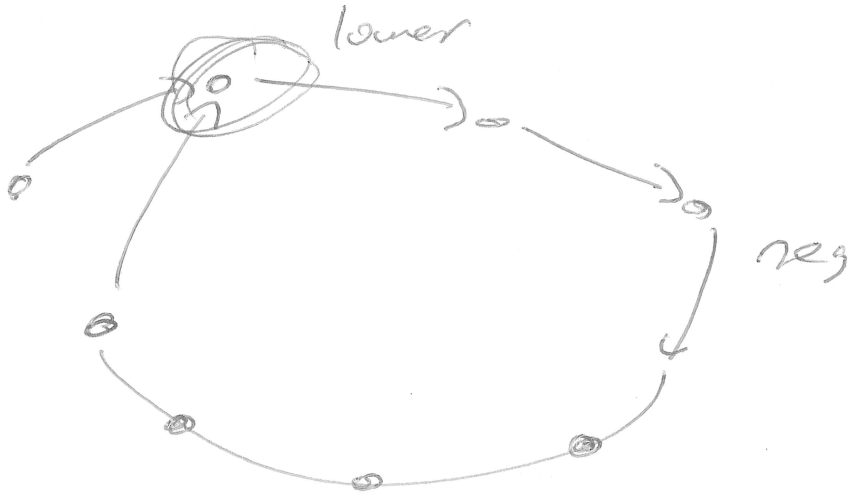


first duplicate update after n iterations so note that d isn't in cycle.

Do: d ← a ← c ← b ← a → reverse this.

→ repeat

Neg cycle detection



any updates on ~~iii~~ iteration n to 2n-1 then there's a neg cycle

Rolling Hashing (for string)

looking for a substring in a larger string.

s "cad" t

abracadabra

Simple alg

```
// i = start in terset of match.
for (i=0; i<=t.length-s.length; i++) {
    boolean ok = true;
    for (j=0; j<s.length; j++) {
        if ( s[j] != t[i+j]) {
            ok=false;
            break;
        }
    }
    if (ok) //ret true or add list of matches
}
```

|t|×|s|

worst case

1 2 ~~18~~ 1 3 1 4 1 2 18 1

a b r a c a d a b r a

$3\times27^2+1\times27^1+4\times27^0$

cad

$(1\times27^{10}+2\times27^9+18\times27^8+\dots+1\times27^0,)$ mod large prime

interpret the string in some base

hash value for a string is this

abr a ⟹ bra (added)

$2\times27^2+18\times27^1+1\times27^0$

$1\times27^2+2\times27^1+18\times27^0$

curhash

$27\times($curhash$)+$ newsymbol $-($oldsymbol$*$ base$^{k-1})$ %mod

precompute base^{k-1}

27 × base

contribution of leftmost term

k = length substrings

Runs in linear time

$O(|t|)$

Obs include

Tip of Your Tongue

To reduce probability of error use 2 different primes for hashing

for each length, we stored the hash value of

(a) each prefix

(b) each suffix

(c) each combined prefix + suffix

mapped each hash value to how many times we saw it!

* Never double checked actual letters.