Bloom Filters

Problem: Billions of web sites URL's — small number of malicious web sites.

User → URL ← is this malicious?

B

List of malicious sites

mal

k hash functions $h_1, h_2, \ldots, h_k$.

Table of size n.

$n \approx \log B$

```
for i ← 1 to B do
    for j ← 1 to k
        index ← h_j (mal[i])
        Table[index] ← 1
```

Download:
Table[0..n-1],
Hash functions $h_1, \ldots h_k$

User wants to visit URL x

Within browser: Calculate $\underbrace{h_1(x)}_{\text{index}_1}, \underbrace{h_2(x)}_{\text{index}_2}, \ldots \underbrace{h_k(x)}_{\text{index}_k}$

if ~~Table~~[index_i] $\neq 1$ for any $i = 1..k$ → then x is "safe".

if Table[index_i] $= 1$ for $i = 1..k$
then check with google if URL x is safe.

Ideally: Table[ ] has large number of 0 entries ($> 50\%$)

Prob of false positive $\sim 2^{-k}$

size of bloom filter: Table, Hash function — ~ 20 kb, say

should be small

Dijkstra's algorithm

Input: Directed graph $G=(V,E)$, source $s \in V$.
Nonnegative weights $w: E \to \mathbb{R}^+$.

Output: For each $u \in V$, $u.\text{distance} = \delta(s,u)$, shortest path distance from s to u, and $u.\text{parent}$ = predecessor of u in shortest path.

Idea: - Maintain a set S of nodes for which shortest paths from s are known.
- For each $v \in V-S$, $v.\text{distance}$ stores length of shortest path from s to v that goes through only nodes of S.
- In each iteration, select a node u in $V-S$ with minimum $u.\text{distance}$ among all nodes in $V-S$, add it to S. Update distance field of other nodes in $V-S$ (a shortest path can now go through v).
- Use an indexed priority queue of vertices, where priority ~~of~~ u = $u.\text{distance}$.

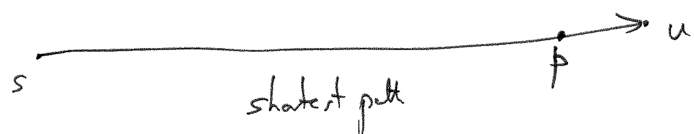
```
// Initialize
for u ∈ V do      u.seen ← false
    u.distance ← ∞    u.parent ← null
s.distance ← 0
Q ← Indexed priority queue of vertices
    using v.distance as priority
while Q is not empty do
    u ← Q.remove() ; u.seen ← true
    // Relax edges out of u
    for each edge e=(u,v) in u.Adj do
        if v.distance > u.distance + w(e)
          and !v.seen then
            v.distance ← u.distance + w(e)
            v.parent ← u
            Q.decreaseKey(v)
```

// In our graph implementation,
// $w(e)$ = e.Weight
// v = e.otherEnd(u).
// $RT = O(|E| \log |V|)$.

DAG-shortest paths

// Input: $G=(V,E)$, a DAG $\quad w: E \to \mathbb{R}$

Idea:



$$s.\text{top} < p.\text{top} < u.\text{top}$$

If nodes are processed in topological order, then when u is reached, $u.\text{distance} = \delta(s,u)$.

$RT = O(V+E)$

Algorithm: Find a topological ordering of G.
Initialize(s)
for each $u \in V$ in topological order do
  // Invariant: $u.\text{distance} = \delta(s,u)$.
    for each edge $e=(u,v) \in E$ do
      Relax(u, v, e).