**PAGERANK**

**Goal**: Determine which Web Pages are “important”

**Approach**: Ignore page contents; focus on hyperlinks

* Treat web as graph: page = vertex, hyperlink = di-edge
* Pages with many incoming hyperlinks are important
* Need to compute “incoming degree” for vertices / page

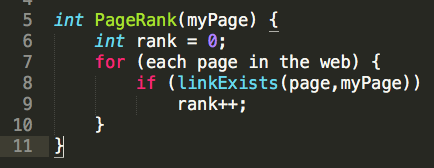
Problem: The web is a very large directed graphs

* Approx 1011 pages, 1013 hyperlinks

Assume for the moment that we could build a graph..

**Most frequent operation in algorithm “Does edge(v,w) exist?”**

Simple PageRank algorithm:



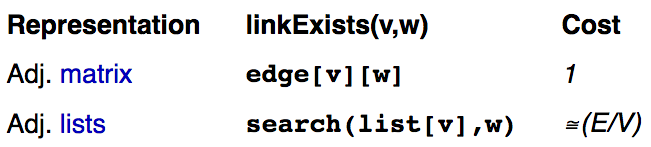
Implementation of **linkExists()**

* For adj matrix: **if (g->edges[page][myPage])**
* For adj list: **searchList(g->edges[page],myPage)**

For analysis:

* V = #pages in Web
* E = #hyperlinks in Web

Costs for computing PageRank for each representation:

****

Not feasible:

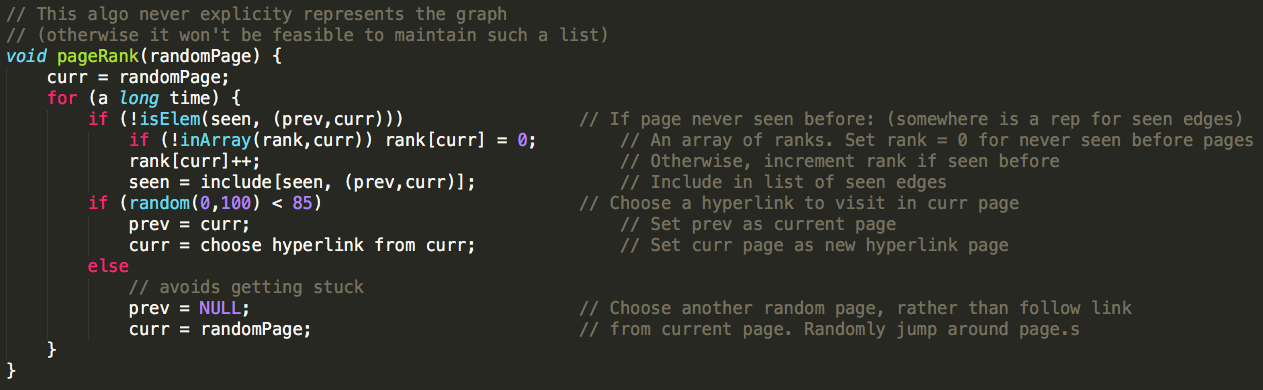
* Adj matrix … V = 4x1010 🡪 matrix has 1021 cells
* Adj list… V lists, each with ~ 1011 list nodes

I.e. we can’t store the entire web as a Graph Structure

So how do we approach the problem?

**Approach: The Random Web Surfer**

* If we randomly follow links in the web
* More likely to re-discover pages with many inbound links  
  (page that is heavily referenced is likely to occur)



This algorithm is PROBABALISTIC: We assume that the more references a page has, the higher chance it will be visited again

This can be accomplished while we are crawling the web to build a search index (i.e. Google Search).

PageRank = ordering the index

**REACHABILITY**

Given a DiGraph g, it is potentially useful to know;

* Is Vertex T reachable from Vertex S?
* Alternatively, is there a path from S to T?

Could be encapsulated as:

* **bool reachable(Graph g, Vertex S, Vertex T)**

Example applications:

* Can I complete a schedule from the current state?
* Is a malloc’d object being reference by any pointer? (to check for memory leak / garbage memory)

**WE CAN USE A REACHABILITY FUNCTION TO SEE IF A PATH EXISTS, BEFORE DOING A BFS OR DFS  
(SAVES TIME, AS BFS/DFS WILL FAIL IF NO PATH EXISTS)**

How to implement an efficient reachability test?

* Implement it via. **hasPath(S,T)**
* Feasible if **reachable(S,T)** is an infrequent operation. Sometimes, even if it is an expensive operation, if the operation is infrequent, then it may be feasible.

Another possibility is a **loop-up table** **(reachability matrix 🡪 tells us if one edge is reachable to another edge)**



If there are a large #vertices, having a matrix like this won’t be feasible.

**REACHABILITY MATRIX**

Create a reachability matrix: (tc = transitive closure)

* If tc[s][t] = 1 🡪 t is reachable from s
* If tc[t][s] = 0 🡪 t is NOT reachable from s
* **Warshall’s Algorithm**
  + Looking for paths of length = 1,2,3 … V-1
  + We make a copy of the edge array (adjacency matrix) + takes already marked adjacent vertices
  + We loop

Cost analysis **tc[ ][ ]**:

* Storage: additional **V2 items** **(each item may be 1 bit)**
* Computation of *makeClosure( ):* **V3** on first call to *reachable( );*
* Computation of *reachable( ):* **O(1)** after first call to *reachable( );*

Amortisation: Many calls to *reachable( )* would justify other costs

Alternative: use DFS in each call to *reachable( )* i.e. using a search rather than a reachability matrix to find if path exists

Cost Analysis of using **DFS**:

* Storage: cost of queue and set during reachable
* Computation of *reachable( ):* **O(V2)** for adjacency matrix