

GRAPHS AND TREES; WEB-PAGE RANKING

CS 505 and CS 506 material

Chapters 1.4, 2.3-2.4, 7.1-7.2 of GT

DISCLAIMER: *Report an error / typo to the author so that he resolves it.*
These notes are NOT to be distributed outside of NJIT without the expressly written permission of A. Gerbessiotis

We present some definitions re CS241 material the way they are being used in the designated textbook; they might have been used slightly differently in CS241. Get used to this notation. This is the purpose of the first few slides of Subject 3: to remind you and familiarize you with the usage of graph theory definitions.

3.1 Directed Graphs.

Definition 1. A **directed graph**, also known as digraph, G is a pair $G = (V, E)$, where V is a finite set and E is a binary relation on V (i.e. E is a subset of $V \times V$, $E \subseteq V \times V$). We shall customarily use n, m to denote the cardinalities of V and E , i.e. $n = |V|$ and $m = |E|$. V is called the vertex set (or set of vertices) and E the edge set (or set of edges) of G . The elements of both V and E are called respectively the vertices and edges of G .

For directed graphs, we sometimes use the notation **nodes** and **arcs** to denote the vertices and edges respectively. Under such a notation we may use $G = (N, A)$ instead of $G = (V, E)$.

3.1.1 Edges. In a directed graph G , members of V will be denoted with lower-case letter such as u, v . Then $u, v \in V$. An edge, a member of E , will be denoted by $(u, v) \in E$, where $u, v \in V$, and this represents an edge leaving u and entering v . In G edge $e = (u, v)$ is **incident from** u or **leaves** vertex u and is also, **incident to** v or **enters** v . The endpoints of edge $e = (u, v)$ are u and v . We sometimes also say that u, v are adjacent, but then "adjacency" does not specify the direction of it.

3.1.2 Special cases of edges. In a digraph **self-loops** such as (u, u) are possible. Multiple edges between two vertices are not allowed.

3.1.3 Directed Graphs: In-degree, out-degree, and degree.

Definition 2. For a directed graph G the **out-degree** of a vertex u is the number of edges incident from u . The **in-degree** of a vertex u is the number of edges incident to u . The **degree** of vertex u in a directed graph G is the sum of its out-degree and in-degree. We use $o(u)$, $outd(u)$ and $i(u)$, $ind(u)$ to represent the out-degree and in-degree of u . $d(u)$ or $deg(u)$ is the degree of u .

Exercise 1. Show that $\sum_{u \in V} d(u) = 2m = 2|E|$. How much is $\sum_{u \in V} outd(u)$?

DRAFT

3.2 Undirected Graphs.

Definition 3. An **undirected graph** G is a pair $G = (V, E)$ where V is a finite set and E is a set of unordered pairs of vertices.

3.2.1 Edges. In an undirected graph an edge such as $(u, v) \in E$ really means $\{u, v\}$. By default we are going to assume that our undirected graphs are all simple. In G edge $e = (u, v)$ is **incident on** u and v . Two vertices connected by an edge are called **adjacent**, ie. u is adjacent to v or v is adjacent to u . For an undirected graph (u, v) and (v, u) are the same edge. The endpoints of (u, v) are u and v . We also say $e = (u, v)$ is incident on u and v .

3.2.2 Special cases of edges. An undirected GRAPH is always **simple** by default; this means it has no self-loops (u, u) and no multiple edges.

3.2.3 Undirected Graphs: degree. For an undirected graph G the **degree** of vertex u is the number of edges incident on u . $d(u)$ or $\deg(u)$ is the degree of u . Show that $\sum_{u \in V} d(u) = 2m = 2|E|$.

3.3.1 'from', 'to', 'on' and 'between'. For an undirected graphs and edge $e = (u, v)$ we say e is incident on u and v . For a directed graph we say e is incident from u and incident to v thus the prepositions in the latter case indicate the direction of the edge. In the former case there is no direction so 'on' is neutral. By convention we use the same notation to represent an edge for the directed and undirected case, that is (u, v) is an edge from u to v in the directed case, whereas in the undirected case it represents an edge between u and v .

3.3.2 Set incidence. If S is a set of vertices an edge is **incident** to S if **exactly one** of its endpoints is in S .

3.3.3 Bipartite graphs. A graph $G = (V, E)$ is **bipartite** if its vertices V can be partitioned into two sets so that edges go from one set to the other only, i.e. the end points cannot both be in the same set.

3.3.4 Subgraphs. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. G_1 is a subgraph of G_2 if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

3.3.5 Complete (undirected) graph. A **complete graph** on n vertices is an undirected graph such that every pair of vertices is adjacent. It is denoted by K_n and contains $n(n-1)/2$ edges.

3.3.6 Isomorphic Graphs (graph isomorphism). Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection (surjective and injective function) f from V to V' such that $(u, v) \in E$ is mapped to $(f(u), f(v)) \in E'$.

3.4.1 Paths.

Definition 4.

$$u \rightsquigarrow v \text{ or } u \rightsquigarrow^k v \quad : \quad \langle u_0, u_1, \dots, u_k \rangle = \langle u, u_1, \dots, v \rangle \text{ with } (u_i, u_{i+1}) \in E \text{ for } 0 \leq i < k.$$

We say that there is a path from u to v and indicate it as $u \rightsquigarrow v$ or we say that there is a path of length k from u to v and indicate it as $u \rightsquigarrow^k v$, if there is a sequence $\langle u_0, u_1, \dots, u_k \rangle$ of $k + 1$ distinct vertices such that $u_0 = u$ and $u_k = v$, and consecutive vertices map to distinct edges i.e. $(u_i, u_{i+1}) \in E$, $0 \leq i < k$. The vertices $u_0 = u$ and $u_k = v$ are the endpoints of the path. The length of the path is the number of (distinct) edges in it. We also say that u_i is **reachable** from u .

3.4.2 Simple Paths. A path is by default **simple**, as defined, if there are no duplicate vertices (i.e. all vertices in the path are distinct). A subpath of a path is a contiguous subsequence of its (path's) vertices. In a path we have unique (distinct) vertices and edges.

3.4.3 Tour. A **tour** has a definition similar to that of a path. In a tour all edges are also distinct; but vertices are not necessarily distinct (unique).

3.4.4 Cycles. A **cycle** in a directed graph is a path with at least one edge, and with $k \geq 0$ whose endpoints also define an edge (u_k, u_0) . Several times a cycle is described as a path with vertex u_0 repeated and ending that path instead of u_k . In that case we say the end-points of the path coincide.

$$\langle u_0, u_1, \dots, u_k \rangle \text{ implies also } (u_k, u_0) \in E \quad \text{also written as } \langle u_0, u_1, \dots, u_k, u_0 \rangle$$

A cycle is **simple** if all $u_1, \dots, u_k$ are distinct. A self-loop is a cycle of length 1. A cycle in an undirected graph is defined identically (self-loops are not allowed in undirected graphs as these graphs are simple).

3.4.5 Acyclic graphs. A graph with no cycles is called **acyclic**.

Graphs Example

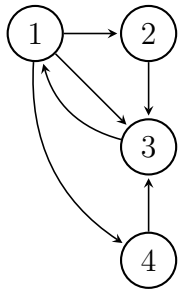


Figure 1: A directed graph

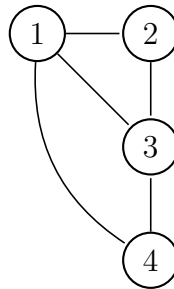


Figure 2: An undirected graph

$|V|=n=4$ and $|E|=m=6$
 $V=\{1,2,3,4\}$
 $E=\{(1,2), (2,3), (4,3), (1,4), (1,3), (3,1)\}$
 $i(1)=1$ $o(1)=3$ $d(1)=4$
 $i(2)=1$ $o(2)=1$ $d(2)=2$
 $i(3)=3$ $o(3)=1$ $d(3)=4$
 $i(4)=1$ $o(4)=1$ $d(4)=2$

$|V|=n=4$ and $|E|=m=5$
 $V=\{1,2,3,4\}$
 $E=\{(1,2), (2,3), (1,3), (3,4), (1,4)\}$
 $d(1)=3$
 $d(2)=2$
 $d(3)=3$
 $d(4)=2$

Cycle: $\langle 1,3 \rangle$ or $\langle 1,3,1 \rangle$
 Cycle: $\langle 1,4,3 \rangle$ $\langle 1,4,3,1 \rangle$

Cycle $\langle 1,2,3 \rangle$ or $\langle 1,2,3,1 \rangle$
 Hamiltonian*Cycle $\langle 1,4,3,2 \rangle$ or $\langle 1,4,3,2,1 \rangle$

* Hamiltonian cycle: contains all vertices of V .

3.5.1 Connected undirected Graph.

Definition 5. An undirected graph is **connected** if every vertex v is reachable from every other vertex u . That is $\forall u, \forall v \in V$ we have that there exists a path $\exists u \rightsquigarrow v$. By symmetry there is also a path $v \rightsquigarrow u$).

$$G \text{ is connected: } \forall u \in V, \forall v \in V, \exists u \rightsquigarrow v$$

3.5.2 Connected components.

Definition 6. The **connected components** of a graph are the equivalence classes under the “**is reachable from**” relation or in other words they are the maximal connected subgraphs of G . An undirected graph is connected if number of connected components is 1.

3.5.3 Strongly connected directed graph.

Definition 7. A directed graph is **strongly connected** if every vertex v is reachable from every other vertex u i.e. $\forall u, \forall v \in V$ we have $\exists u \rightsquigarrow v$ and $\exists v \rightsquigarrow u$. We say then that u, v are mutually reachable.

3.5.4 Strongly connected components.

Definition 8. The **strongly connected components** of a graph are the equivalence classes under the “**are mutually reachable**” relation.

Definition 9. A **forest** is an acyclic undirected graph.

3.6.2 Tree in an (undirected) graph.

Definition 10. A (**free**) **tree** or just a **tree** is a connected acyclic undirected graph.

3.6.3 Tree vs Forest.

Definition 11. A connected forest is thus a tree.

3.6.4 Some tree facts.

Theorem 1 (Tree). *Let G be an undirected graph. The following statements are equivalent*

1. G is a free tree.
2. Any two vertices in G are connected by a unique single path.
3. G is connected but if any edge is removed from E the resulting graph is disconnected.
4. G is connected and $|E| = |V| - 1$.
5. G is acyclic and $|E| = |V| - 1$.
6. G is acyclic but if any edge is added to E the resulting graph contains a cycle.

3.7.1 Rooted Trees.

Definition 12. A **rooted tree** is a free tree in which one of the vertices is distinguished from the others and is called the **root**. The root of a tree T will be represented sometimes by r .

3.7.2 Vertex of a Graph but Node of a Tree.

Definition 13. Vertices of a tree are also called **nodes**. The same u can be a vertex (of the graph) in one context and node (of a tree that is the graph) in another context.

3.7.3 Degree of Vertex vs Degree of Node (in a rooted tree).

Usually we do so when we want to distinguish the degree of a vertex of G from the degree of a node of a tree.

The number of children of the root (of a rooted tree) define the 'degree' of the root. Likewise we can define the degree of a node as the number of its own "children" For a rooted tree we can define relationships such as descendant, ancestor, child and parent.

3.7.4 Ancestor, Descendant. Let P be a path from r to some vertex (node) x of T . Then any node y of the unique path from r to x is called an **ancestor** of x . x is a **descendant** of y (including r and itself). If $x \neq y$ then the x is a proper descendant of y and y is a proper ancestor of x . x is an ancestor/descendant of x . The subtree rooted at x is the tree induced by descendants of x rooted at x .

Trees

Rooted Trees

3.7.5 Parent, Child, Siblings. If in the path the last edge from r to x is (z, x) z is called **parent** of x , x is called **child** of z . If x and u are both children of z they are called **siblings**.

3.7.6 Leaf or external node. Internal nodes. A node with no children is called a **leaf** or an **external node**. A non-leaf is called an **internal node**. (The root is also an internal node.)

Definition 14. The degree of a node in a (rooted) tree is the number of its children.

Note 1. The degree of a vertex in a Graph has already been defined.

DRAFT

Rooted Trees

Depth and Height

3.8.1 Depth.

Definition 15. The length of the path from r to x is the depth of x . A root r has depth $d(r) = 0$. If v is a tree node the $d(v) = d(p(v)) + 1$ where $p(v)$ is parent of v .

3.8.2 Height of a tree.

Definition 16. The largest depth of any node is the height of the tree T .

3.8.3 Height of a node including the root.

Definition 17. The height of a leaf v is $h(v) = 0$. For a non-leaf node u , $h(u) = 1 + \{\max h(w), \text{ where } w \text{ is a child of } u\}$. In other words the height of a node is the length (in terms of number of edges) of the longest path to a descendant leaf.

Definition 18. The height of a rooted tree is the height of its root.

Rooted Trees

Depth and Height

3.8.4 Ordered tree.

Definition 19. An **ordered** tree is a (rooted) tree in which the children of every node are ordered (i.e. we can talk of the first, second etc).

3.8.5 Binary tree.

Definition 20. A **Binary Tree** is a rooted and ordered tree. Definition follows.

DRAFT

Rooted and Ordered Trees

Binary Tree

3.9.1 Binary tree.

Definition 21. A **Binary Tree** is a rooted and ordered tree that either

- contains no nodes (empty), or
- it consists of three disjoint sets of nodes: a **root** node, a binary tree called **left-subtree** and a binary tree called **right-subtree**.

The root of the left subtree is called the left child of **root** and that of the right subtree the right child.

3.9.2 Difference between binary trees and ordered trees. If a node has only one child it matters whether it is a left or a right child in a binary tree, as opposed to ordered trees, where it doesn't matter. Binary trees maintain a left and right notion, ordered trees do not.

3.9.3 Full-binary tree.

Definition 22. A full binary tree is a tree where each node has degree 2 or 0 (leaf).

3.9.4 Complete binary tree.

Definition 23. A complete binary tree, is a tree where each leaf has the same depth and all internal nodes have degree 2.

Several textbooks call a full binary tree what we defined as complete binary tree.

Some of those properties were derived in Subject 1.

3.9.5 Complete Binary Tree (CBT) of height h .

Fact 1. Number of vertices in such a tree of height h is

$$1 + 2 + 2^2 + \dots + 2^h = 2^{h+1} - 1.$$

3.9.6 Number of internal nodes in CBT of height h .

Fact 2. The number of internal nodes is $2^h - 1$.

3.9.7 Number of external nodes in CBT of height h .

Fact 3. The number of external nodes (leaves) is 2^h .

Exercise 2. A Complete binary tree has n nodes. How many of them are leaves, and how many of them are internal nodes ?

3.10.1 Dense graphs vs Sparse graphs.

Definition 24. For a graph $G = (V, E)$ let $n = |V|$ and $m = |E|$, as already noted. 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 as $m = o(n^2)$.

3.10.2 Adjacency matrix and Adjacency list. There are two major ways one can use to represent a graph in the memory of a computer.

- (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 (a matrix). 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 case of a directed graph G and $2m$ entries in the case of an undirected graph G are equal to 1. The remaining entries of A are 0 and map to non-edges. Total space requirements are $\Theta(|V|^2) = \Theta(n^2)$.
- (2) **Edges have weights.** If the edges of the graph have weights (or costs or distances) assigned to them, then instead of storing an 1 we store in A the weight (cost or distance) of that edge. Instead of zeroes for the non-edges we turn them into ∞ . The diagonal elements remain 0. Sometimes we call this modified weighed matrix D or W or C . Total space requirements are still $\Theta(|V|^2) = \Theta(n^2)$.
- (3) **Adjacency list** representation. The graph is represented by n linked lists in the form of an array A of linked lists, one for each vertex of G . The linked list for vertex i is denoted by $A(i)$ and contains all vertices that are neighbors of i , i.e. it contains all the end-points j such that (i, j) is an edge that is incident from i to j . For an undirected graph, j is in the linked list of i that is $A(i)$, and i is in the list of j that is $A(j)$. Thus for a directed graph the length of $A(i)$ is $outd(i)$. Thus for a undirected graph the length of $A(i)$ is $d(i)$. The length of all the $A(i)$ is $\sum_i outd(i)$ for a directed graph, and $\sum_i d(i)$ for an undirected graph. In either case the sum is $\Theta(m)$. Total space requirements are $\Theta(|V| + |E|) = \Theta(n + m)$. The n term is due to the length of the array A of linked lists. The m contribution as derived above. The problem with this representation is that we cannot answer quickly questions of the form "Is j adjacent to i ?" for a directed graph.

Graph representations

Continued

- (1) **Edges have weights.** Along with the endpoint j of the linked list $A(i)$ we store a weight (cost, distance) that describes the weight/cost/distance of edge (i, j) .

The **incidence matrix** of a directed graph $G = (V, E)$ is a $|V| \times |E|$ matrix $B = [b_{ij}]$ such that $b_{ij} = -1$ if edge j leaves vertex i , $b_{ij} = +1$ if edge j enters vertex i , $b_{ij} = 0$ otherwise.

DRAFT

Adjacency Matrix and Adjacency List representation

An example

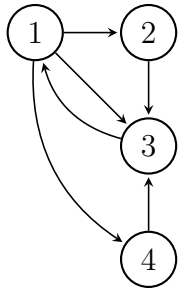


Figure 3: A directed graph

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 4: Its adjacency matrix

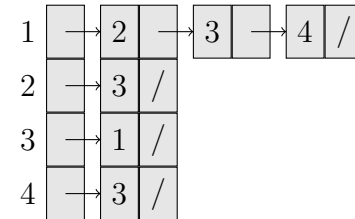


Figure 5: Its adjacency list A

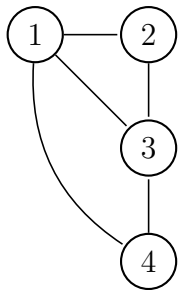


Figure 6: An undirected graph

$$A = \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 7: Its adjacency matrix

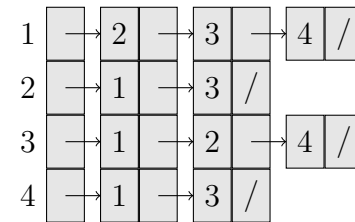


Figure 8: Its adjacency list A

Graphs with weights (distances, edges)

An example

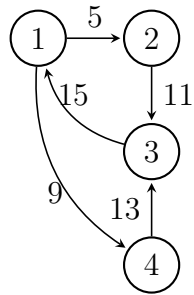


Figure 9: A weighed directed graph

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Figure 10: Its adjacency matrix

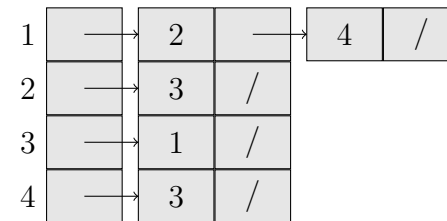


Figure 11: Its adjacency list A

$$W = \begin{pmatrix} 0 & 5 & \infty & 9 \\ \infty & 0 & 11 & \infty \\ 15 & \infty & 0 & \infty \\ \infty & \infty & 13 & 0 \end{pmatrix}$$

Figure 12: Its weight matrix

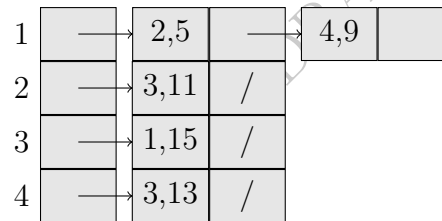


Figure 13: Its weight list W

3.11 Tree traversals. A **tree traversal** is a process of visiting each of the vertices of a rooted tree exactly once. For rooted trees preorder and postorder tree traversals are defined. For the specific case of a **binary** tree three such traversals are quite known: **inorder**, **postorder**, **preorder**. In preorder, parents are visited before their children; in postorder after their children; for binary trees, left children are visited before right children. Inorder traversal has a meaning only for binary trees: left subtree of a parent is visited before the parent followed by right subtree. As a rooted tree may or may not be ordered, inorder is not well-defined/meaningful.

```
pre-visit(u)   :: {pre[u]=pre++; print(<u,pre[u]>);} ;
in-visit(u)    :: {in[u]=in++; print(<u,in[u]>);} ;
post-visit(u)  :: {post[u]=post++; print(<u,post[u]>);} ;
```

```
BT-Inorder(u)
1. if u != NULL {
2.   BT-Inorder (left(u));
3.   in-visit(u) ;
4.   BT-Inorder (right(u));
   }
```

```
BT-Preorder(u)
if u != NULL {
  pre-visit(u);
  BT-Preorder(left(u));
  BT-Preorder(right(u));
}
```

```
BT-Postorder(u)
if u!= NULL {
  BT-Postorder(left(u));
  BT-Postorder(right(u));
  post-visit(u);
}
```

```
BT-Perform-All-Three(u) {
1. pre-visit(u);
2. if left(u) != NULL
3.   BT-Perform-All-Three (left(u));
4. in-visit(u);
5. if right(u) != NULL
6.   BT-Perform-All-Three (right(u));
5. post-visit (u);
}
```

```
Euler-tour(u)
1. left-visit(u); // similar to pre-visit
2. if left(u)!=NULL
3.   Euler-tour(left(u));
4. down-visit(u); // similar to in-visit
5. if right(u) != NULL
6.   Euler-tour(right(u));
7. right-visit(u); // similar to post-visit
```

3.12 BFO (Bread-first Order) traversal. Another useful traversal is the **breadth-first order** (BFO) traversal obtained as follows.

```
RT-BFO(root(RT)) // RT is a rooted (not necessarily binary) tree
1. visit(root(RT)); // visit[root(RT)]=i++; print(visit[root(RT)]);
2. repeat until all vertices are visited
3. visit an unvisited child of the LEAST RECENTLY VISITED
   vertex with an unvisited child
END_BFO
```

If **LEAST RECENTLY** is replaced by **MOST RECENTLY**, a **depth-first order** (DFO) traversal is obtained.

We can extend these two traversals from rooted trees to graphs. If G is a graph and s is an arbitrary starting vertex, we use this vertex as the “root” of the traversal by visiting s first and then repeating the following step until there is no unexamined edge (u, v) such that vertex u has been visited.

Search Step. Select an unexamined edge (u, v) such that u has been visited and examine this edge, visiting vertex v if v has not been visited yet.

3.13 DFO (Depth-First Order) traversal. If u is the **MOST RECENTLY VISITED** vertex we get a **DEPTH-FIRST** order. In **BREADTH-FIRST ORDER** traversal v is the **least recently visited** vertex.

```

  1   |x6| DFO: 1 2 3 4 5 6    BFO: 1 2 6 3 5 4
 / \  |x5|
  2 6 |x4|
 / \  |x3| Tie breaker: Lowest index / vertex/ node /label
  3 5 |x2|
 /    |x1|
4     ----
```

Undirected Graphs

Graph Connectivity

In the first part we will examine the graph connectivity problem for the class of undirected graphs. For this we will use an auxiliary representation of an undirected graph using the Union-Find data structure. Thus starting with the input representation of the graph that utilizes either an adjacency matrix or an adjacency list representation we are going to build a Union-Find data structure. The properties of the Union-Find data structure introduced in the next section are primitive: there are more optimized versions that we do not discuss.

In general a Union-Find data structure is used to represent a disjoining collection of sets of elements. The elements for example can be the vertices of an undirected graph and the sets the connected components (of the undirected graph).

DRAFT

Definition 25 (Disjoint sets of elements: Data). A disjoint-set data structure maintains a collection of $S = \{S_1, S_2, \dots, S_k\}$ of disjoint dynamic sets that is consisting of n elements of a set E . Thus

$$E = \{e_1, e_2, \dots, e_n\}$$

$$S = \{S_1, S_2, \dots, S_k\} \quad \text{for some } 1 \leq k \leq n \quad \text{where}$$

$$S_i \cap S_j = \emptyset \quad \text{and} \quad \bigcup_{i=1}^k S_i = E.$$

Thus a collection of disjoint sets is characterized by the tuple $(n, k, E, S_1, \dots, S_k)$. Each set S_i is identified by a **representative** that is a leader of set S_i which is some arbitrary member of that set. We denote the representative of set S_i by **rep**(S_i).

The collection of disjoint sets of elements is a dynamic collection, thus its structure changes with time. “Dynamic” characterizes the sets $S_1, \dots, S_k$ and thus k . Initially the starting state is $(n, 0, E, \emptyset)$. Operations defined on the collection can change its state

Definition 26 (Disjoint sets of elements: Operations). The following operations are defined on a collection of disjoint sets of elements with initial state $(n, 0, E, \emptyset)$, where n is the number of elements of a set E of elements.

- Operation **MakeSet**(e_r) that creates a single element set provided that the elements is not currently in any of the sets of the collection.
- Operation **Find**(e_r) that finds the set of the collection to which e_r belongs (or not).
- Operation **Union**(e_r, e_s) that forms the (set theoretic) union of the two distinct sets of the collection that contain e_r and e_s .

More information about the semantics of the operations follows. The data structure that implements the data organization that support these three operations is referred to frequently as the Union-Find data structure.

1. Operation: **MakeSet**(e_r)

Input. Element $e_r \in E$ (and collection state $(n, k, E, S_1, \dots, S_k)$).

Output. A set is formed containing e_r with representative e_r itself provided that $e_r \notin S_i$ for all $1 \leq i \leq k$. Thus $\mathbf{rep}(\{e_r\}) = e_r$.

Side effects. The state of the collection changes. As a by product of this operation the number k of sets of the collection increases by one. There is one more set in the collection. Thus MakeSet is a dynamic modifying operation.

2. Operation: **Find**(e_r)

Input. Element $e_r \in E$ (and collection state $(n, k, E, S_1, \dots, S_k)$).

Output. The set S_j that contains e_r is found and the representative $\mathbf{rep}(S_j)$ of S_j is returned.

Side effects. The state of the collection does not change. Thus Find is a static probing operation.

3. Operation: **Union**(e_r, e_s)

Input. Element $e_r, e_s \in E$ such that $e_r \in S_{i_1}$ and $e_s \in S_{i_2}$ for some $1 \leq i_1, i_2 \leq k$ and $i_1 \neq i_2$, (and collection state $(n, k, E, S_1, \dots, S_k)$).

Output. The union $S_{i_1} \cup S_{i_2}$ is formed of the sets containing r and s respectively. The $\mathbf{rep}(S_{i_1} \cup S_{i_2})$ is one of $\mathbf{rep}(S_{i_1})$ $\mathbf{rep}(S_{i_2})$.

Side effects. The state of the collection changes. As a by product of this operation the number k of sets of the collection decreases by one. Sets S_{i_1}, S_{i_2} are replaced by their union $S_{i_1} \cup S_{i_2}$. There is one fewer set in the collection. Thus Union is a dynamic modifying operation.

Definition 27 (Union-Find state). For a Union-Find problem we shall assume that its initial state is given by the tuple $(n, 0, E, \emptyset; n, m)$, where $(n, 0, E, \emptyset)$ is the starting state of the disjoint sets of elements associated with the Union-Find problem, n is the number of elements of E , and m is the number of MakeSet, Find, and Union operations that will be associated with the Union-Find.

Theorem 2. *The number of MakeSet operations can be no more than n as there are n elements in E and the sets are disjoint. The number of Union operations is at most $n - 1$. Thus the number of Find operations is at least $m - n - (n - 1) = m - 2n + 1$.*

Proof. The number of MakeSet operations is at most n , one for each element. The number of elements in E is n . The operations MakeSet, Union and Find maintain a collection of disjoint sets of elements. If e_r is part of the collection e_r was created through a MakeSet operation. There can be no more than n MakeSet operations as for each e_r in the collection at most one such operation can be performed since the sets of the collection are disjoint.

The number of Union operations is (at most) $n - 1$. If the number of MakeSet operations is n all n elements are in the collection of the disjoint sets. Every Union operation decreases the number of sets by 1. If we start with N sets after K Union operations we are left with $N - K$ (disjoint) sets. Since $N \leq n - 1$ and $K > 0$, we have that $N - K \leq n - 1$.

The number of Find operations is m minus the upper bound of n and $n - 1$ of the number of MakeSet and Union operations. ■

Union-Find Example

Example 1. In the example depicted below we have performed a number of MakeSet operations, and a number of Union operations to end up with the two sets as depicted. Answer the following questions

- (a) Who is the **rep**(x)?
- (b) Who is the **rep**(z)?
- (c) How many MakeSet operations were issued if the initial state of the collection was $(?, 0, E, \emptyset)$?
- (d) How many Union operations were issued if the initial state of the collection was $(?, 0, E, \emptyset)$?
- (e) After the Union operation performed to get one (final) set what is the cost of the Union in term of representative element updates?

Example : The representative of the set appears as a superscript

Example : $\begin{matrix} 5 & 5 & 5 & 5 & & 8 & 8 & 8 & 8 & 8 \\ \{ & x, & 2, & 3, & 5 & \}, & \{z, & 8, & 7, & 6, & 4 & \} \end{matrix}$

Union(x, z) = $\begin{matrix} & 8 & 8 & 8 & 8 & & 8 & 8 & 8 & 8 & 8 \\ \{ & x, & 2, & 3, & 5, & & z, & 8, & 7, & 6, & 4 & \} \end{matrix}$

Answers.

- (a) Element x belongs to the set who representative is 5.
- (b) Element z belongs to the set who representative is 8.
- (c) The number of MakeSet operations is equal to number of elements found in the two disjoint sets. This is 9.
- (d) The number of Union operations. is $9 - 2 = 7$. The nine participating elements are the results of 9 MakeSet operations. There are two sets left. Therefore the number of Union operations to go from 9 sets down to 2 is 7.
- (e) After the Union operation Union(x, z) the representative member of the union is the element of the larger of the two sets. This is element 8. We did so because such a choice causes the smallest number of representative member updates. Element x belongs to a set of cardinality 4, and since the cardinality of the set containing z is 5, and $4 < 5$ this choice of a representative member causes the smallest number of representative element updates: 4.

Moreover the cardinality of the new set created after the Union is 9 which is more than twice the cardinality of the smaller of the two sets of the Union (who representative member got updated). ■

Union-Find

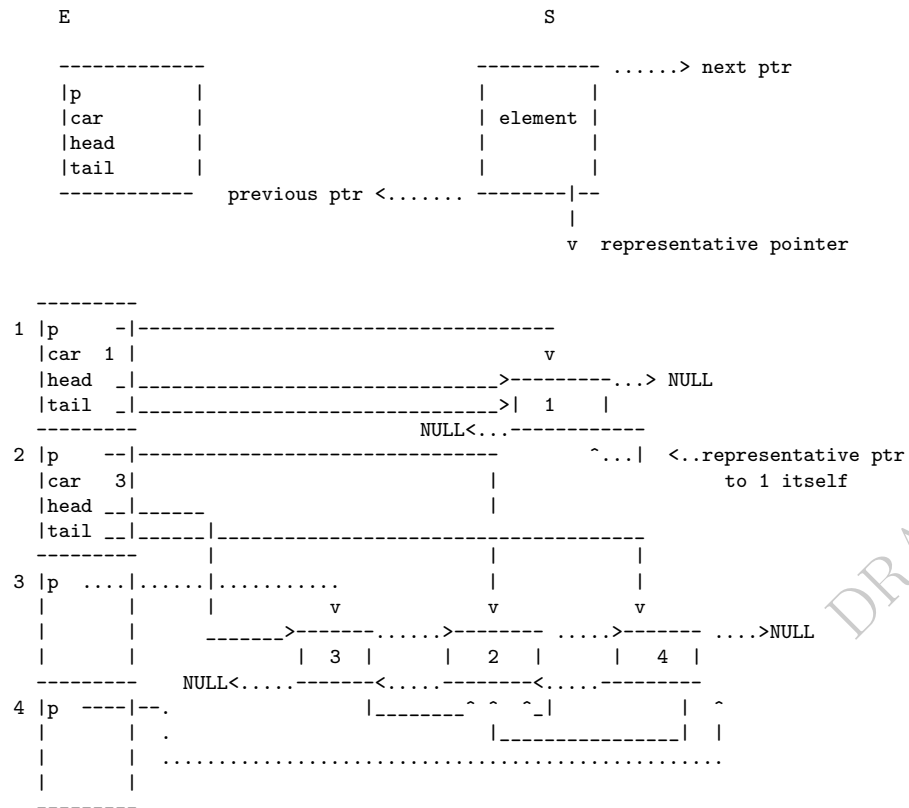
A simple data structure

Definition 28 (A data structure for Union-Find). There is a simple data structure for Union-Find that is based on using an Array of Linked Lists. To keep things simple we assume the elements of E are the subscripts of its elements, thus $E = \{1, 2, \dots, n\}$ rather than $E = \{e_1, e_2, \dots, e_n\}$ or using the latter is also trivial yet a bit cumbersome to describe.

- A set S_i is maintained as a doubly-linked list of elements that are member of S_i . The elements appear in S_i in an arbitrary order. The linked list has a head and tail pointer pointing to the first and last of its elements. Every element of the list has a previous, next pointers associated with the linked list structure, a name field indicate the name of the element (i for e_i), and a representative pointer to the element that is the representative member of the set.
 - a previous pointer called **prev** ,
 - a next pointer called **next** ,
 - a name field called **name** ,
 - a rep pointer called **rep**.
- An array E of linked lists is used to represent the elements and their properties. The length of the array E is n , where n is the number of elements of the set E of elements represented by the name-sake array E . Every element of E is associated with a number of attributes. Thus $E[i]$ maintains the following information for element e_i .
 - A pointer **p** to the element itself, if element e_i has been created and belongs already to a set.
 - A **head** list pointer. If element e_j is the representative member of a set this pointer points to the first element (head) of the linked list representing the set.
 - A **tail** list pointer. If element e_j is the representative member of a set this pointer points to the last element (tail) of the linked list representing the set.
 - A **car** field. If element e_j is the representative member of a set this field shows the cardinality of the set that is the length of the linked list representing the set.

We call such a data structure UF-DS.

Union-Find A simple data structure



Theorem 3. *In a UF-DS data structure over a set of elements E with cardinality n , the running time of operations **MakeSet**, **Find** and **Union** have the following properties.*

1. *In UF-DS the running time of a **MakeSet**(e_r) operation is constant i.e. $\Theta(1)$.*
2. *In UF-DS the running time of a **Find**(e_r) operation is constant i.e. $\Theta(1)$.*
3. *In UF-DS the running time of a **Union**(e_r, e_s) operation is $O(n)$ where n is the cardinality of E , under reasonable assumptions.*
4. *In UF-DS the running time of an arbitrary number of possible **Union** operations that can be defined on the set E of n elements is $O(n \lg n)$, where n is the cardinality of E .*
5. *In UF-DS the running time of $m > n$ operations that include an arbitrary set of **MakeSet**, **Union** and **Find** operations is $O(m + n \lg n)$, under reasonable assumptions.*

Proof.

1. MakeSet(e_r). Operation **MakeSet** requires constant time.

(a) We first create a single element set S represented by a doubly linked list that contains e_r and set its **prev**, **next** pointers to NULL. The **name** field is set to r . The **rep** field points to the linked-list element just created.

(b) Furthermore in array E and in particular E_r we set the **p** pointer to point to element just created in step (a) above, and so do we the **head** and **tail** pointers as e_r is the representative of the set containing e_r . The **car** field is set to 1 as the cardinality of the set which is also the length of the linked list just created is 1.

Obviously all those steps require $\Theta(1)$ time.

2. Find(e_r). Operation **Find** requires constant time.

For element e_r we first access $E[r]$. If pointer p is invalid (e.g. NULL) the element does not belong to a set (yet). Otherwise, through pointer p we access the element, and through its **rep** we find the representative of the set to which r belongs. We return that linked list element (not r , nor e_r not $E[r]$). Obviously all those steps require $\Theta(1)$ time.

Continued.

3. Union(e_r, e_s). Operation Union requires $O(n)$ time.

We first find in constant time the **rep**(e_r) and **rep**(e_s) and let them be a and b respectively. If $a == b$ we stop, there is no need for a Union. Otherwise we proceed as follows. We then access the **car** fields of $E[a]$ and $E[b]$, the larger of the two decides the representative member of the Union **Union**(e_r, e_s). Let e_a be the element whose $E[a].car$ is the largest. This implies that we need to update the representative pointers of the smaller of the two sets which is the one with **rep** the other element, in this example e_b . Through the $E[b]$ information (**head** pointer) we scan the linked list of the smaller of the two sets and update the **rep** information to point to **Find**(e_a) which is **Find**(e_r). We then concatenate the two linked lists updating information in $E[a]$ and invalidating information in $E[b]$ including its **head**, **tail**, **car** fields. The presence of **head**, **tail** pointers allow us Finally the **car** field of $E[a]$ will be set to the sum of its previous value and the **car** field of $E[b]$.

The most expensive step is the update of the representative pointers of the smaller of the two sets (in this example the one having a representative element, element e_b). This update can be $\Theta(n)$ in the worst case and $O(n)$ in general. For example think of a scenario where $n/2$ elements in the larger of the two sets and $n/2 - 1$ elements in the smaller of the two sets and those sets are to be united. Obviously all those steps require $O(n)$ time, in general. The running time of a Union operation is thus $O(n)$.

4. Several Union operations. The collective running time of all $n - 1$ possible Union operations that can involve the n elements of E is no more than $O(n \lg n)$.

The proof is as follows. Pick an arbitrary element 5. How many times do we update the representative pointer of 5? Every time we update it, it means two things: (a) 5 was in the smaller of two sets on which a Union was performed, and (b) the size of the new set is at least double of the size of the set containing 5. (that fact that 5 had its representative pointer updated means that 5 belongs to the smaller of the two sets of the Union operation.) The number of times we update (the representative pointer of 5) is the number of times we can double the size of the set containing 5. Originally 5 is in a set of size 1. Thus $1 \rightarrow 2 \rightarrow 4 \rightarrow \dots$ shows that the number of updates for 5 is no more than $\lg n$. This is true for all elements, not just for element 5. We have n elements, thus the collective cost of updating all the representative pointers of all the elements in all those at most $n - 1$ Union operations is no more than $O(n \lg n)$. Thus any number of Union operations on a set E of n elements has running time $O(n)$.

Union-Find Data Structure

Performance continued

Continued.

5. Running time of $m > n$ operations. The collective running time of all those m operations is no more than $O(m + n \lg n)$.

There can be no more than n MakeSet operations. Each one is constant time thus the total number of MakeSet is $O(n)$ in general. There can be no more than $n - 1$ Union operations. Each one is $O(n)$ time but if there are several of them the collective running time of those several one is $O(n \lg n)$. There can be at least $m - n - (n - 1)$ and thus at most m Find operations. The cost of a Find operations is constant, the collective cost of all of them is $O(m)$. Adding all contributions the running of $m > n$ arbitrarily mixed Find, Union and MakeSet operations is the sum of those three contributions i.e. $O(m + n \lg n)$. ■

Definition 29 (Amortized Analysis of Union). One Union Operation can be $O(n)$ or $\Theta(n)$. Yet $n - 1$ Union operations can and will be $O(n \lg n)$. Thus the amortized cost of a Union operation is defined as the cost of all Union operations divided by the number of Union operations. Thus the **amortized cost of a Union** is $O(\lg n)$.

Undirected Graphs

Connectivity problems

A reminder

Definition 30 (Decision Problem). A decision problem is a problem with a YES or NO answer.

Here is a list of problems that are related to the connectivity of an undirected graph and can be solved using a UF-DS.

Problem 1 (*Is G connected?*).

Input. A (undirected) graph $G = (V, E)$ with the appropriate representation.

Output. YES if G is connected, NO if G is not connected.

Problem 2 (*Path connectivity between $u, v \in V$ in G .*).

Input. A (undirected) graph $G = (V, E)$ with the appropriate representation and two vertices $u, v \in V$.

Output. YES if there is a path from u to v , NO otherwise.

Problem 3 (*Number of connected components of G .*).

Input. A (undirected) graph $G = (V, E)$ with the appropriate representation.

Output. Print number of connected components of G .

Undirected Graphs

Connectivity problems

Problem 4 (*Give connected component of u in G .*)

Input. A (undirected) graph $G = (V, E)$ with the appropriate representation and a vertex $u \in V$ of G .

Output. Find the connected component of u .

Problem 5 (*Give vertex info of connected component of u in G .*)

Input. A (undirected) graph $G = (V, E)$ with the appropriate representation and a vertex $u \in V$ of G .

Output. List all the vertices that are in the same connected component as u .

Problem 6 (*Give cardinality of connected component of u in G .*)

Input. A (undirected) graph $G = (V, E)$ with the appropriate representation and a vertex $u \in V$ of G .

Output. Print the number of vertices in connected component of u .

Connectivity Problems

A solution template

Theorem 4. *Algorithm CC can be utilized for solving a variety of problems associated with the connectivity of an undirected graph $G = (V, E)$.*

```
CC(G)    // CC: Connected Component
1. for each vertex u in V {
2.     MakeSet(u);
3. }
4.     cc=n;
5. for each edge (u,v) in E {
6.     if Find(u) != Find(v)
7.         Union(u,v);
8.         cc--;
9. }
```

Analysis of CC. We use a UF-DS as previously described and analyzed. We build the UF-DS by utilizing the structure of graph $G = (V, E)$. A set of UF-DS will map to a connected component of the graph. The number of connected components of the graphs will be equal to the number of sets available at the end of the execution of the CC algorithm. If edge $e = (u, v)$ exists this means u and v must be in the same connected component. Thus the algorithm starts with a number of candidate connected components equal to n , each connected component being an individual vertex and then considers all the edges of the graph one by one. Edge (u, v) can trigger one of two events: (a) nothing if u, v already belong to the same component i.e. they are in the same set of a collection of disjoint sets maintained, (b) the union of u, v if currently u and v are in different sets (which implies that so far there has been no indication that would put them into the same connected component of G) and then the discovery of $e = (u, v)$ leads to the conclusions that u and v should be in the same connected component along with the vertices u' that are already in the same connected component as u , and the vertices v' that are already in the same connected component as v . Those vertices are in the sets of UF-DS to which u and v currently belong. The set is then extended by the discover of e through the union operation of line 7. The elements of UF-DS are the elements of V that is the n vertices of the graph. Each vertex triggers a MakeSet operation on the vertex itself. The number of MakeSet operations is thus n . The collective running time of MakeSet operations is thus $\Theta(n)$. Each edge of the graph possibly triggers a Union operation. Two vertices joined by an edge i.e. an edge being incident on those two vertices makes those two end-points belonging to the same connected component by way of them belonging to the same set of a UF-DS. The number of Find operations are 2 per edge (two endpoints of an edge). Thus the collective running time of all Find operations is $\Theta(2m) = \Theta(m)$. The Union operations is no more than $n - 1$. It is in fact n minus the number of connected components of graph G or equivalent n minus the number of sets available at the completion of the execution of CC. Based on previous claims the collective running time of all Union is $O(n \lg n)$. Thus the running time of CC is $O(m + n \lg n)$. ■

Connectivity Problems

Problem solutions

Example 2 (Problems 1-3). We provide solutions for Problem 1, Problem 2, Problem 3.

<i>Proof.</i>	Problem 1 solution.	Problem 2 solution	Problem 3 solution
	isGconnected(G)	Path(G,u,v)	nofCC(G)
1.	CC(G);	1. CC(G);	1. CC(G);
2.	if (cc==1)	2. return(sameCC(G,u,v));	2. return(cc);
3.	return(YES)		
4.	else	sameCC(G,u,v)	
5.	return(NO)	1. return(Find(u)==Find(v));	



Exercise 3 (Problems 4-6). Reason about the proof of correctness as shown for Problems 1-3. Solve the remaining Problems 4-6 similarly.

Connectivity Problems

Example run CC(G)

Example 3.

```

      2      4      6
      / \    /
     /   \  /
    /     \ /
   1 ----- 3  5

```

Initially: 1 2 3 4 5 6 : Lines 1-3 of CC.
cc=6 \ / / \ / : Line 4
 2 2 / \ /
(1,2) edge { 1, 2 } / | / : Line 5-7
cc=5 \ / | | : Line 8
 2 \ 2 / 2 | |
(2,3) edge {1, 2, 3 } | | : Line 5-7
 | |
cc=4 2 == 2
(1,3) edge 1 3 | 5 | 5 : Line 5
 | 5 | 5 : Line 8
(4,5) edge { 4, 5 } : Line 5-7
cc=3 : Line 8
nomore edges

Edges examined in this order: (1,2) (2,3), (1,3), (4,5); Edge (1,3) discarded

cc =3 : number of connected components is 3

2 2 2 5 5 6

1 2 3 4 5 6

Vertex set of connected component 2 has {1,2,3}

Vertex set of connected component 5 has {4,5}

Vertex set of connected component 6 has {6}

Document processing

Introduction

We provide now an application of a graph's representation in the problem of Web-based page ranking. Towards this we will introduce two page ranking algorithms: Kleinberg's HITS algorithm, and Brin and Page's PageRank.

Definition 31 (*Document*). A document in its simplest form is a sequence of words separated by some whitespace (e.g. space or tabs).

Definition 32 (*Paragraphs, Sentences, Words*). A document is a sequence of paragraphs. A paragraph is a sequence of sentences. A sentence is a sequence of words ending with a period. The words of a sentence are separate by whitespace (e.g. space or tabs). A word is a sequence of characters.

Definition 33 (*Corpus*). Let a corpus D be a document collection $D = \{d_j : j = 1, \dots, N\}$.

Definition 34 (*Query*). A query q in its simplest form is a document of few words that are known as search terms.

Definition 35 (*Word, keyword, term*). A document contains words, a subset of those words are keywords, and after some manipulation and simplification (e.g. removing suffixes) keywords become terms.

A car, a CAR or cars might get mapped to the same term e.g. car. All are considered keyword. However the article a or connective and might not be considered keyword by some search engine.

Definition 36 (*Index terms, search term*). Similar terms get mapped to the same index-term. A search term is a word that we type in the text box of a web-based search engine.

Thus terms such as a car, a sedan or a coupe might get mapped to the same index-term. Moreover terms such algorithm, algorithms and Algorithm or algorithmicality might also get mapped to the same index-term. Index terms are being used internally for the representation of keywords by a search engine.

A search term input by a user query gets similarly mapped to an index term (and possibly after autocorrecting typos eg algorithm for aglorithm). We call it a search term because in several cases the textbox accepts a primitive language that is supported by a web-based search engine.

Document processing $\rightarrow$ Internal Representation

Word $\rightarrow$ Keyword $\rightarrow$ Term $\rightarrow$ IndexTerm

Search Box processing $\rightarrow$ Internal Representation

Search term is a Word $\rightarrow^*$ IndexTerm

Definition 37 (Information retrieval). It is the field of study (Salton, 1968) that deals with the “structure, analysis, organization, storage, searching, and retrieval of information.”

Information retrieval is different from data retrieval; in the latter case data are more organized.

Definition 38 (Online searching). Online searching is computerized searching of information.

Online searching has been available for some time since the 70s. Information retrieval systems such as LexisNexis, DIA-LOG/ProQuest, MEDLARS, MEDLINE, have provided full-text and bibliographic search capabilities for text and text-related documents. Several early web-based systems searched only lists of documents or their titles. With the development of systems such as WebCrawler and AltaVista or Excite and Infoseek web-users became capable of searching the full text of web-pages.

Definition 39 (Web Search). An instance of information retrieval that involves the search of web-based information. A web search can be realized in a variety of ways including the following.

- **A Search Engine**, that searches and indexes Web-based documents.
- **A List of Web Directories** that classify a list of Web documents by subject, and
- **Link Information** that can be used to exploit the hyperlink structure of the world-wide web, to facilitate the search (rather than rank the results of a search engine).

Definition 40 (Web Search using Search Engine). Web-searching is mostly associated with its realization by a search engine which is an IR system and deals, among other things, with locating and then retrieving the relevant information, if it is not stored locally, organizing it and storing it in an efficient form, and then ranking the results of the retrieval process.

It is noted that web-searching is syntactic search. We search user-specified words or patterns in the text of a web-based document. We do not (ordinarily) do a natural language analysis of the text to say extract the text semantics. However things evolve.

Definition 41 (A query q). Let q be a query that is being input to the textbox of a web-based search engine.

Definition 42 (Corpus and its documents). Let D be a corpus of N documents, and let

$$D = \{d_j : j = 1, \dots, N\}$$

be a corpus with N documents, and let d_j be the j -th document of the corpus.

In the remainder the terms document, Web-based document or web-page or just page will be used interchangeably.

Definition 43 (Similarity measure $s(q, d_j)$). The similarity measure $s(q, d_j)$ establishes how relevant document d_j is to query q .

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 D . This "comparison" checks whether the terms of the query appear (or not) as needed in document d_j . This comparison establishes the similarity between the query q and the particular d_j and the similarity measure is recorded in $s(q, d_j)$. The measure can be Boolean with a **true** or **false** value, or more complicated if the similarity function s is real-valued, usually in the range between 0 and 1, and distinguishes between terms that for example

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

Web search ranking

Similarity based

Thus the **relevance of a query to a document** (and a document d_j in this context is 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). We can then define WebPage ranking as follows.

Problem 7 (*WebPage Ranking based on similarity*).

Input. A corpus D of Web based documents (e.g. web-pages), $D = \{d_j : j = 1, \dots, N\}$ and a query q .

Output. Rank documents based on

$$R(q, d_j) = s(q, d_j), \forall j = 1, \dots, N,$$

where $s(q, d_j)$ is the similarity measure for query q relative to document d_j .

Such an approach however suffers from the following issue.

Issue 1: (Keyword stuffing). Create a document B that contains all relevant words (e.g. all English words of an English language dictionary). Then document B becomes relevant to every query that contains English language words found in a typical dictionary.

Web search ranking

Link based

An alternative approach for a WebPage ranking is needed to avoid issues related to keyword stuffing. The first such approach intended to utilize the link information of Web-based documents (e.g. web-pages). The idea was that heavily linked (Web-)pages are important pages and lightly linked one are less important. However there is still a need to incorporate into this link component the content information of a web-page.

Problem 8 (*WebPage Ranking based on link information*).

Input. A corpus D of web-pages, $D = \{d_j : j = 1, \dots, N\}$, a query q , and a graph representation of the link structure of those web-pages.

Output. Rank documents based on

$$R(q, d_j) = l(d_j), \forall j = 1, \dots, N,$$

where $l(d_j)$ is the link measure of document d_j .

In this definition there is no contribution of q as $R(q, d_j)$ depends solely on $l(d_j)$. What can $l(d_j)$ look like ? It might depend on web-pages d_i pointing to d_j or web-pages d_k pointed by d_j .

Such an approach however suffers from the following issue.

Issue 2: (A Link farm). Several artificial web-pages (possibly each one a byproduct of keyword stuffing) are pointing to one another (intra-cluster), or to other pages of the “farm” (inter-cluster), or to pages of another link farm for which a degree of collusion has been established (e.g. linkage for pay.)

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 (i.e. $l(i, j) = A(i, j)$, where A is the adjacency matrix of the graph of the web) or not, and β is a user defined parameter (e.g. $\beta = 0.2$). Then $R(q, d_j) = s(q, d_j) + \beta \sum_i R(q, d_i)l(i, j)$. We also denote $link(d_j) = \beta \sum_i R(q, d_i)l(i, j)$.

Ranking based on context and web-structure

Overview

Modern web-based search engines (or web-searching methods in general) do not rely on $s(q, d_j)$ exclusively to rank a web-page. 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 web-pages are pointed by d_j and are these pages important (e.g. **authoritative**),
- (b) what web-pages point to d_j and are these pages important (e.g. **hubs of information** such as a directory page).

Problem 9 (*WebPage Ranking*).

Input. A corpus D of web-pages, $D = \{d_j : j = 1, \dots, N\}$, a query q , and a graph representation of the link structure of those web-pages.

Output. Rank documents based on

$$R(q, d_j) = s(q, d_j) + l(d_j), \forall j = 1, \dots, N,$$

where $s(q, d_j)$ is the similarity measure of web-page d_j relative to query q , and $l(d_j)$ is the link measure of document d_j .

HITS Algorithm by Kleinberg (1998)

Introduction

The HITS model was introduced by Kleinberg in 1998. HITS stands for *H*ypertext Induced Topic(al) Search. The then introduced model assigns two ranks to each web-page. One rank signifies how important it is as a *hub* of information, and the other how important it is as an *authoritative* source of information. The resulting ranking algorithm is known as *HITS*. The ideas behind the model can be summarized as follows.

We use the same setup as before of a query q and corpus D . The HITS algorithm first limits the scope of searching to a subset of D and attempts to rank pages that are relevant to a query q based on link structure. The rank is determined based on the following observations.

Definition 44 (Authoritative web-page and Hub web-page). Let q be a query on a corpus D . A web-page of D relative to a query q is considered authoritative, if it is referenced by many hub pages that are relevant to q . A web-page of D relative to a query q is considered a hub of information, if it references (points to) many authoritative pages that are relevant to q . Good authoritative and hub pages reinforce one another.

HITS Algorithm by Kleinberg (1998)

Introduction

Question 1 (Finding authoritative and hub pages for q). How can one find such authoritative and hub pages for a query q over a corpus D ?

Answer.

- Step 1.** Submit query q to a similarity-based engine and record the top results based on $s(q, d_j)$ into the **root set** $RS(q)$ of relevant to q pages.
- Step 2.** Expand $RS(q)$ into a base set $BS(q)$ to include pages pointed by $RS(q)$ pages. But first, find base set $\mathbf{BS}_1(\mathbf{q}) = \{\text{all pages pointed by an } RS(q) \text{ page}\}$.
- Step 3.** Afterwards, include into $BS(q)$ pages pointing to $RS(q)$ but limit them to no more than B , $\mathbf{BS}_2(\mathbf{q}) = \{\text{B pages pointing to an } RS(q) \text{ page}\}$.
- Step 4.** Form then $\mathbf{BS}(\mathbf{q}) = \mathbf{RS}(\mathbf{q}) \cup \mathbf{BS}_1(\mathbf{q}) \cup \mathbf{BS}_2(\mathbf{q})$.
- Step 5.** On the base set $BS(q)$ identify the authority and hub pages by repeating as many times as needed the authority step, then the hub step, and then the scaling step of algorithm HITS.

Note that we start with the document collection that is the corpus D of N documents, and by step 5 we have for a given query q a smaller set, the base set $BS(q)$ that is a subset of D . In the remainder we deal with the documents of the base set. Let n , where $n \leq N$, be the number of documents of $BS(q)$ and for that set we find its authoritative and hub pages as they relate to q . The link structure that will be used is the subgraph of the graph of the Web with vertex set $BS(q)$. Furthermore, when we run through the pages of $BS(q)$ we will use index i as opposed to index j that used to run through the pages of D . ■

Definition 45 (Authority value $a(d_i)$). Each document $d_i \in BS(q)$ would be later referred to as document i from its subscript. The authority score (value) $a(d_i)$ of d_i will be denoted by $a(i) = a_i$.

Definition 46 (Hub value $h(d_i)$). Each document $d_i \in BS(q)$ would be later referred to as document i from its subscript. The hub score (value) $h(d_i)$ of d_i will be denoted by $h(i) = h_i$.

Definition 47 (Authority and Hub vectors). They are vectors of length n formed from the authority and hub scores.

$$\vec{h} = (h(1), \dots, h(n)) \quad \vec{a} = (a(1), \dots, a(n))$$

An alternative depiction is as

$$\vec{h} = (h_1, \dots, h_n) \quad \vec{a} = (a_1, \dots, a_n)$$

Definition 48 (Synchronous update). By performing the authority, hub and scaling steps of HITS, in a synchronous update $\vec{h}, \vec{a}$ for a given time t are obtained from the scores available in $\vec{h}, \vec{a}$ in the previous time step $t - 1$. Thus two copies of $\vec{h}, \vec{a}$ are available, the one of the previous iteration $t - 1$ and the one generated at iteration t .

Definition 49 (Asynchronous update). By performing the authority, hub and scaling steps of HITS, in an asynchronous update $\vec{h}, \vec{a}$ for a given time t are obtained from whatever scores are available in those vector $\vec{h}, \vec{a}$ whether the scores there were derived earlier or at time t . Thus one copy of $\vec{h}, \vec{a}$ is available and used.

In the remainder we shall use $\vec{h}^{(t)}$ and $\vec{a}^{(t)}$ for $\vec{h}, \vec{a}$ at iteration t , whether it is for the synchronous or asynchronous case and whether we actually use that many vectors (we do not) or not. It is for convenience only.

HITS Algorithm by Kleinberg (1998)

The algorithm

Algorithm 1: HITS algorithm (synchronized version, initial values $1/\sqrt{n}$)

Input: Graph G of $BS(q)$ and its representation; bound R or ϵ

Output: $\vec{a}$ and $\vec{h}$, as needed

```
1  $t \leftarrow 0$  ;
2  $\vec{h}^{(0)} \leftarrow 1/\sqrt{n}$  ;
3  $\vec{a}^{(0)} \leftarrow 1/\sqrt{n}$  ;
4 while NotDone do
5    $t \leftarrow t + 1$  ;
6   for  $j = 1, 2, \dots, n$  do                                     /* Authority (update) step */
7      $a_j^{(t)} \leftarrow \sum_{i \rightarrow j} h_i^{(t-1)}$  ;
8   end
9   for  $k = 1, 2, \dots, n$  do                                     /* Hub (update) step */
10     $h_k^{(t)} = \sum_{l \rightarrow k} a_l^{(t)}$  ;
11  end
12  for  $i = 1, 2, \dots, n$  do                                     /* Scaling step */
13     $a_i^{(t)} = a_i^{(t)} / \sqrt{\sum_j a_j^{(t)} * a_j^{(t)}}$  ;
14     $h_i^{(t)} = h_i^{(t)} / \sqrt{\sum_j h_j^{(t)} * h_j^{(t)}}$  ;
15  end
16  if Done then                                               /* Termination check */
17    Break ;
18  end
19 end
```

HITS Algorithm by Kleinberg (1998)

The method

Method 1 (HITS algorithm).

Input: A graph G representing $BS(q)$ of n vertices corresponding to the n web-pages of $BS(q)$. Also, an iteration bound R or an error bound ϵ .

Output: Either

- Vectors $\vec{a}$ and $\vec{h}$ after the scaling step of iteration R , or
- Vectors $\vec{a}$ and $\vec{h}$ after the scaling step of an iteration t such that $|h_i^{(t)} - h_i^{(t-1)}| \leq 10^{-\epsilon}$ and $|a_i^{(t)} - a_i^{(t-1)}| \leq 10^{-\epsilon}$ for all $i = 1, \dots, n$.

There are other alternatives for the initial values in lines 1 and 2 of Algorithm 1 of the previous page. For example, $\vec{h}^{(0)} = \vec{a}^{(0)} = 0$, or $\vec{h}^{(0)} = \vec{a}^{(0)} = 1$, or $\vec{h}^{(0)} = \vec{a}^{(0)} = 1/n$. The check of line 16 of Algorithm 1 determines whether this is the last iteration or not. If there is a bound R on the number of iterations realized, then *Done* is realized by a check $t == R$. Otherwise if there is no explicit R then we go on until a desired error bound is reached. Let this error bound be $10^{-\epsilon}$. Then *Done* evaluates to true at the current iteration t , if $|h_i^{(t)} - h_i^{(t-1)}| \leq 10^{-\epsilon}$ and $|a_i^{(t)} - a_i^{(t-1)}| \leq 10^{-\epsilon}$ for all i .

The synchronous approach steps are individually shown below.

Synchronous Approach

$$\begin{aligned}
 \text{Authority step } j = 1, \dots, n : \quad a_j^{(t)} &= \sum_{i \rightarrow j} h_i^{(t-1)} \\
 \text{Hub step } k = 1, \dots, n : \quad h_k^{(t)} &= \sum_{k \rightarrow l} a_l^{(t)} \\
 \text{Scaling step } i = 1, \dots, n : \quad \text{Scale } a_i^{(t)}, \text{Scale } h_i^{(t)}
 \end{aligned} \tag{1}$$

They can be replaced by an asynchronous execution. This is indicated on the next page.

HITS Algorithm by Kleinberg (1998)

The asynchronous method

Note that when only one copy of a, h is maintained checking for an error bound can not be determined then.

Method 2 (Asynchronous HITS algorithm).

Input: A graph G representing a set $BS(q)$ of n element vertices corresponding to the n web-pages of $BS(q)$ and iteration bound R .

Output: Vectors $\vec{a}$ and $\vec{h}$ after the scaling step of iteration R .

Asynchronous Approach

$$\begin{aligned}
 \text{Authority step } j = 1, \dots, n : \quad & a_j = \sum_{i \rightarrow j} h_i \\
 \text{Hub step } k = 1, \dots, n : \quad & h_k = \sum_{k \rightarrow l} a_l \\
 \text{Scaling step } i = 1, \dots, n : \quad & \text{Scale } a_i, \text{Scale } h_i
 \end{aligned} \tag{2}$$

Example 4. For the three vertex graph shown next, use initial values 1, and complete one iteration.

```

0 -----> 1  Adjacency A = [ 0 1 1 ]  A' = [ 0 0 0 ]  Adjacency List
 \         /      matrix      [ 0 0 1 ]      [ 1 0 0 ]  0: 1 2
  \       /      [ 0 0 0 ]      [ 1 1 0 ]  1: 2
   v     v
   2

A step: a(0) = 0      [ a(0) ]      [ h(0) ]
        a(1) = h(0)    = [ a(1) ] = A'*[ h(1) ] => a=A'* h
        a(2) = h(0)+h(1) [ a(2) ]      [ h(2) ]

```

$$\begin{aligned} \text{H step: } h(0) &= a(1)+a(2) \quad \begin{bmatrix} h(0) \end{bmatrix} && \begin{bmatrix} a(0) \end{bmatrix} \\ h(1) &= a(2) && \begin{bmatrix} h(1) \end{bmatrix} = A * \begin{bmatrix} a(1) \end{bmatrix} \Rightarrow h=A * a \\ h(2) &= 0 && \begin{bmatrix} h(2) \end{bmatrix} && \begin{bmatrix} a(2) \end{bmatrix} \end{aligned}$$

	0/	1/	2/		0/	1/	2/	
Initial Values	h :	1	1	1	a :	1	1	1
t=1	A step:					0	1	2
	H step:	3	2	0				
	Scale step:	0.83	0.55	0.00		0.00	0.44	0.89

Definition 50 (a and h and adjacency matrix A). If the subgraph of the web for $BS(q)$ has adjacency matrix A and A' is the transpose matrix of A , then using the discussion also of Example 4 we conclude that

$$\vec{a}^{(t)} = A' \vec{h}^{(t-1)}$$

and

$$\vec{h}^{(t)} = A \vec{a}^{(t)}.$$

The recursive formulations of Definition 50 are intertwined. If we unfold them and unwind them, then we get the following. This is also known as the **power method**.

$$\begin{aligned} \vec{a}^{(1)} &= A' \vec{h}^{(0)} & \vec{h}^{(1)} &= A \vec{a}^{(1)} \\ \vec{a}^{(2)} &= A' \vec{h}^{(1)} \Rightarrow \vec{a}^{(2)} = A' (A \vec{a}^{(1)}) = (A' A) \vec{a}^{(1)} & \vec{h}^{(2)} &= A \vec{a}^{(2)} \Rightarrow \vec{h}^{(2)} = A (A' \vec{h}^{(1)}) = (A A') \vec{h}^{(1)} \end{aligned}$$

Thus after the first iteration, the values of a, h can then be multiplied by a power of the product AA' or $A'A$ to generate the a, h values in any iteration as shown on the right hand side below.

$$\begin{aligned} \vec{a}^{(t)} &= A' \vec{h}^{(t-1)} \implies \vec{a}^{(t)} = A' A \vec{a}^{(t-1)} \implies \dots \implies \vec{a}^{(t)} = (A' A)^{t-1} \vec{a}^{(1)} \\ \vec{h}^{(t)} &= A \vec{a}^{(t)} \implies \vec{h}^{(t)} = A A' \vec{h}^{(t-1)} \implies \dots \implies \vec{h}^{(t)} = (A A')^{t-1} \vec{h}^{(1)} \end{aligned}$$

Method 3 (Power Method). For the synchronous case the hub and authority scores (values) at iteration t is,

$$\vec{a}^{(t)} = (A'A)^{t-1}\vec{a}^{(1)}, \quad \vec{a}^{(1)} = A'\vec{h}^{(0)},$$

and

$$\vec{h}^{(t)} = (AA')^{t-1}\vec{h}^{(1)}, \quad \vec{h}^{(1)} = A\vec{a}^{(1)}.$$

Note that A is a sparse matrix (too many zeroes in A). Therefore using this formulation to compute a and h is inefficient in time and space. A related question is how large will t be for convergence. And does it converge, why and under what circumstances (e.g. what are the precondition for convergence)?

DRAFT

HITS

Another example

Example 5. For the example with A as shown, we show for $t = 7$ the a and h vectors for a variety of initial values: (a) initial all-one vectors, (b) initial all- $1/n$ vectors, and (c) initial all- $1/\sqrt{n}$ vectors.

Kleinberg's HITS (Hub and Authority) Algorithm

```
0 : 0 1 1
1 : 0 0 1   (Adjacency list)
2 : 0 0 0
```

(a) vectors are initialized to = 1.000000

```
Base   : 0 :A/H[ 0]=1.00000/1.00000 A/H[ 1]=1.00000/1.00000 A/H[ 2]=1.00000/1.00000
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]=0.00000/0.85059 A/H[ 1]=0.52549/0.52582 A/H[ 2]=0.85080/0.00000
Iterat : 5 :A/H[ 0]=0.00000/0.85064 A/H[ 1]=0.52570/0.52574 A/H[ 2]=0.85067/0.00000
Iterat : 6 :A/H[ 0]=0.00000/0.85065 A/H[ 1]=0.52573/0.52573 A/H[ 2]=0.85065/0.00000
Iterat : 7 :A/H[ 0]=0.00000/0.85065 A/H[ 1]=0.52573/0.52573 A/H[ 2]=0.85065/0.00000
```

(b) vectors are initialized to = 0.333333

```
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]=0.00000/0.85059 A/H[ 1]=0.52549/0.52582 A/H[ 2]=0.85080/0.00000
Iterat : 5 :A/H[ 0]=0.00000/0.85064 A/H[ 1]=0.52570/0.52574 A/H[ 2]=0.85067/0.00000
Iterat : 6 :A/H[ 0]=0.00000/0.85065 A/H[ 1]=0.52573/0.52573 A/H[ 2]=0.85065/0.00000
Iterat : 7 :A/H[ 0]=0.00000/0.85065 A/H[ 1]=0.52573/0.52573 A/H[ 2]=0.85065/0.00000
```

(c) vectors are initialized to = 0.577350

```
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]=0.00000/0.85059 A/H[ 1]=0.52549/0.52582 A/H[ 2]=0.85080/0.00000
Iterat : 5 :A/H[ 0]=0.00000/0.85064 A/H[ 1]=0.52570/0.52574 A/H[ 2]=0.85067/0.00000
Iterat : 6 :A/H[ 0]=0.00000/0.85065 A/H[ 1]=0.52573/0.52573 A/H[ 2]=0.85065/0.00000
Iterat : 7 :A/H[ 0]=0.00000/0.85065 A/H[ 1]=0.52573/0.52573 A/H[ 2]=0.85065/0.00000
```

HITS

Another example

Example 6. For the graph with the adjacency matrix $A = L$ shown and a synchronous version with initial values 1 we have.

	0	1	2	3	4	t=0	a=(1	1	1	1	1)	h =(	1	1	1	1	1)

A=	0	0	0	1	1	t=1 A:	a=(1	0	0	3	2)						
L=	1	0	0	1	0	H:						h =(	5	3	5	1	0)
	2	0	0	1	1	Scale:	a=(.26	0	0	.80	.53)	h =(	.64	.38	.64	.12	0)
	3	1	0	0	0												
	4	0	0	0	0												
						t=?	a=(0	0	0	.78	.61)	h =(	.65	.36	.65	0	0)

[Use $\sqrt{14}=3.74$ and $\sqrt{60}=7.74$]

DRAFT

HITS Algorithm

A discussion

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 ranking, or the h vector should be used instead, or it makes sense to use a combination of the two and what that combination should be! The a vector makes more sense to use though.

The iterative (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 web-page connections.

Both matrices AA' and $A'A$ are well-defined and by matrix analysis (e.g. Perron-Frobenius theorem) vectors $\vec{a}^{(t)}$ and $\vec{h}^{(t)}$ will then converge to the dominant eigenvector of one or the other matrix. The rate of convergence however depends on the difference (i.e. **eigengap**) between the largest λ_1 and second largest λ_2 eigenvalues of the two matrices. HITS usually converges (but no guarantee is offered) to a solution after 10-20 iterations. One problem with HITS is the existence of multiple connected components in the graph represented by A . 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 and we stop this discussion here.

Brin and Page's PageRank (1998)

Overview

The algorithm used in the ranking of Google query results is now presented. The original algorithm, named **PageRank** was introduced in 1998 by Page and Brin. How much it is still being used in Google is open to discussion.

We first compare PageRank to HITS to highlight similarities and differences of the two. Both algorithms view the web in the form of a directed graph $G = (V, E)$.

The HITS algorithm however computes a and h values for a subgraph of the graph of the web, the base set $BS(q)$. In order to obtain $BS(q)$ one has to submit q to a similarity-based search engine, obtain $RS(q)$ and then enhance it into eventually set $BS(q)$.

Google's view of the web is a direct view skipping all those intermediate steps. 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.

Method 4 (PageRank models WebBrowsing behavior). 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. Initially the user is at some page p .

- R1.** 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 starting page p has 5 links the user picks one of them, let it be t at random.) Then the user follows such a selected outgoing link and browses page t and repeats steps R1 or R2 as needed.
- R2.** 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 R1 the user decides to pick a random page (not pointed by the current page) to visit next. (So if the user is at the starting page p and it is decided to use this rule, a random page x is chosen uniformly at random among the n web-pages available, including page p !)

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

Question 2. What is the probability that a given page A will be visited? This determines the "importance" i.e. the "rank" of the page!

Utilizing notation from Kleinberg's algorithm description, we note that for the PageRank algorithm n will denote NOT the number of pages of $BS(q)$ that was the case for Kleinberg's algorithm, but it will be NOW the number of pages of the web, a much larger number (and a less refined set of pages).

For page j , let $Pr(j)$ be the probability that page j is being visited by the 'random browsing' WebBrowsing behavior. The page rank $PR(j)$ of page j would be related to $Pr(j)$.

Definition 51 (Rank of a web-page). The rank $PR(j)$ of a web-page j is related to the probability $Pr(j)$ that the web-page j gets visited under the random browsing WebBrowsing behavior.

Theorem 5 ($Pr(j)$ for the graph of the Web). *Let us have a graph of web-pages connected through links. Let j be a web-page and let $k = \text{ind}(j)$ be the in-degree of page j . Let $i_1, \dots, i_k$ be vertices such that (i_l, j) is incident from vertex i_l to vertex j for all $l = 1, \dots, k$. Then, the probability $Pr(j)$ of page j being visited would be determined as follows.*

$$Pr(j) = \sum_{l=1}^k \frac{Pr(i_l)}{\text{outd}(i_l)}.$$

This is Method 4 for $d = 1$ i.e. rule R1 is always chosen, and rule R2 is not used. The summand $Pr(i_l)/\text{outd}(i_l)$ can express the probability of visiting j if we are at i_l and then $Pr(i_l)$ is the probability of visiting i_l when we perform a random walk on the vertices of the graph following edges.

Brin and Page's PageRank (1998) overview

Strongly Connected Graph

The problem with the previous approach (Method 4 for $d = 1$) is that if the graph is NOT strongly connected, the random walk is restricted on a strongly connected component of the graph.

Consider a graph with five vertices. Vertex 1 points to 2, 2 points to 3 and 3 points 1. The vertex 4 points to 5 and 5 points 4. The graph has two connected components one consisting of vertices 1,2,3 and the other of 4,5. If we start the random walk on one of 1, 2, 3 we stay at that strongly connected component; otherwise we stay at the 4,5 strongly connected component. Thus, probability-wise, if we start at say 2, the probability of visiting 4 or 5 is 0.

For this reason Brin and Page introduced the notion of teleportation or equivalently of occasional random browsing. They relaxed the $d = 1$ into something lower than one e.g. $d = 0.85$ (i.e. rule R1 is followed not always but with probability $d = 0.85$) and allowed for the introduction of rule R2 with probability $1 - d = 0.15$. Rule R2 is the teleportation (also known as random web-browsing rule).

DRAFT

Brin and Page's PageRank (1998) overview

Continued

Thus we introduce the random browsing WebBrowsing behavior where links are followed only a fraction at the time but we allow the possibility that from time to time a (uniformly at) random web-page is selected. $Pr(j)$ is then defined so that it is non-zero even if $Pr(j)$ was to be zero. We add to $Pr(j)$ a small positive component $(1 - d)/n$ and adjust the previous expression for $Pr(j)$ accordingly. A justification that this makes sense is the random browsing WebBrowsing behavior.

Theorem 6 ($Pr(j)$ under the random browsing WebBrowsing behavior). *Let us have a graph of web-pages connected through links. Let j be a web-page and let $k = \mathbf{ind}(j)$ be the in-degree of page j . Let $i_1, \dots, i_k$ be vertices such that (i_l, j) is incident from vertex i_l to vertex j for all $l = 1, \dots, k$. Under the random browsing WebBrowsing behavior, the probability $Pr(j)$ of page j being visited would be determined as follows.*

$$Pr(j) = \frac{(1 - d)}{n} + d \sum_{l=1}^k \frac{Pr(i_l)}{outd(i_l)},$$

where d is user-definable and $0 < d < 1$. A typical value for d is $d = 0.85$.



Definition 52 (PageRank of page j). The PageRank $PR(j)$ of page j is then defined as follows.

$$PR(j) = \frac{(1 - d)}{n} + d \cdot \sum_{l=1}^k \frac{PR(i_l)}{outd(i_l)},$$

where d is user-definable and $0 < d < 1$. A typical value for d is $d = 0.85$.

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.

PageRank

The (synchronous) method

Method 5 (PageRank algorithm).

Input: A graph G representing the n vertices of the Web. A parameter d such that $0 < d < 1$. Either an iteration bound R or an error bound ϵ .

Output: Run the steps of the PageRank algorithm and either print

- Vector $\vec{PR}$ at the end of iteration R , or
- Vector $\vec{PR}$ at the end of an iteration t such that $|PR_i^{(t)} - PR_i^{(t-1)}| \leq 10^{-\epsilon}$ for all $i = 1, \dots, n$.

In the follow up discussion we simplify notation. What we called before vertex j is now vertex A . The edges (i_l, j) incident to vertex j have become (T_i, A) and incident to vertex A , where $m = \text{ind}(A)$ and $i = 1, \dots, m$. Therefore, suppose we have pages $T_1, \dots, T_m$ pointing to page A and let $\text{outd}(T_i)$ be the out-degree of T_i and m be the in-degree of vertex A i.e. $m = \text{ind}(A)$. Let $n = N$ be the corpus size. The pagerank $PR(A)$ of A is then defined by the pagerank of the web-pages pointing to A . A parameter d is used where $0 < d < 1$.

$$PR(A) = \frac{(1-d)}{n} + d \cdot \left(\frac{PR(T_1)}{\text{outd}(T_1)} + \dots + \frac{PR(T_m)}{\text{outd}(T_m)} \right) = \frac{1-d}{n} + d \cdot \sum_{i=1}^m \frac{PR(T_i)}{\text{outd}(T_i)} \quad (3)$$

This equation represents the asynchronous version of algorithm PageRank. Note that for initialization we prefer $PR_i^{(0)} = 1/n$ although $PR^{(0)} = 1$ may also be used. This is because $\sum_i PR_i^{(0)} = 1$ and thus $PR_i^{(0)}$ conforms to a probability property. The synchronous version is shown below and in Algorithm 2 that follows.

$$PR^{(t)}(A) = \frac{1-d}{n} + d \cdot \sum_{i=1}^m \frac{PR^{(t-1)}(T_i)}{\text{outd}(T_i)} \quad (4)$$

PageRank

The algorithm

Algorithm 2: PageRank algorithm (synchronized version, initial values $1/n$)

Input: Graph G of $BS(q)$ with n vertices and its representation. Parameter d , such that $0 < d < 1$. Bound R or ϵ , as needed

Output: $\vec{PR}$, as needed

```
1  $t \leftarrow 0$  ;
2  $\vec{PR}^{(0)} \leftarrow 1/n$  ;
3 while NotDone do
4    $t \leftarrow t + 1$  ;
5   for  $i = 1, 2, \dots, n$  do                                /* Synchronous update step */
6      $PR_i^{(t)} = \frac{1-d}{n} + d \cdot \sum_{j \rightarrow i} \frac{PR_j^{(t-1)}}{outd(j)}$ 
7   end
8   if Done then                                           /* Termination check */
9     Break ;
10  end
11 end
```

PageRank Example(s)

Example 7. With reference to the graph used previously we show a synchronous and asynchronous execution for the regular initial condition of $1/n$ as used in PageRank.

(a) Synchronous all $1/3$

```
Base   : 0 :PR[ 0]=0.33333 PR[ 1]=0.33333 PR[ 2]=0.33333
Iter   : 1 :PR[ 0]=0.05000 PR[ 1]=0.19167 PR[ 2]=0.47500
Iter   : 2 :PR[ 0]=0.05000 PR[ 1]=0.07125 PR[ 2]=0.23417
Iter   : 3 :PR[ 0]=0.05000 PR[ 1]=0.07125 PR[ 2]=0.13181
Iter   : 4 :PR[ 0]=0.05000 PR[ 1]=0.07125 PR[ 2]=0.13181
```

(b) Asynchronous all $1/3$

```
Base   : 0 :PR[ 0]=0.33333 PR[ 1]=0.33333 PR[ 2]=0.33333
Iterat : 1 :PR[ 0]=0.05000 PR[ 1]=0.07125 PR[ 2]=0.13181
Iterat : 2 :PR[ 0]=0.05000 PR[ 1]=0.07125 PR[ 2]=0.13181
```

Exercise 4. Find the pagerank values for the collusion graph with the adjacency matrix $A = L$ shown next. Is this consistent with your expectations? Use both the synchronous and asynchronous approach for initial values 1, and $1/n$.

```
A=L= [ 0 1 0 ]
      [ 0 0 1 ]
      [ 1 0 0 ]
```

PageRank

A discussion

Each page has its own PageRank (eg. $PR(A), PR(T_i)$). Each page is recording its number of outgoing links i.e. $outd(T_i)$. The PageRank 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 $PR(T_i)/outd(T_i)$. 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.)

The normalized rank of Eq. (3) 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, **R1** and **R2**.

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 . 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. 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. What if we have a cycle or a very long back-chain?

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.

Definition 53 (Eigenvalues and Eigenvectors). For an $n \times n$ matrix A , vector x is called the (right) eigenvector of eigenvalue λ of A if and only if $Ax = \lambda x$.

Theorem 7 (Spectral Theorem). *The spectral theorem for matrices states that if there are n eigenvectors of A that form a basis (i.e. they are linearly independent) of dimension n , then A can be written as $A = Q\Lambda Q^{-1}$, where Q is a matrix whose i -th column is the i -th eigenvector and Λ is a diagonal matrix whose i -th diagonal entry is λ_i (i.e. the eigenvalue corresponding to the i -th eigenvector).*

Theorem 8 (Determinant of A and its eigenvalues). *The determinant of A is the products of its eigenvalues,*

$$|A| = \lambda_1 \dots \lambda_n.$$

Definition 54 (Convention (real eigenvalues)). Let the eigenvalues of A be relabeled and numbered so that $\lambda_1 > \lambda_2 > \dots$ that is we list the eigenvalues from the largest to the smallest one.

This convention is valid for positive and real eigenvalues. For negative or complex eigenvalues it is augmented as follows.

Definition 55 (Convention (negative or complex eigenvalues)). Let the eigenvalues of A be relabeled and numbered so that $|\lambda_1| > |\lambda_2| > \dots$ that is we list the eigenvalues from the one with the largest measure to the smallest one. The $|\cdot|$ represents the measure (magnitude) of a complex number or the absolute value of a real number.

Definition 56 (Unit vector $\mathbf{e}$). Vector e is the unit vector, an all-one vector where $\mathbf{e}' = (1, 1, \dots, 1)$.

Definition 57 (Adjacency matrix of G is L). L is the adjacency matrix of a graph $G = (V, E)$ whose (i, j) -th entry is 1 if $(i, j) \in E$ and 0 otherwise. L' is the transpose of L whose (i, j) -th entry is 1 if $(j, i) \in E$ and 0 otherwise.

Definition 58 (Weighed adjacency matrix C). The weighed adjacency matrix C can be obtained from L if its (i, j) -th entry is normalized so that $c_{ij} = 1/d(i)$ for $(i, j) \in E$ and 0 otherwise. Here, $d(i)$ is the degree (out-degree for directed graphs) of vertex i . If the i -th row of L is $(0 \quad 1 \quad 1)$, then the corresponding row of C would become $(0 \quad 1/2 \quad 1/2)$.

Definition 59 (An eigenvector of C). Matrix C obtained from the adjacency matrix of a graph has a nice property: $Ce = 1e$, i.e. for matrix C the unit vector is an eigenvector for eigenvalue 1. This is because the rows of C add up to 1.

Definition 61 (Transpose C' of C). Transposing matrix C to get C' we obtain a matrix that has an (i, j) -th entry that is $1/d(j)$ for $(j, i) \in E$.

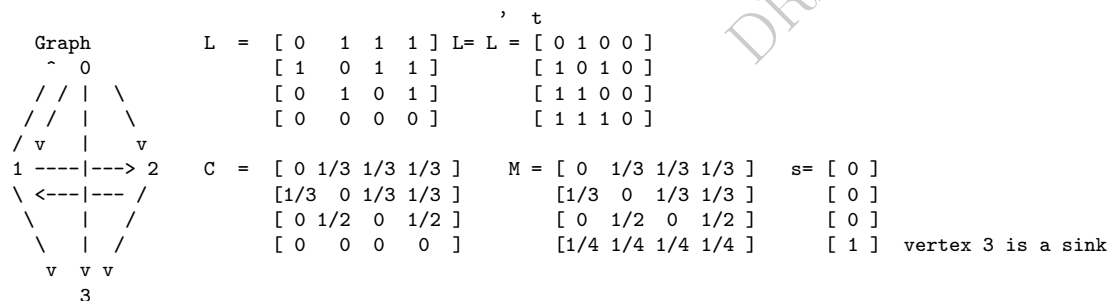


Figure 14: Sinks and their effects

Example 8 (Example of Figure 14). We use the example of Figure 14 as guidance to the rest of this discussion. We simplify notation and use r_i to denote the rank of page i . Then r is the (column) vector of ranks i.e. an $n \times 1$ matrix. Therefore r' is a row-vector of $1 \times n$ ranks, the transpose of r . L is the adjacency matrix and L' is the transpose of the adjacency matrix. Moreover C is derived from L if row i is divided by the number of ones of row i (for non sink rows). Thus rows 0 and 1 are divided by 3, row 2 by 2 and row 3 (last row) is a sink. We can then derive from C matrix M (modified C) that for a sink it has $1/n$ on every element of a row. Then $M = C + s(1/n)e'$. In this expression e is an $n \times 1$ vector of ones, and e' its transpose, a $1 \times n$ row vector. Moreover s is an $n \times 1$ vector with entries zeros except for the sinks that have an 1. Thus for our example only the last entry of s is a one. Then the PageRanks of the 4 vertices are computed by Equation 4 as follows.

$$\begin{aligned} r_0 &= \frac{1-d}{4} + d \left(\frac{1}{3} r_1 \right) \\ r_1 &= \frac{1-d}{4} + d \left(\frac{1}{3} r_0 + \frac{1}{2} r_2 \right) \\ r_2 &= \frac{1-d}{4} + d \left(\frac{1}{3} r_0 + \frac{1}{3} r_1 \right) \\ r_3 &= \frac{1-d}{4} + d \left(\frac{1}{3} r_0 + \frac{1}{3} r_1 + \frac{1}{2} r_2 \right) \end{aligned}$$

Theorem 9 (Matrix M and eigenvalue property). *Matrix $M = C + s(1/n)e'$ has a interesting property: $M \cdot e = 1 \cdot e$, where e is the all-one vector. This is because all rows of M add up to 1. Thus $(e, 1)$ is an eigenvector and eigenvalue pair of M : we might call it an eigenpair. Moreover 1 is the largest eigenvalue of M .*

Theorem 10 (Google matrix G from M or C). *Then let*

$$G = dM + \frac{1-d}{n} e e' = d(C + s(1/n)e e') + \frac{1-d}{n} e e' = d \cdot C + (ds + (1-d)e) \frac{1}{n} e'$$

Example 9 (Example of Figure 14 continued). Then for the example in hand,

$$M = \begin{bmatrix} 0 & 1/3 & 1/3 & 1/3 \\ 1/3 & 0 & 1/3 & 1/3 \\ 0 & 1/2 & 0 & 1/2 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{bmatrix}$$

Theorem 11 (Large eigenvalues of M , G , eigenpair $(e, 1)$ and λ_2). *In M and G the elements of any row add up to 1. Thus for G the $(e, 1)$ is still an eigenpair and thus 1 is the largest eigenvalue of both. All eigenvalues (other than the largest one which is one) are smaller than 1 (or in general their magnitude is smaller than 1) because the sum of the rows is 1. The difference between G and M is that in G the second largest eigenvalue $\lambda_2(G)$ is guaranteed to be less than d . Thus $\lambda_2(G) \leq d$. Thus the $|\lambda_1(G) - \lambda_2(G)| = |1 - \lambda_2(G)| \leq 0.15$. The distance 0.15 (in fact, 0.85) determines how fast the algorithm on the next page converges!*

PageRank

Math background (may skip)

Theorem 12 (G and G' , M and M'). *The two matrices G and G' have the same eigenvalues. This is a fact from linear algebra! So do M and M' .*

It is one thing though to have the same eigenvalues and another thing to also have the same eigenvectors.

Theorem 13 (Rank vector r of G). *For G the rank vector r is a probability vector: its i -th element r_i denotes the probability of visiting page i . (Note that the subscript i is not in parentheses!) Moreover,*

$$r_1 + r_2 + \dots + r_n = \sum r_i = 1.$$

By linear algebra/probability theory (Markov chains) r is the solution of $r' = r'G$. That is, r is the dominant "left" eigenvector of G i.e. the "left" eigenvector that corresponds to the largest eigenvalue of it. In $1 \cdot r' = r'G$, the r' is on the "left" of G , that's why it is a "left eigenvector"; the left hand side implies an eigenvalue of 1, as noted. Since this is the largest eigenvalue, it is the dominant or principal eigenvalue and its eigenvector is the dominant or principal eigenvector as well.) By a prior theorem G and G' have the same eigenvalues and the largest eigenvalue of either is equal to 1. If we transpose both sides of $r' = r'G$ we get that $r = G'r$. That is the solution r is the "right eigenvector" of G' as it gets multiplied with it from the right. Likewise its corresponding eigenvalue is 1. We drop the right from "right eigenvector" to just say eigenvector. Given that G and G' have the same eigenvalues this does not affect their eigenvalues. In conclusion, to find the dominant "left" eigenvector of G we find the eigenvector corresponding to the dominant eigenvalue (1) of G' .

PageRank

Math background (may skip)

Method 6 (Computing r using the Power Method). We can use iterative methods such as the **power method** where

$$r'_{(i)} = r'_{(i-1)}G$$

The subscript indicates the iteration. That is why it is in parentheses. The solution at convergence will have an i -th element whose value would be r_i or an approximation to r_i . We move from iteration $i - 1$ to i by performing a multiplication with G as indicated above. The process stops when at some iteration k we have convergence i.e. the entries of the r vector remain the same to a given accuracy, i.e. $|r_{(k)} - r| < \text{error}$ or given that we do not know r , when $|r_{(k)} - r_{(k-1)}| < \text{error}$. The Google authors, Brin and Page, claimed that $k = 50$ works for most cases.

Corollary 1 (Initial values of PageRank). For the iterative process $r'_{(i)} = r'_{(i-1)}G$. What is $r_{(0)}$, the initial or boundary value? Any vector $r_{(0)}$ is fine as long as it is orthogonal to the solution r ! For example the unit vector e or e/n could work. A requirement for convergence implies the need for scaling! This would require the use of e/n over the unscaled e . A $e/\sqrt{n}$ will also work! Never try however an initial vector of zeroes!

Theorem 14 (Linear algebra and $(e, 1)$ eigenpair of G). *Eigenvalue 1 is the largest eigenvalue of non-negative matrix M (or G or M' or G'). Every other eigenvalue is smaller. Therefore $\lambda_1 = 1 > \lambda_2 > \dots$. Let the corresponding eigenvectors be $x_1, \dots, x_n$. From matrix theory we have that $G'x = \lambda x$ for eigenpair (x, λ) . Let us call in the remainder $A = G'$ to avoid problems with transposes. Thus $Ax = \lambda x$ given that A is an alias for G' . Therefore $A^2x = A(Ax) = A\lambda x = \lambda(Ax) = \lambda^2x$ and in general $A^kx = \lambda^kx$. Let the starting vector $r_{(0)}$ be a linear combination of the eigenvectors i.e. $r_{(0)} = x_1 + a_1x_2 + \dots + a_nx_n$. Then using the fact that $A^kx_i = \lambda_i^kx_i$ we obtain the following.*

PageRank

Math background (may skip)

Proof.

$$\begin{aligned} r_{(0)} &= x_1 + a_1 x_2 + \dots + a_n x_n \\ r_{(1)} = Ar_{(0)} &= Ax_1 + Aa_2 x_2 + \dots + Aa_n x_n \\ &= 1x_1 + \dots + a_n \lambda_n x_n \\ r_{(k)} = Ar_{(k-1)} &= 1x_1 + a_2 \lambda_2^k x_2 + \dots + a_n \lambda_n^k x_n \end{aligned} \tag{5}$$

Therefore

$$r_{(k)} = x_1 (1 + a_2 (\lambda_2/1)^k x_2 + \dots + a_n (\lambda_n/1)^k x_n) \tag{6}$$

From the latter we obtain that $r_{(k)}$ converges to x_1 as long as $\lambda_2^k, \dots, \lambda_n^k \rightarrow 0$.

Given that $\lambda_1 = 1 > \lambda_2 > \dots > \lambda_n$, this is indeed the case.

Moreover, the rate of convergence depends on λ_2^k .

■

Corollary 2 (Rate of convergence of PageRank). The rate of converge of PageRank is λ_2^k after k iterations. Given that $\lambda_2(G) = \lambda_2(G') \leq d = 0.85$, this implies indeed that the rate of convergence is 0.85^k . The "Given" part of the claim of the previous paragraph has not been proven yet! Its proof appears next. Note that $0.85^{50} \approx 0.00029$ and $0.85^{60} \approx 0.000058$.

Theorem 15. If matrix M has eigenvalues $\langle 1 \geq \lambda_2 \geq \dots \geq \lambda_n \rangle$, then matrix G and also G' has eigenvalues $\langle 1 \geq d\lambda_2 \geq \dots \geq d\lambda_n \rangle$. The two matrices G, M are defined as follows: $G = dM + \frac{1-d}{n} e e'$.

In the remainder for clarity we use superscript t instead of $'$ to denote the transpose operation.

Proof. Consider $G = ((1-d)/n)ee' + dM$. For M , an eigenpair is $(e, 1)$ i.e. $Me = e$. Consider matrix Q formed by the eigenvectors of M listed so that e appears first, i.e. $Q = (e \ X)$ then $Q^{-1} = \begin{pmatrix} z' \\ Z' \end{pmatrix}$. Since $Q^{-1}Q = I$ we obtain that

$$Q^{-1}Q = \begin{pmatrix} z'e & z'X \\ Z'e & Z'X \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & I \end{pmatrix}. \text{ From the last equality we conclude that}$$

$$z'X = 0, \quad z'e = 1, \quad Z'e = 0, \quad Z'X = I \quad (7)$$

Consider now the product $Q^{-1}MQ$. We have

$$Q^{-1}MQ = \begin{pmatrix} z' \\ Z' \end{pmatrix} \cdot (Me \ MX) = \begin{pmatrix} z'Me & z'MX \\ Z'Me & Z'MX \end{pmatrix} = \begin{pmatrix} 1 & z'MX \\ 0 & Z'MX \end{pmatrix} \quad (8)$$

The last equality was derived because $Me = e$ implies $z'Me = z'e = 1$ and the fact that $z'e = 1$ by Eq. (7). In addition $Z'Me = 0$ by $Z'e = 0$ again from Eq. (7).

$$Q^{-1}GQ = Q^{-1}(dM + ((1-d)/n)ee')Q = dQ^{-1}MQ + (1-d)/n Q^{-1}ee'Q \quad (9)$$

$$= \begin{pmatrix} d & dz'MX \\ 0 & dZ'MX \end{pmatrix} + \frac{(1-d)}{N} \cdot \begin{pmatrix} z'e \\ Z'e \end{pmatrix} \cdot (e'e \ e'X). \quad (10)$$

The last two matrices are the expanded form of $Q^{-1}e$ and $e'Q$ respectively. Therefore we get after observing again from Eq. (7) that $z'e = 1, Z'e = 0$.

Continued.

$$\begin{aligned}
 Q^{-1}GQ &= \begin{pmatrix} d & dz'MX \\ 0 & dZ'MX \end{pmatrix} + \frac{(1-d)}{N} \cdot \begin{pmatrix} z'e(e'e) & z'e(e'X) \\ Z'e(e'e) & Z'e(e'X) \end{pmatrix} = \begin{pmatrix} d & dz'MX \\ 0 & dZ'MX \end{pmatrix} + \frac{(1-d)}{N} \cdot \begin{pmatrix} (e'e) & (e'X) \\ 0 & 0 \end{pmatrix} \\
 &= \begin{pmatrix} d + ((1-d)/N) * N & dz'MX + ((1-d)/N)e'X \\ 0 & dZ'MX \end{pmatrix} = \begin{pmatrix} 1 & dz'MX + ((1-d)/N)e'X \\ 0 & dZ'MX \end{pmatrix}
 \end{aligned}$$

since $Z'X = I$ from Eq. (7), and from the rightmost side of Eq. (8) we have $Z'MX$ has eigenvalues the vector $(\lambda_2, \dots, \lambda_n)$, and thus we can conclude that $dZ'MX$ will have as eigenvalues the d multiples of them, i.e. $(d\lambda_2, \dots, d\lambda_n)$. ■

DRAFT

Notice

Problem Set with Solutions

These Problem Sets are not for credit. They are provided as are for your practice. Several problems are provided with solutions. These can be discussed with the instructor after a student has studied the relevant material. Some other problems are provided with either no solutions at all or a hint of a solution. They are there for your reference and will not be discussed further. For some solutions provided you might have to fill in details related to calculations or intermediate steps.

Before you try these problems: (a) Study the designated textbook, (b) use the Subject notes (typewritten notes available through the course web-page from the password-protected section C) to review your study of the designated textbook material, (c) resolve the examples done in class, (d) then practice with some of these problems PS (Problem Set), and then if you have time, (e) solve some of the problems of the designated textbook, and (f) solve more problems from the PS especially the ones with minimal solution information.

Several PS problems appeared in past exams or homeworks.

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 1.

A wolf, a goat, and a cabbage are on one bank of a river. A boatman will take them across, but can take only one at a time. The wolf and goat cannot be left together at either bank of the river without the boatman, nor can the goat and the cabbage. Model this problem with a graph, and then based on this modeling determine the shortest time required for the boatman to achieve his task, without any loss of life for its passengers. **Note: Your solution is an acceptable one if it uses graph modeling, and is based on this modeling.**

Proof. Draw a graph, solve the problem. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 2.

If every vertex in a undirected graph has degree at least $(n - 1)/2$ then the graph is connected.

Proof. Pick x, y with degree $(n - 1)/2$ or more. Then x, y either they are adjacent and we are done. Or they have a common neighbor since their degrees add up to $((n - 1)/2 + (n - 1)/2) = n - 1$ and there are only $n - 2$ other vertices. One of them must be common to x, y . If there is a common neighbor, then x, y are connected through it. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 3.

You are given a directed graph $G = (V, E)$ represented through its adjacency matrix. Thus you only have n and the adjacency matrix A which $n \times n$. You are given m . Provide running time (tight asymptotic bounds) for each one of the following problems/questions.

1. Does edge (u, v) exist?
2. $\text{Outd}(u)$?
3. $\text{Ind}(u)$?
4. $\text{Deg}(u)$?
5. Find number of edges of G . (Saying it is m is not enough. You have got to compute m itself!)

Proof. 1. $\Theta(1)$. 2. $\Theta(n)$, 3. $\Theta(n)$. 4. $\Theta(n)$ 5. $\Theta(n^2)$.

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's webpage

Problem Set

Problem 4.

You are given a directed graph $G = (V, E)$ represented through its adjacency list. Thus you only have n and the adjacency array A which is an array/vector of length n . In $A[i]$ we can retrieve the heads of linked list $A(i)$ that points to the adjacency list of vertex i . Provide running time (tight asymptotic bounds) for each one of the following problems/questions.

1. Does edge (u, v) exist?
2. $\text{Outd}(u)$?
3. Find number of edges of G . (Saying it is m is not enough. You have got to compute m .)

Proof. 1. $O(\text{outd}(u))$. $\text{outd}(u)$ is the length of linked list $A(u)$.

2. $\Theta(\text{outd}(u))$. Cute isn't it? To avoid this some optimized implementation store in $A[i]$ the length of the linked list.

3. $\Theta(n + m)$. Cuter isn't it? The time to find m is proportional to m and n . Traverse all linked lists $A(i)$ for all i . Do not forget $\sum_{u \in V} \text{outd}(u) = \Theta(m)$. ■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web page

Problem 5.

You are given an undirected graph G represented through its adjacency list. Based on Subject 3, How long does it take to answer the following questions? How?

1. Find the number c of connected components of G .
2. Is G connected? (i.e. is $c = 1$?)
3. For input u, v in addition to G does there exist a path from u to v ?
4. Modify existing code to collect this info. (Eg 3.40 of Subject 3.)

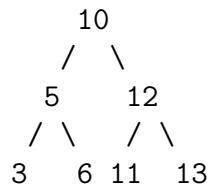
Proof. 1. Union-Find $O(m + n \lg n)$.
2. If from 1, $c == 1$ done in $\Theta(1)$ additional time.
3. If 1 is done already, $\Theta(1)$ additional time.
4. Practice makes perfect!

■

Copyright (c) 2020-2024. Alex. Gerbessiotis. All rights reserved.
Not to be displayed outside of copyright holder's web-page

Problem Set
Problem 6.

In the tree shown perform



(a) inorder, (b) preorder, (c) postorder traversals. Show your results.

Proof. (a) An inorder traversal yields the sequence: 3 5 6 10 11 12 13.

(b) A preorder traversal yields the sequence: 10 5 3 6 12 11 13.

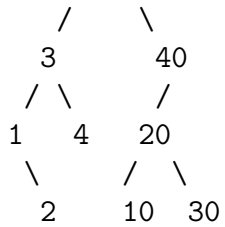
(c) A postorder traversal yields the sequence: 3 6 5 11 13 12 10.

■

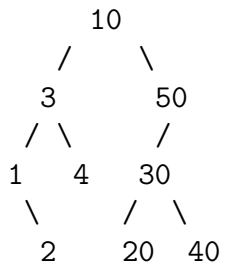
Problem Set
Problem 7.

What is the statement of the problem that has the following solution? No kidding!

Proof. 5



Preorder Traversal: 5 3 1 2 4 40 20 10 30



Postorder Traversal: 2 1 4 3 20 40 30 50 10

■