Data Structures – LECTURE 12

Graphs and basic search algorithms

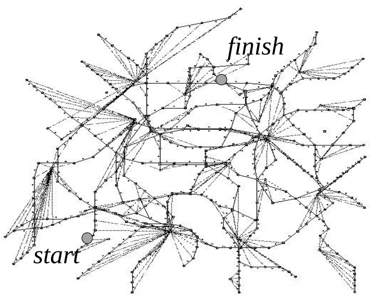
- Motivation
- Definitions and properties
- Representation
- Breadth-First Search
- Depth-First Search

Chapter 22 in the textbook (pp 221—252).

Motivation

- Many situations can be described as a binary relation between objects:
  – Web pages and their accessibility
  – Roadmaps and plans
  – Transition diagrams
- A *graph* is an abstract structure that describes a binary relation between elements. It is a generalization of a *tree*.
- Many problems can be reduced to solving graph problems: shortest path, connected components, minimum spanning tree, etc.

Example: finding your way in the Metro



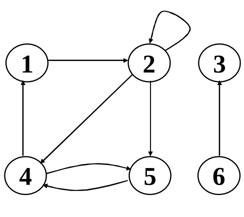
- Stations are vertices (nodes)
- Line segments are edges.
- Shortest path = shortest distance, time.
- Reachable stations.

Graph (גרפים): definition

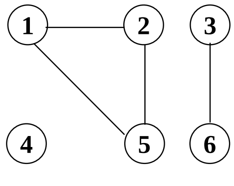
- A *graph* $G = (V,E)$ is a pair, where $V = \{v_1, .. v_n\}$ is the *vertex set* (nodes) and $E = \{e_1, .. e_m\}$ is the *edge set.* An edge $e_k = (v_i ,v_j)$ connects (is incident to) two vertices v_i and v_j of V.
- Edges can be undirected or directed (unordered or odered): $e_{ij}: v_i — v_j$ or $e_{ij}: v_i —> v_j$
- The graph G is finite when $|V|$ and $|E|$ are finite.
- The size of graph G is $|G| = |V| + |E|$.

Graphs: examples

Let $V = \{1,2,3,4,5,6\}$



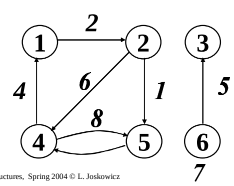
Directed graph



Undirected graph

Weighted graphs

- A *weighted graph* is graph in which edges have *weights* (*costs*) $c(v_{i,} v_j) > 0$.
- A graph is a weighted graph in which all costs are 1. Two vertices with no edge (path) between them can be thought of having an edge (path) with weight ∞.



The cost of a path is the sum of the costs of its edges:

Directed graphs

- In a directed graph, we say that an edge $e = (u,v)$ leaves u and enters v (v is adjacent, a neighbor of u).
- Self-loops are allowed: an edge can leave and enter u.
- The *in-degree* $d_{in}(v)$ of a vertex v is the number of edges entering v. The *out-degree* $d_{out}(v)$ of a vertex v is the number of edges leaving v. $\Sigma d_{in}(v_i) = \Sigma d_{out}(v_i)$
- A *path* from u to v in $G = (V,E)$ of length k is a sequence of vertices $<u = v_0, v_1, \ldots, v_k = v>$ such that for every i in $[1,\ldots,k]$ the pair (v_{i-1},v_i) is in E.

Undirected graphs

- In an undirected graph, we say that an edge $e = (u,v)$ is incident on u and v.
- Undirected graphs have no self-loops.
- Incidency is a symmetric relation: if $e = (u,v)$ then u is a neighbor of v and v is a neighbor of u.
- The degree of a vertex $d(v)$ is the total number of edges incident on v. $\Sigma d(v_i) = 2|E|$.
- Path: as for directed graphs.

Graphs terminology

- A *cycle* (*circuit*) is a path from a vertex to itself of length ≥ 1
- A *connected graph* is an undirected graph in which there is a path between any two vertices (every vertex is *reachable* from every other vertex).
- A *strongly-connected graph* is a directed graph in which for any two vertices u and v there is a directed path from u to v and from v to u.
- A graph $G' = (V',E')$ is a *sub-graph* of $G = (V,E)$, $G' \subseteq G$ when $V' \subseteq V$ and $E' \subseteq E$.
- The (*strongly*) *connected components* G_1, G_2, … of a graph G are the largest (*strongly*) connected sub-graphs of G.

Size of graphs

- There are at most $|E| = O(|V|^2)$ edges in a graph.
  Proof: each node can be in at most $|V|$ edges.
  A graph in which $|E| = |V|^2$ is called a *clique*.
- There are at least $|E| \geq |V|-1$ edges in a connected graph.
  Proof: By induction on the size of V.
- A graph is *planar* if it can be drawn in the plane with no two edges crossing. In a planar graph, $|E| = O(|V|)$. The smallest non-planar graph has 5 vertices.

Trees and graphs

- A tree is a connected graph with no cycles.
- A tree has $|E| = |V|-1$ edges.
- The following four conditions are equivalent:
  1. G is a tree.
  2. G has no cycles; adding a new edge forms a cycle.
  3. G is connected; deleting any edge destroys its connectivity.
  4. G has no self-loops and there is a path between any two vertices.
- Similar definitions for a directed tree.

Graphs representation

Two standard ways of representing graphs:

1. Adjacency list: for each vertex v there is a linked list L_v of its neighbors in the graph. Size of the representation: $\Theta(|V|+|E|)$.
2. Adjacency matrix: a $|V| \times |V|$ matrix in which an edge $e = (u,v)$ is represented by a non-zero (u,v) entry. Size of the representation: $\Theta(|V|^2)$.

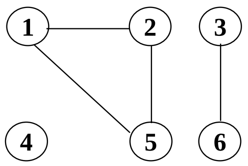
Adjacency lists are better for sparse graphs.

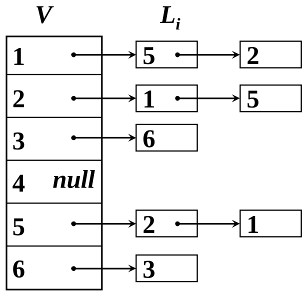
Adjacency matrices are better for dense graphs.

Example: adjacency list representation

$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(1,5),(2,5),(3,6)\}$

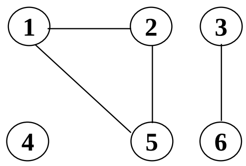




Example: adjacency matrix representation

$V = \{1,2,3,4,5,6\}$

$E = \{(1,2),(1,5),(2,5),(3,6)\}$



A

	1	2	3	4	5	6
1		**1**			**1**	
2	**1**				**1**	
3						**1**
4						
5	**1**	**1**				
6			**1**			

For undirected graphs, $A = A^T$

Graph problems and algorithms

- Graph traversal algorithms
  - Breath-First Search (BFS)
  - Depth-First Search (DFS)
- Minimum spanning trees (MST)
- Shortest-path algorithms
  - Single path
  - Single source shortest path
  - All-pairs shortest path
  - Strongly connected components
- Other problems: planarity testing, graph isomorphism

Shortest path problems

There are three main types of shortest path problems:

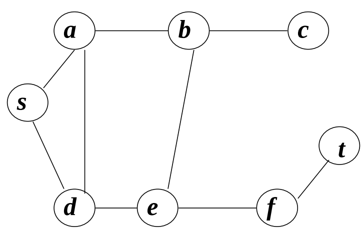
1. Single path: given two vertices, *s* and *t*, find the shortest path from *s* to *t* and its length (distance).
2. Single source: given a vertex *s*, find the shortest paths to all other vertices.
3. All pairs: find the shortest path from all pairs of vertices (*s*, *t*).

We will concentrate on the single source problem since 1. ends up solving this problem anyway, and 3. can be solved by applying 2. $|V|$ times.

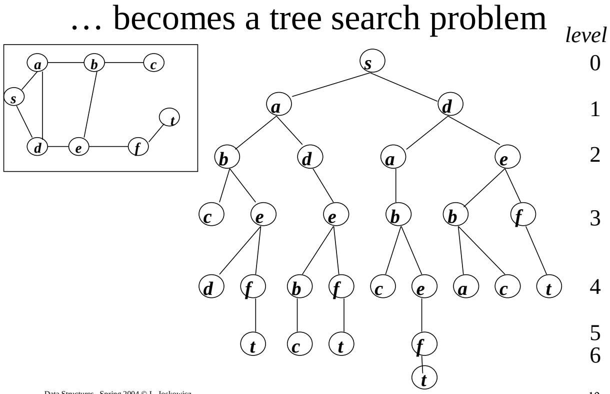
Intuition: how to search a graph

- Start at the vertex *s* and label its level at 0.
- If *t* is a neighbor of *s*, stop. Otherwise, mark the neighbors of *s* as having level 1.
- If *t* is a neighbor of a vertex at level *i*, stop. Otherwise, mark the neighbors of vertices at level *i* as having level *i*+1.
- When *t* is found, trace the path back by going to vertices at level *i*, *i* –1, *i* –2, …0.
- The graph becomes in effect a shortest-path neighbor tree!

Example: a graph search problem...



… becomes a tree search problem

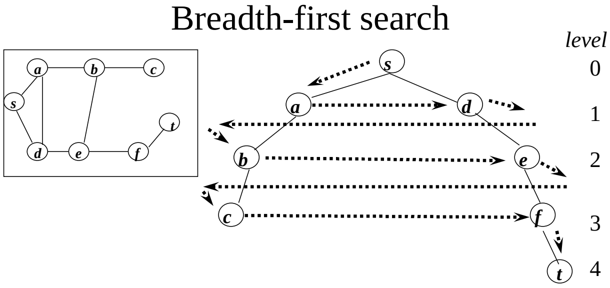


How is the tree searched?

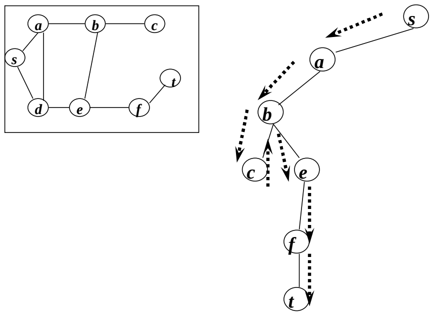
The tree can be searched in two ways:

- Breadth: search all vertices at level *i* before moving to level *i*+1 → Breadth-First Search (BFS).
- Depth: follow the vertex adjacencies, searching a node at each level *i* and backing up for alternative neighbor choices → Depth-First Search (DFS).

Breadth-first search



Depth-first search



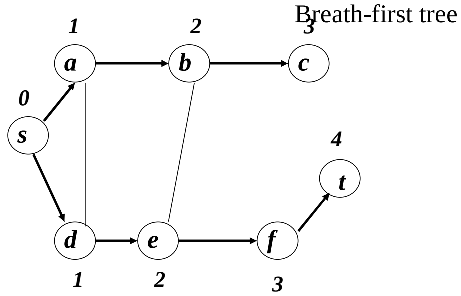
The BFS algorithm: overview

- Search the graph by successive levels (expansion wave) starting at *s*.
- Distinguish between three types of vertices:
  - *visited*: the vertex and all its neighbors have been visited.
  - *current*: the vertex is at the frontier of the wave.
  - *not_visited*: the vertex has not been reached yet.
- Keep three additional fields per vertex:
  - the type of vertex *label*[*u*]: *visited, current, not_visited*
  - the distance from the source *s*, *dist*[*u*]
  - the predecessor of *u* in the search tree, $\pi[u]$.
- The *current* vertices are stored in a queue *Q*.

The BFS algorithm

BFS(*G*, *s*)
label[*s*] ← *current; dist*[*s*] = 0; $\pi[s]$ = ***null***
for all vertices *u* in *V* – {*s*} **do**
  label[*u*] ← *not_visited; dist*[*u*] = ∞; $\pi[u]$ = ***null***
EnQueue(*Q*,*s*)
while *Q* is not empty **do**
  u ← DeQueue(*Q*)
  for each *v* that is a neighbor of *u* **do**
    if *label*[*v*] = *not_visited* **then** *label*[*v*] ← *current*
    dist[*v*] ← *dist*[*u*] + 1; $\pi[v]$ ← *u*
    EnQueue(*Q*,*v*)
  label[*u*] ← *visited*

Example: BFS algorithm



BFS characteristics

- *Q* contains only current vertices.
- Once a vertex becomes *current* or *visited*, it is never labeled again *not_visited*.
- Once all the neighbors of a *current* vertex have been considered, the vertex becomes *visited*.
- The algorithm can be easily modified to stop when a target *t* is found, or report that no path exists.
- The BSF algorithm builds a *predecessor sub-graph*, which is a *breath-first tree:* $G_\pi = (V_\pi, E_\pi)$
  $V_\pi = \{v \in V: \pi[v] \neq \textbf{\textit{null}}\} \cup \{s\}$ and $E_\pi = \{(\pi[v], v),\ v \in V - \{s\}\}$

Complexity of BFS

- The algorithm removes each vertex from the queue only once. There are thus $|V|$ DeQueue operations.
- For each vertex, the algorithm goes over all its neighbors and performs a constant number of operations. The amount of work per vertex in the **if** part of the **while** loop is a constant times the number of outgoing edges.
- The total number of operations (**if** part) for all vertices is a constant times the total number of edges $|E|$.
- Overall: $O(|V|) + O(|E|) = O(|V|+|E|)$, at most $O(|V|^2)$

BFS correctness (1)

- Define $\delta(s,u)$ to be the *shortest distance* from *s* to *u* (the minimum number of edges).
  $\delta(u,u) = 0$ and $\delta(s,u) = \infty$ when there is no path.
- Let $G = (V,E)$ be a graph (directed or undirected) and $s \in V$.

Lemma 1: For every edge $(u,v) \in E$,

$$\delta(s,v) \leq \delta(s,u) + 1$$

Proof: the shortest path from *s* to *v* cannot be longer than the shortest path from *s* to *u* plus edge (u,v). If *u* is not reachable from *s*, $\delta(s,u) = \infty$.

BFS correctness (2)

Lemma 2: Upon termination of BFS:

$$\forall v \in V,\ dist[v] \geq \delta(s,v).$$

Proof: By induction on the number of EnQueue operations. The hypothesis is that $\forall v \in V\ dist[v] \geq \delta(s,v)$

Basis: when *s* is first enqueued, $dist[s] = \delta(s,s) = 0$ and $dist[v] = \infty \geq \delta(s,v)\ \ \forall v \in V - \{s\}$.

Inductive step: let *v* be a *not_visited* vertex that is discovered during the search from *u*. By the inductive hypothesis, $dist[u] \geq \delta(s,u)$. After the assignment,

$$\begin{aligned} dist[v] &= dist[u] + 1 \\ &\geq \delta(s,u) + 1 \\ &\geq \delta(s,v) \quad \text{(the value is never changed again)} \end{aligned}$$

BFS correctness (3)

Lemma 3: Suppose that during the execution of BFS on the graph the queue *Q* contains vertices $<v_1, .. v_r>$ where v_1 and v_r are *Q*'s head and tail. Then:

$$dist[v_r] \leq dist[v_1] + 1 \text{ \underline{and} } dist[v_i] \leq dist[v_{i+1}] \text{ for } i=1,2,..,r-1$$

Proof: By induction on the number of queue operations.

Basis: When $Q = <s>$, $dist[s] \leq dist[s] + 1$ and $\leq dist[s]$.

Inductive step: lemma holds after enqueuing and dequeueing *v*.

Dequeuing: when v_1 is removed, v_2 becomes the new head. The inequalities $dist[v_1] \leq dist[v_2] \leq \ldots \leq dist[v_r]$ still hold.

BFS correctness (4)

Enqueuing: when v is enqueued, it becomes v_{r+1}. At this time, vertex u whose adjacency list is currently been scanned has been removed from Q. By the induction hypothesis, the new head v_1 has $dist[v_1] \geq dist[u]$. Thus,

$$dist[v_{r+1}] = dist[v] = dist[u] + 1 \leq dist[v_1] + 1$$

From the inductive hypothesis, $dist[v_r] \leq dist[u]$, and so

$$dist[v_r] \leq dist[u] + 1 = dist[v] = dist[v_{r+1}]$$

and the other inequalities remain unaffected.

Corollary: If vertex v_i was enqueued before vertex v_j during BFS, then $dist[v_i] \leq dist[v_j]$ when v_j is enqueued.

BFS correctness (4)

Theorem: During its execution, BFS discovers every vertex $v \in V$ that is reachable from s. Upon termination, $dist[v] = \delta(s,v)$. For all reachable vertices v except for s, one of the shortest paths from s to v is a shortest path from s to $\pi[v]$ followed by the edge $(\pi[v],v)$.

Proof outline:

by contradiction, assume that there is a vertex v that receives a distance value such that $dist[v] > \delta(s,v)$.

– Clearly, v cannot be s.

BFS correctness (5)

– Vertex v must be reachable from s for otherwise $\delta(s,v) = \infty \geq dist[v]$ and thus $dist[v] \geq \delta(s,v)$.

– Let u be the vertex immediately preceding v on a shortest path from s to v so that $\delta(s,v) = \delta(s,u) + 1$. Because $\delta(s,u) < \delta(s,v)$, $dist[u] = \delta(s,u)$. Therefore:

$$dist[v] > \delta(s,v) = \delta(s,u) + 1 = dist[u] + 1$$

– Consider now the time when u is dequeued. Vertex v is either *not_visited*, *current*, or *visited*.

– Each case leads to a contradiction! Thus, $dist[v] = \delta(s,v)$

– In addition, if $\pi[v] = u$, then $dist[v] = dist[u] + 1$. Thus, we obtain the shortest path from s to v by taking a shortest path from s to $\pi[v]$ and then traversing the edge $(\pi[v],v)$.

The DFS algorithm: overview (1)

- Search the graph starting at s and proceed as deep as possible (*expansion path*) until no unexplored vertices remain. Then go back to the previous vertex and choose the next unvisited neighbor (*backtracking*). If any undiscovered vertices remain, select one of them as the source and repeat the process.
- Note that the result is a forest of *depth-first trees*:

  $G_\pi = (V,E_\pi) \quad E_\pi = \{(\pi[v],v),\ v \in V \textit{ and } \pi[v] \neq \textbf{\textit{null}}\}$

  where $\pi[v]$ is the predecessor of v in the search tree
- As for BFS, there are three three types of vertices: *visited*, *current*, and *not_visited.*

The DFS algorithm: overview (2)

- Two additional fields holding *timestamps.*
  - $d[u]$: timestamp when u is first discovered (u becomes *current*).
  - $f[u]$: timestamp when the neighbors of u have all been explored (u becomes visited).
- Timestamps are integers between 1 and $2|V|$, and for every vertex u, $d[u] < f[u]$.
- Backtracking is implemented with recursion.

The DFS algorithm

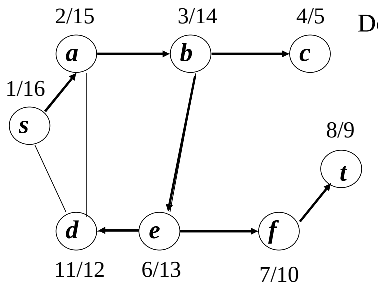
DFS(G, s)

label[*s*] ← *current; dist*[*s*] = 0*;* π[*s*] = ***null;*** *time* ← 0.
for each vertex *u* in **do**
  if *label*[*u*] = *not_visited* **then** DFS-Visit(*u*)

DFS-Visit(*u*)

label[*u*] = *current; time* ← *time* +1; *d*[*u*] ← *time*
  for each *v* that is a neighbor of *u* **do**
    if *label*[*v*] = *not_visited* **then**
      π[*v*] ← *u;* DFS-Visit(*v*)
  label[*u*] ← *visited*
  f[*u*] ← *time* ← *time* + 1

Example: DFS algorithm



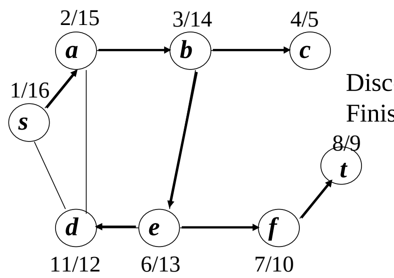
Depth-first tree

Time: discovery/finish

DFS characteristics

- The *depth-first forest* that results from DFS depends on the order in which the neighbors of a vertex are selected to deepen the search.
- The DFS program be easily modified to search only from start vertex s, and to find the shortest path from s to t.
- Instead of recursion, a LIFO queue can be used (instead of FIFO for BFS).
- The history of discovery and finish times, $d[v]$ and $f\,[v]$, has a *parenthesis structure.*

DFS: parenthesis structure (1)



Discovery: open (*push*
Finish: close) *pop*

(***s*** (***a*** (***b*** (***c c***) (***e*** (***f*** (***t t***) ***f***) (***d d***) ***e***) ***b***) ***a***) ***s***)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

DFS: parenthesis structure (2)

(***s*** (***a*** (***b*** (***c c***) (***e*** (***f*** (***t t***) ***f***) (***d d***) ***e***) ***b***) ***a***) ***s***)
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

(***s*** ——————————————— ***s***)
1 (***a*** ————————————— ***a***) 16
2 (***b*** ——————————— ***b***) 15
3 (***c***-***c***) (***e*** ————————— ***e***) 14
4 5 6 (***f*** —— ***f***) (***d***—***d***) 13
7 (***t***-***t***) 10 11 12
8 9

Complexity of DFS

- The algorithm visits every node $v \in V \rightarrow \Theta(|V|)$
- For each vertex, the algorithm goes over all its neighbors and performs a constant number of operations.
- Overall, DFS-Visit is called only once for each v in V, since the first thing that the procedure does it label v as *current.*
- In DFS-Visit, the recursive call is made for at most the number of edges incident to v:

$$\Sigma_{v \in V} |neighbors[v]| = \Theta(|E|)$$

- Overall: $\Theta(|V|) + \Theta(|E|) = \Theta(|V|+|E|)$, at most $\Theta(|V|^2)$
- Same complexity as BFS!

DFS correctness (1)

Theorem 1 (*parenthesis theorem*):

In any DFS of a graph $G = (V,E)$ for any two vertices u and v, exactly one of the next conditions hold:

1. The intervals $[d[u], f\,[u]]$ and $[d[v], f\,[v]]$ are entirely disjoint and neither u nor v is a descendant of each other.
2. The interval $[d[u], f\,[u]]$ is contained entirely within the interval $[d[v], f\,[v]]$ and u is a descendant of v.
3. The interval $[d[v], f\,[v]]$ is contained entirely within the interval $[d[u], f\,[u]]$ and v is a descendant of u.

DFS correctness (2)

Proof: Assume first that $d[u] < d[v]$. Then either

1. $d[v] < f[u]$ → v was discovered while u was still *current*. Therefore, v is a descendant of u. Also, since v was discovered more recently than u, all of its (outgoing) edges are explored and v is labeled *visited* before the search returns and finishes u → $[d[v], f[v]]$ is included in $[d[u], f[u]]$.
2. $f[u] \leq d[v]$. Since $d[u] < d[v]$ by definition, intervals $[d[v], f[v]]$ and $[d[u], f[u]]$ are entirely disjoint. Also, neither vertex was discovered while the other was *current*, so neither is a descendant of the other.

The proof for $d[v] < d[u]$ is symmetrical.

Corollary: v is a descendant of u iff $d[u] < d[v] < f[v] < f[u]$.

DFS correctness (3)

Theorem 2 (*visited path theorem*):

In a depth-first forest of graph $G = (V,E)$ vertex v is a descendant of u iff at the time $d[u]$ when the search discovers u, node v can be reached from u along a path consisting entirely of *not_visited* nodes.

Proof:

→ assume v is a descendant of u. Let w be a node on the path between u and v in the depth-first tree so that w is a descendant *of* u. By the previous corollary, $d[u] < d[w]$ and so w is *not_visited* at time $d[u]$.

DFS correctness (4)

←Suppose v is reachable from u along a path with visited vertices at time $d[u]$, but v does not become a descendant of u. Without loss of generality, assume that every other vertex along the path becomes a descendant of u. Let w be a predecesor of v and a descendant of u. Then $f[w] \leq f[v]$. Note that v must be discovered after u is discovered, but before w is finished. Therefore, $d[u] < d[v] < f[w] \leq f[u]$.

This implies that $[d[v], f[v]]$ is included in $[d[u], f[u]]$.

Therefore, v must be a descendant of u.

Classification of edges

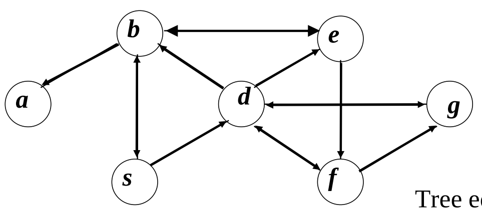
Edges in the depth-first forest

$G_\pi = (V,E_\pi)$ and $E_\pi = \{(\pi[v],v), v \in V \text{ and } \pi[v] \neq \textbf{null}\}$

can be classified into four categories:

1. *Tree edges*: depth-first forest edges in E_π
2. *Back edges*: edges (u,v) connecting a vertex u to an ancestor v in a depth-first tree (includes self-loops)
3. *Forward edges*: non-tree edges (u,v) connecting a vertex u to a descendant v in a depth-first tree.
4. *Cross edges*: all other edges. Go between vertices in the same depth-first tree without an ancestor relation between them.

Example: DFS edge classification



Tree edges
Back edges
Forward edges
Cross edges

DFS: classification of edges

Theorem 3: In a depth-first search of an undirected graph $G=(V,E)$, every edge in E is either a tree edge or a back edge.

Proof: Let (u,v) be an an edge in E, and suppose that $d[u] < d[v]$. Then v must be discovered and finished before u is finished (*current*) since v is on u's adjacency list. If the edge (u,v) is explored in the direction u → v, then u is *not_visited* until that time. If the edge is explored in the other direction, u ← v, then it is a back edge since u is still current at the time the edge is first explored.

Summary: Graphs, BFS, and DFS

- A graph is a useful representation for binary relations between elements. Many problems can be modeled as graphs, and solved with graph algorithms.
- Two ways of finding a path between a starting vertex *s* and all other vertices of a graph:
  - Breath-First Search (BFS): search all vertices at level i before moving to level i+1.
  - Depth-First search (DFS): follow vertex adjacencies, one vertex at each level i and backtracking for alternative neighbor choices.
- Complexity: linear in the size of the graph: $\Theta(|V|+|E|)$