Graph Algorithms

Chapters 6 and 7

Disclaimer: These abbreviated notes DO NOT substitute the textbook for this class. They should be used IN CONJUNCTION with the textbook and the material presented in class. If there is a discrepancy between these notes and the textbook, ALWAYS consider the textbook to be correct. Report such a discrepancy to the instructor so that he resolves it. These notes are only distributed to the students taking this class with A. Gerbessiotis in Spring 2016; distribution outside this group of students is NOT allowed.
Graph representations

Introduction

For a graph $G = (V, E)$ let $n = |V|$ and $m = |E|$.

A graph $G$ is dense if $m \gg n$, and sparse otherwise. In some cases the former condition is restated as $m = \Theta(n^2)$ and the latter condition $m = o(n^2)$.

There are two major ways one can use to represent a graph

1. **Adjacency matrix** representation. A graph $G = (V, E)$ on $n$ vertices and $m$ edges is represented by an $n \times n$ two-dimensional array $A$. Element $A(i, j)$ is 1 if there is an edge $(i, j)$ in the graph, otherwise it is 0. For undirected graphs, $A$ will be symmetric since for every edge $(i, j)$ both $A(i, j)$, $A(j, i)$ are set to 1. In general, $m$ entries in the directed case and $2m$ entries in the undirected case are 1, whereas the remaining ones are 0. If the edges of the graph have weights, then instead of storing a 1 we store in $A$ the weight of the edge. Total space requirements are $O(|V|^2)$.

2. **Distance matrix** is similar to the adjacency matrix; an 1 entry is replaced by the distance between the two vertices (in the direction implied by the edge), and a 0 becomes $\infty$ to denote that no edge exists.

3. **Adjacency list** representation. The graph is represented by $n$ linked lists, one for each vertex of $G$. The linked list for vertex $i$ contains all vertices that are neighbors of $i$, ie it contains all $j$ such that $(i, j)$ is an edge. For an undirected graph, $j$ is in the linked list of $i$ and $i$ is in the list of $j$. Total space requirements are $O(|E|)$. The problem with this representation is that we cannot answer quickly questions of the form ”Is $j$ adjacent to $i$?”

**Note.** The number of ones in the adjacency matrix is $2m$ for an undirected graph and $m$ for a directed graph. This is a consequence of the following facts.

**Fact Undirected.** In an undirected graph $\sum_{v \in V} \deg(v) = 2m$.

**Fact Directed.** In a directed graph $\sum_{v \in V} \text{out}(v) - \deg(v) = m$. 
Graph search processes
DFS: Depth First Search

Depth-Search Search (DFS for short) is a method for traversing graphs. In essence it is a generalization on a graph of the depth-first search method on a tree described earlier. Below we give a very brief outline of how the method works for the case of an undirected graph.

1. Allocate discovered[] such that initially discovered[u] = false for all u
   // It just explores all vertices reachable from u i.e. the connected component of u.
   U-DFS(G,u) // Graph G=(V,E) is undirected.
   0. discovered[u] = true;
   1. for all edges e incident on u
   2. if edge e has not been labeled
   3. let v be the other end-point of e i.e. e=(u,v)
   4. if v is unexplored // i.e. discovered[v] == false
   5. label[e] = tree;
   6. U-DFS(G,v);
   7. else
   8. label[e] = back;

Starting from u we check the adjacency list of u. For every edge e = (u,v) in this adjacency list we identify the other end-point leading from u to it. If v is undiscovered, the edge e is labeled as a tree edge, and a new depth-first-search exploration is initiated at v. Otherwise, if v has already been explored, we label e as a back edge. We can keep track of explored/unexplored vertices v through an array of vertices where we mark an explored vertex.

A more general framework for DFS is shown on the following page(s). It searches a whole graph, not only the portion reachable from a given vertex such as u. It works for both directed and undirected graphs.

This generalized framework labels all vertices with two timestamps:
(a) first visit or discovery (D[] array) of the vertex for the first time, (b) final visit (F[] array) when all vertices in the adjacency list have been discovered.

In addition it labels the edges of the graph with a total of four different labels for the directed case tree (t), back (b), cross (c), forward (f) and the same labels as above (for U-DFS) tree (t), back (b) for the undirected case. Although the labelings are different for an undirected and a directed graph, in both cases, an edge labeled b indicates the existence of a cycle in the graph.

The collection of all this information in depth-first-search can facilitate the easy solution of the problem of, for example, topological sorting.
Depth First Search

Applications

Procedure U-DFS has some interesting applications. The following problem can be solved easily.

**Application 1.** Determine whether $w$ is reachable from $u$.

**Solution.** Run U-DFS($u$). If $\text{discovered}[w]$ is true it means $w$ is reachable from $u$.

**Application 2.** Determine whether graph is connected.

**Solution.** Run U-DFS($u$). If for some vertex $z$ $\text{discovered}[z]$ is false it means $z$ is unreachable from $u$, i.e. the graph is not connected.

**Application 3.** Does $G$ have a cycle that includes $u$?

**Solution.** Run U-DFS($u$). If there is a cycle that includes $u$ then in the U-DFS one of its edges would be labeled as back. The existence of such an edge indicates a cycle. To determine whether that cycles contains $u$, we must trace the cycle. For that in line 5 of U-DFS we add a line that does $\text{parent}[v]=u$ i.e. it keeps track of tree edges in an alternate form. Alternately, we can trace the cycle by walking back the path/cycle in a more laborious way. If for example we are in $v$ we check the adjacency list of $v$ and identify the edge $e$ that is tree that leads to $v$ from some other vertex. The other end-point (i.e. $u$) is traced similarly and so on.

**Application 4.** Find the connected components of $G$.

**Solution.** We start with U-DFS($u$) for an arbitrary $u$. This gives the connected component of $u$. If there is a vertex $w$ that is still undiscovered (we can check this by checking $\text{discovered}[]$) we then initiate a U-DFS($w$) to discover the connected component of $w$, and go on until we discover all vertices.

**Application 5.** Determine whether the graph is bipartite, i.e whether we can color the vertices R or B so that no two adjacent vertices have the same color.

**Solution.** An exam-level question.
Depth First Search
DFS: A general framework

// For an undirected graph DFS labels some edges of the graph as tree edges. Every other edge
// can be labeled implicitly as back edge; a back edge is an indication of a cycle in the graph
// For directed graph we can insert the lines in the far right to label non tree edges accordingly.

dfs(u) // discover u by searching it ***FOR A DIRECTED GRAPH THE FOLLOWING LINES

1. color[u] = GRAY; // Previsit u: Discovery time *** CAN BE ADDED TO LABEL THE EDGES
2. time++; // Previsit u: Discovery time *** OF THE TRAVERSED GRAPH
3. D[u] = time; // Previsit u: Discovery time *** AS tree, forward, cross, back
4. foreach (edge (u,v) in E) { // Visit edge (u,v)
5. if (color[v] == WHITE) { // if WHITE, first visit
6. p[v] = u; (u,v) is a tree edge;
7. dfs(v); // Go Visit v: Discover v
8. } // if
9. } // foreach
10. color[u] = BLACK; // Post visit u: Finalize visit
11. time++; // Post visit u: Finalize visit
12. F[u] = time; // Post visit u: Finalize visit

DFS(G,V,E)

1. for each vertex u { //Initialize
2. color[u] = WHITE;
3. p[u] = NIL;
4. }
5. time = 0; // Start timer
6. for each vertex u { // Start Search
7. if color[u] == WHITE { // Node u non visited yet?
8. dfs(u); // Go for u!
9. }
10. }
Depth First Search
DFS: The algorithm

In order to assist the search process, vertices are colored with three colors during DFS, WHITE, GRAY and BLACK. Initially all vertices are WHITE (i.e. undiscovered). Whenever a WHITE vertex is discovered it is colored GRAY and timestamped (array D[]) and remains so until all the vertices in its adjacency list are discovered and search recursively. When this discovery end, the vertex is colored BLACK and timestamped again (array F[] this time). The process terminates when there are no undiscovered vertices (i.e. WHITE) left.

What is the running time of the Algorithm? $O(|V| + |E|)$. (a) WHITE vertices get visited. As soon as the Visit begins, a WHITE vertex becomes GRAY and remains so until it becomes BLACK. Every vertex is colored WHITE, GRAY or BLACK once for a total of $O(|V|)$ time. A vertex is visited either through line 8 of DFS of line 7 of DFS-Visit.

(b) Every directed edge $(u,v)$ is visited once through line 4 of DFS-Visit. Every undirected edge is visited twice (once as edge $(u,v)$ and once as edge $(v,u)$).
**DFS**

**Directed and Undirected Graphs**

If graph G is an undirected graph we have two types of edges:

(a) **TREE** edges \((u,v)\) such as those visited in line 5 of DFS-Visit for color\((v)\) = WHITE.

(b) **back-edges** \((u,v)\) that connect an ancestor to a descendant such as those visited in line 5 of DFS-Visit for color\((v)\) = GRAY.

If graph G is a directed graph we have four types of edges:

(a) Spanning **TREE** out-edges such as those visited in line 5 of DFS-Visit for color\((v)\) = WHITE.

(b) **BACK** edges directed from descendants to ancestors such as those visited in line 5 of DFS-Visit for color\((v)\) = GRAY. Self loops are back-edges (if allowed).

(c) **FORWARD** edges directed from ancestors to descendants, such as those visited in line 5 of DFS-Visit for color\((v)\) = BLACK and with \(D[u] < D[v]\).

(d) **CROSS** edges that connect vertices that neither is descendant of the other, such as those visited in line 5 of DFS-Visit for color\((v)\) = BLACK and \(D[u] > D[v]\).

**CONVENTIONS:** In examples and exercises, we assume that vertices have labels that are numbers, and the vertices in the adjacency list of any vertex \(u\) are accessed in lexicographic order i.e. smallest label to largest label.
**Graph search processes**

**DFS: Example**

**Assumptions**: Nodes in adjacency list are visited in increasing node number. Nodes are laid out top-to-bottom left-to-right based on increasing node number. We label (directed or undirected as necessary) edges $t$, $f$, $c$, $b$ for tree, forward, cross, and back edges. We label vertex $u$ by pair $(D[u], E[u])$.

Edges $(1,2)$, $(1,3)$, $(3,4)$, $(1,4)$, $(1,5)$, $(5,6)$, $(5,4)$ in directed graph below.

```
(1,8) 1--------
   / \       |
  t/  t\     |
 /   \      |
(2,3) 2 -- (4,7) |
  t/    | b
 /     |
/      |
(5,6) 4--------
```

(c) Copyright A. Gerbessiotis. CS610: Spring 2016. All rights reserved.
**An application of DFS**

**Topological sorting**

A topological sort on a directed acyclic graph \( G = (V,E) \) is a linear ordering of all its vertices such that if \( G \) contains an edge \((u,v)\) then \( u \) appears before \( v \) in the ordering. Note that if the graph is NOT acyclic (i.e. it has cycles) such a linear ordering is not possible.

A linear ordering can be viewed as a listing of the vertices on a horizontal line, so that all directed edges go from left to right. The “sorting” in topological sorting differs from the traditional definition of sorting.

The following algorithm performs topological sorting.

```
TopologicalSort(G)
1. DFS(G);
2. When vertex u is colored BLACK insert u as the head
   of a linked list.
   //i.e. update line 11 to include insert(u,LinkedList);
   //At the conclusion the linked list is represented by
   //using the graph links not the linked-list links that are
   //not interesting (each one points to next element in the list)
3. return(linked list)
```
Another application of DFS

Strongly Connected Components

A directed graph is strongly connected if for every two vertices \(u, v\) there is a path from \(u\) to \(v\) and also a path from \(v\) to \(u\).

One way to determine whether \(G\) is strongly connected is to initiate a DFS (or to be consistent with the notation of page 5, a \texttt{dfs}(u)) starting from every vertex \(u\) of the graph (every time we complete the top-level call we reset all variables and start from scratch with the next vertex). If the first DFS discovers all the vertices (i.e., all vertices get \texttt{color}[u]=B), then \(u\) is connected to every other vertex. Repeating this \(n-1\) additional times we can determine whether the graph is strongly connected or not. Total running time is \(n \times \Omega(n + m)\) i.e. \(O(n(n + m))\). We show next how we can determine strong connectivity in just \(O(n + m)\) time instead.

```
StrongConnectivity(G, V, E)
1. Pick arbitrary \(u\) of \(V\).
2. \texttt{dfs}(u,G); // Use \(G\) as input; otherwise identical to \texttt{dfs}(u) of page 5
3. If there is a vertex \(w\) with \texttt{color}[w] == WHITE
4. return("\(G\) is not strongly connected");
5. else // all vertices are reachable from \(u\)
6. create \(G'=(V, E')\) where \(E'\) contains the reverse of the edges of \(E\)
7. i.e. for every \(e=(a, b)\) of \(E\) add \(e'=(b, a)\) to \(E'\)
8. \texttt{dfs}(u,G'); // page 5 dfs; use \(G'\) not \(G\) as input
9. if there is a vertex \(w\) with \texttt{color}[w] == WHITE
10. return("\(G\) is not strongly connected");
11. else
12. return("\(G\) is strongly connected");
```

The \texttt{dfs} call of line 2 can only determine whether \(G\) is NOT strongly connected. If lines 6-12 are executed, this means that all vertices are reachable from \(u\). We then reverse the directions of all the edges of \(G\) to get \(G'\). In \(G'\) we run DFS from \(u\) again as we did in line 2 for \(G\). If all vertices are reachable from \(u\) in \(G'\) (i.e., we are in line 12 rather than line 10), this means that in \(G'\) we can go from \(u\) to every other vertex \(w\). Given that \(G'\) is the reverse of \(G\) this means that we can go from every \(w\) to \(u\) in \(G\).

Therefore reaching line 12 we can do two things: (a) From \(u\) we can reach every other \(w\) (line 2 leads to line 6), and (b) from every \(w\) we can reach \(u\) (line 6 leads to line 12). This is equivalent to saying that \(G\) is strongly connected. If we want to go from \(a\) to \(b\), we first go from \(a\) to \(u\) and then from \(u\) to \(b\). Of course this defines not necessarily a path, but probably a tour. But it suffices to show that \(G\) is strongly connected anyway. A refinement can show how one can determine a path from this potential tour.
Graph search processes

BFS: Breadth First Search

In BFS we explore vertices starting from a given vertex \( s \) by first finding all vertices adjacent to \( s \), then all vertices reachable from \( s \) within two edge traversals, i.e. at distance two, then those at distance three, and so on.

This graph exploration process can be summarized as follows.

\[
\text{BFS}(s) \\
1. \text{label}[s] = \text{REACHED}; \\
2. \text{QUEUE} = <s>; \\
3. \text{while QUEUE is not empty} \\
4. \quad \text{remove u from queue; // A Dequeue operation} \\
5. \quad \text{for every vertex v in adjacency list of u do} \\
6. \quad \quad \text{if label[v] != REACHED} \\
7. \quad \quad \quad \text{add v to QUEUE; // An Enqueue operation} \\
8. \quad \quad \quad \text{label v REACHED;} \\
9. \quad \quad \quad \text{Maintain fact that v was discovered while traversing (u,v);} \\
10. \quad \} \\
11. \} \\
\]

A more robust implementation of the pseudocode, label vertices by assigning them colors as before for DFS. Step 9 requires a single array and is equivalent to assigning \( p[v] = u \) i.e. indicating that the parent of \( v \) is vertex \( u \). The QUEUE is implemented as a FIFO queue. In addition we may keep track of the distance of a vertex \( v \) from \( s \) by using a distance array \( d[] \) and in step 9 also performing the update step \( d[v] = d[u] + 1 \).
Breadth First Search
BFS: A general framework

BFS(s) //s is the start vertex of the search process on Graph G=(V,E)
1. for each u in V-{s} {
2. \hspace{1em} color[u]=WHITE;
3. \hspace{1em} d[u] = infinity; // d[u] gives the distance of u from s.
4. \hspace{1em} p[u] = NIL // p[u] represents the parent of u in BFS search
5. }
6. color[s]=GRAY;
7. d[s]=0;
8. p[s]=NIL; // s is to be expanded first
9. queue= <s> //A single queue containing s.
10. while queue != <> { // <> stands for an empty queue
11. \hspace{1em} u = Dequeue(queue) //remove an element from queue
12. \hspace{1em} foreach ((u,v) an edge of E) {
13. \hspace{2em} if (color(v)== WHITE) {
14. \hspace{3em} color[v]=GRAY;
15. \hspace{3em} d[v]=d[u]+1;
16. \hspace{3em} p[v]=u;
17. \hspace{3em} queue = Enqueue(queue,v);
18. \hspace{2em} } // if statement ends
19. \hspace{1em} } // foreach statement ends
20. \hspace{1em} color[u]=BLACK // We are done with u (all its neighbors examined)
21. } // while loop ends
Dynamic Programming
Review

Dynamic programming, like the divide-and-conquer method, solves problems by dividing a problem into subproblems, solving the subproblems, and combining the solutions to subproblems. In d-and-c subproblems are usually independent of each other. DP however is used in cases where subproblems are not independent. A DP algorithm solves subproblems once, saves the solutions in a table and avoids recomputing the answer for a problem that has already been solved before. The development of a DP algorithm is broken into a sequence of steps.

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from computed information.

We are going to use dp techniques in solving the following problem.

All-pairs shortest path problem
Given a graph $G = (V, E)$ such that $d_{ij} \geq 0$ is the weight of the edge $(i, j)$ (if no such edge exists, $d_{ij} = \infty$), find the cost of the path with minimal total weight between any two vertices in $G$.

Note that such a path may not be the shortest in terms of edges path. If the cost of a shortest path is $\infty$, then it will mean that no such path exists. We will denote with $D_{ij}$ the total weight of the minimal weight path from $i$ to $j$.

A dynamic-programming based algorithm due to Floyd and Warshall can be used to solve the all-pair shortest path problem. The algorithm works correctly even in the presence of negative edge weights as long as there are NO NEGATIVE WEIGHT CYCLES.
Floyd-Warshall’s algorithm
All-pairs shortest paths

FloydWarshall(G,A,d) //G must NOT HAVE NEGATIVE WEIGHT CYCLES
0 //G MAY HAVE NEGATIVE WEIGHT EDGES.
1. D (i,j)=d(i,j) // Diagonal elements are 0, non-edges are infinity
2. for k=1 to n
3. for i=1 to n
4. for j=1 to n
5. D (i,j)=MIN(D (i,j) , D (i,k)+D (k,j));

Proof of correctness (by Induction).
Inductive assumption. After loop $k = l$ is executed, $D^l_{ij}$ is the cost of the minimal weight path from $i$ to $j$ that ONLY USES internal vertices from \{1,\ldots,l\}.

Basis of induction. Before the loop is executed for the first time, all paths with no internal vertices are single edges; $D^0_{ij} = d_{ij}$ is the cost of an edge path between $i$ and $j$ with NO internal vertices. Subsequently, let us assume that the Inductive assumption is true for $l$. We shall prove it for $l+1$.

Case 1. A shortest path $P$ from $i$ to $j$ using internal vertices from \{1,\ldots,l,l+1\} goes from $i$ into $l+1$ at most once (with internal vertices from \{1,\ldots,l\}) and then from $l+1$ goes to $j$ (with internal vertices from \{1,\ldots,l\}). It cannot go through $l+1$ more than once because this would indicate the existence of a cycle. If such a cycle was part of $P$ as all edge weights are positive (or if negative weights are allowed, there are no negative weight cycles), this cycle could be removed and the resulting path $P'$ from $i$ to $j$ would have total weight less than that of $P$, a contradiction to the minimality of $P$. In $P$, $D^l_{i,l+1}$ is the cost of the first subpath of $P$, and $D^l_{l+1,j}$ is the cost of the second one, and line 5 would detect such a path $P$ if it exists. Therefore $D^{l+1}_{ij}$ will be correctly updated.

Case 2. A shortest path $P$ from $i$ to $j$ with intermediate vertices in \{1,\ldots,l,l+1\} never goes through $l+1$. Then line 5 of the $k = l+1$ loop does nothing, i.e. no update takes place.

Note. Floyd’s algorithm finds ALL minimal weight paths between any two vertices. Its running time is $O(|V|^3)$. It works even with negative edge weights as long as there are no negative weight cycles.
Floyd-Warshall’s algorithm
Transitive Closure and Shortest Paths

The transitive closure problem is defined as follows

**Definition Transitive Closure.** Given a graph $G = (V, E)$ determine whether $v$ is reachable from $u$ for every $u, v \in E$.

**Solution 1.** Run $n$ DFS from every vertex $u$ in total time $O(n(n + m))$.

**Solution 2.** Form a distance matrix $d$ with $d(i,j) = 1$ if edge $(i,j) \in E$ and 0 otherwise. Maintain $\infty$ in the diagonal entries. Run the all-pairs shortest path FloydWarshall. If for a non-diagonal entry of $D^k$ we have $D^k(i,j) > 0$ it means there is a path from $i$ to $j$ of length that value. This is equivalent to solving the transitive closure problem. We have also solved the strong connectivity problem by extension.

**Question.** Is Floyd-Warshall faster than **Solution 1**? Explain.

**Solution 3.** Modify line 5 of Floyd Warshall. The $\text{MIN}$ can be replaced by an $\text{OR}$ and the $+$ by an $\text{AND}$. Explain why this works as claimed.

**Question.** How can we keep track of the paths rather than just the lengths of these paths in Floyd-Warshal? Explain.

**Question.** Think of the $\text{MIN}$ as a $+$ and the $+$ as a $\times$ (times). Can you relate Floyd-Warshall with matrix multiplication? What is $A^n$?

**Question.** How fast can you compute $A^n$? (Hint. Think of Strassen and the exponentiation problem applied to matrices).
Greedy Algorithms

Dijkstra’s algorithm for single-source shortest path

Single-source shortest path problem

Given a graph $G = (V, E)$ with $d_{ij} > 0$ the weight of an edge $(i, j)$ (if no such edge exists, $d_{ij} = \infty$), and a source vertex $s$, find the cost of every minimal total weight path between $s$ and any other vertex of $G$. Edge weights MUST BE POSITIVE for the algorithm to work properly. 

**NOTE:** Remember that shortest refers to total edge weight NOT number of edges of the path!

Dijkstra($G, d, s$)
1. for every $v$ in $V$ do
2. $D[v] = \infty$.
3. $D[s] = 0$;
4. $Q = V$; Build-MinHeap($Q, D$); //Build a min-Heap using $D$ for priorities
5. while $Q$ is not empty do // First node to be extracted is $s$ since $D[s] = 0$
6. EXTRACT from $Q$ vertex $u$ with MINIMUM $D(u)$ //EXTRACT-MINIMUM
7. $S = S \cup \{u\}$;
8. for each $v$ in $Q$ such that $(u, v)$ is an edge do
9. update $D(v) = \min(D(v), D(u) + d(u, v))$ // Also do parent[$v$] = $u$ to keep track of the path

The above algorithm is what we call a greedy algorithm. A greedy algorithm is an algorithm that always makes the choice that looks best at the moment. For a split $S$ and $V - S$ in Dijkstra’s algorithm, that vertex of $V - S$ is added into $S$ that is closest (lightest) to any node of $S$.

Greedy algorithms do not yield in general optimal results. This is however the case with Dijkstra’s Algorithm. Spanning tree problems is another class of problems that can be solved by using greedy algorithms.

Exercise. Show why Dijkstra may not work if a negative weight edge exists.

Exercise. Suppose $G$ has negative weight edges. Does the following method solve the single source shortest path problem? We identify the most negative edge of weight $-W$ in $G$, and then add $W + 1$ to every weight so that all weights become positive. Then we apply to $G$ with the new edge weights Dijkstra’s algorithm. In the reported $D[v]$ at the end we subtract $W + 1$ as many times as the edges from $s$ to $v$. Will this work? Explain.
Dijkstra’s Algorithm
Correctness and Running time

Proof of correctness (by induction).

Inductive hypothesis.
1. For \( u \in S \), \( D(u) \) is the length of the shortest path from \( s \) to \( u \).
2. For \( u \notin S \), \( D(u) \) is the shortest path from \( s \) to \( u \) with intermediate vertices in \( S \) other than \( u \) of course).

Base case In the beginning \( S = \{s\} \). Therefore the minimum \( D(u) \) gives the weight of the lightest edge connecting \( s \) directly to a vertex in \( V - S \). This vertex is \( u \), which is added to \( S \) and (1) becomes true. All neighbors \( v \) of \( u \) then update their \( D(v) \): the shortest path which is internal to \( S \) from \( s \) to \( v \) is (a) either a single edge path from \( s \) to \( v \) (i.e. an edge) of cost \( D[v] = d(u, v) \) (Step 2 of the algorithm), or a path from \( s \) that goes through \( u \); in that case \( D[v] \) is updated, if necessary, by the length of that path, i.e. \( D[u] + d(u, v) \).

Inductive Step. Let the inductive hypothesis apply to any set \( S \) of size \( k \), i.e. \( |S| = k \). We are going to prove that the hypothesis is true for a set \( S \) such that \( |S| = k + 1 \) (inductive step). Let \( u \) be the vertex outside \( S \) with minimum \( D(u) \). Then the shortest path from \( s \) to \( u \) is internal in \( S \) except for a single edge that goes from a vertex of \( S \) directly to \( u \). For otherwise (i.e. the path was not internal in \( S \)) there would be another vertex in that path that from \( s \) to \( u \) that is also in \( V - S \) and that is closer to \( s \) than \( u \), a contradiction to the minimality of \( D(u) \) (NOTE that this is the case because positive weight edges are only allowed).

Running time of the algorithm. Suppose that \( Q \) uses an adjacency list representation of \( G \). Line 6 is performed \(|V| - 1\) times and finding the minimum takes time \( O(|V|) \) each iteration for a total time for line 5 of \( O(|V|^2) \). A vertex extracted from \( Q \) gets into \( S \) once and only once. Whenever this happens all its neighbors are examined. Therefore lines 8-9 are executed once for every edge for a total of \( O(|E|) \). Total running time is \( O(|E| + |V|^2) \).

If we implement \( Q \) with a priority queue (MIN-heap), lines 1-2 build a queue \( Q \) in \( O(|V|) \) time. Line 6 is performed \(|V| - 1\) times for the cost of Extract-Min from a heap in \( O(lg |V|) \) steps for a total cost of \( O(|V| lg |V|) \). Lines 8-9 are executed once for every edge. Each time a Decrease-Key operation is performed for a total cost of \( O(|E| lg |V|) \). Therefore total time is \( O(|E| + |V|) lg |V|) \). Note this running time is better for sparse graphs than \( O(|E| + |V|^2) \).
Greedy Algorithms for minimum spanning trees

Introduction

A spanning tree of an undirected connected graph $G$ is a tree that saturates/touches every vertex of the graph.

**Fact 1.** Every spanning tree of a graph with $n$ vertices has $n - 1$ edges.

If graph edges have weights, a minimum spanning tree is a spanning tree for which the sum of its weights is minimum.

We present below two algorithms for finding minimum spanning trees. Both algorithms are greedy.

Kruskal’s algorithm is described below.

Kruskal’s Algorithm($G, V, E$)
1. $T = \{ \}$; // empty set
2. while (T has fewer than $n - 1$ edges)
3. add to T the shortest edge that does not make T have a cycle

Prim’s algorithm is described below.

Prim’s Algorithm($G, V, E$)
1. $T = \text{empty}; X=\{r\}; Q=V$; // $r$ is an arbitrary vertex of $V$
2. while ($X \neq V$) {
3. let $e=(u,v)$ be lowest cost edge from $u$ in $X$ to $v$ in $Q$
4. $T = T \cup \{(u,v)\}$;
5. $X=X \cup \{v\}$; $Q = Q \setminus \{v\}$;
6. }

**Lemma.** If $G = (V, E)$ is a connected undirected graph and $S = (V, T)$ is a spanning tree of $G$, then

(1) For all $v_1, v_2 \in V$ a path between $v_1, v_2$ is unique.

(2) Any edge in $E - T$ added to $S$ creates a unique cycle.

(3) If an edge in $E - T$ is added to $T$ and then another edge is removed from the resulting cycle, we get a new spanning tree.
Greedy Algorithms for spanning trees
Prim’s Algorithm

Prim(G,V,E,W) // W: weight matrix i.e. weight of edge (u,v) is w(u,v)
1. Q=V;
2. foreach v in V do {
3.     cost[v] = infinity;
4.     p(v) = NIL ; //parent of v initialization
5. }
6. cost[r] = 0; p(r)= NIL; // Set X= {r} in previous pseudocode
7. while Q is not empty {
8.     u=EXTRACT-MIN(Q);
9.     for each v in Adj(u) and v in Q {
10.        if w(u,v) < cost[v] {
11.            p(v)=u;
12.            cost[v]=w(u,v);
13.        }
14.     }
15. }

(c) Copyright A. Gerbessiotis. CS610 : Spring 2016 . All rights reserved.
Proposition Prim. Let \( X \subset V \). If \((u,v)\) is the lowest cost edge from \( X \) to \( Q = V - X \), then there is a minimum cost spanning tree that includes \((u,v)\).

Proof (by contradiction).
1. Let us assume by way of contradiction that all minimum spanning trees DO NOT contain \((u,v)\).
2. Let \( T' \) be a minimum spanning tree and by 1. it does not include \((u,v)\).
3. Consider edge \((u,v)\). Since \( T' \) is a spanning tree there is a path in that tree that connects \( u \) to \( v \). This path does not include of course edge \((u,v)\) by way of 1.
4. Consider \( T' \cup \{ (u,v) \} \). Subgraph has a cycle. The remainder applies to this subgraph of \( G \).
5. In the path from \( u \in X \) to \( v \in V - X \) there must exists another vertex \( u' \in X \), and \( v' \in V - X \) such that \((u',v')\) \( \in E \).
6. Prim’s algorithm chose \((u,v)\) over \((u',v')\), i.e. \( w(u,v) \leq w(u',v') \).
7. Consider \( N = T' - (u',v') + \{ (u,v) \} \).
8. \( N \) has \((u,v)\) and \( \text{cost}(N) = \text{cost}(T') - w(u',v') + w(u,v) \leq \text{cost}(T') \)
9. \( N \) is a minimum spanning tree that includes \((u,v)\), which is a contradiction to 1.

Running Time. We implement \( Q \) with a binary heap. Steps 1-5 take \( O(|V|) \) time. Each vertex is extracted exactly once and the number of vertices checked in line 9 is at most \( 2|E| \) ie \( O(|E|) \). Line 8 takes \( O(|V| \lg |V|) \) time, lines 9-12 are implemented using a Decrease-Key (see Dijkstra’s algorithm) operation that requires \( O(|V| \lg |V|) \) time per call for a total of \( O(|E| \lg |V|) \). Total time is therefore \( O((|E| + |V|) \lg |V|) \). Note that the test in line 9 of membership in \( Q \) is implemented by having one extra bit per vertex. This bit is set to 1 for a vertex in \( Q \). For a removed vertex it is set to 0.
Greedy Algorithms for minimum spanning trees

Kruskal’s Algorithm

\[
\text{Kruskal}(G,V,E,w) \\
1. \text{T}=\text{empty}; \\
2. \text{for each } u \text{ in } V \\
3. \quad \text{do Make-Set}(u); \\
4. \quad \text{sort edges of } E \text{ by nondecreasing weight } w; \\
5. \quad \text{for each edge } (u,v) \text{ in } E \text{ in nondecreasing weight order} \\
6. \quad \text{do if } \text{Find-Set}(u) \neq \text{Find-Set}(v) \\
7. \quad \quad \text{then } \text{T}=\text{T} \cup \{(u,v)\}; \\
8. \quad \quad \text{Union}(u,v);
\]

**Proof of correctness (by contradiction).** Graph \( T = \{e_1, \ldots, e_{n-1}\} \) generated by Kruskal’s algorithm is definitely a tree that spans \( G \) if \( G \) is connected. Suppose that \( T \) is not a minimal spanning tree. Let \( T' \) be the minimal spanning tree of \( G \) with the maximum number of edges common with \( T \), and let this maximum number of common edges be \( k-1 \). Then \( e_k = (a,b) \) is the first edge in \( T \) not in \( T' \). Let \( P \) be the path in \( T' \) that connects \( a, b \) (as edge \( (a,b) \) is not in \( T' \)). Let \( G_k \) be the connected component containing \( \{e_1, \ldots, e_{k-1}\} \) that saturates (touches) \( a \). Then there exists an edge \( e \) in \( P \) connecting \( G_k \) to a vertex outside \( G_k \). Since the algorithm chooses \( e_k \) over \( e \) in \( T \) \( w(e_k) \leq w(e) \). But in \( T' - e + e_k \) is a spanning tree at least as cheap as \( T' \), but with one more edge in common with \( T \) than \( T' \) a contradiction to the choice of \( T' \).

**Running Time.** Steps 1-3 take \( O(|V|) \) time. Sorting takes \( O(|E| \lg |E|) \) time. Steps 5, 6, 7 are executed \( O(|E|) \) times if an adjacency list representation of the graph is considered, and total cost by prior discussion is \( O(|E| \lg |E|) \) as well.
1.1 Ranking based on similarity first: $s(q,d_j)$. In order to solve the Web-search problem, for a given query $q$ we need to "compare" the query to every document $d_j$ of the collection (we call the collection the corpus). This "comparison" checks whether the terms of the query appear (or not) as needed in document $d_j$ for every $j$ (i.e. for all documents of the corpus). This comparison establishes the similarity between the query $q$ and the particular $d_j$ and the similarity measure is stored in $s(q,d_j)$. The measure can be as simple as a Boolean true or false or more complicated if the similarity function $s$ distinguishes between terms

- that appear in the title of a page (e.g. HTML <TITLE> context),
- emphasized (<B> or <EM>) or increased font size,
- appear as text in a hypertext link
- appear in the document more than once
- several terms of the query appear close together as needed.

Thus the **relevance of a query to a document** (e.g. a web-page) is established by first computing the function $s(q,d_j)$ that determines the similarity between query $q$ and document $d_j$. This similarity measure is based solely on retrieval-based knowledge (e.g. the searchable contents of document $d_j$).

1.2 Ranking based on link structure of $d_j$. Search engines (or web-searching methods in general) do not rely on this measure exclusively to rank a web-page (e.g. document $d_j$ relative to query $q$). Search engines also use additional information such as the **link structure** of $d_j$ to generate the rank $R(q,d_j)$. The link structure relates to information such as:

- (a) what documents/pages are pointed by $d_j$ and are these pages important (e.g. **authoritative**),
- (b) what documents/pages point to $d_j$ and are these pages important (e.g. **hubs of information such as a directory page**).

1.3 First ranking model: Vector Spread Activation Model. One of the first attempts to use linkage information was the **vector spread activation model** by Yuwono. The ranking score $R(q,d_j)$ is the sum of the similarity $s(q,d_j)$ of $d_j$ plus a portion of the similarity of each document that points to $d_j$. This is under the assumption that if relevant documents point to $d_j$, then $d_j$ should also be relevant. Let $l(i,j)$ be 1 or 0 based on whether document $d_i$ points to $d_j$ or not and $\beta$ is a user defined parameter (e.g. $\beta = 0.2$). Then $R(q,d_j) = s(q,d_j) + \beta \sum_i R(q,d_i)l(i,j)$.
2.1 Another model: Kleinberg’s HITS model. Kleinberg (1998) introduced a model that assigns two ranks to each webpage/document. One rank signifies how important the document is as a hub of information, and the other how important it is as an authoritative source of information. The resulting ranking algorithm is sometimes referred to as HITS i.e. a Hypertext Induced Topic Search. The ideas behind the model can be summarized as follows.

2.2 HITS modeling: Hubs and Authorities.

(1) A page is an authoritative page if it is referenced by many hub pages that are relevant to the query,

(2) a page is a hub page for a query if it points to many authoritative pages for that query, and

(3) good authoritative and hub pages reinforce one another.

2.3 Finding pages for a query \( q \). How can one find such authoritative and hub pages for a query \( q \)?

2.3.1 Step 1: Submit query \( q \) to a similarity-based engine and record the top pages i.e. the root set \( RS(q) \) pages based on \( s(q,d_j) \) results.

2.3.2 Step 2: Expand set \( RS(q) \) into the base set \( BS(q) \) to include pages pointed by \( RS(q) \) pages. That is,

\[
BS_1(q) = \{ \text{all pages pointed by an } RS(q) \text{ page} \}.
\]

2.3.3 Step 3: Also include into \( BS(q) \) pages pointing to \( RS(q) \) pages (if possible, but restrict the number for each \( RS(q) \) page to no more that a number say \( B \)) that is

\[
BS_2(q) = \{ B \text{ pages pointing to an } RS(q) \text{ page} \}, \text{ for some fixed } B.
\]

2.3.4 Step 4: Thus

\[
BS(q) = RS(q) \cup BS_1(q) \cup BS_2(q).
\]

2.3.5 Authority and Hub computation on \( BS(q) \) only: \( d_j \) and \( a(d_j) \) and \( h(d_j) \). For the subgraph induced by \( BS(q) \), let \( E \) be the set of links (i.e. edges) induced by it. A link from document \( d_i \) to \( d_j \) will be denoted by the pair \((i,j)\). For each document \( d_j \), we shall compute the authority score \( a(d_j) \) of \( d_j \), and the hub score \( h(d_j) \) of \( d_j \) by performing an A (for authority) update followed by an H (for hub) update in that order.
2.4 Authority and Hub vectors for all \(d_j\). All the \(a(d_j)\) and \(h(d_j)\) values are collected into two vectors \(a^t, h^t\), i.e. containing \(n\) rows and one column! The \(^t\) indicates the transpose, i.e. a vector of \(n\) rows \(a, h\) is shown as row vector \(a^t, h^t\) of one row and \(n\) columns.

\[
h^t = (h(d_1), \ldots, h(d_n)) \quad a^t = (a(d_1), \ldots, a(d_n))
\]

Those two vectors \(a, h\) will be updated synchronously (most often) for a number of iterations. The \(i\)-th iteration will be denoted by a superscript \(^{(i)}\) and values of those vectors will be denoted accordingly by \(a^{(i)}\) and \(h^{(i)}\) respectively.

2.5.1 A: Authority update step. The first update involves the authority scores. The previously available hub scores are being used for this update. If an edge points from \(i\) to \(j\) i.e. \((i, j) \in E\) then the authority of \(d_j\) is to be updated using the score of \(h(d_i)\). Therefore, for every link \((i, j) \in E\) that points to document \(d_j\)

\[
a(d_j) = \sum_{(i, j) \in E} h(d_i).
\]  

2.5.2 H: Hub update step. For the hub update step the just computed authority values are to be used. For every link \((j, k) \in E\) of documents \(d_k\) pointed by \(d_j\)

\[
h(d_j) = \sum_{(j, k) \in E} a(d_k).
\]

2.6 Example 0. For the three vertex graph shown below

```
0 -----> 1
 \ / \
 v v
 v v
\ / \
0: 1 2
```

Adjacency matrix \(L = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}\) \(L^t = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}\)

**A step:**

\[a(0) = 0\]
\[a(1) = h(0) = \begin{bmatrix} a(1) \end{bmatrix} = L^t \ast \begin{bmatrix} h(1) \end{bmatrix} \Rightarrow a = L^t \ast h\]

**Adjacency List**

\[a(2) = h(0)+h(1)\]

```
2
\```

**H step:**

\[h(0) = a(1)+a(2) = \begin{bmatrix} h(0) \end{bmatrix}\]
\[h(1) = a(2) = \begin{bmatrix} h(1) \end{bmatrix} = L \ast \begin{bmatrix} a(1) \end{bmatrix} \Rightarrow h = L \ast a\]
\[h(2) = 0\]

(c) Copyright A. Gerbessiotis. CS610 : Spring 2016 . All rights reserved.
2.7 An A step precedes an H step (An H step follows an A step). The two steps of Eq. 1 and Eq. 2 are interwined. One first uses the \( h \) values of a previous computation (iteration) to get the \( a \) values in the A step. The newly computed \( a \) values are then used to generate a new set of \( h \) values in the H step.

2.8 Adjacency Matrix relationship of the \( a^{(i)} \) and \( h^{(i)} \) vectors. Let \( L \) be the adjacency matrix over the set of links \( E \) of \( BS(q) \). With this we mean that \( L(i, j) = 1 \) if \( (i, j) \in E \). Otherwise, \( L(i, j) = 0 \) for \( (i, j) \notin E \). Let \( L^t \) be the transpose matrix of \( L \) i.e. one that results if the rows of \( L \) become columns in \( L^t \) and the columns respectively rows. Let \( h^{(i)}, a^{(i)} \) be the hub, authority vectors after \( i \) iterations of the A,H steps, for the \( n \) documents of the corpus.

\[
\begin{align*}
    h^{(i)' } &= (h^{(i)}(d_1), h^{(i)}(d_2), \ldots, h^{(i)}(d_n)), \\
    a^{(i)' } &= (a^{(i)}(d_1), a^{(i)}(d_2), \ldots, a^{(i)}(d_n)),
\end{align*}
\]

Then by rewriting Eq. 1 (see also Example 0 of the previous page) we have

\[
a^{(i)} = L^t h^{(i-1)}
\]

and by rewriting Eq. 2 we obtain

\[
h^{(i)} = L a^{(i)}.
\]

2.8.1 Note the superscripts above! Note the superscripts on the right side of the equation mark in both equations. In the former case it refers to iteration \( i - 1 \) and in the latter case to iteration \( i \). Note also that the order of execution matters. First the authority values are updated for a given iteration \( i \) using the \( i - 1 \) iteration hub values, and then the hub values get updated using the iteration \( i \) authority values just computed.

2.9 Example 0 continued. For the previous example, let the initial values for the \( a, h \) vectors be all ones.

<table>
<thead>
<tr>
<th>A step:</th>
<th>a(0) = 0</th>
<th>h(0)</th>
</tr>
</thead>
</table>
| a(1) = h(0) | [ a(1) ] = [ h(1) ] | \( \Rightarrow a = L^t \ast h \)
| a(2) = h(0)+h(1) | [ a(2) ] = [ h(2) ] |

<table>
<thead>
<tr>
<th>Adjacency List</th>
</tr>
</thead>
<tbody>
<tr>
<td>0: 1 2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>H step:</th>
<th>h(0) = a(1)+a(2)</th>
</tr>
</thead>
</table>
| h(1) = a(2) | [ h(1) ] = L \ast [ a(1) ] \( \Rightarrow h = L \ast a \)
| h(2) = 0 | [ h(2) ] = [ a(2) ] |

<table>
<thead>
<tr>
<th>Step 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
</tr>
<tr>
<td>1 1 1</td>
</tr>
<tr>
<td>a: 1 1 1</td>
</tr>
</tbody>
</table>

| H: |
| 3 2 0 |

| scale : |
| 0.83 0.55 0.0 |
| 0.0 0.44 0.89 |
2.10 Power method. These recursive formulae of (2.8) are interwined. If we unfold them and unwind them, then we get the following. This is also known as the power method.

\[ a^{(1)} = L^t h^{(0)} \quad h^{(1)} = L a^{(1)} \]
\[ a^{(2)} = L^t h^{(1)} \Rightarrow a^{(2)} = L^t (La^{(1)}) = (L^t L) a^{(1)} \]
\[ h^{(2)} = La^{(2)} \Rightarrow h^{(2)} = L (L^t h^{(1)}) = (LL^t) h^{(1)} \]

Thus after the first iteration, the values of \(a,h\) can then be multiplied by a power of the product \(LL^t\) or \(L^tL\) to generate the \(a,h\) values in any iteration as shown on the right hand side below.

\[ a^{(i)} = L^t h^{(i-1)} \Rightarrow a^{(i)} = L^t La^{(i-1)} \Rightarrow \ldots \Rightarrow a^{(i)} = (L^tL)^{i-1} a^{(1)} \]
\[ h^{(i)} = La^{(i)} \Rightarrow h^{(i)} = LL^t h^{(i-1)} \Rightarrow \ldots \Rightarrow h^{(i)} = (LL^t)^{i-1} h^{(1)} \]

2.11 Instability and Scaling: Avoiding growing values. One step is missing in this computation. The values \(a^{(i)}, h^{(i)}\) can grow large as they are dependent on \((L^tL)^i\) and \((LL^t)^i\). Thus a scaling needs to be performed immediately after a round of \(A\) and \(H\) updates gets completed to make sure than those values are no more than 1. This scaling is shown below. The scaling step can appear as step 2.5.3 immediately after the 2.5.2 \(H\) step of page 24.

\[ a^{(i)}(d_j) = \frac{a^{(i)}(d_j)}{\sqrt{\sum_k (a^{(i)}(d_k))^2}} \quad h^{(i)}(d_j) = \frac{h^{(i)}(d_j)}{\sqrt{\sum_k (h^{(i)}(d_k))^2}} \]

For the computation to work properly and this iterative process to converge, and Algorithm \textit{HubAuthority-Rank} (shown on the next page) to be complete, we need to explain two missing parts from this exposition.

\textbf{Issue 1.} What are the initialization vectors \(a^{(0)}\) and \(h^{(0)}\)? (The answer is in line 4 of Algorithm \textit{HubAuthority-Rank}.)

\textbf{Issue 2.} Why does the algorithm converge? How fast is convergence? What are the preconditions for convergence?
Algorithm HubAuthority-Rank
1. Obtain RS(q) for query q.
2. Expand RS(q) into BS(q) as described.
3. Use BS(q) to obtain a graph G=(V,E) of links with adjacency matrix L
4. For each dj initialize hub and authority values to 1 i.e. a(dj) = h(dj) = 1;
5. for every document dj component of a,h repeat {
   6. Run the authority update step A to get a(dj) from h(...)
   7. Run the hub update step H to get h(dj) from a(...)
   8. Scale a(dj) h(dj) as explained.
} until convergence is reached
9. Return documents ordered by authority scores (high first)

Figure 1: Kleinberg’s HITS algorithm computation

In the scaling step one can remove the superscripts that indicate iteration and have no other significance in the computation to obtain a simpler

\[ a(d_j) = \frac{a(d_j)}{\sqrt{\sum_i a^2(d_i)}}, \quad h(d_j) = \frac{h(d_j)}{\sqrt{\sum_i h^2(d_i)}}. \]

(Note that the denominator is computed partially and only once by the end of step 7, then inverted so that multiplication rather than the more expensive division is performed.)
**2.12 Initial values to address Issue 1.** The initial values of \( a(0) \) and \( h(0) \) affect convergence. In order to have convergence, \( a(0) \) and \( h(0) \) must be chosen in such a way that they are not orthogonal to the solution \( a(i) \) and \( h(i) \) of the \( i \)-th iteration. Note also that only \( h(0) \) needs to be initialized, not \( a(0) \) since the former is only used in the first A step to generate \( a(1) \). In the subsequent H step, the \( h(1) \) will use the \( a(1) \) values just computed.

2.12.1 **One choice of initial values: all-one values.** A choice \( a(0) = h(0) = e \), where \( e \) is the unit vector (i.e. all its components are ones) suffices.

2.12.2 **One choice of initial values: all-1/n values.** Another alternative is to set \( a(0) = h(0) = e/n \) i.e. scale the initial values.

2.12.3 **A choice of initial values: all-1/√n values.** Another alternative is to set \( a(0) = h(0) = e/√n \) i.e. scale the initial values according to the scaling method of the A and H steps.

2.12.4 **Full Answer to Issue 1.** Initial values are all ones for the hub and authority vector elements as indicated in step 4 of the Algorithm HubAuthority-Rank. A ”more appropriate” alternative is \( e/√n \) though.

**2.13 Convergence to address Issue 2.** Both matrices \( LL^t \) and \( L^tL \) are well-defined and by matrix analysis (e.g. Perron-Frobenius theorem) vectors \( a(i) \) and \( h(i) \) will then converge to the dominant eigenvector of one or the other matrix.

2.13.1 **Rate of convergence.** The rate of convergence however depends on the difference (i.e. eigengap) between the largest \( \lambda_1 \) and second largest \( \lambda_2 \) eigenvalues of the two matrices. HITS usually converges (but no guarantee is offered) to a solution after 10-20 iterations.

2.14 **A mild critique of the mathematical properties of HITS.** One problem with HITS is the existence of multiple connected components in the graph represented by \( L \). Then the algorithm might ignore all but one of these components when it converges. We are brief at this point on the mathematical properties of HITS; a more thorough discussion that also relates to HITS will deal with Google’s PageRank algorithm.
For the example of page 25 or so, we show the computations involved for 7 iterations using (a) initial all-one vectors, (b) initial all-1/n vectors, and (c) initial all-1/√n vectors.

Kleinberg's HITS (Hub and Authority) Algorithm

(a) vectors are initialized to = 1.00000
Iterat : 1 : A/H [ 0]=0.00000/0.83205 A/H [ 1]=0.44721/0.55470 A/H [ 2]=0.89443/0.00000
Iterat : 2 : A/H [ 0]=0.00000/0.84800 A/H [ 1]=0.51450/0.53000 A/H [ 2]=0.85749/0.00000
Iterat : 3 : A/H [ 0]=0.00000/0.85027 A/H [ 1]=0.52410/0.52635 A/H [ 2]=0.85166/0.00000
Iterat : 4 : A/H [ 0]=0.00000/0.85069 A/H [ 1]=0.52849/0.52882 A/H [ 2]=0.85080/0.00000
Iterat : 5 : A/H [ 0]=0.00000/0.85064 A/H [ 1]=0.52780/0.5274 A/H [ 2]=0.85067/0.00000
Iterat : 6 : A/H [ 0]=0.00000/0.85065 A/H [ 1]=0.52783/0.5275 A/H [ 2]=0.85065/0.00000
Iterat : 7 : A/H [ 0]=0.00000/0.85065 A/H [ 1]=0.52783/0.5275 A/H [ 2]=0.85065/0.00000

(b) vectors are initialized to = 0.33333
Base : 0 : A/H [ 0]=0.33333/0.33333 A/H [ 1]=0.33333/0.33333 A/H [ 2]=0.33333/0.33333
Iterat : 1 : A/H [ 0]=0.00000/0.83205 A/H [ 1]=0.44721/0.55470 A/H [ 2]=0.89443/0.00000
Iterat : 2 : A/H [ 0]=0.00000/0.84800 A/H [ 1]=0.51450/0.53000 A/H [ 2]=0.85749/0.00000
Iterat : 3 : A/H [ 0]=0.00000/0.85027 A/H [ 1]=0.52410/0.52635 A/H [ 2]=0.85166/0.00000
Iterat : 4 : A/H [ 0]=0.00000/0.85069 A/H [ 1]=0.52849/0.52882 A/H [ 2]=0.85080/0.00000
Iterat : 5 : A/H [ 0]=0.00000/0.85064 A/H [ 1]=0.52780/0.5274 A/H [ 2]=0.85067/0.00000
Iterat : 6 : A/H [ 0]=0.00000/0.85065 A/H [ 1]=0.52783/0.5275 A/H [ 2]=0.85065/0.00000
Iterat : 7 : A/H [ 0]=0.00000/0.85065 A/H [ 1]=0.52783/0.5275 A/H [ 2]=0.85065/0.00000

(c) vectors are initialized to = 0.57735
Base : 0 : A/H [ 0]=0.57735/0.57735 A/H [ 1]=0.57735/0.57735 A/H [ 2]=0.57735/0.57735
Iterat : 1 : A/H [ 0]=0.00000/0.83205 A/H [ 1]=0.44721/0.55470 A/H [ 2]=0.89443/0.00000
Iterat : 2 : A/H [ 0]=0.00000/0.84800 A/H [ 1]=0.51450/0.53000 A/H [ 2]=0.85749/0.00000
Iterat : 3 : A/H [ 0]=0.00000/0.85027 A/H [ 1]=0.52410/0.52635 A/H [ 2]=0.85166/0.00000
Iterat : 4 : A/H [ 0]=0.00000/0.85069 A/H [ 1]=0.52849/0.52882 A/H [ 2]=0.85080/0.00000
Iterat : 5 : A/H [ 0]=0.00000/0.85064 A/H [ 1]=0.52780/0.5274 A/H [ 2]=0.85067/0.00000
Iterat : 6 : A/H [ 0]=0.00000/0.85065 A/H [ 1]=0.52783/0.5275 A/H [ 2]=0.85065/0.00000
Iterat : 7 : A/H [ 0]=0.00000/0.85065 A/H [ 1]=0.52783/0.5275 A/H [ 2]=0.85065/0.00000
3.0.1 Limitations of the HITS algorithm. The HITS algorithm has certain limitations. To start with we need access to a similarity search engine to obtain $RS$ plus the additional steps to formulate $BS$. Subsequent ranking depends on the $BS$ that depends on the given query $q$. For this reason we write $BS(q)$ to indicate that $BS$ depends on the query $q$. Moreover, HITS uses two vectors $a, h$. If the web-graph related to $BS(q)$ has isolated components the HITS algorithm may focus on one such component and miss the other ones (midredirection). It is not clear whether the $a$ vector should be used for rank, or the $h$ vector or it makes sense to use a combination of the two and what that combination should be!

3.0.2 Nice features of HITS. The iterative (aka power method) form of the HITS algorithm makes it easy (though time consuming) to establish hub and authority values and makes matters of convergence and existence of $a, h$ values contingent of explicit conditions of the matrix of document (web-page) connections.

3.1 Google’s PageRank algorithm. The algorithm used to ranking Google query results is now presented. The original algorithm, named PageRank was introduced in 1998. How much it is still being used in Google is open to discussion.
**Page Rank**

**Introduction**

3.2 Idea behind Google’s view of the Web: PageRank vs HITS. View the web as a directed graph \( G = (V, E) \). The HITS algorithm views as a directed graph not the web but only the base set \( BS(q) \); to obtain \( BS(q) \) one has to submit \( q \) to a similarity-based search engine, obtain \( RS(q) \) and then enhance it into \( BS(q) \). Google’s view of the web is a direct view skipping all those intermediate steps.

3.2.1 Forward (outgoing) links and backlinks (incoming) establish a rank independent of \( q \). Each page \( p \) has outgoing/forward links and incoming/backlinks. Each backlink is a citation of the named page \( p \); each outgoing link cites the page pointed by \( p \). Google’s original ranking algorithm named **PageRank** is a measure of global web-page importance not of an importance based on the search query results of a similarity search-engine.

3.2.2 1998 PageRank and Google Toolbar PageRank. With respect to that measure Google’s rank is (naturally) very high, MIT’s is a bit less, and NJIT scores a little less than both. Note that this **PageRank** algorithm that we will be discussing in the remainder of this Subject reflects Google’s state in 1998. Google is currently using a slightly different “PageRank” measure and makes it available to Google users through what is known as Google Toolbar (Internet Explorer); around 2012 it was a number between 0 and 10 (Google itself used to be a 10, MIT had a 9 rank, and NJIT an 8).

3.3 Graph of the Web and Browsing: Random Surfing. The idea behind PageRank is simple. Consider the web to be a graph and do the following random walk on it by following the protocol described by the following two rules.

**Rule 1.** For \( d \times 100\% \) (say 85\%) of the time, a user follows (outgoing) links (uniformly) at random when the user browses a web-page. (For example, if a web-page has 5 links the user picks one of them at random.) Then the user follows such a selected outgoing link and browses it.

**Rule 2.** For \( (1 - d) \times 100\% \) of the time, a web-page might become a sink (no linked pages) or the user just gets bored browsing the current web-page. So instead of using Rule 1 the user decides to pick a random page (not pointed by the current page) to visit next. (The choice of a random page is one among the \( n \) web-pages of the whole Web.)

3.4 Parameter \( 1 - d \) decides how often the user gets ”bored”. Parameter \( d \) is a number between 0 and 1. For example one might choose \( d = 0.85 \).

3.5 Fundamental Question. What is the probability that a given page \( A \) will be visited? This determines the ”importance” i.e. the ”rank” of the page!
3.6 Idea of ranking web-pages. If a page $p$ is linked by many pages then $p$ is important. If a page $p$ is linked by few but important pages then $p$ should also be important. The importance or not of a page is divided evenly and also inherited to the pages pointed by it. Suppose we have pages $T_1, \ldots, T_m$ pointing to page $A$. The PageRank $PR(A)$ of $A$ is then defined by the pagerank of the pages pointing to $A$.

$$PR(A) = \frac{(1-d)}{n} + \frac{1}{d} \cdot \left( \frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_m)}{C(T_m)} \right),$$ (3)

where $PR(T_i)$ is the page rank of $T_i$ and $C(T_i)$ is the number of outgoing links of $T_i$, and $n$ is the total number of pages in the corpus (web). (Sometimes instead of $(1-d)/n$ one can use $(1-d)$ this cause some matrix or algebra-related problems though.)

3.6.1 Technical issues to resolve: How to avoid sinks. Each page has its own PageRank (eg. $PR(A), PR(T_i)$). Each page is recording its number of outgoing links i.e. $C(T_i)$. The RageRank of a page $T_i$ is evenly distributed among the pages pointed by $T_i$ and one of them is $A$. Each such page gets a rank contribution of $PR(T_i)/C(T_i)$. Even if a page has no citations (i.e. incoming links) it will still have a non-zero rank of $(1-d)/n$ (though for $n$ a billion or so it would be negligibly small). To be fair to every web page, all pages’ ranks get pumped up by $(1-d)/n$. The choice of $d$ is important in controlling the rate of convergence in a PageRank computation. (We have $(1-d)/n$ instead of $1-d$ to avoid instability issues by keeping PageRanks smaller than one, and also because we do not want to be too generous.)

3.6.2 PageRank $PR(A)$ for $A$. The PageRank $PR(A)$ of web-page $A$ thus

$$PR(A) = \frac{(1-d)}{n} + \frac{1}{d} \cdot \left( \frac{PR(T_1)}{C(T_1)} + \ldots + \frac{PR(T_m)}{C(T_m)} \right) = \frac{1-d}{n} + d \cdot \sum_{i=1}^{m} \frac{PR(T_i)}{C(T_i)}$$ (4)

3.6.3 Note: Rank as ”probability”. The normalized rank of Eq. (4) could be interpreted to signify probabilities. Thus one might interpret $PR(A)$ as the probability of visiting web-page $A$ during the random walk obtained through the application of the two rules, Rule 1 and Rule 2.
3.7.1 **In order to compute** $PR(A)$ **we need to know** $PR(T_i)$. To evaluate the rank $PR(A)$ of $A$ one needs to know the ranks $PR(T_i)$ of all pages $T_i$ pointing to $A$.

3.7.2 **In order to obtain** $PR(T_i)$ **we need to know** $T_i$. And this requires that we know in advance all the pages that point to $A$ which is a little bit more elaborate than finding the pages pointed by $A$. The latter is easy: scan document $A$ and locate all anchor fields to determine pages pointed by $A$. But finding $T_i$ has not yet been resolved.

3.7.3 **Even if we know** $T_i$ **to find** $PR(T_i)$ **we need to find the pages pointing to** $T_i$! Not only that but in order to find these ranks $PR(T_i)$ of $T_i$’s one needs to know the ranks of the pages pointing to the $T_i$’s.

3.7.4 **Chicken and egg problem?** What if we have a cycle or a very long back-chain?.

3.7.5 **Matrix theory to the rescue.** Matrix theory and the power method alluded in the HITS algorithm (see pages 25 and 26) can be used to show that using a simple iterative algorithm (that was also employed in a different form in HITS), $PR(A)$ can be computed in a few iterations and the rank vector (of all $n$ web-pages) computed corresponds to the principal eigenvector of the normalized (and weighed) link matrix of the web.
Page Rank
An Example and the Algorithm

We present the two cases synchronous and asynchronous for initial conditions $1/n$ as used in the algorithm.

(a) Synchronous all $1/3$
Base : 0 : $P[0]=0.33333$ $P[1]=0.33333$ $P[2]=0.33333$
Iter : 1 : $P[0]=0.05000$ $P[1]=0.19167$ $P[2]=0.47500$
Iter : 2 : $P[0]=0.05000$ $P[1]=0.07125$ $P[2]=0.23417$
Iter : 3 : $P[0]=0.05000$ $P[1]=0.07125$ $P[2]=0.13181$
Iter : 4 : $P[0]=0.05000$ $P[1]=0.07125$ $P[2]=0.13181$

(b) Asynchronous all $1/3$
Base : 0 : $P[0]=0.33333$ $P[1]=0.33333$ $P[2]=0.33333$
Iterat : 1 : $P[0]=0.05000$ $P[1]=0.07125$ $P[2]=0.13181$
Iterat : 2 : $P[0]=0.05000$ $P[1]=0.07125$ $P[2]=0.13181$

The PageRank algorithm (final version). We present below algorithm PageRank that uses Eq. (4) to compute the rank of web-pages. The initialization vector is $e^t/n$ i.e. a vector whose all entries are $1/n$. (Avoid all zeroes, and it is better than an all-one initialization.) For the algorithm to be efficient a proper representation of the Web-graph must be available. A compact form of it is through an adjacency list representation of the link structure i.e. for every vertex we maintain its out-degree and a list of end-points for outgoing edges as shown below

<table>
<thead>
<tr>
<th>Source-vertex</th>
<th>Out-degree</th>
<th>Endpoints-of-outgoing edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>u m k1 k2 k3 ... km</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{PageRank}(G,V,E) \quad //\text{The code performs a SYNCHRONOUS Rank update}\n\]
1. for all vertices $u$ in $V$ /* Initialization Step */
2. \quad $\text{Src}[u] = 1/n$;
3. \quad small = something-small;
4. \quad while (convergence-distance > small) {
5. \quad \quad for all $v$ in $V$
6. \quad \quad \quad $D[v]=0$;
7. \quad \quad \quad for (i=0; i<|V|; i++) {
8. \quad \quad \quad \quad Read-Adjacency-List(u,m,k1,k2,...,km);
9. \quad \quad \quad \quad for (j=1; j<=m; j++)
10. \quad \quad \quad \quad \quad D[kj] = D[kj] + $\text{Src}[u]/m$
11. \quad \quad \quad }
12. \quad \quad for all $v$ in $V$
13. \quad \quad \quad $D[v] = d \cdot D[v] + (1-d)/n$
14. \quad \quad \quad convergence-distance = ||\text{Src}-D|| \quad /*\text{Euclidean distance}*/
15. \quad \quad \quad $\text{Src}=D$;
16. \quad }

(c) Copyright A. Gerbessiotis. CS610: Spring 2016. All rights reserved.