Introduction to Graph Algorithms

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 2019; distribution outside this group of students is NOT allowed.*

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Chapter 21 (21.3 and 21.4 are out)
Pages 3-8 are from the Appendix (Graphs and Trees).
Chapter 22 (proofs are out, 22.4 no-proofs just the three line algorithm)
Chapter 23 (emphasis on Section 23.2; these notes are sufficient to follow the discussion)
Chapter 24 introduction and Section 24.2; in addition introduction of Chapter 16.
Chapter 25 introduction and 25.2 and Introduction of Chapter 15.
If something is not clear in the textbook, consult these lecture notes.
Some material in these lecture notes are not available in the textbook (eg related to rooted tree traversals, binary tree traversals).

A very well written treatise on data structures and graph algorithms that I used for the discussion on tree traversals is the following reference.

R.E.Tarjan, "Data Structures and Network Algorithms", Society for Industrial and Applied Mathematics, Dec 1983, ISBN 0898711878.

(c) Copyright A. Gerbessiotis. All rights reserved.

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 in a tree described earlier. Below we give a very brief outline of how the method works.

A more general framework for DFS is shown on the next page. It searches a whole graph not only the portion reachable from a given vertex. It 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 graphs. Although the labelings are different for an undirected and a directed graph, in either case, a b labeled edge indicates the existence of a cycle in the graph.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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    ----------------------------------------> 9a else if (color[v]==GRAY)
10. color[u]=BLACK;           // Post visit u: Finalize visit   9b          (u,v) is a back edge
11. time++;                   // Post visit u: Finalize visit   9c      else if (color[v]==BLACK)
12. F[u]=time;                // Post visit u: Finalize visit   9d            if D[u]<D[v] (u,v) is a forward edge
                                                               9e            else         (u,v) is a cross edge
    DFS(G,V,E)                                                 9g } //This is the line 9 closing brace of the line 4
1.  for each vertex u {   //Initialize                        9h   //foreach loop
2.    color[u]=WHITE;
3.    p[u] = NULL;
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. }
```

(c) Copyright A. Gerbessiotis.

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 (arry 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)).

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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)** 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].

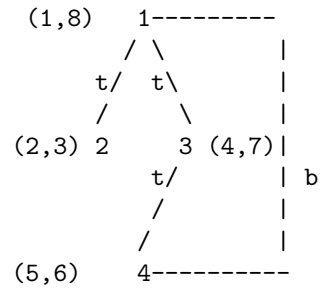
(d) **CROSS** edges that connect vertices that neither is descendant of the other, such as those visited in line 5 of DFS-Visit for color(v)=BLACK and D[u] $>$ D[v].

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.

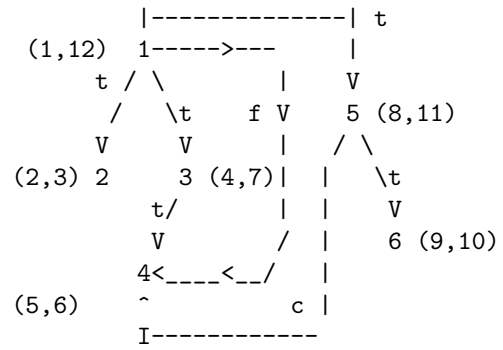
(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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.



(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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)
```

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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.

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

(c) Copyright A. Gerbessiotis. All rights reserved.

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

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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 o 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 ∞, 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 presenece of negative edge weights **as long as there are NO NEGATIVE WEIGHT CYCLES**.

```
 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)
2.  for k=1 to n
3.    for i=1 to n
4.      for j=1 to n
           k           k-1         k-1      k-1
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 $\{1, \ldots, l\}$ i.e. vertices $k \leq l$.
Basis of induction $l = 0$**.** 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 i.e. $k \leq 0$, noting that vertex indexes are positive.
Inductive Step. Assume that the inductive hypothesis is true for $k = l - 1$ and consider the triangular inequality for $k = l$, ie. $D^l_{i,j} = \min\{D^{l-1}_{i,j}, D^{l-1}_{i,l} + D^{l-1}_{l,j}\}$. If the shortest path from i to j with intermediate vertices $k \leq l$ does not go through l, $D_{i,j}$ remains unchanged from $k = l - 1$ to $k = l$. Inductive step is concluded then; if it does pass through l, then by the inductive hypothesis $D^{l-1}_{i,l}$ is the shortest path from i to l with intermediate vertices $k < l$, and likewise $D^{l-1}_{l,j}$ is the shortest path from l to j. So the sum $D^{l-1}_{i,l} + D^{l-1}_{l,j}$ is optimal for intermediate vertices $k \leq l$. This concludes the inductive step.

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.

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)=infinity; p(v) = NULL ; // latter for line 9, if needed
3.  S={ }; D(s)=0;
4.  Q=V; BuildMinHeap(H, V,D); // BuildMinHeap H using the D values of vertices in V
5.  while Q is not empty do  // First time D(s)=0, s is extracted
6.        u= ExtractMin(H);  //EXTRACT node u with min D(u)
7.        S=S U {u}; Q=Q-{u};
8.        for each (u,v) an edge and v in Q do
9.          update D(v) = min(D(v),D(u)+d(u,v));  // Also p(v)=u to trace 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 an S, Q split in Dijkstra's algorithm, that vertex of Q 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.

(c) Copyright A. Gerbessiotis. All rights reserved.

Proof of correctness (induction on the length of S). The following invariants will be maintained during the induction as inductive hypothesis. In conclusion $|S| = n$ with $S = V$ and $D(u)$ is the length of the shortest path from s to u in V.

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 length of the shortest path from s to u with intermediate vertices in S (other than s,u).

Base case $|S| = 1$. In the first execution of line 6, s is extracted from Q and moved to S. $D(s)$ is 0 by default and given that there are no non-positive weights we are done. Thus invariant 1 is true for s, the only member of S by default. For invariant 2, we note that $D(.) = d(s, .)$ in line 9 is the length of the shortest *edge* path from s to an arbitrary . if such edge exists.
Inductive Step: from $|S| = k$ to $|S| = k + 1$. 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 $D(u)$. Thus any other $b \in Q$ is such that $D(b) > D(u)$. Then the shortest path from s to u (by rule 2 above) is internal in S except for a single edge that goes from a vertex of S directly to u and $u \in Q$. Let us call this path P. For otherwise, if P was not the shortest path, then there would be another path P' whose intermediate vertices were not internal in S and thus there would be another vertex say b in P' from s to u that is also in Q and that is closer to s than u, a contradiction to the minimality of $D(u)$. That is, if the shortest path would not be internal in S, there would be a path P' of the form $s \rightsquigarrow a$, $a \rightarrow b$ and $b \rightsquigarrow u$, where all vertices of the path from s to a are internal in S, b is the first node in Q, i.e. outside of S, and then the path from b to u might also include nodes internal or external to S. Then $cost(P') < cost(P)$ and therefore $cost(s \rightsquigarrow^{P'} b) + cost(b \rightsquigarrow^{P'} u) < cost(s \rightsquigarrow^{P} u)$. The first term is $D(b)$ and the last term is $D(u)$ thus $D(b) + cost(b \rightsquigarrow^{P'} u) < D(u)$. Given that all edges are positive this means $D(b) < D(u)$ which contradicts $D(b) > D(u)$ i.e. the minimality of $D(u)$. This satisfies invariant 1 for u just inserted into S. Moreover, the line 9 update with respect to $D(u)$ guarantees the maintenance of invariant 2, for vertices still in Q.

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 $O(|E| + |V|^2)$.

(c) Copyright A. Gerbessiotis.

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,d)
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,d)
1. T=empty; X={r}; Q= V: // r is an arbitrary vertex of V
2. while ( X != V) {
3.   let e=(x,q) be lowest cost edge from x in X to q in Q
4.   T= T U {(x,q)};
4.   X=X U {x}; Q= Q - {q};
5. }
```

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.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

Prim's Algorithm

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

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. 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. $d(u, v) \leq d(u', v')$.
7. Consider $N = T' - (u', v') + \{(u, v)\}$.
8. N has (u, v) and $cost(N) = cost(T') - d(u', v') + d(u, v) \leq cost(T')$
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 $O(\lg |V|)$ 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.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

```
Kruskal(G,d)
1. T=empty;
2. for each u in V
3.  do  Make-Set(u);
4. sort edges of E by nondecreasing weight w;
5. for each edge (u,v) in E in nondecreasing weight order
6.   do if Find-Set(u) != Find-Set(v)
7.     then T=T U {(u,v)} ;
8.          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. If a vertex a is not saturated by an edge in T, then an edge must exist between a and T since G is connected and Kruskal never rejects an edge unless it causes a cycle and this is not a case involving a. Likewise if a is in a component not connected to T (rather than an isolated vertex). Let T be not a minimal spanning tree. Let T' be the minimal spanning tree of G with the maximum number of edges (starting with e_1) common with T, and let this number be $k-1$. Then $e_k = (a, b)$ is the first edge in T and not in T'. It can't be the other way i.e. $e_k \notin T$ and $e_k \in T'$ because this would imply e_k caused a cycle in Kruskal (and thus was not picked). Then it would also cause a cycle in T' as up to that point both of the algorithms have common edges! Let P be the path in T' that connects a, b since $e_k = (a, b)$ is not in T'. (The addition of e_k would cause P plus e_k to become a cycle.) There must be an edge $e' > e_k$ in P. If there was no such edge, Kruskal would have picked all those edges for T (they are lighter than e_k) and by including then a heavier edge e_k would have generated a cycle: a contradiction. Thus an $e' > e_k$ exists. Consider $T' - e' + e_k$, it is a spanning tree better that T', a contradiction.

(Alternatively, let G_k be the connected component containing $\{e_1, \ldots, e_{k-1}\}$ that touches a. Then there exists an edge e in P connecting G_k to a vertex outside G_k. Since Kruskal chooses e_k over e in T, $e_k \leq e$. But $T' - e + e_k$ 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 Suppose that T is not a minimal spanning tree.

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

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 β is a user defined parameter (e.g. $\beta = 0.2$). Then $R(q, d_j) = s(q, d_j) + \beta \sum_i R(q, d_i) l(i, j)$.

(c) Copyright A. Gerbessiotis. All rights reserved.

HITS: Authority and Hub page rank (Kleinberg 1998)

2.1 Another model: Kleinberg's HITS model. Kleinberg (1998) introduced a model that assigns two ranks to each web-page/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 n 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,
$\mathbf{BS_1(q)} = \{$**all pages pointed by an** $RS(q)$ **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 $\mathbf{BS_2(q)} = \{$**B pages pointing to an** $RS(q)$ **page**$\}$, for some fixed B.

2.3.4 Step 4: Thus $\mathbf{BS(q)} = \mathbf{RS(q)} \cup \mathbf{BS_1(q)} \cup \mathbf{BS_2(q)}$.

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**.

(c) Copyright A. Gerbessiotis. All rights reserved.

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.

$$h^t = (h(d_1), \ldots, h(d_n)) \qquad a^t = (a(d_1), \ldots, a(d_n))$$

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.

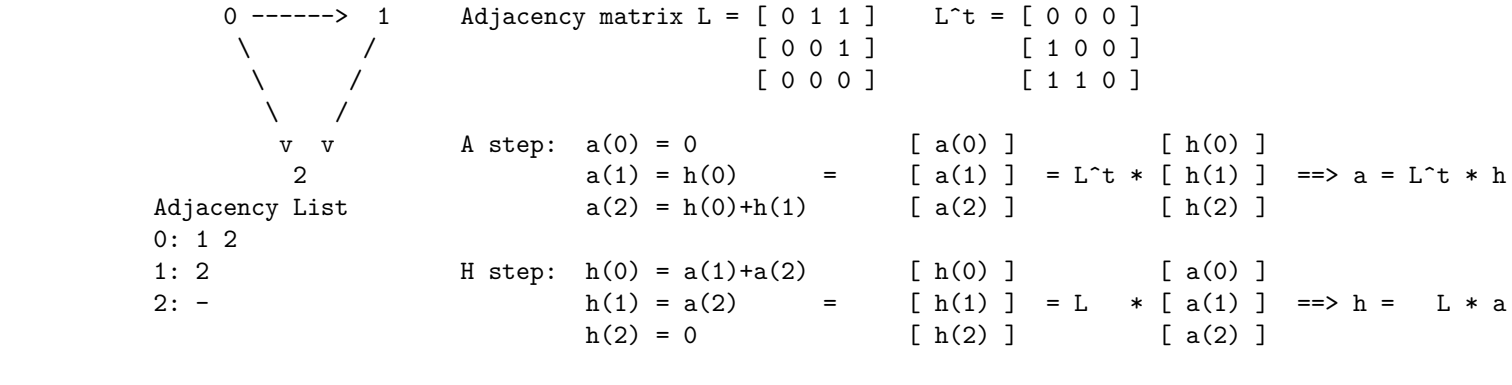
2.5.1 A: Authority update step. The first update involves the authority scores. The previously available hub scores are being used for this update. If an edge points from i to j i.e. $(i, j) \in E$ then the authority of d_j is to be updated using the score of $h(d_i)$. Therefore, for every link $(i,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

$$a(d_j) = \sum_{(i,j)\in E} h(d_i). \tag{1}$$

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

$$h(d_j) = \sum_{(j,k)\in E} a(d_k). \tag{2}$$

2.6 Example 0. For the three vertex graph shown below



2.7 Example 0 continued. For the previous example, let the initial values for the a, h vectors be all ones. After every iteration is completed (an A and H step) a scaling takes place.

$$a(d_j) = \frac{a(d_j)}{\sqrt{\sum_k (a(d_k))^2}}, \quad h(d_j) = \frac{h(d_j)}{\sqrt{\sum_k (h(d_k))^2}},$$

The process goes on until convergence to the desired level is achieved. (If we want 5 decimal digit convergence, this would mean that we stop when the 5 decimal digits do not change from iteration p to $p+1$ for some p for all vector values.)

```
                    A step:  a(0) = 0              [ a(0) ]          [ h(0) ]
                             a(1) = h(0)      =    [ a(1) ]  = L^t * [ h(1) ]  ==> a = L^t * h
Adjacency List               a(2) = h(0)+h(1)     [ a(2) ]          [ h(2) ]
0: 1 2               H step: h(0) = a(1)+a(2)     [ h(0) ]          [ a(0) ]
1: 2                         h(1) = a(2)      =    [ h(1) ]  = L   * [ a(1) ]  ==> h =   L * a
2: -                         h(2) = 0             [ h(2) ]          [ a(2) ]
               h :  1    1    1     a :  1    1   1
Step 1     A:                             0    1   2
           H:        3    2    0
     scale  :      0.83 0.55 0.0         0.0 0.44 0.89
```

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.

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!

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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.

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

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

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

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 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.

(c) Copyright A. Gerbessiotis. CS435-102: Spring 2019. All rights reserved.

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

```
  Source-vertex Out-degree  Endpoints-of-outgoing edges
          u        m        k1 k2 k3 ...  km

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