TOPIC-SENSITIVE PAGE RANK

- choose independent categories in advance

$C_1, C_2, \dots C_k$ example: $\left.\begin{array}{l}\text{bussiness}\\ \text{economics}\\ \text{sports}\\ \text{news}\\ \text{travel}\dots\end{array}\right\}$

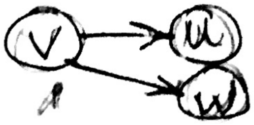
- compute page rank for each page u inside a set of "relevant" pages for each category k

$$PR_k(u) = PR\left\{\begin{array}{l}\text{of page } u \text{ in set}\\ \text{retrieved for category } k\end{array}\right\}$$

- at query time q

$$PR_q(u) = \sum_k \text{prob}(c_k|q) \cdot PR_k(u)$$

- $\text{prob}(c_k|q) \propto \underbrace{\text{prob}(c_k)}_{\substack{\text{estimate}\\ \text{in}\\ \text{advance}}} \cdot \underbrace{\text{prob}(q|c_k)}_{\text{language model.}}$



$$A(W) = \sum_{V}\sum_{W} \frac{A(W)}{in(w)\,out(V)}$$

The *outlinking* [illegible] link-selection pred[illegible] to [illegible]

$$O^P \; [illegible] \; , v) \; [illegible]$$

The *inlinking* [illegible] selection predicate P a[illegible]

$$I_s^P = [illegible]$$

The *base set* [illegible] P [illegible] defined to be:

$$B_s^P = [illegible] I_s^P \cup [illegible]$$

The *neighborhood* [illegible] N_s^P [illegible] the base set B_s^P as its vertex set and an edge set N_s^P containing those edges in E that are covered by B_s^P and permitted by P.

$$N_s^P = \{(u, [illegible]) \in E : [illegible] B_s^P \wedge [illegible] P(u, v)\}$$

To simplify notation, we write B to denote B_s^P, and N to denote N_s^P.

5.2 The HITS algorithm

For each node [illegible] neighborhood graph, HITS computes two scores: [illegible] score [illegible]), estimating how authoritative [illegible] by the query, and a hub score $H($ [illegible] whether [illegible] a good [illegible] to many authoritative pages. This is done using the following algorithm:

HITS-Hub-and-Authority-Scores:

```
1. For all $u \in B$ do $H(u) := \sqrt{\frac{1}{|B|}}$, $A(u) := \sqrt{\frac{1}{|B|}}$.
2. Repeat until $H$ and $A$ converge:
    (a) For all $v \in B$ : $A'(v) := \sum_{[illegible]} H(u)$
    (b) For all $u \in B$ : $H'(u) := \sum_{(u,v) \in N} A(v)$
    (c) $H := \frac{1}{\|H'\|_2} H'$, $A := \frac{1}{\|A'\|_2} A'$
```

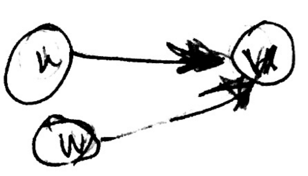
where $\|X\|_2$ is the euclidean norm of vector X.

5.3 The SALSA algorithm

For each node u in the neighborhood graph, SALSA computes an *authority score* $A(u)$ and a *hub score* $H(u)$ using the following two *independent* algorithms:

SALSA-Hub-Scores:

```
1. Let $B^H$ be $\{u \in B : out(u) > 0\}$.
2. For all $u \in B$
     $H(u) = \begin{cases} \frac{1}{|B^H|} & u \in B^H \\ 0 & \text{other} \end{cases}$
3. Repeat until $H$ converges:
    (a) For all $u \in B^H$:
          $H'(u) = \sum_{(u,v) \in N} \sum_{(w,v) \in N} \frac{H(w)}{in(v)\,out(w)}$
    (b) For all $u \in B^H$ : $H(u) = H'(u)$
```



$$H(u) = \sum_{V}\sum_{W} \frac{H(w)}{out(w) \cdot in(V)}$$

B^A $u \in B$ $in(u) > 0\}$

all $u \in B$

$$A(u) = \begin{cases} \frac{1}{|B^A|} & u \in B^A \\ 0 & \text{other} \end{cases}$$

[illegible] converges:

[illegible] for all $u \in B^A$

$$A'(u) = \sum_{(v,u) \in N} \sum_{(v,w) \in N} \frac{A(w)}{out(v)\,in(w)}$$

(b) For all $u \in B^A$: $A(u) := A'(u)$

6. THE SCALABLE HYPERLINK STORE

[illegible] the *Scalable* [illegible] for the [illegible]paq Link [illegible] distributed [illegible] in main memory [illegible] to nodes [illegible] data compression [illegible] properties (namely, [illegible] web graph to achieve fairly [illegible] the full 17.7 billion link graph [illegible] 3 requires six machines, each with 16 GB of main memory.

The two principal abstractions [illegible] in SHS are a *URL store* and two *link stores*, one to trace links forward and another to trace them back. Clients use SHS by linking against a library containing classes that implement clerks for the URL store and the link stores; all the intricacies common to distributed systems are handled by the clerks and the SHS servers.

The URL store maintains a bijection between URLs (strings) and UIDs (integers that serve as short-hands for URLs). Clients can map URLs to UIDs and UIDs back to URLs. The API of the URL store clerk looks as follows:

```
class UrlStoreClerk {
  ... // omitting private members
 public:
  UrlStoreClerk(char *serverNameFile);
  ~UrlStoreClerk();
  INT64 UrlToUid(char *url);
  char *UidToUrl(INT64 uid);
  SeqInt64 BatchedUrlToUid(SeqString& urls);
  SeqString BatchedUidToUrl(SeqInt64& uids);
  ... // omitting methods irrelevant to this paper
};
```

The *UrlStoreClerk* constructor takes the name of a file that contains the names of the SHS servers maintaining the graph. The central methods are *UrlToUid*, which maps a URL to a UID, and *UidToUrl*, which maps a UID back to a URL. The methods *BatchedUrlToUid* and *BatchedUidToUrl* are variants of the previous two methods that allow the mapping of entire batches of URLs or UIDs; their purpose is to allow client applications to amortize RPC overheads. As a point of reference, mapping a URL to a UID takes about 3 microseconds, while performing a null RPC takes about 100 microseconds; so providing a mechanism to batch up requests is performance-critical. Our implementations of