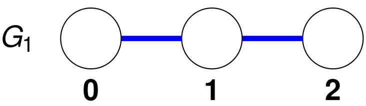
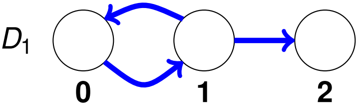
CS250: Discrete Math for Computer Science

L28: Searching Undirected Graphs: Depth First Search

Walks versus Paths

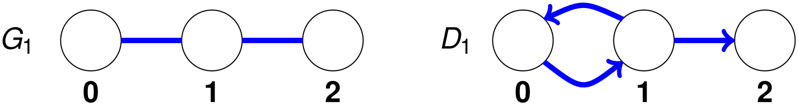
A **path** is a walk that **never visits the same edge or vertex twice**, **except** a path may **start and end** at the **same vertex** in which case it is called a **cycle**.





Walks versus Paths

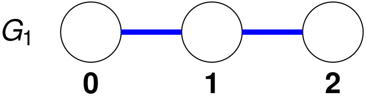
A **path** is a walk that **never visits the same edge or vertex twice**, **except** a path may **start and end** at the **same vertex** in which case it is called a **cycle**. A **loop** is a **cycle of length 1**. There are no cycles of length 0.



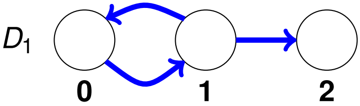
Walks versus Paths

A **path** is a walk that **never visits the same edge or vertex twice**, **except** a path may **start and end** at the **same vertex** in which case it is called a **cycle**. A **loop** is a **cycle of length 1**. There are no cycles of length 0.

Note: this definition differs for directed and undirected graphs. The undirected graph G_1 is acyclic.



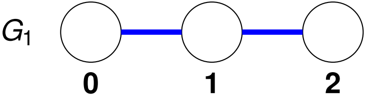
acyclic



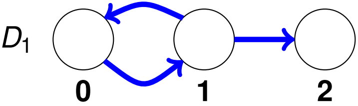
Walks versus Paths

A **path** is a walk that **never visits the same edge or vertex twice**, **except** a path may **start and end** at the **same vertex** in which case it is called a **cycle**. A **loop** is a **cycle of length 1**. There are no cycles of length 0.

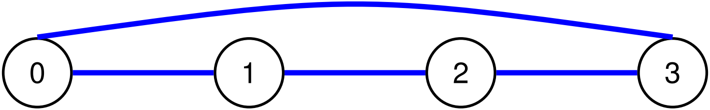
Note: this definition differs for directed and undirected graphs. The undirected graph G_1 is acyclic.
However, the directed graph, D_1, has a cycle: $(0, 1, 0)$.



acyclic

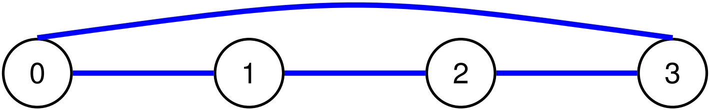


cyclic



$w_1 = (0)$ $w_2 = (0, 1)$ $w_3 = (0, 1, 2)$

$w_4 = (0, 1, 0)$ $w_5 = (0, 1, 2, 3)$ $w_6 = (0, 1, 2, 3, 0)$

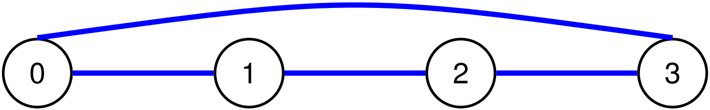


$w_1 = (0)$ $w_2 = (0, 1)$ $w_3 = (0, 1, 2)$

$w_4 = (0, 1, 0)$ $w_5 = (0, 1, 2, 3)$ $w_6 = (0, 1, 2, 3, 0)$

iClicker 28.1 In the above undirected graph, which of the above walks are paths?

A: all of them **B: all except w_4** **C: all except w_4 and w_6**



$w_1 = (0)$ $w_2 = (0, 1)$ $w_3 = (0, 1, 2)$

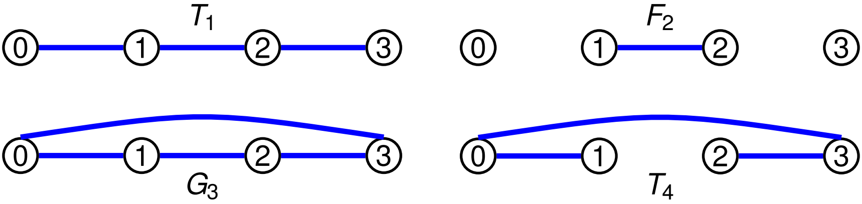
$w_4 = (0, 1, 0)$ $w_5 = (0, 1, 2, 3)$ $w_6 = (0, 1, 2, 3, 0)$

iClicker 28.2 Which of the above walks are cycles?

A: w_4 and w_6 **B: just w_6**

Cyclic versus Acyclic

A graph with at **at least one cycle** is called **cyclic**.

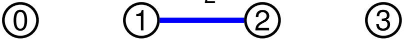


Cyclic versus Acyclic

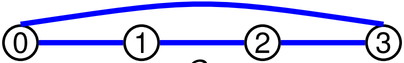
A graph with at **at least one cycle** is called **cyclic**.

A graph that has **no cycles** is called **acyclic**.

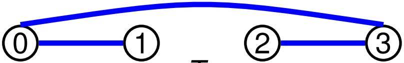
T_1



F_2



G_3

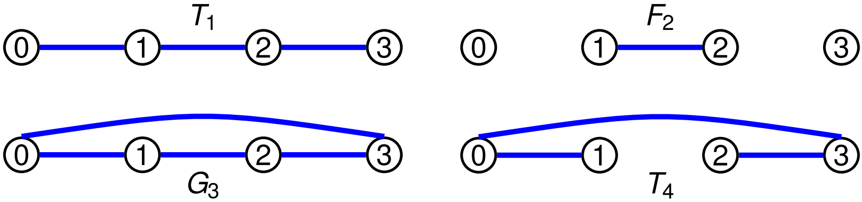


T_4

Cyclic versus Acyclic

A graph with at **at least one cycle** is called **cyclic**.

A graph that has **no cycles** is called **acyclic**.



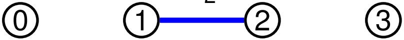
Rest of today: all graphs will be **undirected** and **loop free**.

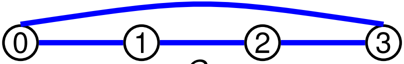
Cyclic versus Acyclic

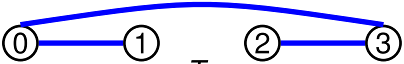
A graph with at **at least one cycle** is called **cyclic**.

A graph that has **no cycles** is called **acyclic**.


T_1


F_2


G_3


T_4

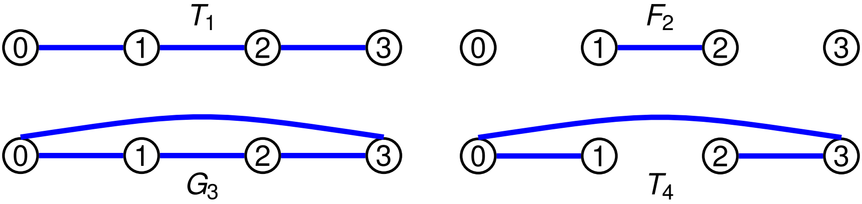
Rest of today: all graphs will be **undirected** and **loop free**.

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

Cyclic versus Acyclic

A graph with at **at least one cycle** is called **cyclic**.

A graph that has **no cycles** is called **acyclic**.



Rest of today: all graphs will be **undirected** and **loop free**.

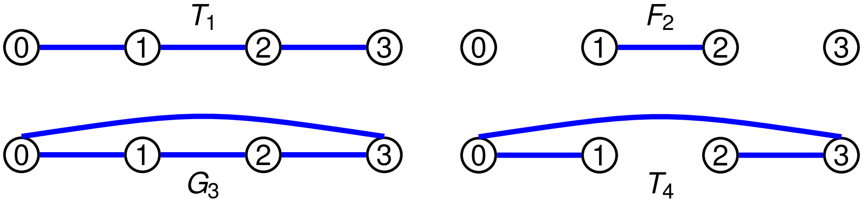
A **forest** is an **undirected acyclic graph**.

An undirected graph is **connected** if every pair of vertices has a path between them.

Cyclic versus Acyclic

A graph with at **at least one cycle** is called **cyclic**.

A graph that has **no cycles** is called **acyclic**.



Rest of today: all graphs will be **undirected** and **loop free**.

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

An undirected graph is **connected** if every pair of vertices has a path between them.

An undirected **tree** is a **connected forest**.

Connected Components

A **connected component**, C, of an undirected graph, G, is a **maximal connected** subgraph of G.

Connected Components

A **connected component**, C, of an undirected graph, G, is a **maximal connected** subgraph of G.

If G is **connected** then G itself is G's **unique connected component**.

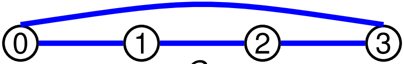
Connected Components

A **connected component**, C, of an undirected graph, G, is a **maximal connected** subgraph of G.

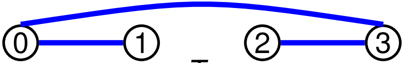
If G is **connected** then G itself is G's **unique connected component**.

T_1

F_2



G_3

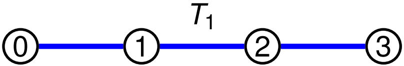


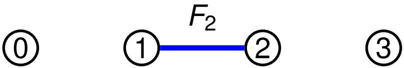
T_4

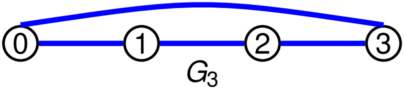
Connected Components

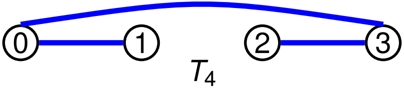
A **connected component**, C, of an undirected graph, G, is a **maximal connected** subgraph of G.

If G is **connected** then G itself is G's **unique connected component**.









iClicker 28.3 How many connected components does F_2 have? $V^{F_2} = \{0, 1, 2, 3\}$ $E^{F_2} = \{(1, 2), (2, 1)\}$

A: 1 **B:** 2 **C:** 3

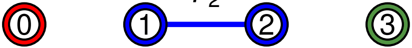
Connected Components

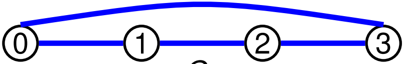
A **connected component**, C, of an undirected graph, G, is a **maximal connected** subgraph of G.

If G is **connected** then G itself is G's **unique connected component**.

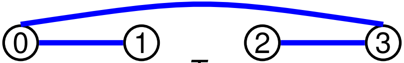
T_1

F_2





G_3

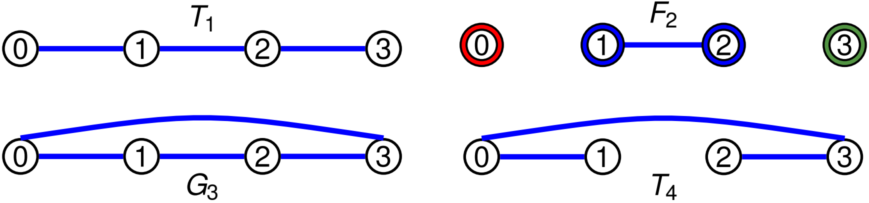


T_4

Connected Components

A **connected component**, C, of an undirected graph, G, is a **maximal connected** subgraph of G.

If G is **connected** then G itself is G's **unique connected component**.



Recall E^* is the reflexive transitive closure of E. For undirected graphs, E^* is an equivalence relation and it partitions the vertices into connected components:

$$[v]_{E^*} \;=\; \left\{ u \in V \;\middle|\; E^*(u,v) \right\}$$

Wanted: fast algorithm to

- **search** a **graph**

Wanted: fast algorithm to

- **search** a **graph**
- compute its **connected components**

Wanted: fast algorithm to

- **search** a **graph**
- compute its **connected components**
- determine if it is **cyclic** or **acyclic**

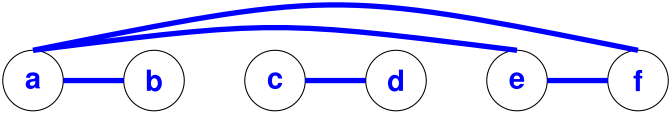
Depth First Search (undirected graphs)

```
DFSmain(G)
 for each u in V :
   color[u] = white
   parent[u] = NULL
 time=0
 for each u in V :
   if (color[u] == white):
     DFSVisit(u)
```

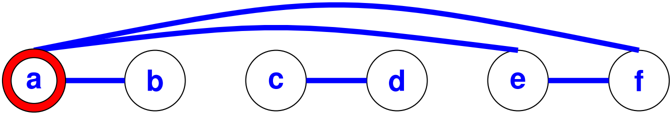
```
DFSVisit(u)
  color[u] = red                    // in process
  d[u] = ++time                     // discover
  for each v in Adj(u) :
    if (color[v] == white) :        // unseen
      parent[v] = u                 // tree edge
      DFSVisit(v)
  color[u] = black                  // done
  f[u] = ++time                     // finish
```

black Tree edge brown Back edge

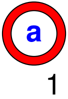
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



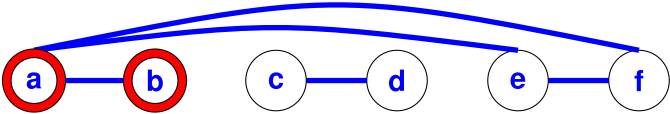
black Tree edge brown Back edge



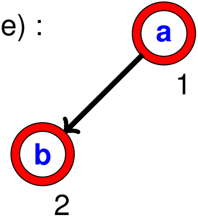
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



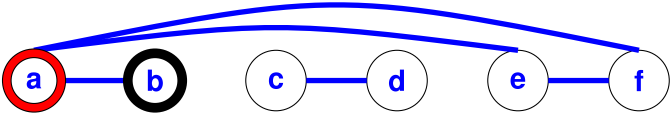
black Tree edge   brown Back edge



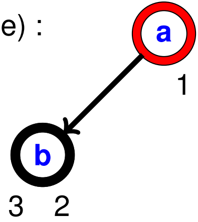
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



black Tree edge brown Back edge

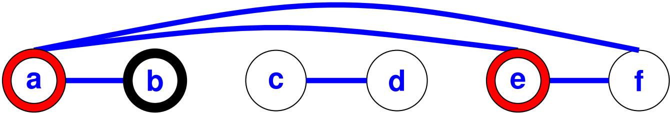


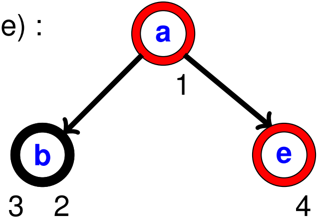
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



black Tree edge   brown Back edge

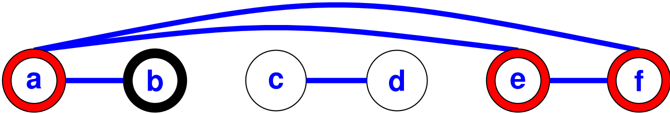
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```

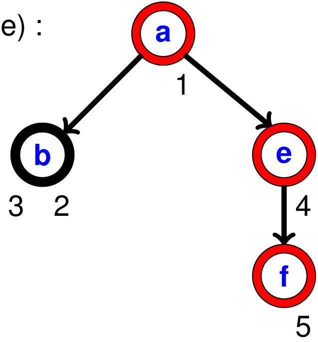




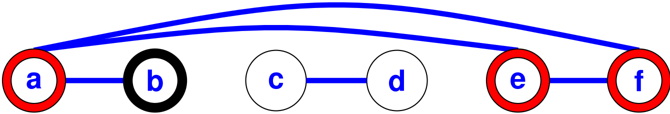
black Tree edge brown Back edge

```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```

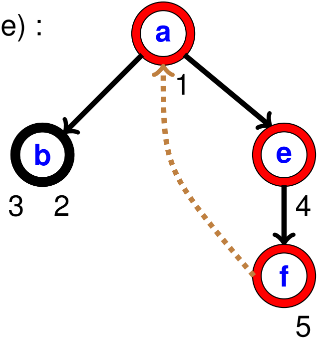




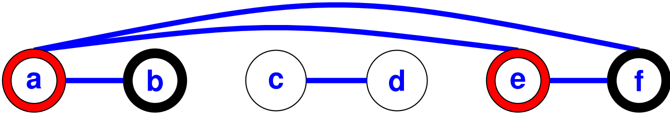
black Tree edge   brown Back edge



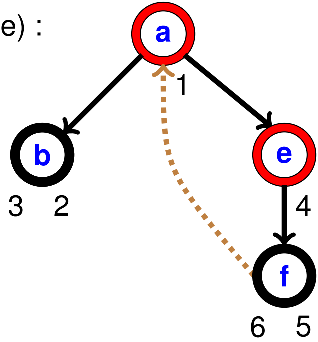
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



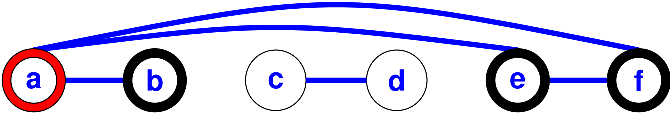
black Tree edge brown Back edge



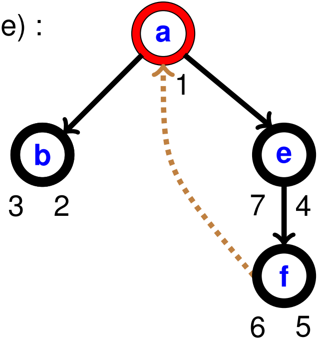
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



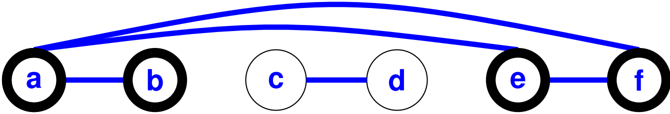
black Tree edge   brown Back edge



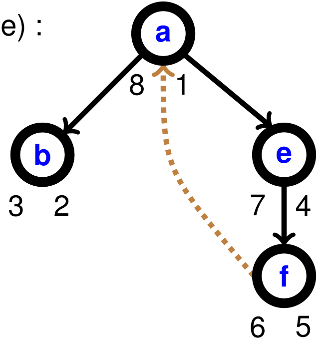
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



black Tree edge   brown Back edge

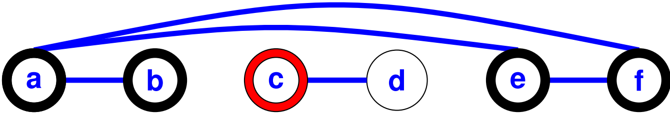


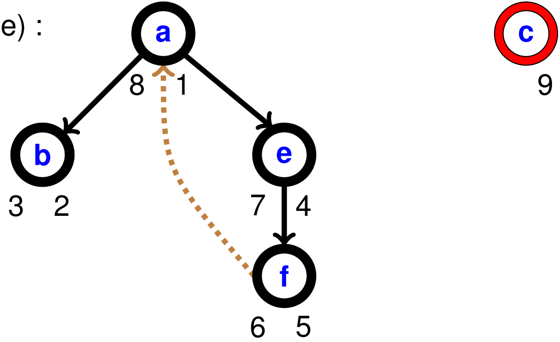
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



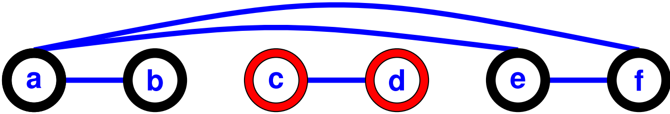
black Tree edge   brown Back edge

```
DFSVisit(u)
 color[u] = red
 d[u] = ++time
 for each v in Adj(u) :
   if (color[v] == white) :
     parent[v] = u
     DFSVisit(v)
 color[u] = black
 f[u] = ++time
```

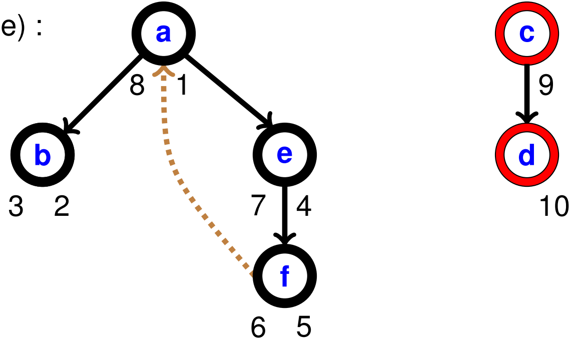




black Tree edge brown Back edge

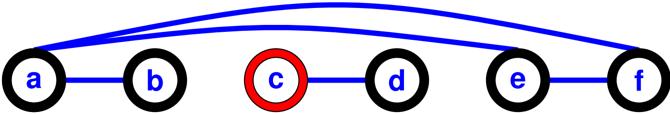


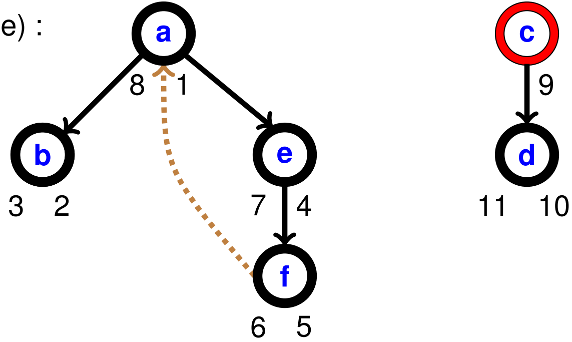
```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



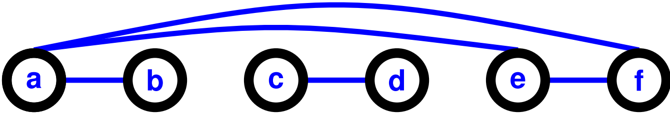
black Tree edge   brown Back edge

```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```

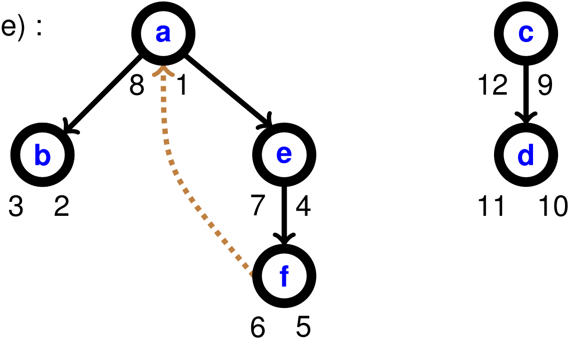




black Tree edge   brown Back edge



```
DFSVisit(u)
  color[u] = red
  d[u] = ++time
  for each v in Adj(u) :
    if (color[v] == white) :
      parent[v] = u
      DFSVisit(v)
  color[u] = black
  f[u] = ++time
```



Depth First Search (undirected graphs)

```
DFSmain(G)
 for each u in V :
   color[u] = white
   parent[u] = NULL
 time=0
 for each u in V :
   if (color[u] == white):
     DFSVisit(u)
```

```
DFSVisit(u)
 color[u] = red                   // in process
 d[u] = ++time                    // discover
 for each v in Adj(u) :
   if (color[v] == white) :       // unseen
     parent[v] = u                // tree edge
     DFSVisit(v)
 color[u] = black                 // done
 f[u] = ++time                    // finish
```

Thm. **(Properties of DFS on Undirected Graphs)**
Let G be an undirected graph with n vertices and m edges.
Then:

Thm. (Properties of DFS on Undirected Graphs)
Let G be an undirected graph with n vertices and m edges.
Then:

1. DFS(G) runs in **linear time**, i.e., $O(n + m)$.

Thm. (Properties of DFS on Undirected Graphs)
Let G be an undirected graph with n vertices and m edges.
Then:

1. DFS(G) runs in **linear time**, i.e., $O(n + m)$.
2. DFS computes **connected components** of G.

Thm. (Properties of DFS on Undirected Graphs)
Let G be an undirected graph with n vertices and m edges.
Then:

1. DFS(G) runs in **linear time**, i.e., $O(n + m)$.
2. DFS computes **connected components** of G.
3. DFS determines which of these components is cyclic: a component is **cyclic** iff it has a **backedge**.

Proof:

1. DFS runs in linear time, i.e., $O(n+m)$ time.

   DFSmain has constant number of steps per vertex. $O(n)$

Proof:

1. DFS runs in linear time, i.e., $O(n+m)$ time.

DFSmain has constant number of steps per vertex. $O(n)$

DFSVisit(v) called once for each vertex v. $O(n)$

Proof:

1. DFS runs in linear time, i.e., $O(n+m)$ time.

DFSmain has constant number of steps per vertex. $O(n)$

DFSVisit(v) called once for each vertex v. $O(n)$

DFSVisit(v) performs a bounded number of steps, $O(n)$

Proof:

1. DFS runs in linear time, i.e., $O(n+m)$ time.

DFSmain has constant number of steps per vertex. $O(n)$

DFSVisit(v) called once for each vertex v. $O(n)$

DFSVisit(v) performs a bounded number of steps, except for walking down v's adjacency list, $O(n)$

Proof:

1. DFS runs in linear time, i.e., $O(n+m)$ time.

DFSmain has constant number of steps per vertex. $O(n)$

DFSVisit(v) called once for each vertex v. $O(n)$

DFSVisit(v) performs a bounded number of steps, $O(n)$
except for walking down v's adjacency list,
one step for each outgoing edge.

Proof:

1. DFS runs in linear time, i.e., $O(n+m)$ time.

DFSmain has constant number of steps per vertex. $O(n)$

DFSVisit(v) called once for each vertex v. $O(n)$

DFSVisit(v) performs a bounded number of steps, $O(n)$
except for walking down v's adjacency list,
one step for each outgoing edge.
Each edge examined once in each direction. $O(m)$

Proof:

1. DFS runs in linear time, i.e., $O(n+m)$ time.

DFSmain has constant number of steps per vertex. $O(n)$

DFSVisit(v) called once for each vertex v. $O(n)$

DFSVisit(v) performs a bounded number of steps, $O(n)$
except for walking down v's adjacency list,
one step for each outgoing edge.
Each edge examined once in each direction. $O(m)$

O(n+m)

✓

2. **Claim** The trees of the *DFS* forest are exactly the connected components of G.

2. **Claim** The trees of the *DFS* forest are exactly the connected components of *G*.

We show that all the vertices reachable from *r* are included in the DFS tree whose root is *r*.

2. **Claim** The trees of the *DFS* forest are exactly the connected components of G.

We show that all the vertices reachable from r are included in the DFS tree whose root is r.

Prove by induction on the number of vertices in $[r]$.

2. **Claim** The trees of the *DFS* forest are exactly the connected components of G.

We show that all the vertices reachable from r are included in the DFS tree whose root is r.

Prove by induction on the number of vertices in $[r]$.

base case: $1 = |[r]| = \{r\}$ visited first step of DFSVisit(r). ✓

inductive case: Assume **indHyp**: for all $H, r \in V^H$,
$|V^H| \leq n_0 \Rightarrow$ if we start at r then DFSVisit(r) visits all of $[r]$.

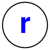
inductive case: Assume **indHyp**: for all $H, r \in V^H$, $|V^H| \leq n_0 \ \Rightarrow$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

inductive case: Assume **indHyp**: for all $H, r \in V^H$,
$|V^H| \leq n_0 \Rightarrow$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

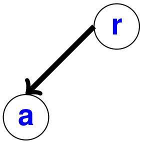
Call DFSVisit(r),



inductive case: Assume **indHyp**: for all $H, r \in V^H$,
$|V^H| \leq n_0 \ \Rightarrow$ if we start at r then DFSVisit(r) visits all of [r].

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

Call DFSVisit(r), let a be first neighbor of r.

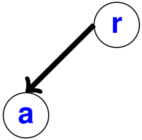


inductive case: Assume **indHyp**: for all $H, r \in V^H$, $|V^H| \leq n_0 \;\Rightarrow\;$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

Call DFSVisit(r), let a be first neighbor of r.

$H_1 \overset{\text{def}}{=}$ vertices reachable from a without going through r.

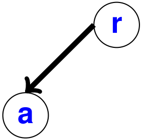


inductive case: Assume **indHyp**: for all $H, r \in V^H$, $|V^H| \leq n_0 \;\Rightarrow\;$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

Call DFSVisit(r), let a be first neighbor of r.

$H_1 \overset{\text{def}}{=}$ vertices reachable from a without going through r. H_1 is connected and has at most n_0 vertices.

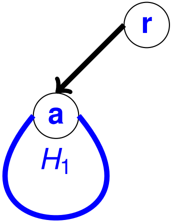


inductive case: Assume **indHyp**: for all $H, r \in V^H$, $|V^H| \leq n_0 \;\Rightarrow\;$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

Call DFSVisit(r), let a be first neighbor of r.

$H_1 \overset{\text{def}}{=}$ vertices reachable from a without going through r. H_1 is connected and has at most n_0 vertices. By **indHyp**, DFSVisit(a) visits all of H_1.

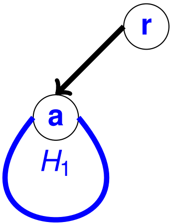


inductive case: Assume **indHyp**: for all $H, r \in V^H$, $|V^H| \leq n_0 \ \Rightarrow$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

Call DFSVisit(r), let a be first neighbor of r.

$H_1 \stackrel{\text{def}}{=}$ vertices reachable from a without going through r. H_1 is connected and has at most n_0 vertices. By **indHyp**, DFSVisit(a) visits all of H_1.



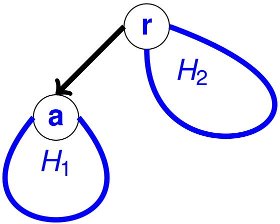
$H_2 \stackrel{\text{def}}{=} G - H_1$. Remainder of DFSVisit($r$) is the same as if H_1 didn't exist and we are just doing the DFS of H_2.

inductive case: Assume **indHyp**: for all $H, r \in V^H$,
$|V^H| \le n_0 \;\Rightarrow\;$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

Call DFSVisit(r), let a be first neighbor of r.

$H_1 \overset{\text{def}}{=}$ vertices reachable from a without going through r. H_1 is connected and has at most n_0 vertices. By **indHyp**, DFSVisit(a) visits all of H_1.



$H_2 \overset{\text{def}}{=} G - H_1$. Remainder of DFSVisit($r$) is the same as if H_1 didn't exist and we are just doing the DFS of H_2.

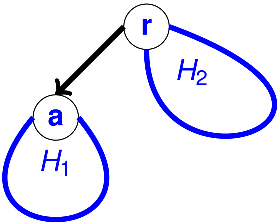
By **indHyp**, DFSVisit(r) visits all of H_2.

inductive case: Assume **indHyp**: for all $H, r \in V^H$, $|V^H| \le n_0 \;\Rightarrow\;$ if we start at r then DFSVisit(r) visits all of $[r]$.

Let G be arbitrary connected, $|V^G| = n_0 + 1$, $r \in V^G$.

Call DFSVisit(r), let a be first neighbor of r.

$H_1 \overset{\text{def}}{=}$ vertices reachable from a without going through r. H_1 is connected and has at most n_0 vertices. By **indHyp**, DFSVisit(a) visits all of H_1.



$H_2 \overset{\text{def}}{=} G - H_1$. Remainder of DFSVisit($r$) is the same as if H_1 didn't exist and we are just doing the DFS of H_2.

By **indHyp**, DFSVisit(r) visits all of H_2.

Thus initial call of DFSVisit(r) visits all of $G = H_1 \cup H_2$. ✓

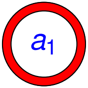
3. **Claim** *G* is cyclic iff DFS(*G*) finds a backedge.

3. **Claim** *G* is cyclic iff DFS(*G*) finds a backedge.

If there is a backege from *b* to *a*, then there is a path in the tree from *a* down to *b*, so this path plus the backedge forms a cycle. ✓

3. **Claim** G is cyclic iff DFS(G) finds a backedge.

If there is a backege from b to a, then there is a path in the tree from a down to b, so this path plus the backedge forms a cycle. ✓

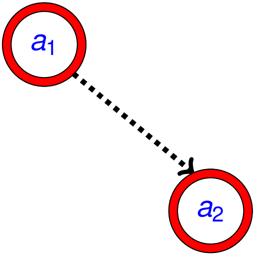


Conversely, suppose that G contains a cycle and let $C = a_1, a_2, \ldots, a_{k-1}, a_1$ be a cycle. Let a_1 be the first vertex of the cycle that is visited in DFS(G).

3. **Claim** G is cyclic iff DFS(G) finds a backedge.

If there is a backege from b to a, then there is a path in the tree from a down to b, so this path plus the backedge forms a cycle. ✓

Conversely, suppose that G contains a cycle and let $C = a_1, a_2, \ldots, a_{k-1}, a_1$ be a cycle. Let a_1 be the first vertex of the cycle that is visited in DFS(G). By (2), we know that a_2 and a_{k-1} are visited during the call of DFSVisit(a_1). Assume that a_2 is visited first, the other case is symmetrical.

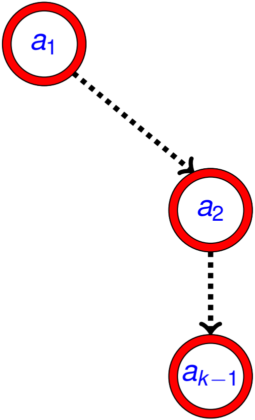


3. **Claim** G is cyclic iff DFS(G) finds a backedge.

If there is a backege from b to a, then there is a path in the tree from a down to b, so this path plus the backedge forms a cycle. ✓

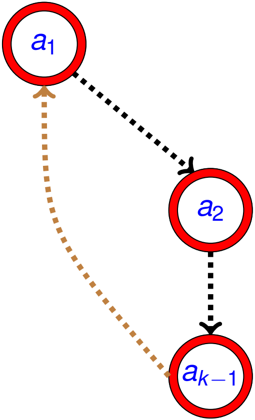
Conversely, suppose that G contains a cycle and let $C = a_1, a_2, \ldots, a_{k-1}, a_1$ be a cycle. Let a_1 be the first vertex of the cycle that is visited in DFS(G). By (2), we know that a_2 and a_{k-1} are visited during the call of DFSVisit(a_1). Assume that a_2 is visited first, the other case is symmetrical.

By (2), we know that a_{k-1} is visited during the call of DFSVisit(a_2).



3. **Claim** G is cyclic iff DFS(G) finds a backedge.

If there is a backege from b to a, then there is a path in the tree from a down to b, so this path plus the backedge forms a cycle. ✓

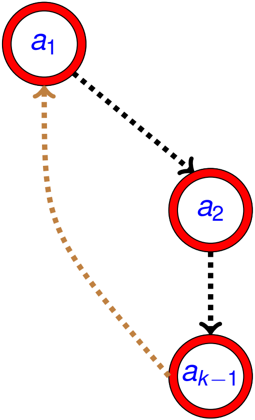


Conversely, suppose that G contains a cycle and let $C = a_1, a_2, \ldots, a_{k-1}, a_1$ be a cycle. Let a_1 be the first vertex of the cycle that is visited in DFS(G). By (2), we know that a_2 and a_{k-1} are visited during the call of DFSVisit(a_1). Assume that a_2 is visited first, the other case is symmetrical.

By (2), we know that a_{k-1} is visited during the call of DFSVisit(a_2).

When a_{k-1} is visited, a_1 is red and not a_{k-1}'s parent.

3. **Claim** G is cyclic iff DFS(G) finds a backedge.



If there is a backege from b to a, then there is a path in the tree from a down to b, so this path plus the backedge forms a cycle. ✓

Conversely, suppose that G contains a cycle and let $C = a_1, a_2, \ldots, a_{k-1}, a_1$ be a cycle. Let a_1 be the first vertex of the cycle that is visited in DFS(G). By (2), we know that a_2 and a_{k-1} are visited during the call of DFSVisit(a_1). Assume that a_2 is visited first, the other case is symmetrical.

By (2), we know that a_{k-1} is visited during the call of DFSVisit(a_2).

When a_{k-1} is visited, a_1 is red and not a_{k-1}'s parent. Thus the edge (a_{k-1}, a_1) is a backedge. □