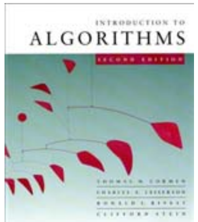
CS 5633 -- Spring 2008



Graphs

Carola Wenk

Slides courtesy of Charles Leiserson with changes and additions by Carola Wenk

Graphs (review)

Definition. A ***directed graph (digraph)*** $G = (V, E)$ is an ordered pair consisting of

- a set V of ***vertices*** (singular: ***vertex***),
- a set $E \subseteq V \times V$ of ***edges***.

In an ***undirected graph*** $G = (V, E)$, the edge set E consists of *unordered* pairs of vertices.

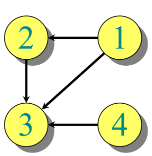
In either case, we have $|E| = O(|V|^2)$. Moreover, if G is connected, then $|E| \geq |V| - 1$.

(Review CLRS, Appendix B.4 and B.5.)

Adjacency-matrix representation

The ***adjacency matrix*** of a graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, is the matrix $A[1 \,..\, n, 1 \,..\, n]$ given by

$$A[i, j] = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{if } (i, j) \notin E. \end{cases}$$

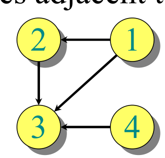


A	1	2	3	4
1	0	1	1	0
2	0	0	1	0
3	0	0	0	0
4	0	0	1	0

$\Theta(|V|^2)$ storage $\Rightarrow$ ***dense*** representation.

Adjacency-list representation

An ***adjacency list*** of a vertex $v \in V$ is the list $Adj[v]$ of vertices adjacent to v.



$Adj[1] = \{2, 3\}$
$Adj[2] = \{3\}$
$Adj[3] = \{\}$
$Adj[4] = \{3\}$

For undirected graphs, $|Adj[v]| = degree(v)$.

For digraphs, $|Adj[v]| = out\text{-}degree(v)$.

Adjacency-list representation

Handshaking Lemma:

Every edge is counted twice

- For undirected graphs:
  $\sum_{v \in V} degree(v) = 2|E|$
- For digraphs:
  $\sum_{v \in V} in\text{-}degree(v) + \sum_{v \in V} out\text{-}degree(v) = 2\,|\,E\,|$

$\Rightarrow$ adjacency lists use $\Theta(|V| + |E|)$ storage

$\Rightarrow$ a ***sparse*** representation

Graph Traversal

Let $G=(V,E)$ be a (directed or undirected) graph, given in adjacency list representation.

$|V| = n$, $|E| = m$

A graph traversal visits every vertex:

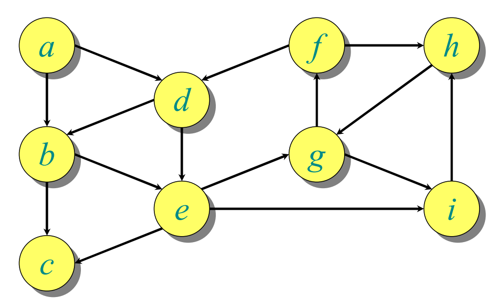
- Breadth-first search (BFS)
- Depth-first search (DFS)

Breadth-First Search (BFS)

```
BFS(G=(V,E))
    Mark all vertices in G as "unvisited" // time=0
    Initialize empty queue Q
    for each vertex v ∈ V do
        if v is unvisited
            visit v // time++
            Q.enqueue(v)
            BFS_iter(G)
```

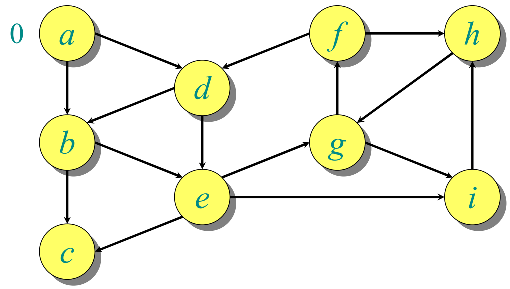
```
BFS_iter(G)
    while Q is non-empty do
        v = Q.dequeue()
        for each w adjacent to v do
            if w is unvisited
                visit w   // time++
                Add edge (v,w) to T
                Q.enqueue(w)
```


Example of breadth-first search



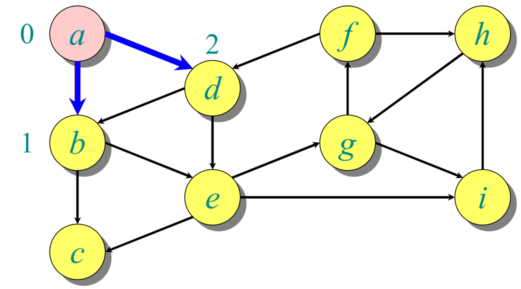
Q:

Example of breadth-first search



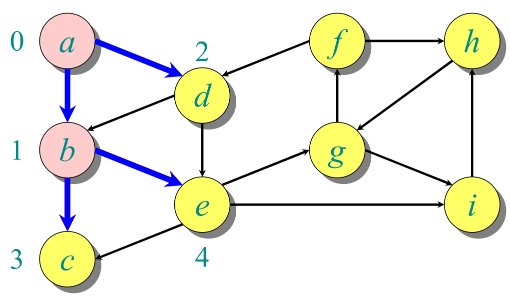
0
Q: a

Example of breadth-first search



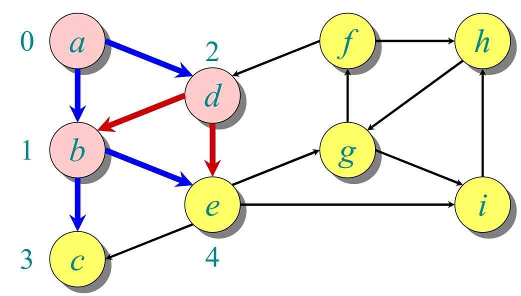
1 2
Q: a b d

Example of breadth-first search



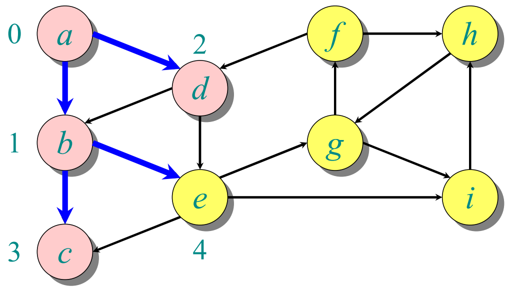
2 3 4
Q: a b d c e

Example of breadth-first search



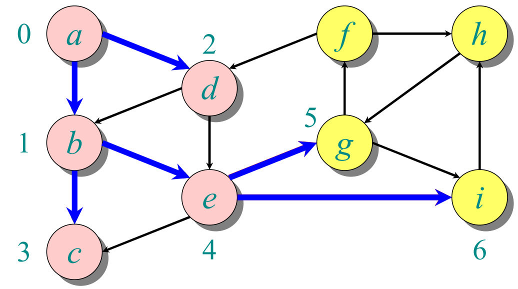
3 4
Q: a b d c e

Example of breadth-first search



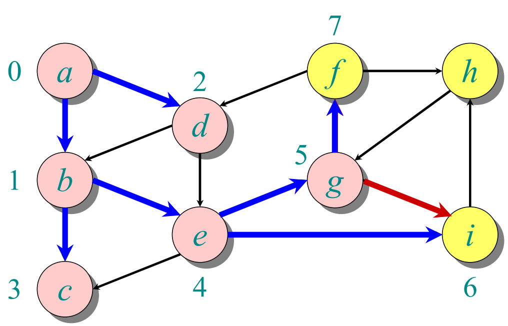
4
Q: a b d c e

Example of breadth-first search



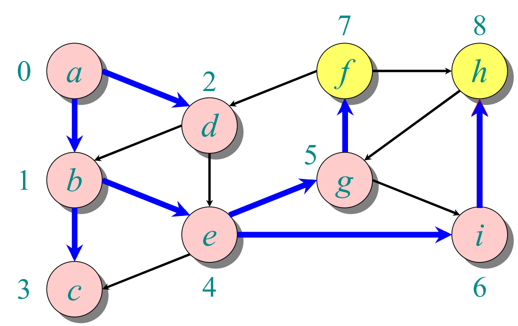
5 6
Q: a b d c e g i

Example of breadth-first search



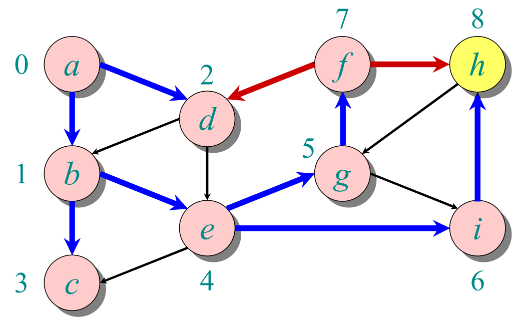
6 7
Q: a b d c e g i f

Example of breadth-first search



7 8
Q: a b d c e g i f h

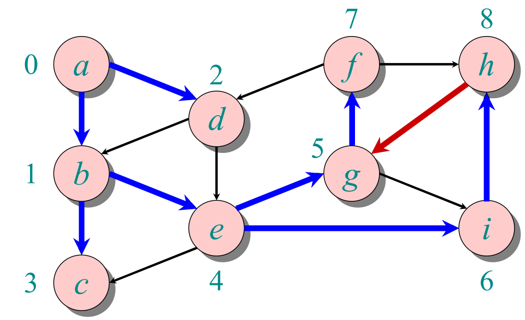
Example of breadth-first search



8

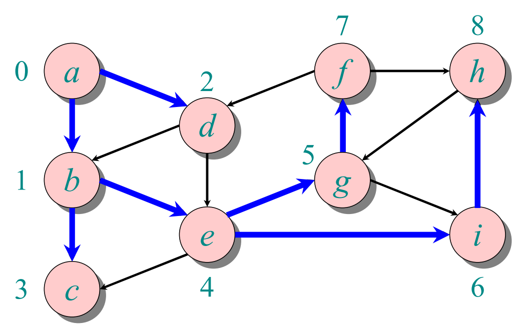
Q: *a b d c e g i f h*

Example of breadth-first search



Q: *a b d c e g i f h*

Example of breadth-first search



Q: *a b d c e g i f h*

Breadth-First Search (BFS)

```
        BFS(G=(V,E))
O(n)        Mark all vertices in G as "unvisited" // time=0
O(1)        Initialize empty queue Q
            for each vertex v ∈ V do
O(n)            if v is unvisited
without             visit v // time++
BFS_iter            Q.enqueue(v)
                    BFS_iter(G)

        BFS_iter(G)
            while Q is non-empty do
                v = Q.dequeue()
O(m)            for each w adjacent to v do
    O(deg(v))       if w is unvisited
                        visit w   // time++
                        Add edge (v,w) to T
                        Q.enqueue(w)
```

BFS runtime

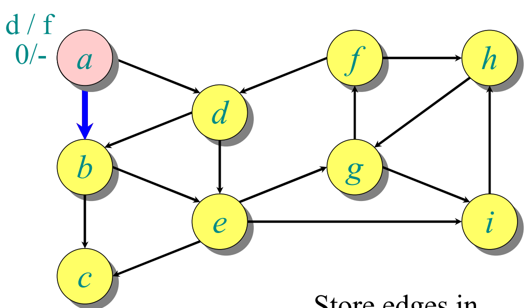
- Each vertex is marked as unvisited in the beginning ⇒ $O(n)$ time
- Each vertex is marked at most once, enqueued at most once, and therefore dequeued at most once
- The time to process a vertex is proportional to the size of its adjacency list (its degree), since the graph is given in adjacency list representation
  ⇒ $O(m)$ time
- Total runtime is $O(n+m) = O(|V| + |E|)$

Depth-First Search (DFS)

```
DFS(G=(V,E))
    Mark all vertices in G as "unvisited"  // time=0
    for each vertex v ∈ V do
        if v is unvisited
            DFS_rec(G,v)
```

```
DFS_rec(G, v)
    visit v  // d[v]=++time
    for each w adjacent to v do
        if w is unvisited
            Add edge (v,w) to tree T
            DFS_rec(G,w)
    // f[v]=++time
```

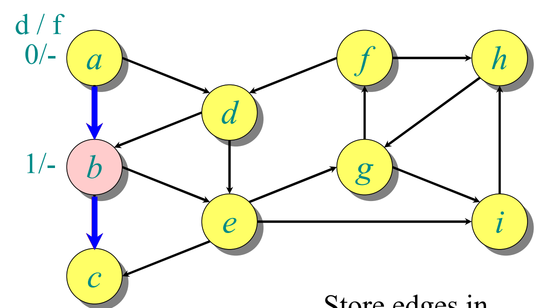

Example of depth-first search



π: a b c d e f g h i
- a

Store edges in predecessor array

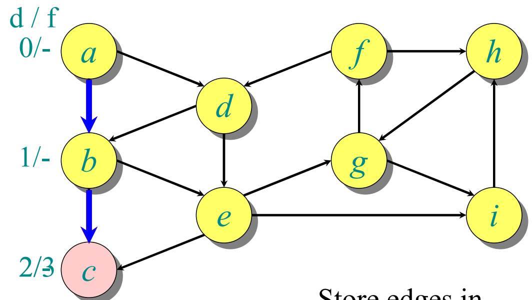
Example of depth-first search



π: a b c d e f g h i
- a b

Store edges in predecessor array

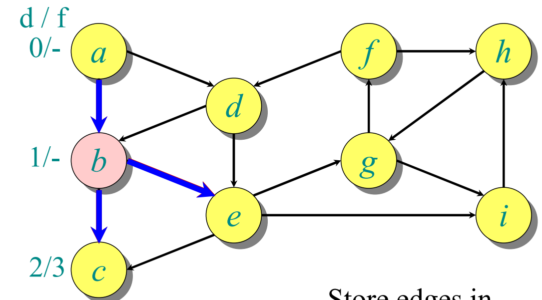
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b						

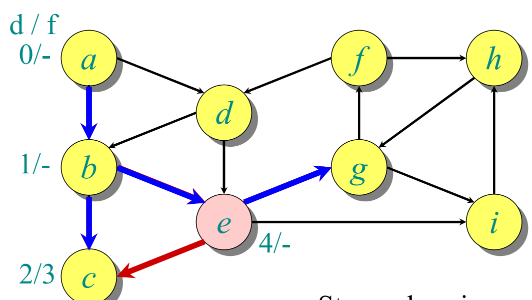
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b				

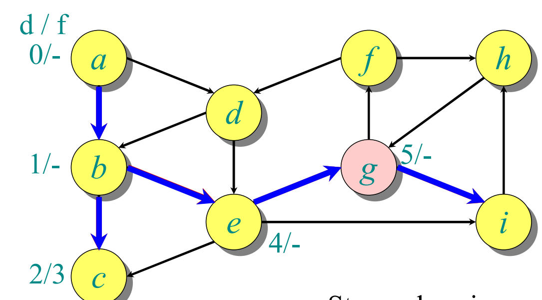
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b		e		

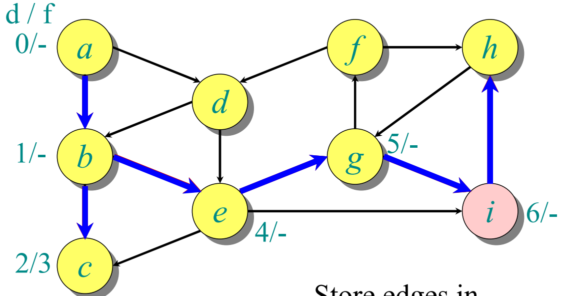
Example of depth-first search



Store edges in
predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b		e		g

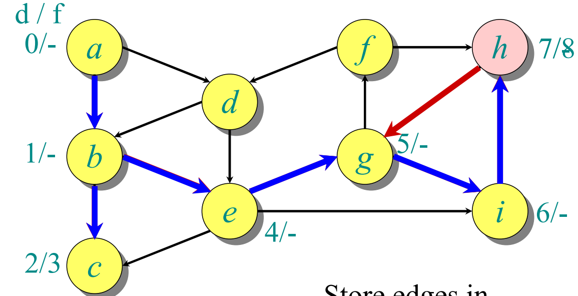
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b		e	i	g

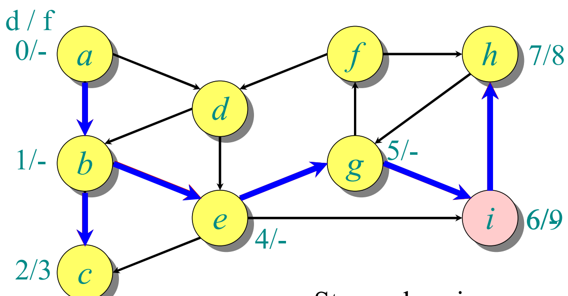
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b		e	i	g

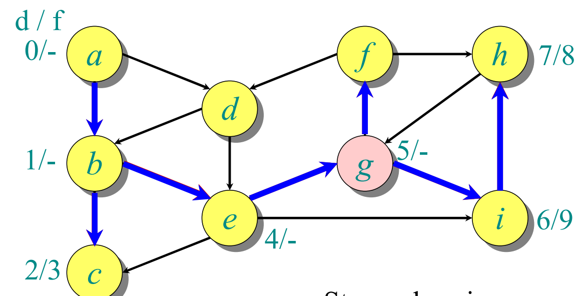
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b		e	i	g

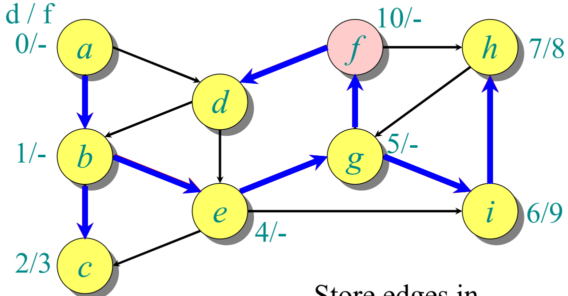
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b		b	g	e	i	g

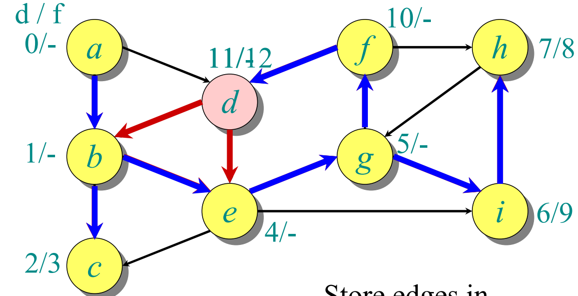
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	f	b	g	e	i	g

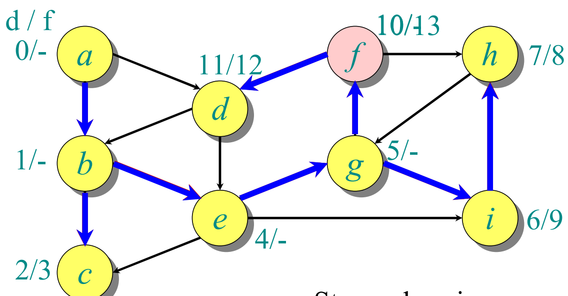
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	f	b	g	e	i	g

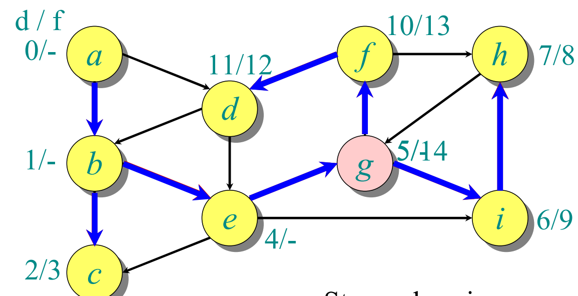
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	f	b	g	e	i	g

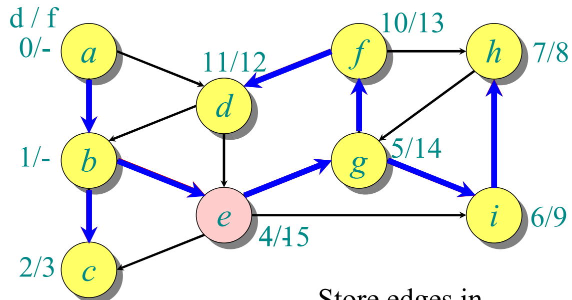
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	f	b	g	e	i	g

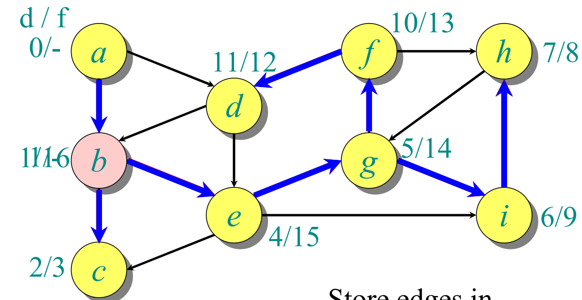
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	f	b	g	e	i	g

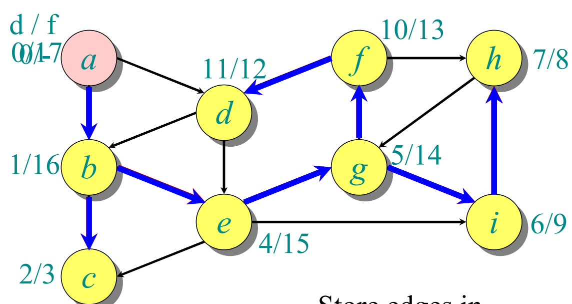
Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	f	b	g	e	i	g

Example of depth-first search



Store edges in predecessor array

π:	a	b	c	d	e	f	g	h	i
	-	a	b	f	b	g	e	i	g

Depth-First Search (DFS)

DFS($G=(V,E)$)
  Mark all vertices in G as "unvisited" // time=0 — $O(n)$
  for each vertex $v \in V$ **do**
    if v is unvisited
      DFS_rec(G,v)
— $O(n)$ without DFS_rec

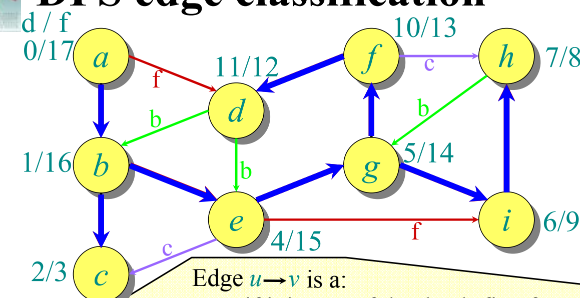
DFS_rec(G, v)
  visit v // $d[v]$=++time — $O(1)$
  for each w adjacent to v **do**
    if w is unvisited
      Add edge (v,w) to tree T
      DFS_rec(G,w)
  // $f[v]$=++time
— $O(deg(v))$ without recursive call

⇒ With Handshaking Lemma, all recursive calls are $O(m)$, for a total of $O(n + m)$ runtime

DFS runtime

- Each vertex is visited at most once ⇒ $O(n)$ time
- The body of the **for** loops (except the recursive call) take constant time per graph edge
- All **for** loops take $O(m)$ time
- Total runtime is $O(n+m) = O(|V| + |E|)$

DFS edge classification



Edge $u \rightarrow v$ is a:

- **tree edge**, if it is part of the depth-first forest.
- back edge, if u connects to an ancestor v in a depth-first tree. It holds $d(u)>d(v)$ and $f(u)<f(v)$.
- forward edge, if it connects u to a descendant v in a depth-first tree. It holds $d(u)<d(v)$.
- cross edge, if it is any other edge. It holds $d(u)>d(v)$ and $f(u)>f(v)$.