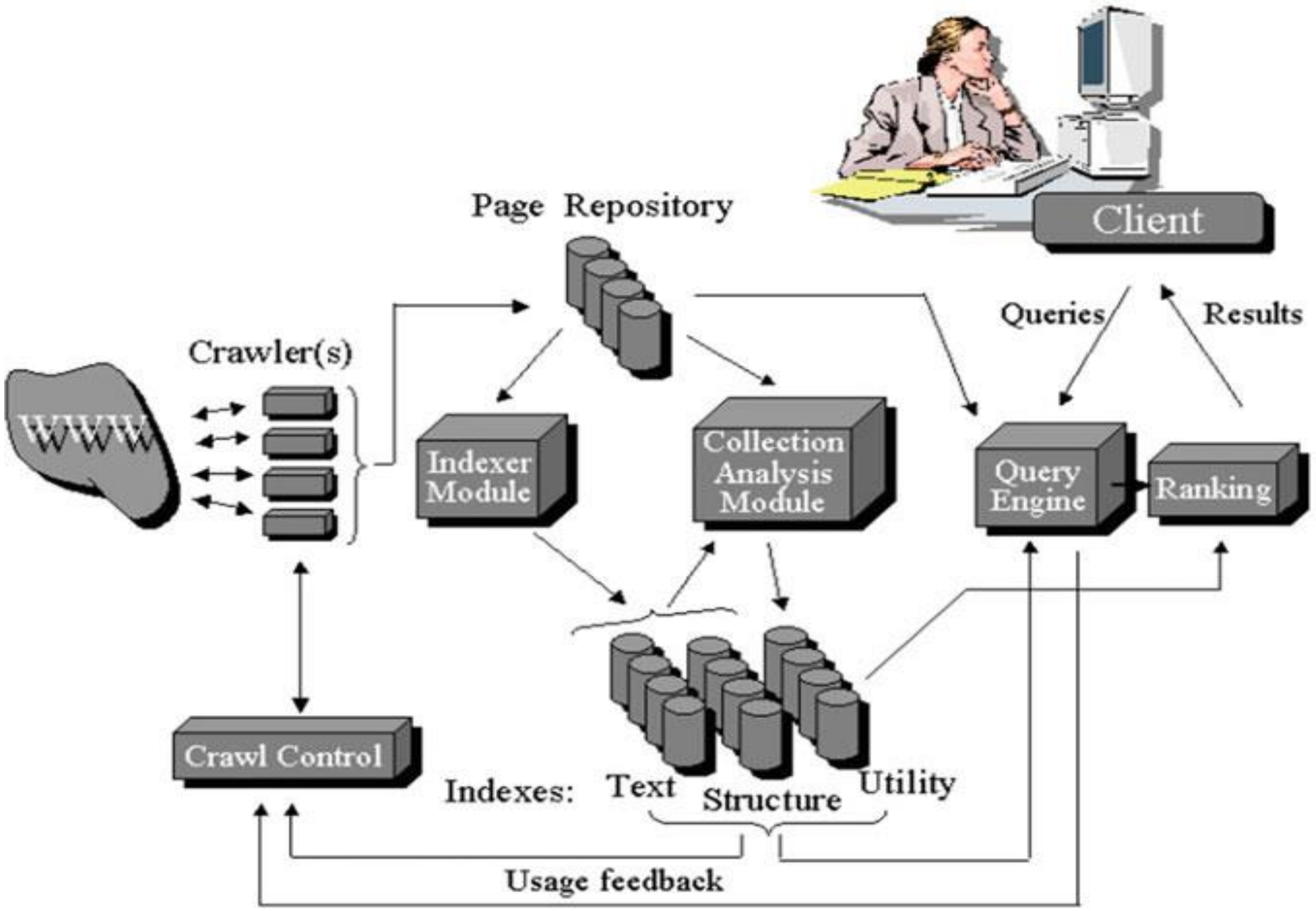
LINK ANALYSIS- SEARCH ENGINES

MODULE 5

Search Engine

- A search engine is an application which takes as input a search query and returns a list of relevant Webpages.
- Search engine do a lot of preparation work so that when you click search you are presented with a set of precise and quality results that answers your question and queries.
- Preparation works divide into three
  - ***Crawling***
  - ***Indexing***
  - ***Ranking***

Architecture of a Search Engine



- Every engine requires a ***crawler module***.
- A crawler is a small program that browses the Web by following links.
- The crawlers are given a starting set of pages.
- They then retrieve the URLs appearing in these pages and give it to the crawl control module.
- The crawl control module determines what links to visit next, and feeds the links to visit back to the crawler.
- The crawler also stores the retrieved pages in the page repository.

- The ***indexer module*** extracts all the words from each page, and records the URL where each word occurred.
- The result is a very large “lookup table” that has URLs mapped to the pages where a given word occurs.
- This index is the ***text index***.
- The indexer module also generate a structure index which reflects the links between pages.
- The collection analysis module is responsible for creating other kinds of indexes, such as the utility index.

- Utility indexes may provide access to pages of a given length, pages of a certain “importance”, or pages with some number of images in them.
- The text and structure indexes may be used when creating utility indexes.
- The crawl controller can change the crawling operation based on these indexes.

- The ***query engine module*** receives and files search requests from users.
- For the queries received by the query engine, the results may be too large to be directly displayed to the user.
- The ***ranking module*** is therefore tasked with the responsibility to sort the results.

Crawling

- The crawler module starts with an initial set of URLs S0 which are initially kept in a priority queue.
- From the queue, the crawler gets a URL, downloads the page, extracts any URLs in the downloaded page, and puts the new URLs in the queue.
- This process repeats until the crawl control module asks the crawler to stop.

Page Selection

- Due to the large number of webpages and limit in the available resources, the crawl control program has to decide which pages to crawl, and which not to.
- The “importance” of a page can be defined in one of the following ways:

- 1. Interest Driven:
  - The important pages can be defined as those of “interest to” the users.
  - This could be defined in the following manner: given a query *Q*, the importance of the page *P* is defined as the textual similarity between *P* and *Q* .
  - This metric is referred to as ***I S(·)***.

- 2. Popularity Driven:
  - Page importance depends on how “popular” a page is.
  - Popularity can be defined in terms of a page’s in-degree.
  - A page’s popularity is the in-degree with respect to the entire Web.
  - This can be denoted as ***I B(·)***.

- 3. Location Driven:
  - This importance measure is computed as a function of a page’s location instead of its contents.
  - Denoted as ***I L(·)***
  - If URL *u* leads to a page *P*, then *I L(P)* is a function of *u*.
- The importance of a page *P* can be defined as
- $IC(P) = k_1 I S(P) + k_2 I B(P) + k_3 I L(P)$, for constants k_1, k_2, k_3 and query *Q*.

Ways to Compute the Performance of a Crawler

- 1. Crawl and Stop:
  - A crawler *C* starts at a page P_0 and stops after visiting *K* pages, *K* denotes the number of pages that the crawler can download in one crawl.
  - At this point a *perfect crawler* would have visited $R_1, \ldots, R_K$ where R_1 is the page with the highest importance value, R_2 is the next highest, and so on.
  - These pages R_1 through R_K are called the ***hot pages***.

- The K pages visited by the real crawler will contain only $M(\leq K)$ pages with rank higher than or equal to that of R_K .
- The performance of the crawler C is computed as
    - $PCS(C) = (M \cdot 100)/K.$
- Although the performance of the *perfect crawler* is 100%, a crawler that manages to visit pages at random would have a performance of $(K \cdot 100)/T$, where T is the total number of pages in the Web.

- 2. Crawl and Stop with Threshold:
  - Assume that a crawler visits K pages.
  - If we are given an importance target *G*, then any page with importance target greater than *G* is considered ***hot***.
  - If we take the total number of hot pages to be *H*, then the performance of the crawler, PST (C), is the percentage of the *H* hot pages that have been visited when the crawler stops.

- If $K < H$, then an *ideal crawler* will have performance $(K \cdot 100)/H$.
- If $K = H$, then the *ideal crawler* has 100% performance.
- A *purely random crawler* that revisits pages is expected to visit $(H/T)K$ hot pages when it stops.
- Its performance is $(k \cdot 100)/T$.
- If the random crawler visits all T pages, then its performance expected to be 100%.

Page Refresh

- Once the crawler has downloaded the “important” pages, these pages must be periodically refreshed to remain up-to-date.
- The strategies used to refresh the pages:
  - 1. Uniform refresh policy:
    - All pages are revisited at the same frequency *f* , regardless of how often they change.

- 2. Proportional refresh policy:
  - The crawler visits a page proportional to its change.
  - If λ_i is the change frequency of a page P_i , and f_i is the crawler’s revisit frequency for P_i , then the frequency ratio λ_i / f_i is the same for any i .

Storage

- The storage repository of a search engine must perform two basic functions.
  1. It must provide an interface for the crawler to store pages.
  2. It must provide an efficient access API that the indexer module and the collection analysis module can use to retrieve pages.

Issues deals with a Repository

- **1**. It must be capable of supporting both *random access* and *streaming access* equally efficiently.
- Random access for quickly retrieving a specific Webpage, given the page’s unique identifier to serve out cached copies to the end-user.
- Streaming access to receive the entire collection as a stream of pages to provide pages in bulk for the indexer and analysis module to process and analyze.

- **2**. It must be *scalable*, i.e., it must be capable of being distributed across a cluster of systems.
- **3**. Since the Web changes rapidly, the repository needs to handle a high rate of modifications since pages have to be refreshed periodically.
- **4**. The repository must have a mechanism of detecting and removing obsolete pages that have been removed from the Web.

- Issues that affect the *characteristics and performance* of the repository.
    - Page Distribution Policies
    - Physical Page Organization Methods
    - Update Strategies

1. Page Distribution Policies

- Pages can be assigned to nodes using a number of different policies.
- *Uniform distribution policy*:
  - All nodes are treated identically, and a page can be assigned to any of these nodes, independent of its identifier.
  - Thus each node will store portions of the collection proportionate to its storage capacity.

- *Hash distribution policy:*
  - Allocates pages to nodes depending on the page identifiers.
  - A page identifier would be hashed to yield a node identifier and the page would be allocated to the corresponding node.

2. Physical Page Organization Methods

- Physical page organization at each node determines how well each node supports *page addition/insertion, high-speed streaming and random page access*.
- A ***hash-based organization*** treats a disk as a set of hash-buckets
- Pages are assigned to hash buckets depending on their page identifier.

- For page additions, a ***log-structured organization*** is used in which the entire disk is treated as a large contiguous log to which incoming pages are appended.
- In ***hybrid hashed-log organization***, where storage is divided into large sequential “extents”, as opposed to buckets that fit in memory.
- Pages are hashed into extents, and each extent is organized like a log-structured file.

3. Update Strategies

- 1. *Batch-mode or steady crawler*:
  - A *batch-mode crawler* is periodically executed and allowed to crawl for a certain amount of time or until a targeted set of pages have been crawled, and then stopped.
  - A *steady crawler* runs continuously, without pause, supplying updates and new pages to the repository.

- 2. Partial or complete crawls:
  - A *partial crawl* re-crawls only a specific set of pages, while a *complete crawl* performs a total crawl of all the pages.
  - This makes sense only for a batch- mode crawler while a steady crawler cannot make such a distinction.

- 3. In-place update or shadowing:
  - With *in-place updates*, pages received from the crawl are directly integrated into the repository’s existing collection, probably replacing the older version.
  - With *shadowing*, pages from a crawl are stored separate from the existing collection and updates are applied in a separate step.

- The advantage of shadowing is that the reading and updating tasks can be delegated to two separate nodes thereby ensuring that a node does not have to concurrently handle both page addition and retrieval.
- While avoiding conflict by simplifying implementation and improving performance, there is a delay between the time a page is retrieved by the crawler and the time it is available for access.

Indexing

- The indexer and the collection analysis modules are responsible for generating the text, structure and the utility indexes.
- Different indexes are.
  - Structure index
  - Text index
  - Utility index

- Structure Index
  - To build structure index, the crawled content is modelled as a graph with nodes and edges.
  - This index helps search algorithms by providing
    - *Neighborhood information*: For a page P, retrieve the set of pages pointed to by P or the set of pages pointing to P,
    - *Sibling pages*: Pages related to a given page P.

- An *inverted index* over a collection of Webpages consists of a set of inverted lists, one for each word.
- The inverted list for a term is a sorted collection of locations (page identifier and the position of the term in the page) where the term appears in the collection.

- To make the inverted index scalable, there are two basic strategies for partitioning the inverted index across a collection of nodes.
- In *local inverted file (IFL)* organization each node is responsible for a disjoint subset of pages in the collection.
- A *search query* would be broadcast to all nodes, each of which would return disjoint lists of page identifiers containing the search terms.

- Text Index
  - Traditional information retrieval methods using suffix arrays, inverted indices and signature files, can be used to generate these text index.
- Utility Index
  - The number and type of utility indexes built by the collection analysis module depends on the features of the query engine and the type of information used by the ranking module.

Ranking

- Page rank is a measure of how ‘important’ a web page is.
- It works on the basis that when another website links to your web page, it’s like a recommendation or vote for that web page.
- Each link increases the web page’s page rank.
- The amount it increases depends on various factors, including how important the voting page is and how relevant it is.

- Why is page rank important?
  - Page rank is important because it’s one of the factors a search engine like Google takes into account when it decides which results to show at the top of its search engine listings – where they can be easily seen.

- Reasons for ranking
  - **1.** Before the task of ranking pages, one must first retrieve the pages that are relevant to a search.
  - This information retrieval suffers from problems of synonymy (multiple terms that more or less mean the same thing) and polysemy (multiple meanings of the same term),
  - (By the term “jaguar”, do you mean the animal, the automobile company, the American football team, or the operating system).

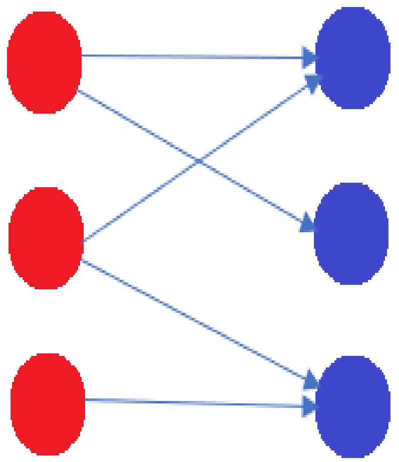
- **2.** The time of retrieval.
- For instance, in the case of an event such as a calamity, government and news sites will update their pages as and when they receive information.
- The search engine not only has to retrieve the pages repeatedly, but will have to re-rank the pages depending on which page currently has the latest report.

- 3. With every one capable of writing a Web page, the Web has an abundance of information for any topic.
- The task is to rank these results in a manner such that the most important ones appear first.

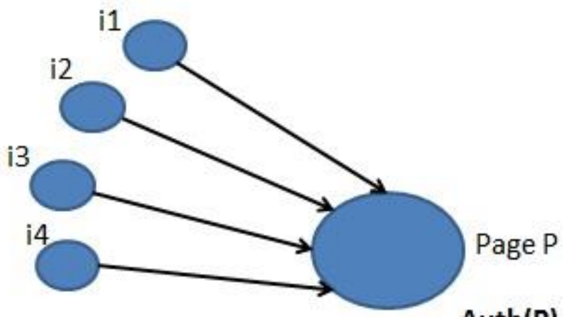
Hyperlink Induced Topic Search (HITS) Algorithm

- It is a link analysis algorithm that rates the web pages.
- Developed by John Kleinberg
- A good ***Hub*** is a page that pointed to many other pages
- A good ***Authority*** represents a page that was linked to many other different hubs

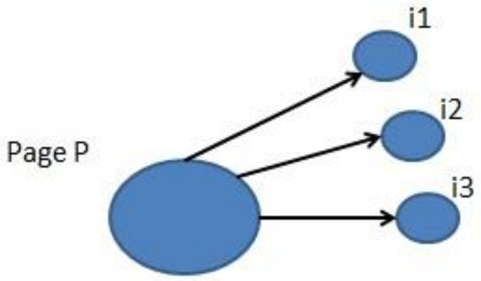
HUBS



AUTHORITIES



Auth(P) = Sum of **Hub(i)** for all **i** pointing to **P**.



Hub(P) = Sum of **Auth(i)** for all **i** pointed to by **P**.

HITS Algorithm

- 1. Start with all hub and authority scores equal to 1.
- 2. Choose a number of steps k.
- 3. Perform a sequence of k hub-authority updates. Each update works as follows:
  - Apply the Authority Update rule to the current set of scores.
  - Apply the Hub Update rule to the resulting set of scores.
- 4. The resultant hub and authority scores may involve numbers that are very large. Normalize these values

- To perform the spectral analysis of HITS algorithm, first represent the network of n nodes as an adjacency matrix M.
- The hub and authority scores are represented as vectors in *n* dimensions.
- The vector of hub and authority scores are denoted as *h, a*
- Thus, the Hub Update and Authority Update rules can be represented by
  - $h \leftarrow M \cdot a$
  - $a \leftarrow M^T \cdot h$

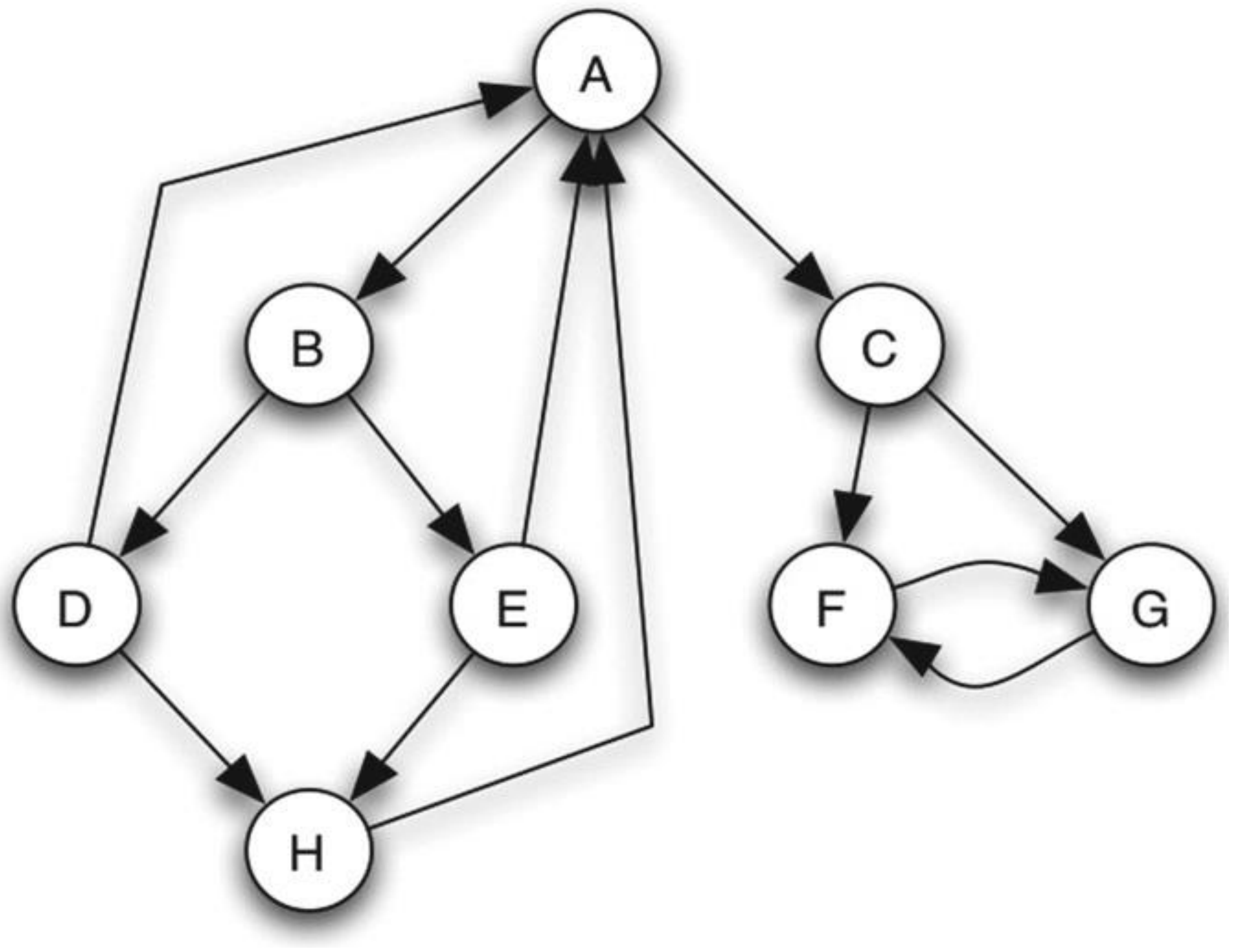
PageRank

- PageRank also works on the same principle as HITS algorithm.
- It starts with simple voting based on in-links, and refines it using the Principle of Repeated Improvement.
- PageRank can be thought of as a “fluid” that circulates through the network, passing from node to node across edges, and pooling at the nodes that are most important.
- It is computed as follows:

Page Rank Algorithm

- 1. In a network with n nodes, all nodes are assigned the same initial PageRank, set to be 1/n.
- 2. Choose a number of steps k.

- 3. Perform a sequence of k updates to the PageRank values, using the following Basic PageRank Update rule for each update:
- Each page divides its current PageRank equally across its out-going links, and passes these equal shares to the pages it points to.
- If a page has no out-going links, it passes all its current PageRank to itself.
- Each page updates its new PageRank to be the sum of the shares it receives.


A
B
C
D
E
F
G
H

- Any network containing reciprocating links will have the clogging of PageRanks at such nodes.
- To prevent such a situation, the PageRank rule is updated by a scaling factor ***s*** that should be strictly between 0 and 1
- According to this rule, first apply the Basic PageRank Update rule.
- Then scale down all PageRank values by a factor of ***s*** , thereby shrinking the total PageRank from 1 to ***s***.
- The residual ***1 − s*** units of PageRank are divided equally over all the nodes, giving ***(1 − s)/n*** to each.
- This scaling factor makes the PageRank measure less sensitive to the addition or deletion of small number of nodes or edges.

- Under the basic rule, each node takes its current PageRank and divides it equally over all the nodes it points to.
- This suggests that the “flow” of PageRank specified by the update rule can be naturally represented using a matrix ***N***.
- Let ***Ni j*** be the share of ***i*** ’s PageRank that ***j*** should get in one update step.
- ***Ni j*** = 0 if ***i*** doesn’t link to ***j*** , and when ***i*** links to ***j*** , then ***Ni j*** = ***1/li*** , where ***li*** is the number of links out of ***i*** .
- (If ***i*** has no outgoing links, then ***Nii = 1***, in keeping with the rule that a node with no outgoing links passes all its PageRank to itself.)

- If the PageRank of all the nodes using a vector ***r*** , where $\boldsymbol{r_i}$ is the PageRank of node ***i*** .
- The Basic PageRank Update rule is
- $\boldsymbol{r \leftarrow N^T \cdot r}$
- We can similarly represent the Scaled PageRank Update rule using the matrix $\hat{N}$ to denote the different flow of PageRank.
- To account for the scaling, we define $\hat{N}_{ij}$ to be
- $\boldsymbol{s N_{ij} + (1 - s)/n}$, this gives the scaled update rule as
- $\boldsymbol{r \leftarrow \hat{N}^T \cdot r}$

- Starting from an initial PageRank vector $r^{[0]}$, a sequence of vectors $r^{[1]}$, $r^{[2]}$, . . . are obtained from repeated improvement by multiplying the previous vector by $\hat{N}^T$.
- This gives us $r^{[k]} = (\hat{N}^T)^k r^{[0]}$
- This means that if the Scaled PageRank Update rule converges to a limiting vector
- $r^{[*]}$, this limit would satisfy $\hat{N}^T r^{[*]} = r^{[*]}$. This is proved using Perron’s Theorem.

Random Walk

- Consider the situation where one randomly starts browsing a network of Webpages.
- They start at a random page and pick each successive page with equal probability.
- The links are followed for a sequence of k steps: in each step, a random out-going link from the current page is picked.
- (If the current page has no out-going links, they just stay where they are.)

- *Remark 1*
  - Taking a random walk in this situation, the probability of being at a page *X* after *k* steps of this random walk is precisely the PageRank of *X* after *k* applications of the Basic PageRank Update rule.
- *Proof*
  - If $b_1, b_2, \dots, b_n$ denote the probabilities of the walk being at nodes $1, 2, \dots, n$ respectively in a given step, then the probability it will be at node *i* in the next step is computed as follows:

1. For each node *j* that links to *i* , if we are given that the walk is currently at node *j* , then there is a $1/l_j$ chance that it moves from *j* to *i* in the next step, where l_j is the number of links out of *j* .
2. The walk has to actually be at node *j* for this to happen, so node *j* contributes $b_j (1/ l_j) = b_j / l_j$ to the probability of being at *i* in the next step.
3. Therefore, summing b_j / l_j over all nodes *j* that link to *i* gives the probability the walk is at bi in the next step.

- So the overall probability that the walk is at *i* in the next step is the sum of *bj* /*l j* over all nodes that link to *i* .
- If we represent the probabilities of being at different nodes using a vector *b*, where the coordinate *bi* is the probability of being at node *i* , then this update rule can be written using matrix-vector multiplication as
- $\boldsymbol{b} \leftarrow \boldsymbol{N}^T \cdot \boldsymbol{b}$
- Since both PageRank values and random walk probabilities start out the same (they are initially 1/*n* for all nodes), and they then evolve according to exactly the same rule, so they remain same forever. This justifies the claim.

- Remark 2
  - The probability of being at a page X after k steps of the scaled random walk is precisely the PageRank of X after k applications of the Scaled PageRank Update Rule.
- Proof
  - If b1, b2, . . . , bn denote the probabilities of the walk being at nodes 1, 2, . . . , n respectively in a given step, then the probability it will be at node *i* in the next step, is the sum of s bj / l j , over all nodes j that link to i, plus (1 − s)/n.
  - If we use the matrix ˆN , then we can write the probability update as
  - $\boldsymbol{b} \leftarrow \boldsymbol{N}^{\hat{}T} \boldsymbol{b}$

SALSA Algorithm

- SALSA algorithm starts with a “Root Set” short list of Webpages relevant to the given query retrieved from a text-based search engine.
- This root set is augmented by pages which link to pages in the Root Set, and also pages which are linked to pages in the Root Set, to obtain a larger “Base Set” of Webpages.

- Perform a random walk on this base set by alternately
  a) Going uniformly to one of the pages which links to the current page, and
  b) Going uniformly to one of the pages linked to by the current page.
- The authority weights are defined to be the stationary distribution of the two-step chain first doing (a) and then (b),
- The hub weights are defined to be the stationary distribution of the two-step chain first doing (b) and then (a).

- Let ***B(i) = {k : k →i }*** denote the set of all nodes that point to i , i.e, the nodes we can reach from i by following a *backward* link and
- Let ***F(i) = {k : i → k}*** denote the set of all nodes that we can reach from i by following a *forward* link.

- The Markov Chain for the authorities has transition probabilities

$$P_a(i, j) = \sum_{k: k \in B(i) \cap B(j)} \frac{1}{|B(i)|} \frac{1}{|F(k)|}$$

- If the Markov chain is irreducible, then the stationary distribution $a = (a1, a2, \ldots, an)$ of the Markov chain satisfies $a_i = |B(i)|/|B|$, where $B = \cup_i B(i)$ is the set of all backward links.

- The Markov chain for the hubs has transition probabilities

$$P_h(i, j) = \sum_{k:k \in F(i) \cap F(j)} \frac{1}{|F(i)|} \frac{1}{|B(k)|}$$

- Stationary distribution h = (h1, h2, . . . , hn) of the Markov chain satisfies $h_i = |F(i)|/|F|$, where $F = \cup_i F(i)$ is the set of all forward links.

- SALSA is similar as a one-step truncated version of HITS algorithm, i.e, in the first iteration of HITS algorithm.
- If we perform the Authority Update rule first, the authority weights are set to a = AT u, where u is the vector of all ones.
- If we normalize in the L1 norm, then ai = |B(i)|/|B|, which is the stationary distribution of the SALSA algorithm.
- A similar observation can be made for the hub weights.