**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

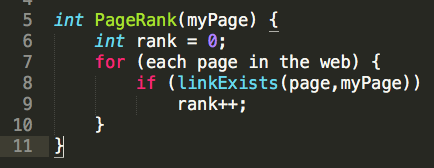
Problem: The web is a very large directed graphs

* 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:



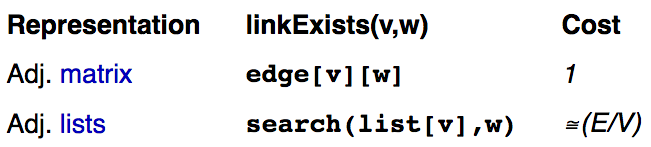
Implementation of **linkExists()**

* 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:

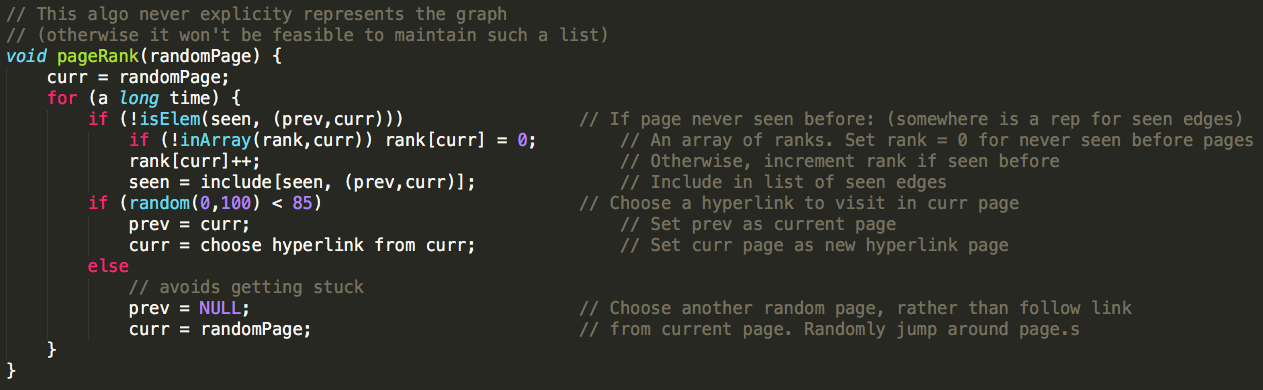
* Adj matrix … V = 4x1010 🡪 matrix has 1021 cells
* Adj list… V lists, each with ~ 1011 list nodes

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

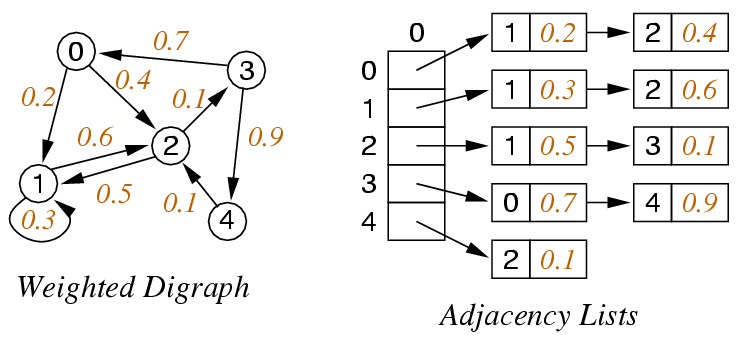
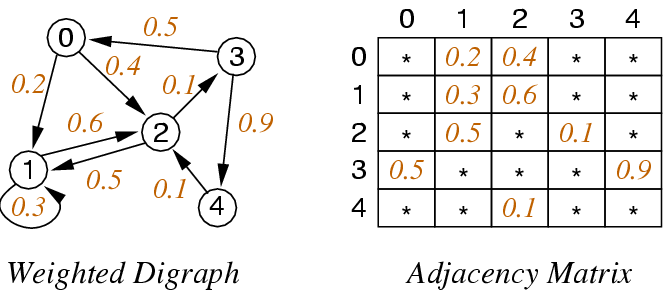
**WEIGHTED GRAPHS**

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:

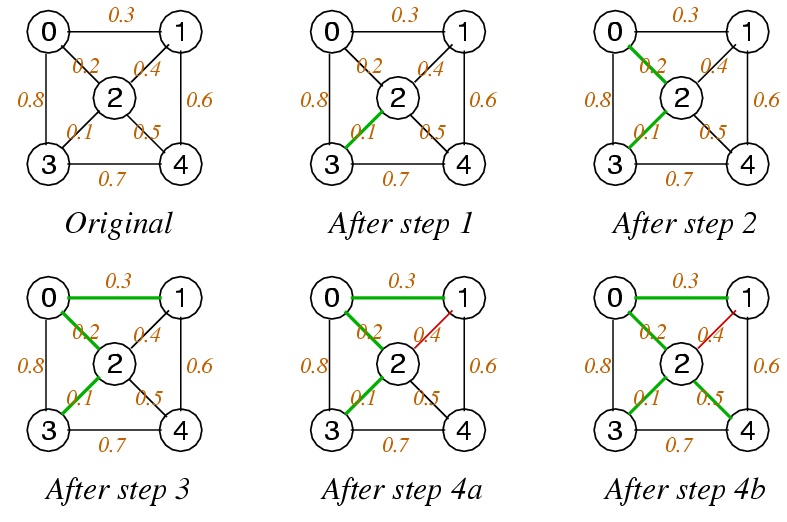
* Edges in G are not directional (MST for Digraphs is harder)
* No edge weights are negative
* All edges weights are distinct

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

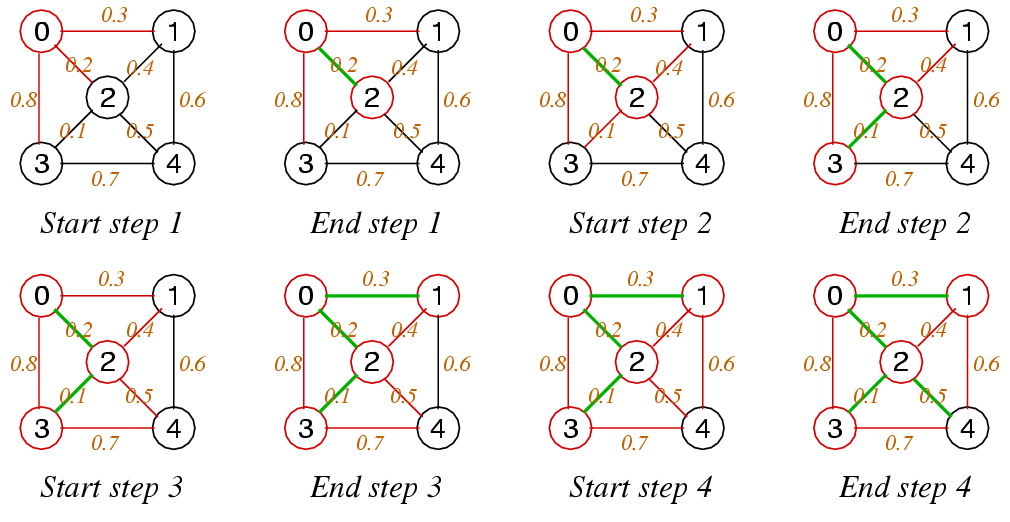
* Sorting edge list is **O(E Log E)**
* At least **V** iterations over sorted edges
* On each iteration:
  + Getting next lowest cost edge is **O(1)**
  + Checking whether adding it forms a cycle: **cost = ??** **potentially expensive for DFS**
* Possibilities for cycle checking:
  + Use DFS… too expensive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

**PRIM’S ALGORITHM (MST)**

Execution trace of Pim’s Algorithm

Another approach to computing MST for graph(G,E) **(Spreading-activation, similar to Breadth-First-Search)**

* Create empty MST graph
* Create usedV set = start from any vertex S
* Create unusedEdgeList = edges(g)
* While size of usedVertices < nV  
  **(Always have a connected graph)**
  + Find an edge in unusedEdgeList, where:  
    1. **e** = S (usedV) 🡪 T (!usedV)  
    2. **w** = W min weight for all edges
  + Add e = (s,t,w) to MST
  + Add vertice T to usedV set
  + Remove edge from unusedE



Critical operations:

* Checking whether vertex is actually connected to the MST graph  
  **(adj matrix 🡪 Go to row for that vertex to check if there is a non-zero value)  
  (adj list 🡪 Go to row for that vertex and traverse list to see if vertex is within list)**
* Finding min weight edge in a set of edges

Cost analysis for Prim’s Algorithm

* **V iterations** of outer loop
* In each iteration:
  + Set operations could be **O(1)** (e.g. with Bit-map)
  + Find min edge with edge Set O(E) 🡪 **O(VE) overall**
  + Find min edge with PQueue O(LogE) **🡪 O(VLogE) overall**
* Notes:
  + Prim’s algo is a variation of graph traversal with PQueue vs. LIFO/FIFO
  + Has a similar structure to **Dijkstra’s Shortest Path Algorithm**

**SHORTEST PATH**

**Path** = Sequence of edges in G

**Cost**(Path) = Sum of edge weights along path

**Shortest Path** between vertices S to T:

* A simple path p(S,T) where S = first trip / T = last trip
* No other simple path q(S,T) where cost(Q) < cost(P)

**Assumptions** = Weight digraph, no –ve weights

**Variations**

* Source-target (Google Maps 🡪 Starting to End)
* Single-source (Start from single source to all other dest)
* All-pairs (single source for every vertex in the graph 🡪 most difficult one)

**Applications** = Robot navigation, Routing in data networks

**SHORTEST PATH – SINGLE SOURCE**

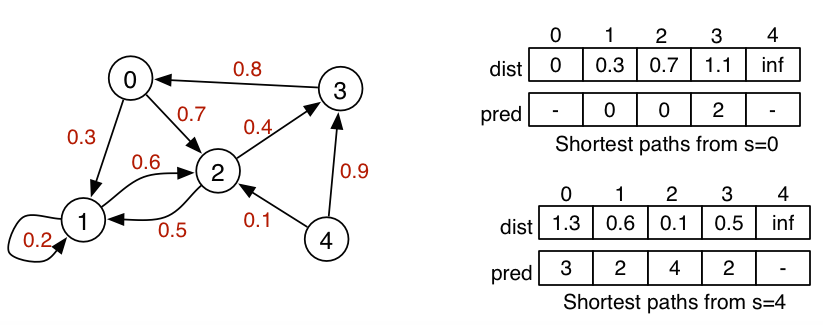
**Dist[ ]**: V-indexed array of cost of shortest path from S **Pred[ ]**: V-indexed array of predecessor in shortest path

With S = 0:

* Shortest path from 0 🡪 4 = infinite  
  as it is impossible to reach Vertex 4

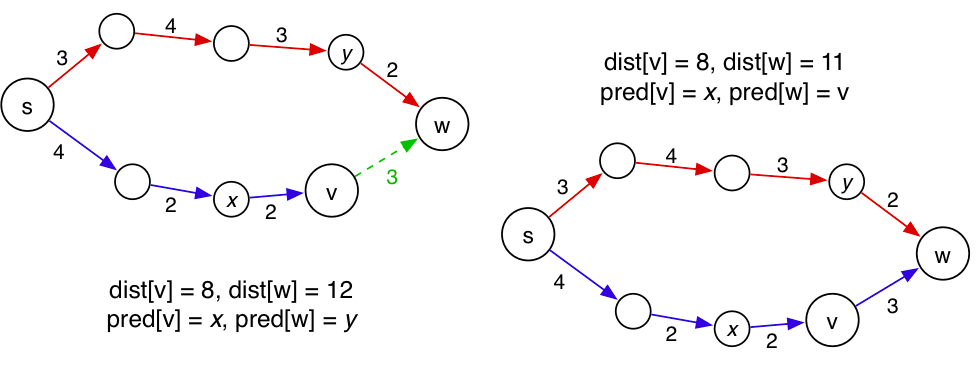
Given = Weighted digraph G, source Vertex S

Result = Shortest paths from S to all other vertices



**EDGE RELAXATION**

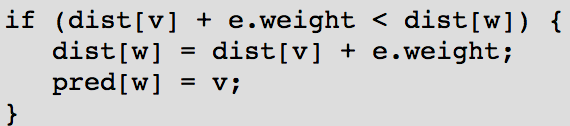
Assume: **dist[]** and **pred[]** as above (but containing data for shortest paths **discovered so far**)



**Relaxation** along edge E from V 🡪 W

* Dist[v] is length of shortest known path from S🡪V
* Dist[w] is length of shortest known path from S🡪W
* If E gives shorter path from S🡪W via. V, then update update **dist[w]** and **pred[w]**

Implementation:

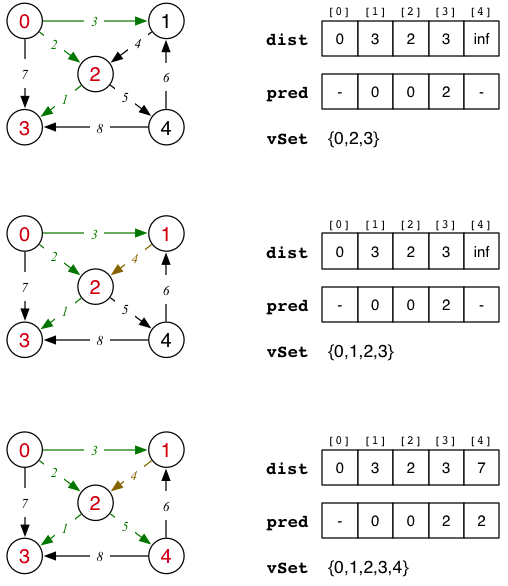


**Relaxation updates dist[ ] / pred[ ] data for W if we find a shorter path from S 🡪 W**

**DIJKSTRA’S ALGORITHM**

This solves the single-source shortest path problem.

**Data** = G, S, dist[], pred[] and **vSet** (set of vertices whose shortest path (so far) from S is known)



Section of execution trace of Dijkstra’s Algorithm.

Steps:

* Start from source vertex **V0**
* Find min weight edge + **relax** along edge **V0 🡪 V2**
  + Add edge weight in **dist[ ] dist[2] = 2**
  + Add prev vertex to **pred[ ] pred[2] = 0**
  + Add new vertex to **vSet**. **vSet {0}**
* Find next set of **known edges** to consider
* Continue to **relax** along min weight edges.
* Reject edges where **cost(Q) > cost(P)** if there is a shorter known path.
* Continue until **all edges are considered**.

Complexity analysis:

* Outer loop = **O(V)** iterations
* PQ updates = **O(LogV)**
* Implementing **find edge e=(S,T,W)**
  + Try all E in EdgeSet = **O(VE)**
  + Classical Djikstra approach = **O(V2)**
    - Consider only edges (s,t,w) where  
      S in vSet and t !in vSet
* Use a PQ to find edge to relax
  + Cost = **V\*CostJOIN + V\*CostLEAVE + ~E\*CostREORDER**

In case 3, cost is dependent on efficiency of PQueue

**SEARCHING**

An extremely common operation in computing

* Given a large collection of **items** and a **key value**, find the item(s) in the collection that contains the key.

As previously:

* **Item** = {key, val1, val2, …} (a struct)
* **Key** = value used to distinguish items (e.g. student ID)

Keys may: