**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

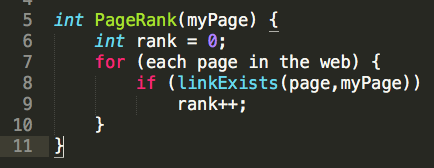
Problem: The web is a very large directed graphs

* 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:



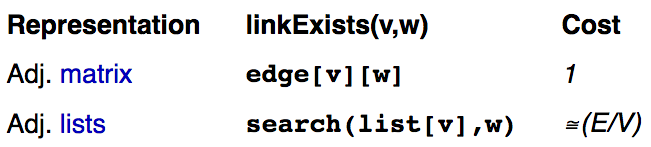
Implementation of **linkExists()**

* 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:

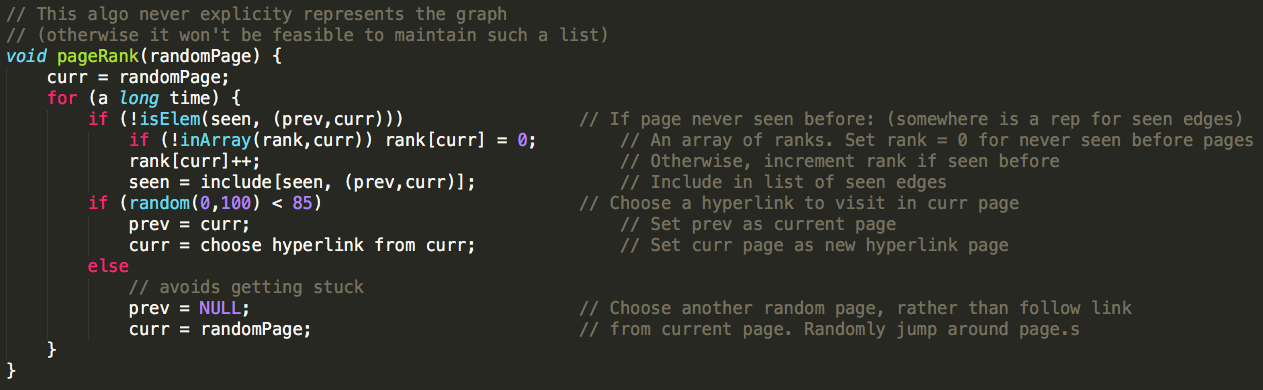
* Adj matrix … V = 4x1010 🡪 matrix has 1021 cells
* Adj list… V lists, each with ~ 1011 list nodes

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

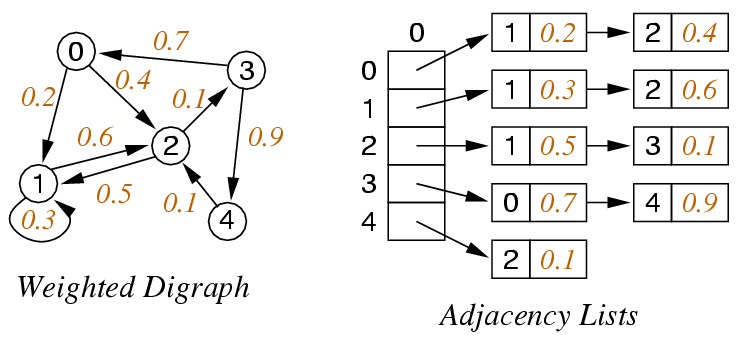
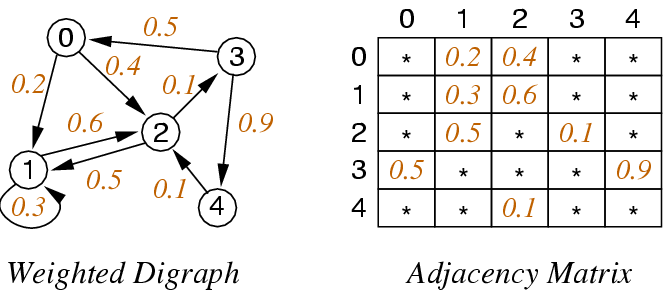
**WEIGHTED GRAPHS**

Some applications require us to consider a **cost** or **weight**. We assign values to edges (+ve, -ve, 0..100, floating pt. etc.)

Weights lead to minimisation-type questions:

* Cheapest way to connect all vertices?
  + **Minimum Spanning Tree** problem
  + Assumes: edges are weighted and non-directed
* Cheapest way to get from A 🡪 B?
  + **Shortest Path** problem
  + Assumes: edges are weighted and directed

Adjacency Matrix Representation with weights:



* Use **–ve** **numbers as weights for NO EDGE** (usually **-1**).
* Adjacency List representation is similar, except instead of just vertex number that we store, we store the cost of the edge between two vertices.

**MINIMUM SPANNING TREES**

A **Spanning Tree (ST)** is a subset of Graph G, which has all vertices covered with a minimum possible number of edges.

* Spanning Trees do not have cycles + cannot be disconnected (as all vertices must be covered)

A **Minimum Spanning Tree (MST)** of Graph G:

* MST is a spanning tree of G
* Sum of edge weights is no larger than any other ST

Problem: **How to efficiently find a MST for a particular Graph?**

* Brute force solution: Generate spanning trees.
  + Take a set of edges, remove edges until all cycles are eliminated. As long as edges cover all the vertices, you have a spanning tree.
  + Check sum of edges weights in the spanning tree. Find the spanning tree with the best cost.
  + This approach is NOT useful as there are possibly a large number of Spanning Trees in a graph

A **cost(t)** function gives sum of edge weights in **t.**

* Function to iterative over edge weights, adding the costs and returning the total cost

Simplifying assumptions:

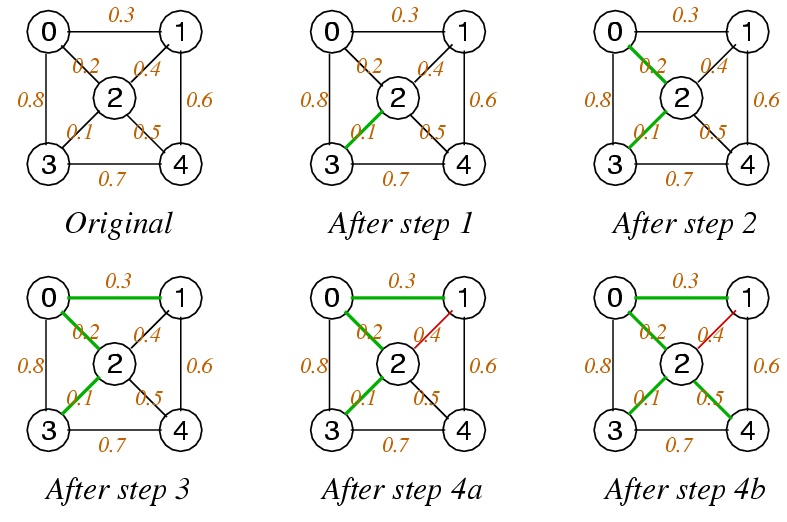
* Edges in G are not directional (MST for Digraphs is harder)
* No edge weights are negative
* All edges weights are distinct

If edges have the same weights:

* MST may not be unique. It is possible that there may be **multiple Minimum Spanning Trees**
* If we have to choose between edges with same weight:
  + Greedy algorithms might make wrong choice.  
    **A greedy algo is one that makes the locally optimal choice in each stage with the hope of finding a Global optimum. In general, a greedy algo does not produce a Globally optimal solution, but it may build a solution that is close to the Global optimum,**
  + Might not end up with minimum cost solution

**KRUSKAL’S ALGORITHM (MST)**

Execution trace of Kruskal’s Algorithm



One approach to computing MST for graph G(V,E)   
// MST is pretty much just a graph which we call MST

* Create empty MST graph
* Create sortedEdgeList (sorted by weight)
* For each sortedEdge:
  + Add edge to MST  
    (graph doesn’t need to be connected straight away)
  + If MST has a cycle, remove edge from MST
  + If nV(MST) == nV(g) break;

(set of connected vertices = #vertices)

Critical operations:

* Iterating over edges in weight order
* Checking for cycles in a graph

Cost analysis for Kruskal’s Algorithm

* Sorting edge list is **O(E Log E)**
* At least **V** iterations over sorted edges
* On each iteration:
  + Getting next lowest cost edge is **O(1)**
  + Checking whether adding it forms a cycle: **cost = ??** **potentially expensive for DFS**
* Possibilities for cycle checking:
  + Use DFS… too expensive
  + Use Union-Find data structure (Sedgewick) (LogN if done properly)

**PRIM’S ALGORITHM (MST)**

Another approach to computing MST for graph(G,E)

* Start from any vertex S and empty MST
* Choose edge not already in MST to add to MST