Graph Algorithms

Graph Traversals and Connectivity II

Michael Lampis

September 11, 2025

Graph Traversal

Problem

Given (di)graph G, determine connectivity properties:

- Is G (strongly) connected?
- What are the (strongly) connected components of G?
- Which vertices can be reached from a given source s?
- What is the shortest path distance from (given vertex) s to (given vertex) t?

Graph Traversal

Problem

Given (di)graph G, determine connectivity properties:

- Is G (strongly) connected?
- What are the (strongly) connected components of G?
- Which vertices can be reached from a given source s?
- What is the shortest path distance from (given vertex) s to (given vertex) t?

Algorithms:

- BFS (last lecture)
- DFS (today)

Depth-First Search

Depth-first search (DFS) is a basic **graph traversal** algorithm.

- Input: G and specified source vertex s
- Output: Set of vertices reachable from s and tree of shortest paths from s to all such vertices.

Depth-First Search

Depth-first search (DFS) is a basic **graph traversal** algorithm.

- Input: G and specified source vertex s
- Output: Set of vertices reachable from s and tree of shortest paths from s to all such vertices.

Key properties:

- Linear time and space complexity $O(n + m)$.
- Works for both graphs and digraphs.

Depth-First Search

Depth-first search (DFS) is a basic **graph traversal** algorithm.

- Input: G and specified source vertex s
- Output: Set of vertices reachable from s and tree of shortest paths from s to all such vertices.

Key properties:

- Linear time and space complexity $O(n + m)$.
- Works for both graphs and digraphs.

Key idea:

- Explore vertices in order of increasing distance from s, using a queue going as far as possible, until we get stuck, then backtracking.

DFS High-level ideas

- Vertices are colored White, Gray, or Black
  - White: undiscovered
  - Gray: discovered, not processed yet
  - Black: finished
- As vertices are discovered, ~~they are added to the queue~~ we move to the first undiscovered neighbor and continue from there.
  - **NB**: This is essentially equivalent to adding vertices to a **stack** instead of a queue.
- ~~FIFO $\rightarrow$ vertices further away are processed later (proof?)~~
- We will keep track of the **time** when a vertex became Gray and Black.

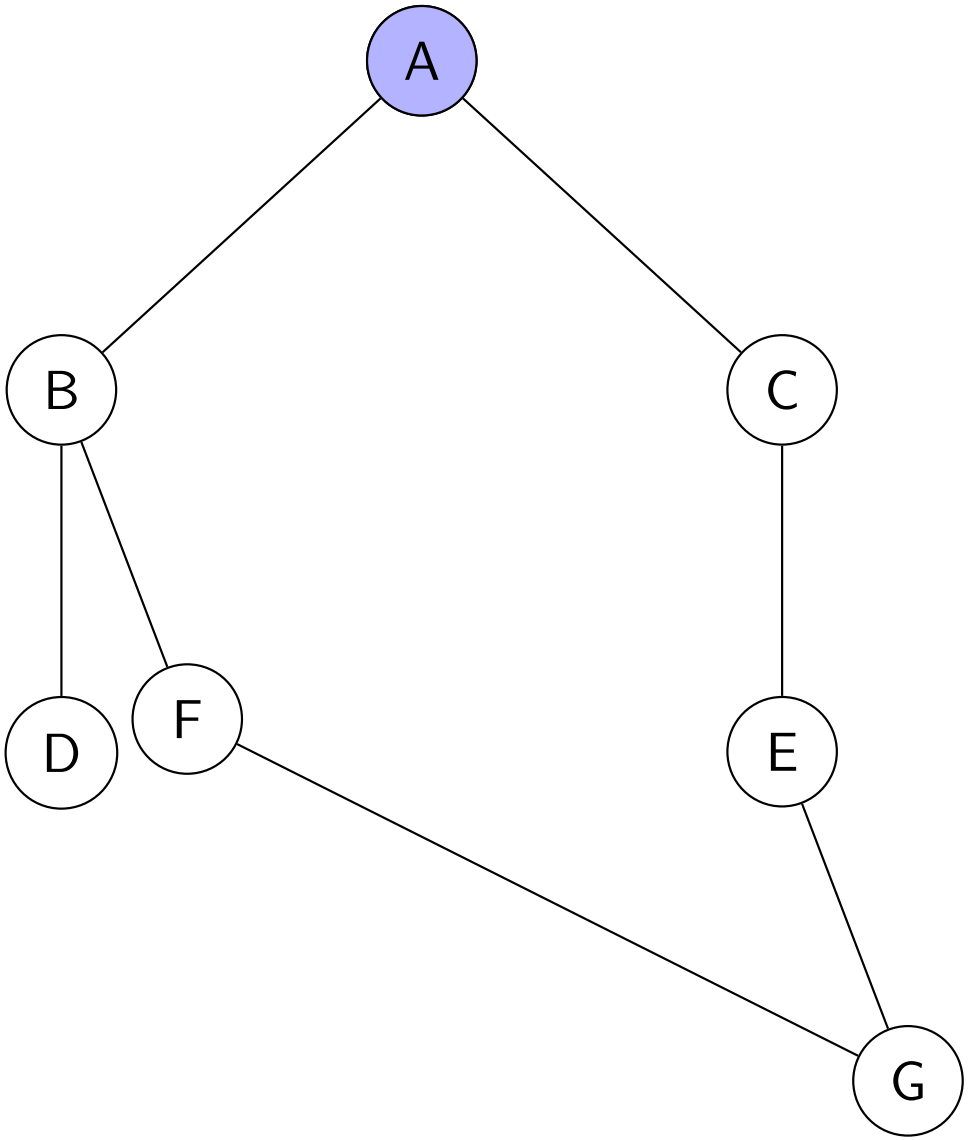
DFS – Initialization

1: **for** $v \in V \setminus \{s\}$ **do** ▷ initialize
2: Color v White, Parent of $v \leftarrow$ NULL
3: **end for**
4: $t = 0$ ▷ universal time variable
5: DFS-Visit(G,s)

DFS – Recursive Procedure

```
1:  procedure DFS-Visit(G,u)
2:      t++
3:      u.d = t, Color u Gray
4:      for v ∈ N(u) do
5:          if v is White then
6:              Set Parent of v to be u
7:              DFS-Visit(G,v)
8:          end if
9:      end for
10:     t++
11:     u.f = t, Color u Black
12: end procedure
```

DFS Example (7 Nodes) — Animation + Stack



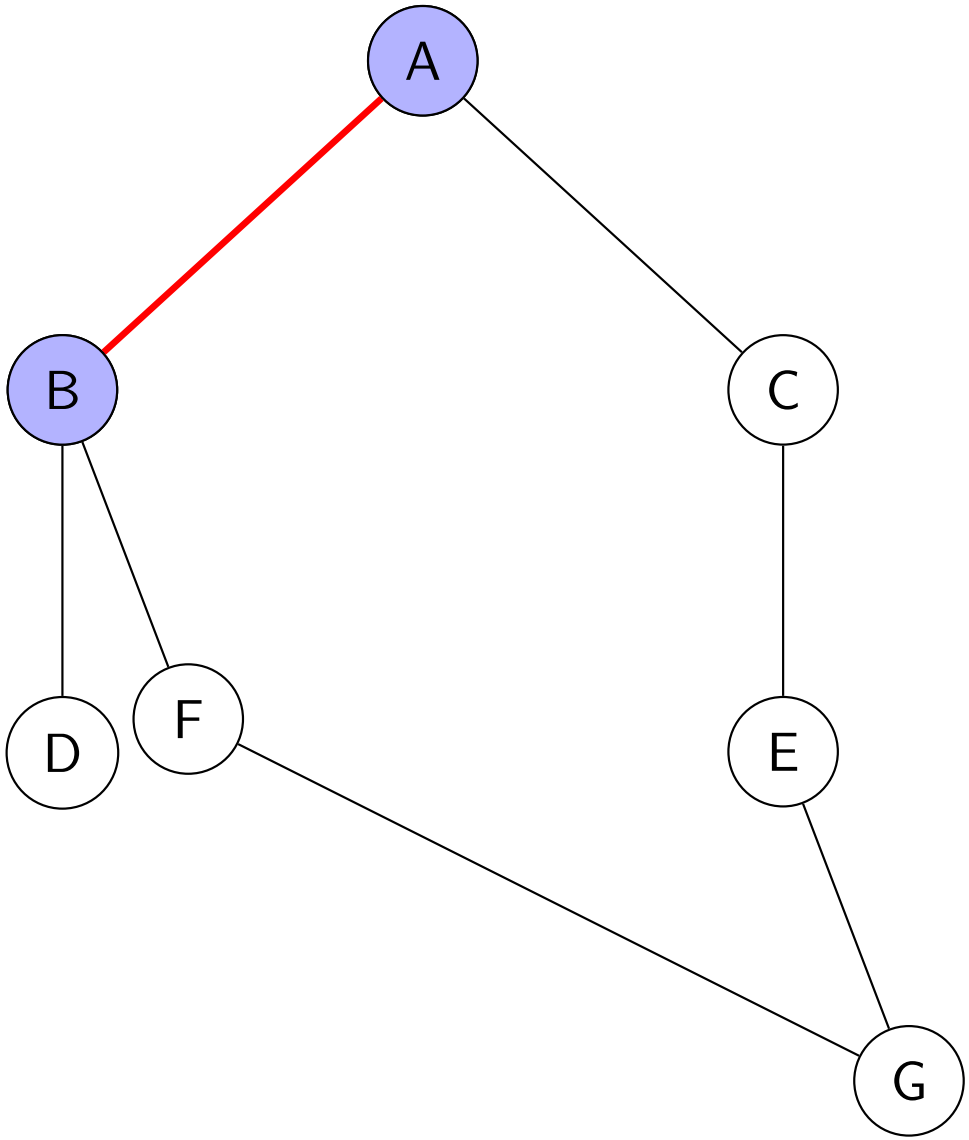
Recursion stack (top at bottom)

A

Visit order

A

DFS Example (7 Nodes) — Animation + Stack



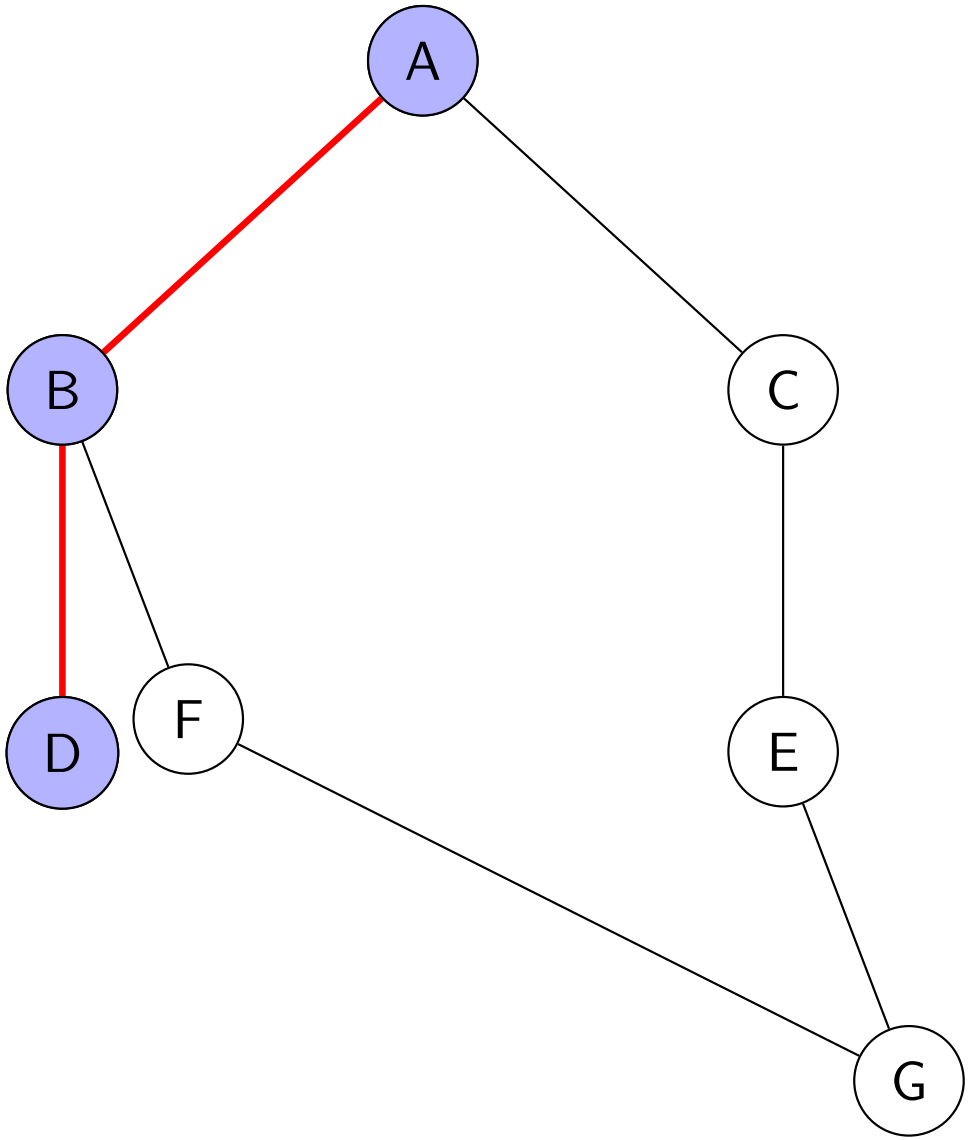
Recursion stack (top at bottom)

A
B

Visit order

A $\rightarrow$ B

DFS Example (7 Nodes) — Animation + Stack



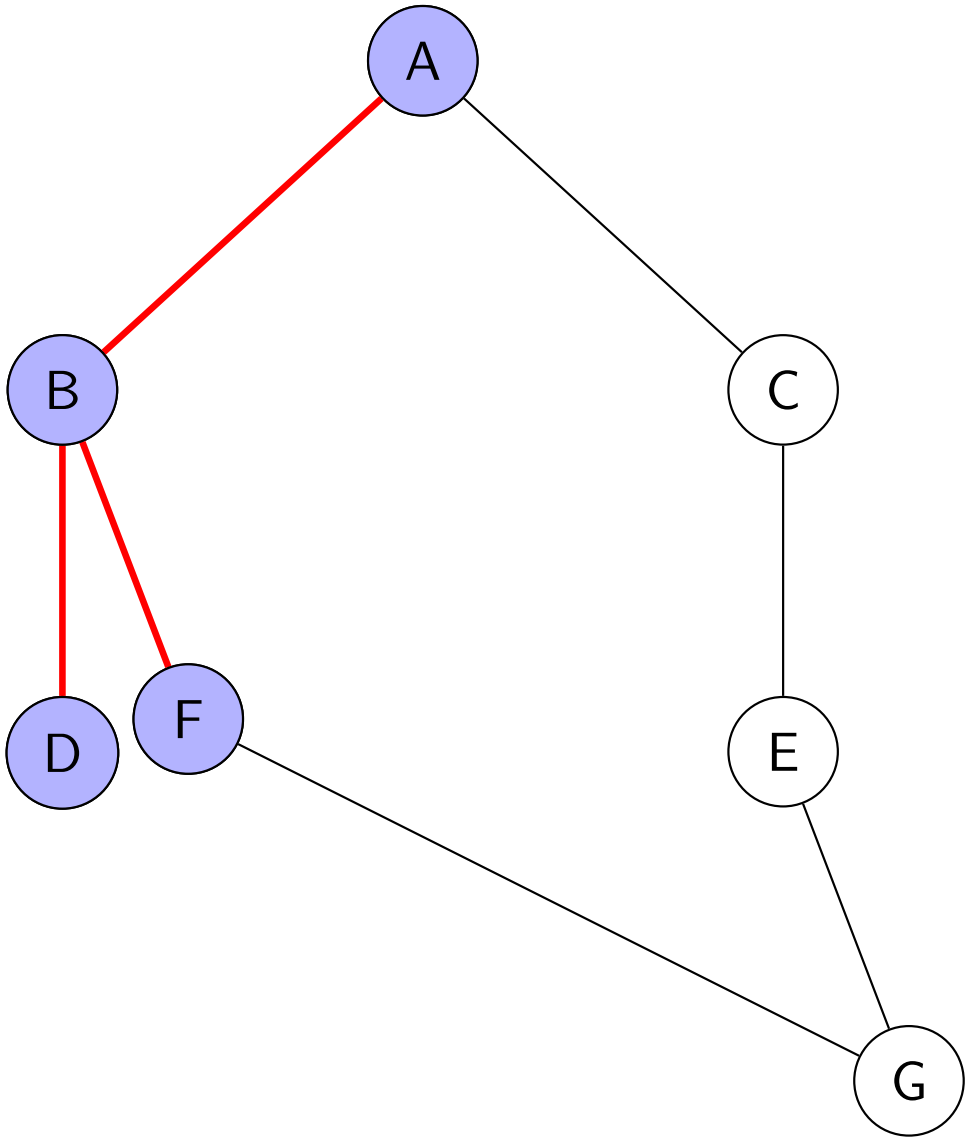
Recursion stack (top at bottom)

A
B
D

Visit order

A $\rightarrow$ B $\rightarrow$ D

DFS Example (7 Nodes) — Animation + Stack



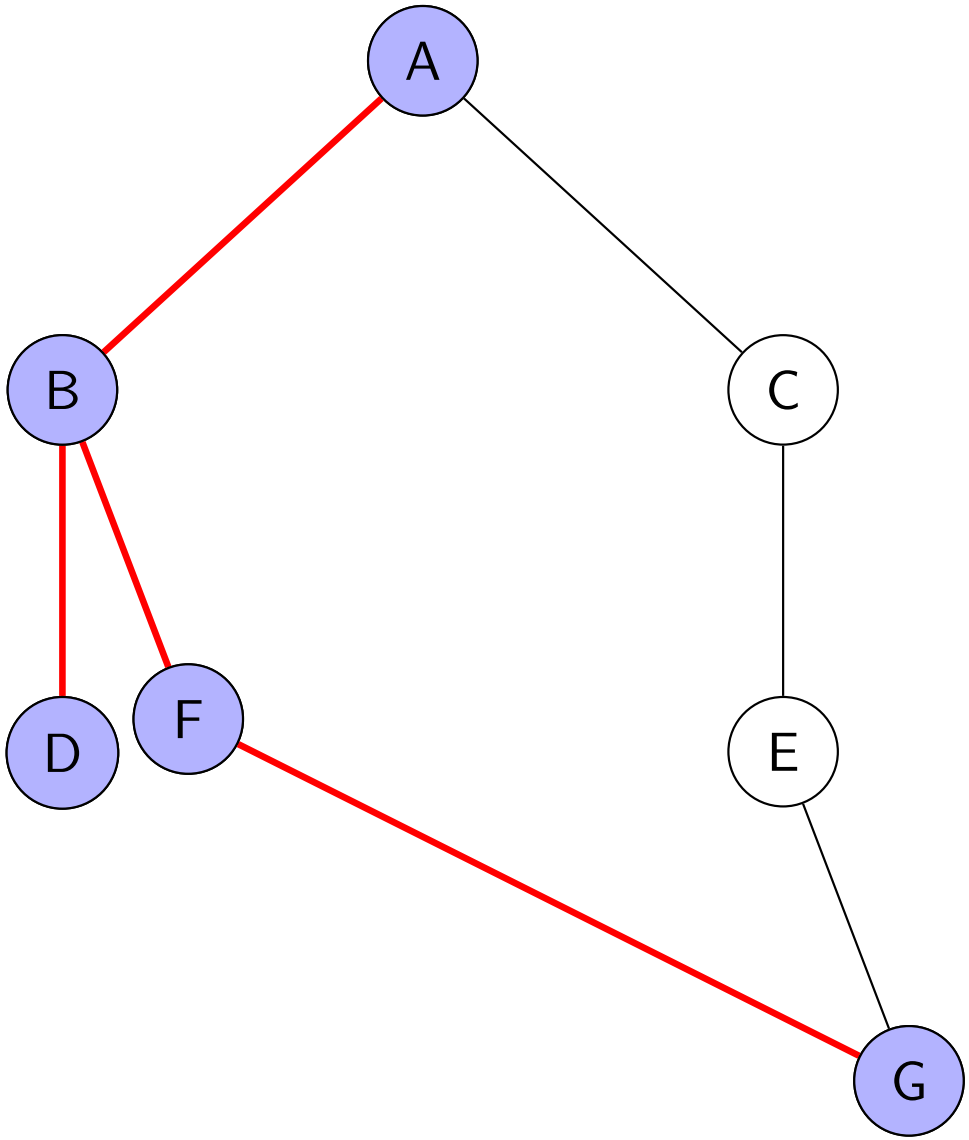
Recursion stack (top at bottom)

A
B
F

Visit order

A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ F

DFS Example (7 Nodes) — Animation + Stack



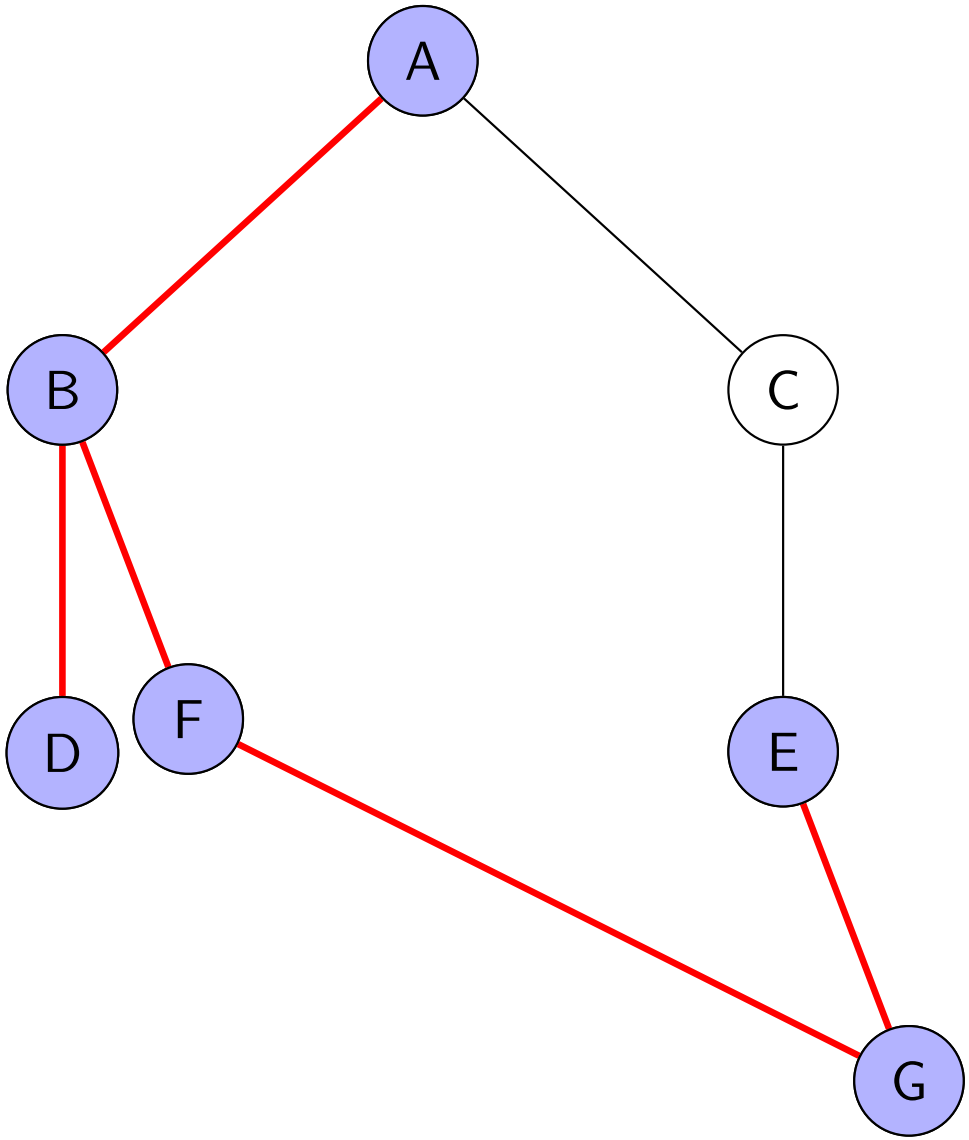
Recursion stack (top at bottom)

A
B
F
G

Visit order

A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ F $\rightarrow$ G

DFS Example (7 Nodes) — Animation + Stack



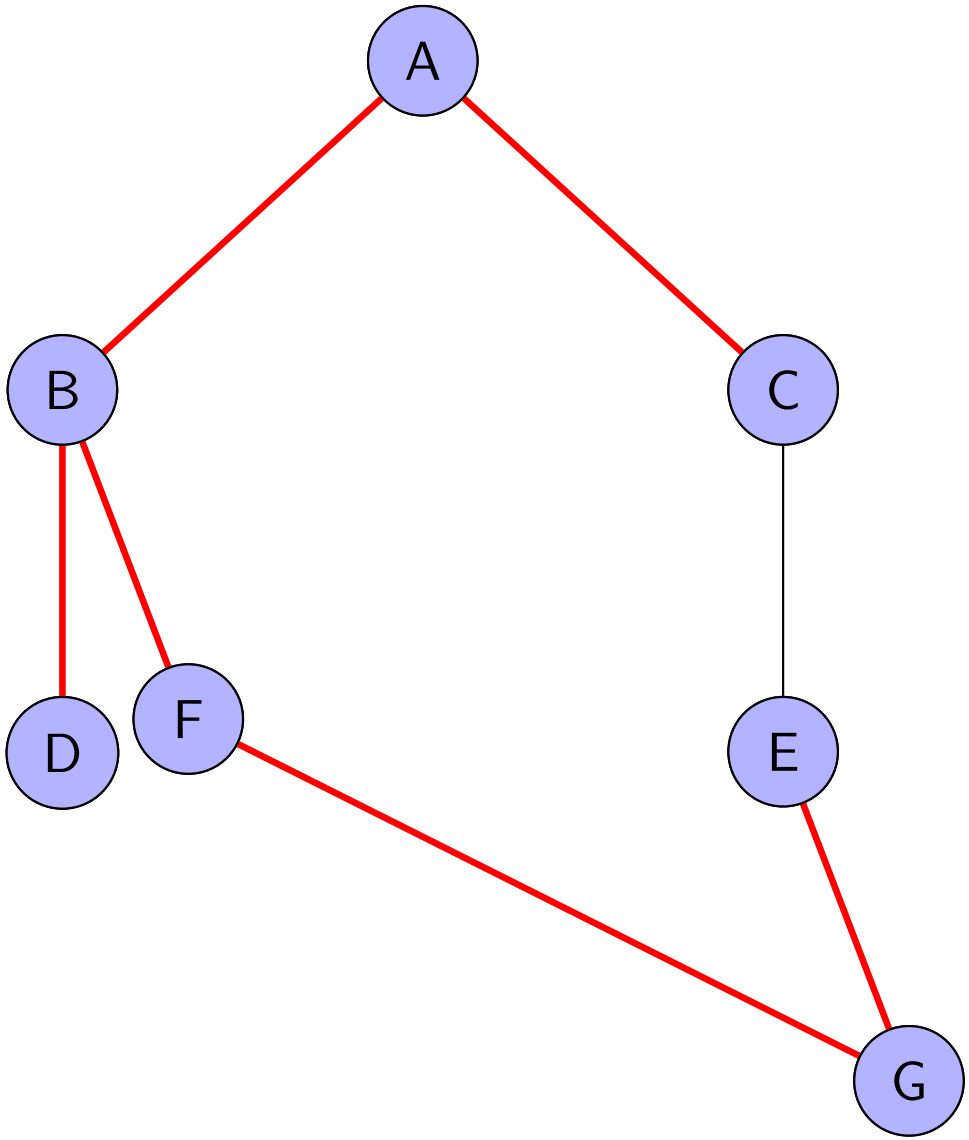
Recursion stack (top at bottom)

A
B
F
G
E

Visit order

A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ F $\rightarrow$ G $\rightarrow$ E

DFS Example (7 Nodes) — Animation + Stack



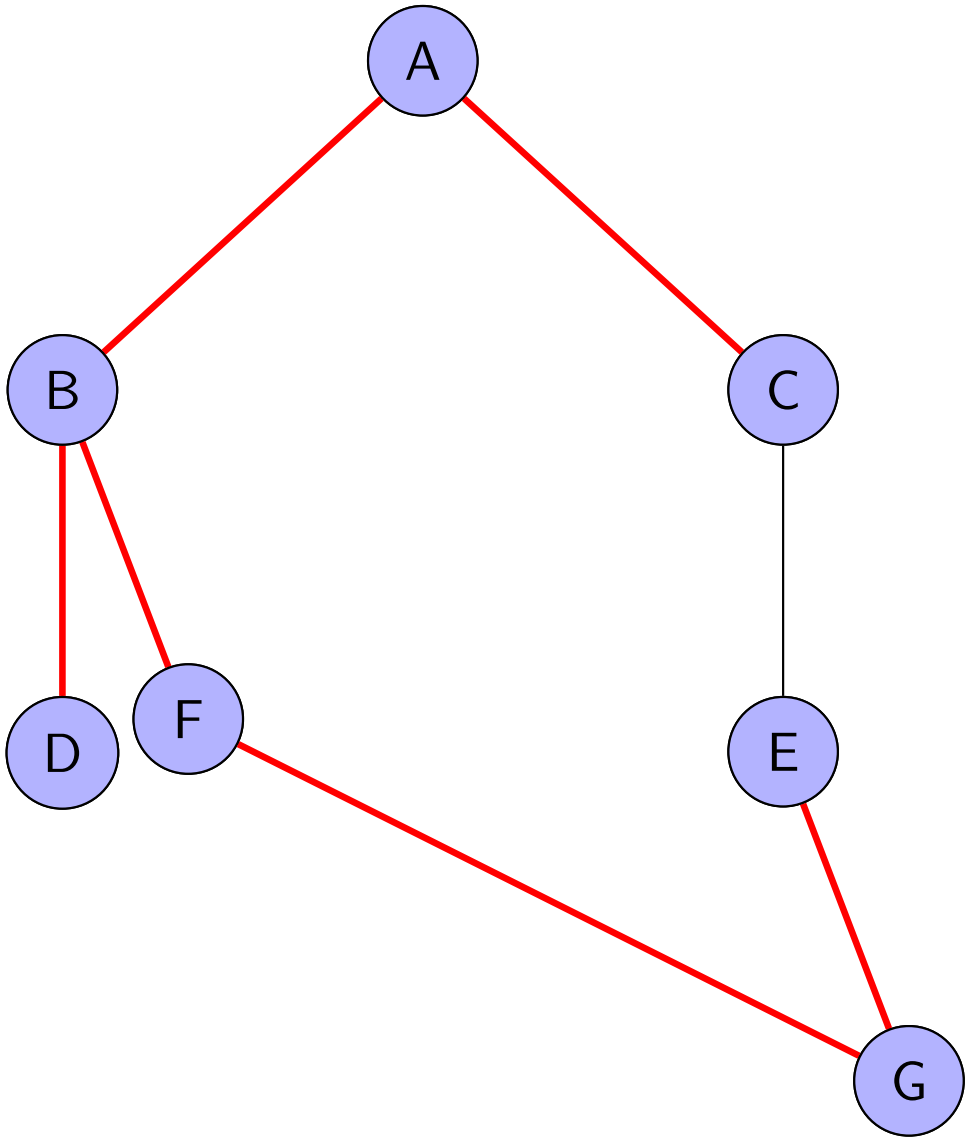
Recursion stack (top at bottom)

A
C

Visit order

A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ F $\rightarrow$ G $\rightarrow$ E $\rightarrow$ C

DFS Example (7 Nodes) — Animation + Stack



Recursion stack (top at bottom)

empty

Visit order

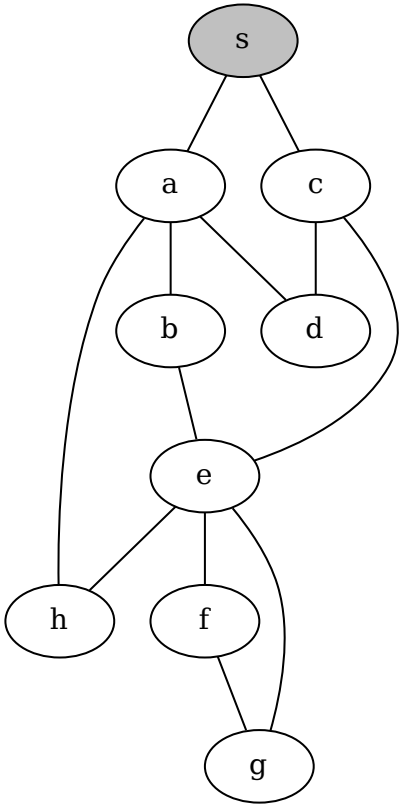
A $\rightarrow$ B $\rightarrow$ D $\rightarrow$ F $\rightarrow$ G $\rightarrow$ E $\rightarrow$ C

Remarks about example

Remarks about example

- Previous example was generated by ChatGPT
  - Only took $2-3$ minutes of computing time, and a couple of iterations!
- It is **mostly** correct and illustrates the high-level idea of DFS
- However, it contains some minor error...
  - Incorrect DFS tree.
- What is the moral lesson of this story? (food for thought...)

Example – (human generated, could still be wrong?)



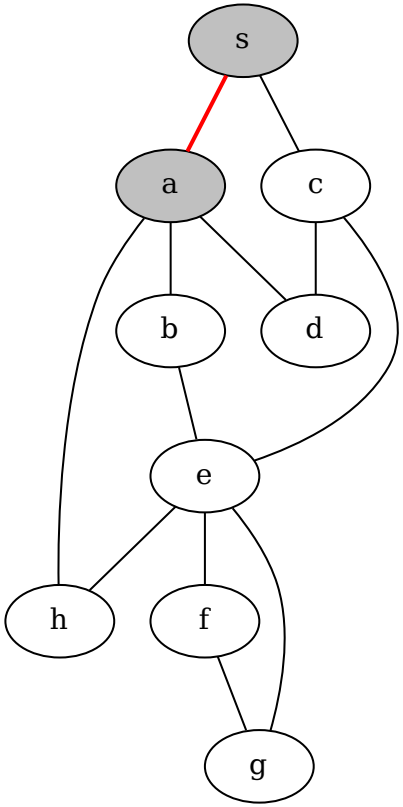
Times:

	d	f
s	1	
a		
b		
c		
d		
e		
f		
g		
h		

Calling Stack:

s						...

Example – (human generated, could still be wrong?)



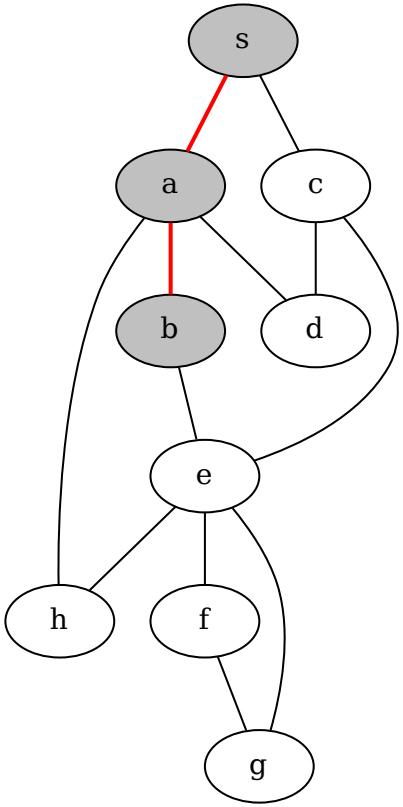
Times:

	d	f
s	1	
a	2	
b		
c		
d		
e		
f		
g		
h		

Calling Stack:

s	a					...

Example – (human generated, could still be wrong?)



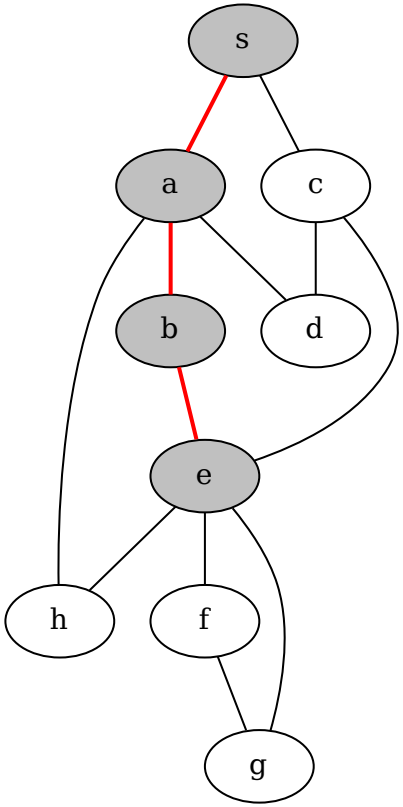
Times:

	d	f
s	1	
a	2	
b	3	
c		
d		
e		
f		
g		
h		

Calling Stack:

s	a	b				...

Example – (human generated, could still be wrong?)



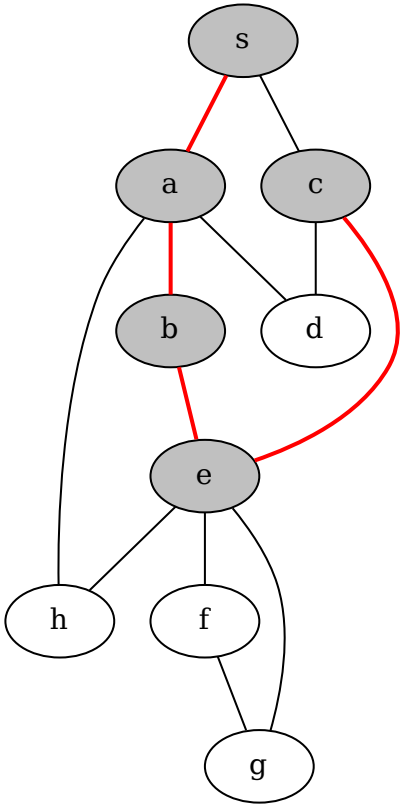
Times:

	d	f
s	1	
a	2	
b	3	
c		
d		
e	4	
f		
g		
h		

Calling Stack:

s	a	b	e			...

Example – (human generated, could still be wrong?)



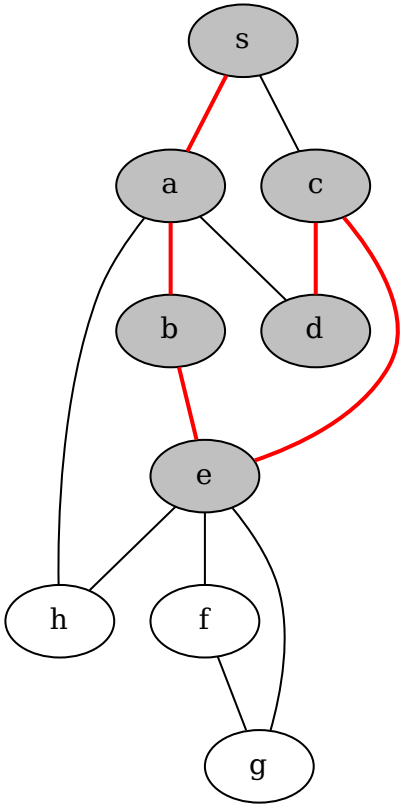
Times:

	d	f
s	1	
a	2	
b	3	
c	5	
d		
e	4	
f		
g		
h		

Calling Stack:

s	a	b	e	c		...

Example – (human generated, could still be wrong?)



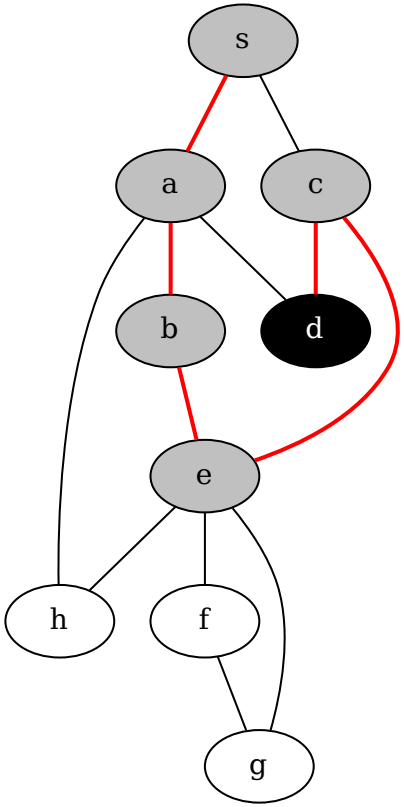
Times:

	d	f
s	1	
a	2	
b	3	
c	5	
d	6	
e	4	
f		
g		
h		

Calling Stack:

s	a	b	e	c	d	...

Example – (human generated, could still be wrong?)



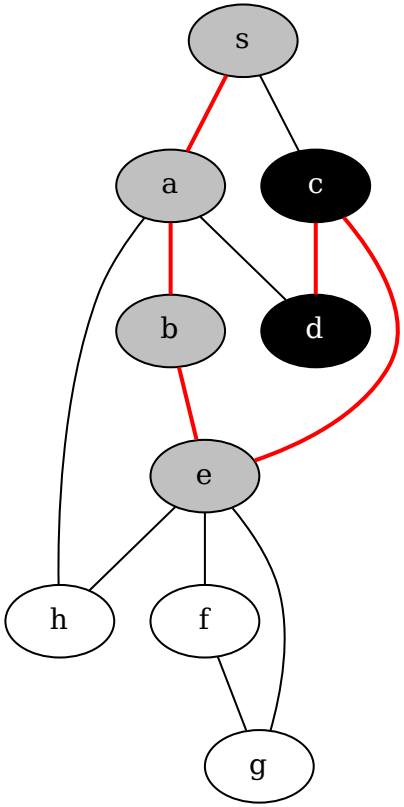
Times:

	d	f
s	1	
a	2	
b	3	
c	5	
d	6	7
e	4	
f		
g		
h		

Calling Stack:

s	a	b	e	c		...

Example – (human generated, could still be wrong?)



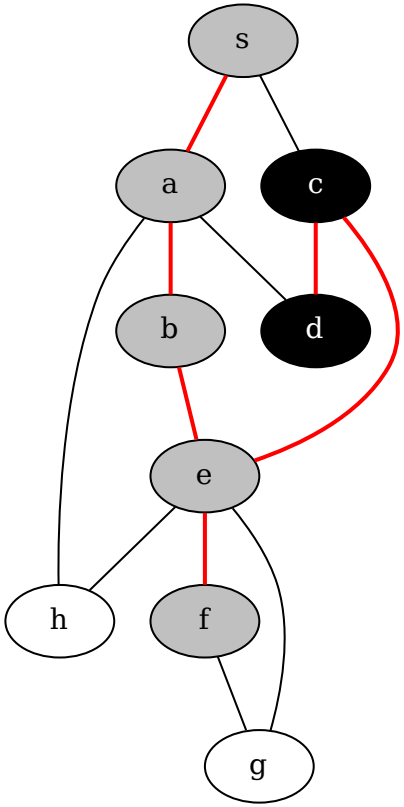
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	
f		
g		
h		

Calling Stack:

s	a	b	e			...

Example – (human generated, could still be wrong?)



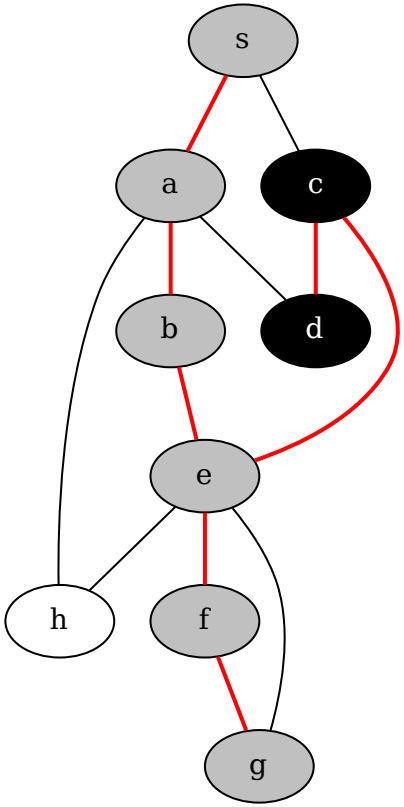
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	
f	9	
g		
h		

Calling Stack:

s	a	b	e	f		...

Example – (human generated, could still be wrong?)



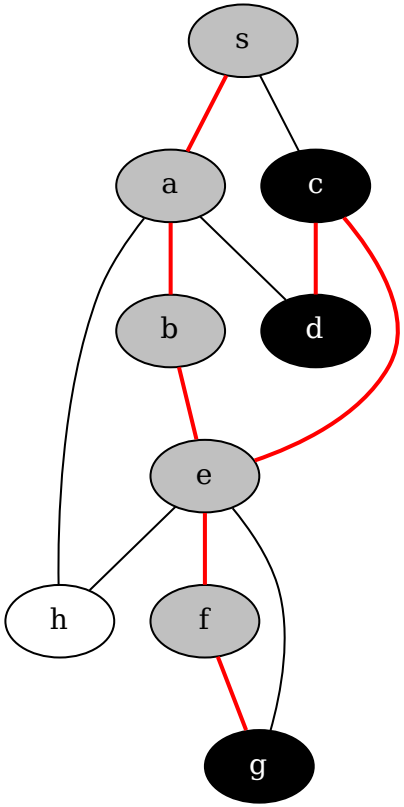
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	
f	9	
g	10	
h		

Calling Stack:

s	a	b	e	f	g	...

Example – (human generated, could still be wrong?)



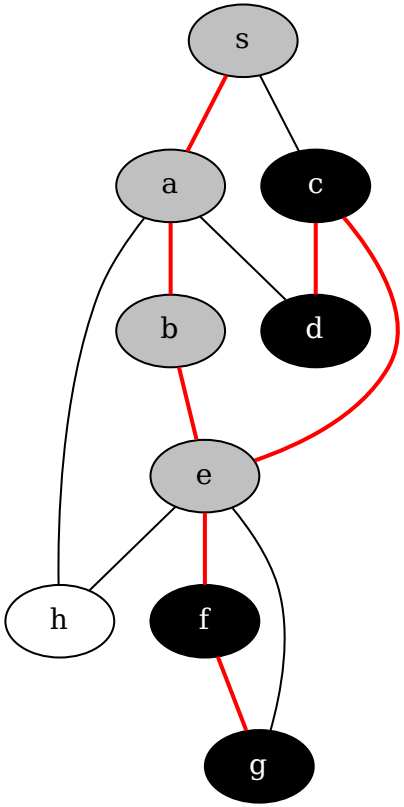
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	
f	9	
g	10	11
h		

Calling Stack:

s	a	b	e	f		...

Example – (human generated, could still be wrong?)



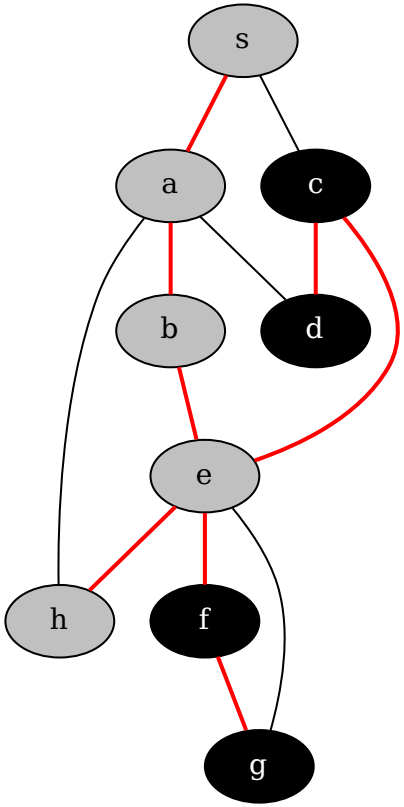
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	
f	9	12
g	10	11
h		

Calling Stack:

s	a	b	e			...

Example – (human generated, could still be wrong?)



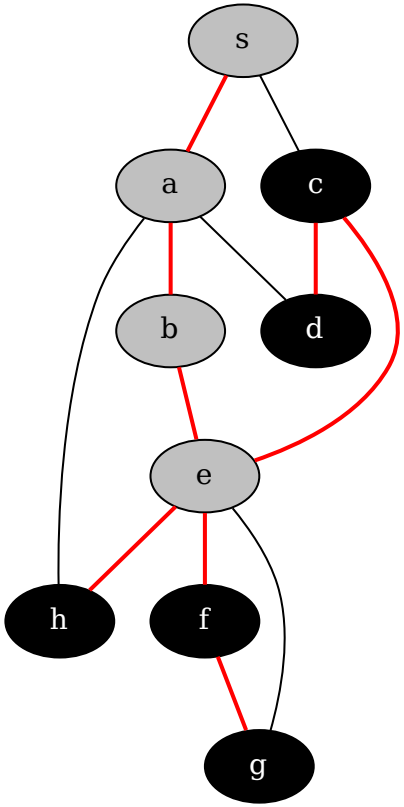
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	
f	9	12
g	10	11
h	13	

Calling Stack:

s	a	b	e	h		...

Example – (human generated, could still be wrong?)



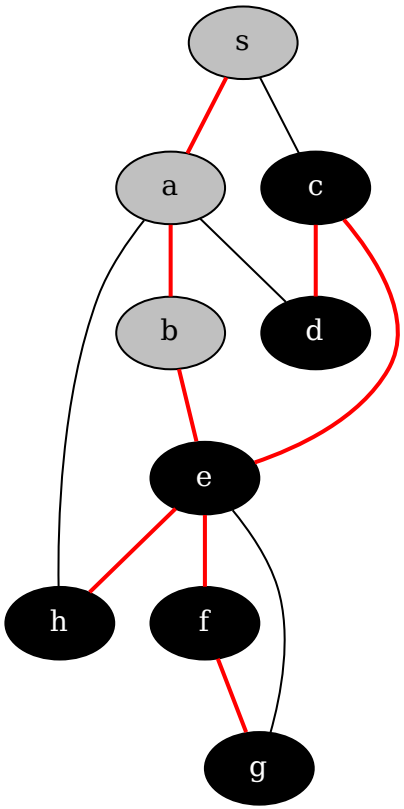
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	
f	9	12
g	10	11
h	13	14

Calling Stack:

s	a	b	e			...

Example – (human generated, could still be wrong?)



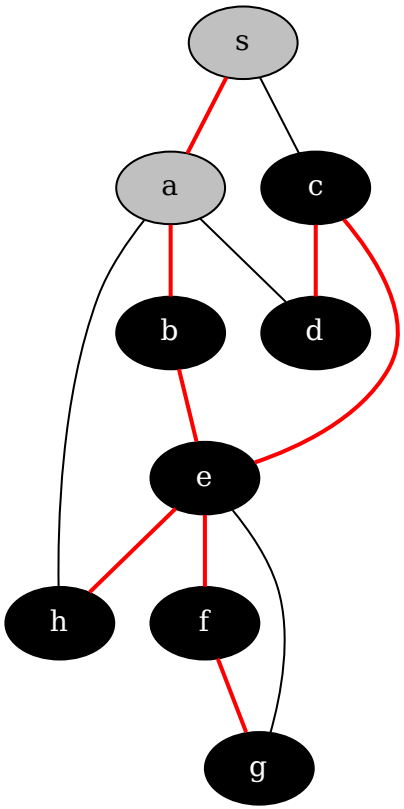
Times:

	d	f
s	1	
a	2	
b	3	
c	5	8
d	6	7
e	4	15
f	9	12
g	10	11
h	13	14

Calling Stack:

s	a	b				...

Example – (human generated, could still be wrong?)



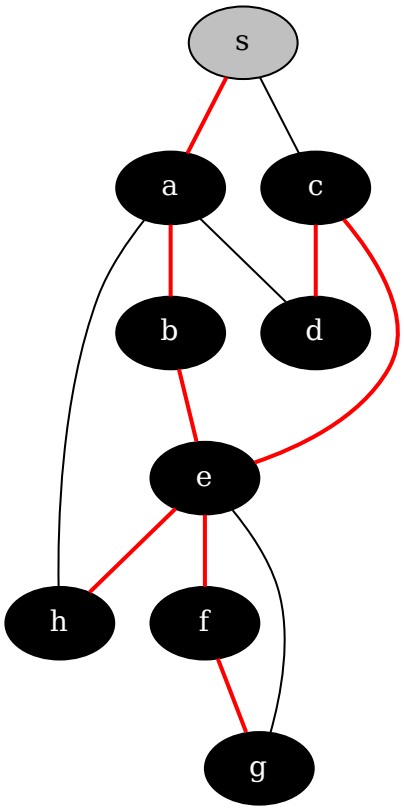
Times:

	d	f
s	1	
a	2	
b	3	16
c	5	8
d	6	7
e	4	15
f	9	12
g	10	11
h	13	14

Calling Stack:

s	a					...

Example – (human generated, could still be wrong?)



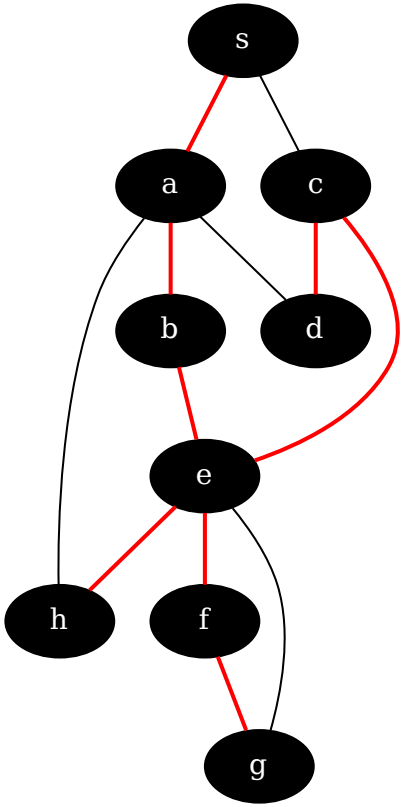
Times:

	d	f
s	1	
a	2	17
b	3	16
c	5	8
d	6	7
e	4	15
f	9	12
g	10	11
h	13	14

Calling Stack:

s						...

Example – (human generated, could still be wrong?)



Times:

	d	f
s	1	18
a	2	17
b	3	16
c	5	8
d	6	7
e	4	15
f	9	12
g	10	11
h	13	14

Calling Stack:

Analysis of DFS

Running Time

- Initialization takes $O(n)$ time (and space).

Running Time

- Initialization takes $O(n)$ time (and space).
- Vertex colors go from White$\rightarrow$Gray$\rightarrow$Black
  - $O(n)$ color changes.

Running Time

- Initialization takes $O(n)$ time (and space).
- Vertex colors go from White$\rightarrow$Gray$\rightarrow$Black
  - $O(n)$ color changes.
- $O(n)$ stack operations:
  - Only add to stack White vertices, which we turn Gray $\rightarrow$ each vertex added to stack at most once.
  - Therefore, $O(n)$ stack operations.

Running Time

- Initialization takes $O(n)$ time (and space).
- Vertex colors go from White$\rightarrow$Gray$\rightarrow$Black
  - $O(n)$ color changes.
- $O(n)$ stack operations:
  - Only add to stack White vertices, which we turn Gray $\rightarrow$ each vertex added to stack at most once.
  - Therefore, $O(n)$ stack operations.
- $O(\deg(v))$ operations when v is discovered.
  - Recall $\sum_{v \in V} \deg(v) = 2m$.

Running Time

- Initialization takes $O(n)$ time (and space).
- Vertex colors go from White$\rightarrow$Gray$\rightarrow$Black
  - $O(n)$ color changes.
- $O(n)$ stack operations:
  - Only add to stack White vertices, which we turn Gray $\rightarrow$ each vertex added to stack at most once.
  - Therefore, $O(n)$ stack operations.
- $O(\deg(v))$ operations when v is discovered.
  - Recall $\sum_{v \in V} \deg(v) = 2m$.
- $\Rightarrow$ Total space: $O(n)$ and total time $O(n + m)$.

Properties of DFS trees

Lemma

For all $u \in V$ if, d_u, f_u are the discovery and finish times for u, then $d_u < f_u$

Properties of DFS trees

Lemma

For all $u \in V$ if, d_u, f_u are the discovery and finish times for u, then $d_u < f_u$

Proof.

- d_u is time that u became Gray.
- f_u is time that u became Black
- u becomes Black **after** becoming Gray..

Properties of DFS trees

Definition

Reminder: in a tree, u is an **ancestor** of v, if $u = v$, or u is an ancestor of the parent of v.

Lemma

For all $u, v \in V$, u is an ancestor of v in the DFS tree if and only if u was Gray when v became Gray ($d_u < d_v < f_u$).

Properties of DFS trees

Definition

Reminder: in a tree, u is an **ancestor** of v, if $u = v$, or u is an ancestor of the parent of v.

Lemma

For all $u, v \in V$, u is an ancestor of v in the DFS tree if and only if u was Gray when v became Gray ($d_u < d_v < f_u$).

Proof.

- When a vertex v is added, all its ancestors are Gray.
  - Put another way: a Black vertex gains no new descendants.

□

Interval Relations

Lemma

For all $u, v \in V$ we have one of the following:

- *$[d_u, f_u]$ and $[d_v, f_v]$ are disjoint, u, v are not ancestor-descendant.*

Interval Relations

Lemma

For all $u, v \in V$ we have one of the following:

- $[d_u, f_u]$ *and* $[d_v, f_v]$ *are disjoint,* u, v *are not ancestor-descendant.*
- $[d_u, f_u]$ *contains* $[d_v, f_v]$*, and* u *is ancestor of* v*.*

Interval Relations

Lemma

For all $u, v \in V$ we have one of the following:

- *$[d_u, f_u]$ and $[d_v, f_v]$ are disjoint, u, v are not ancestor-descendant.*
- *$[d_u, f_u]$ contains $[d_v, f_v]$, and u is ancestor of v.*
- *Same as previous point but with u, v exchanged.*

Interval Relations

Lemma

For all $u, v \in V$ we have one of the following:

- *$[d_u, f_u]$ and $[d_v, f_v]$ are disjoint, u, v are not ancestor-descendant.*
- *$[d_u, f_u]$ contains $[d_v, f_v]$, and u is ancestor of v.*
- *Same as previous point but with u, v exchanged.*

Proof.

(Wlog $d_u < d_v$)

- u, v not related $\Leftrightarrow [d_u, f_u] \cap [d_v, f_v] = \emptyset$
  - Proof in next slide
- u ancestor of $v \Leftrightarrow d_u < d_v < f_v < f_u$
  - Proof in next slide

Interval Relations

Proof.

- Recall we showed u ancestor of $v \Leftrightarrow d_u < d_v < f_u$

Interval Relations

Proof.

- Recall we showed u ancestor of $v \Leftrightarrow d_u < d_v < f_u$
- Therefore u **not** ancestor of $v \Leftrightarrow (d_u > d_v$ or $d_v > f_u)$
- But we assumed wlog $d_u < d_v$
- Therefore u **not** ancestor of $v \Leftrightarrow d_v > f_u$, so intervals disjoint.
- Since $d_u < d_v$, v cannot be ancestor of u
- $\ldots \Rightarrow (u, v$ not related $\Leftrightarrow$ intervals are disjoint)

Interval Relations

Proof.

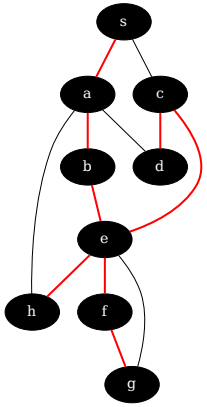
- Recall we showed u ancestor of $v \Leftrightarrow d_u < d_v < f_u$

Interval Relations

Proof.

- Recall we showed u ancestor of $v \Leftrightarrow d_u < d_v < f_u$
- What is missing: u ancestor of $v \Rightarrow f_v < f_u$
- This follows because the recursive call for v will terminate before the recursive call for u terminates (Stack structure)

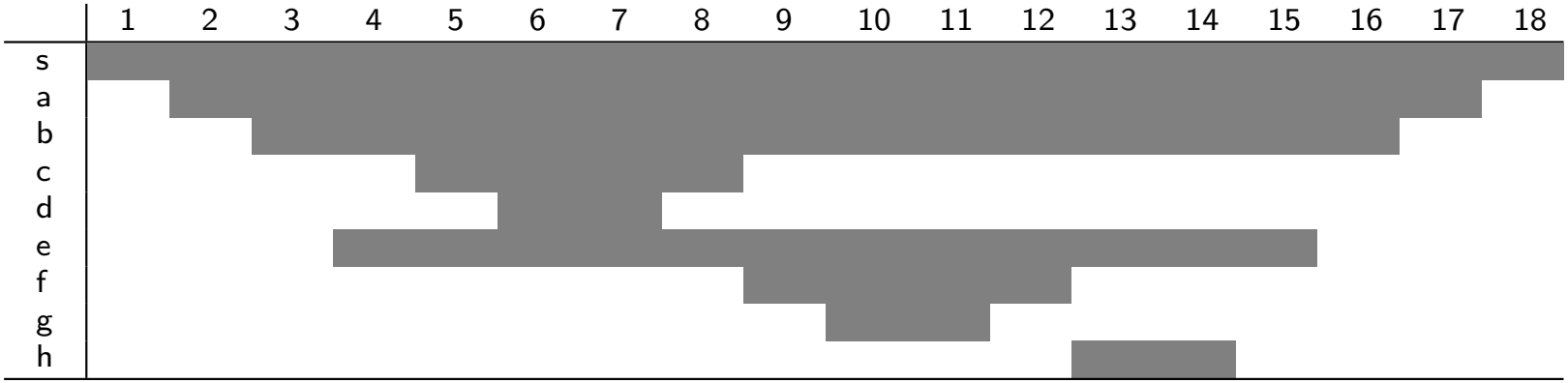
Sanity check



Times:

	d	f
s	1	18
a	2	17
b	3	16
c	5	8
d	6	7
e	4	15
f	9	12
g	10	11
h	13	14

Intervals:



White-Path Theorem

Theorem

For $u, v \in V$, v is a descendant of u if and only if at time d_u there is a White path from u to v.

White-Path Theorem

Theorem

For $u, v \in V$, v is a descendant of u if and only if at time d_u there is a White path from u to v.

Proof.

- $\Leftarrow$:
  - Let u, v be counter-example with v as close as possible to u.
  - w last White vertex (time d_u) in $u \rightarrow v$ path.
  - w descendant of $u \Rightarrow d_u < d_w < f_w < f_u$
  - Also $d_u < d_v$, as v is White at d_u

White-Path Theorem

Theorem

For $u, v \in V$, v is a descendant of u if and only if at time d_u there is a White path from u to v.

Proof.

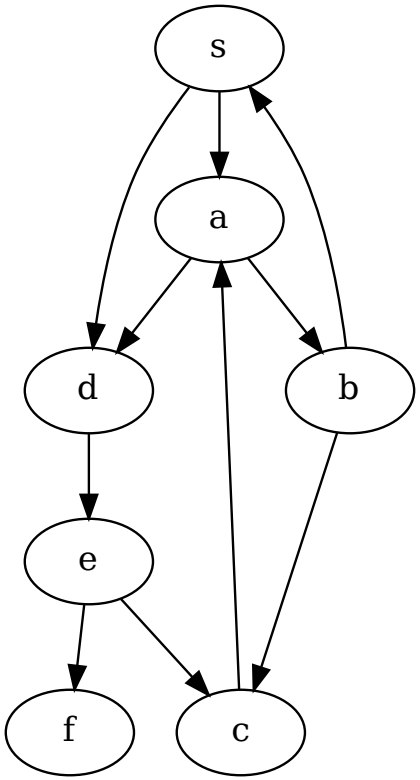
- $\Leftarrow$:
  - Let u, v be counter-example with v as close as possible to u.
  - w last White vertex (time d_u) in $u \to v$ path.
  - w descendant of $u \Rightarrow d_u < d_w < f_w < f_u$
  - Also $d_u < d_v$, as v is White at d_u
  - If $d_v > f_w$, then we would have explored v from w, contradiction!
  - If $d_v < f_w$, then u, v intervals intersect, v must be descendant of u!

Edge classification

DFS partitions the edges of a (di)graph into four types:

- Tree edges: edges which connect a vertex to its parent
- Back edges: edges which connect a vertex to its ancestor
- Forward edges: edges which connect a vertex to its descendant
- Cross edges: edges which connect two vertices without a descendant-ancestor relation

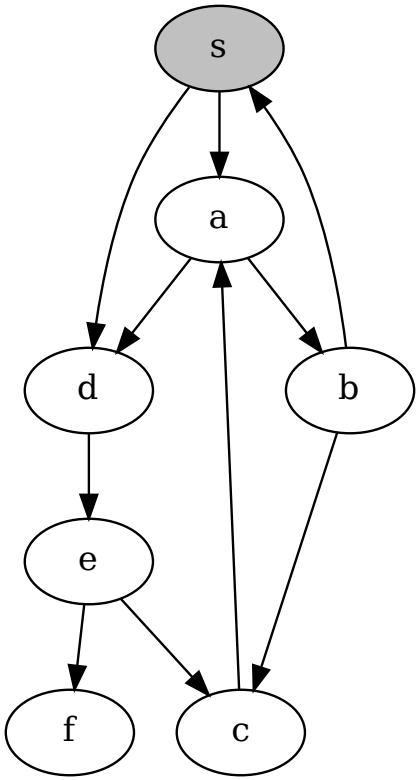
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

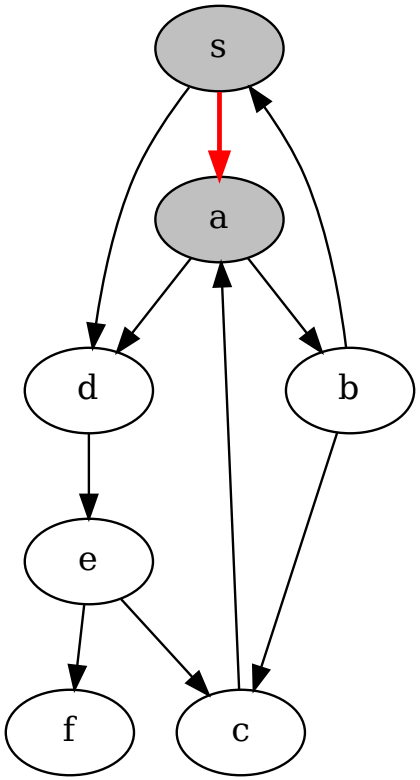
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

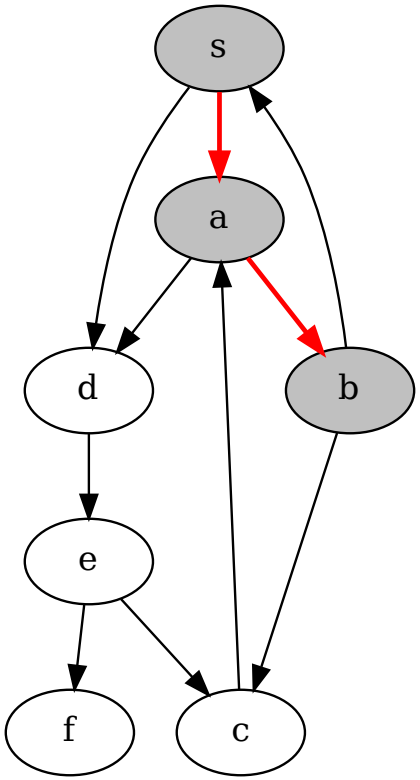
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

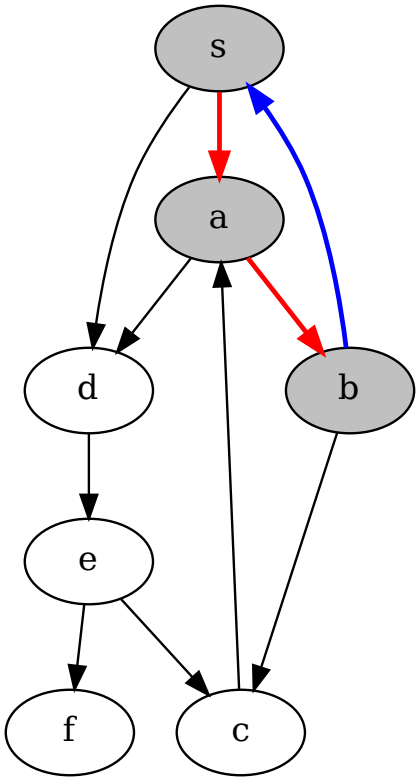
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

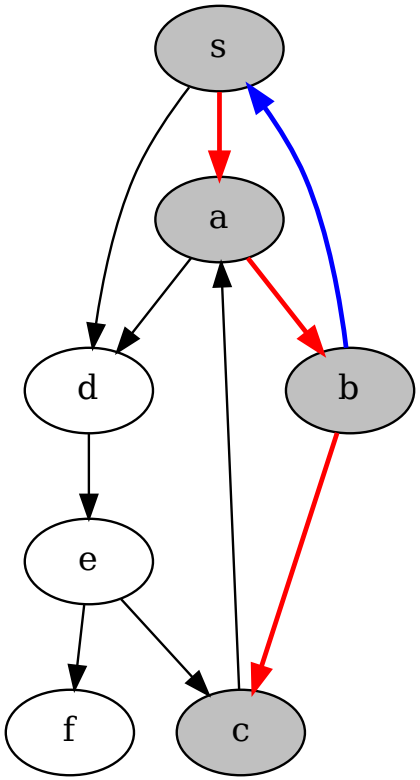
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

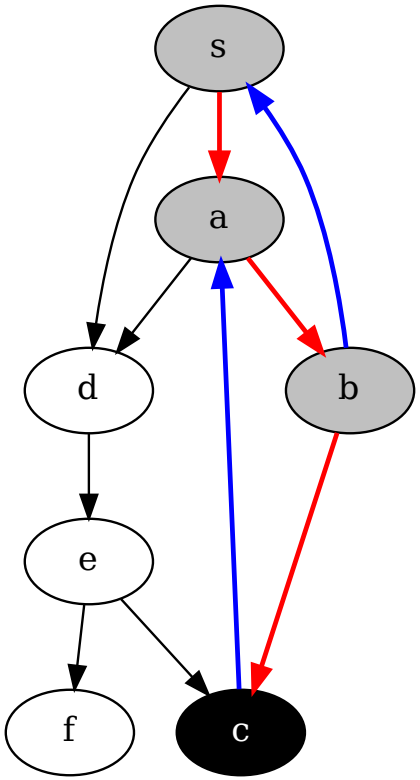
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

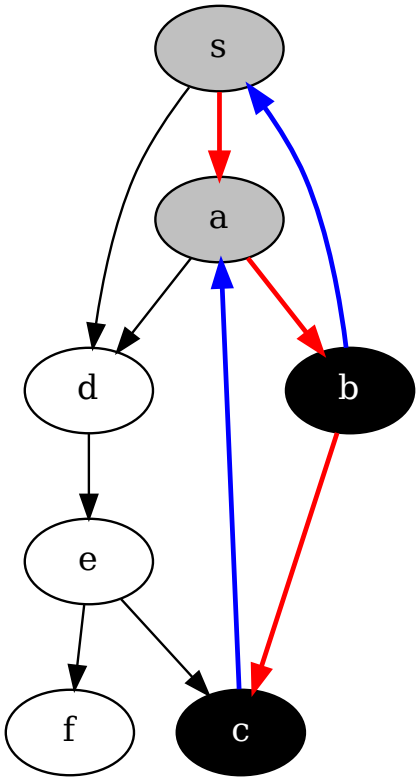
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

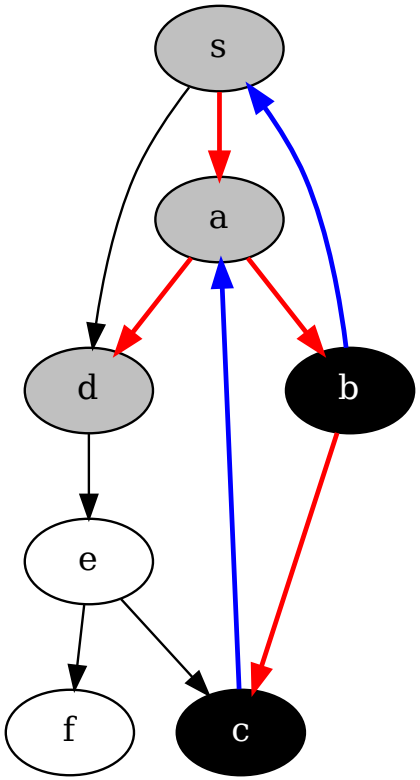
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

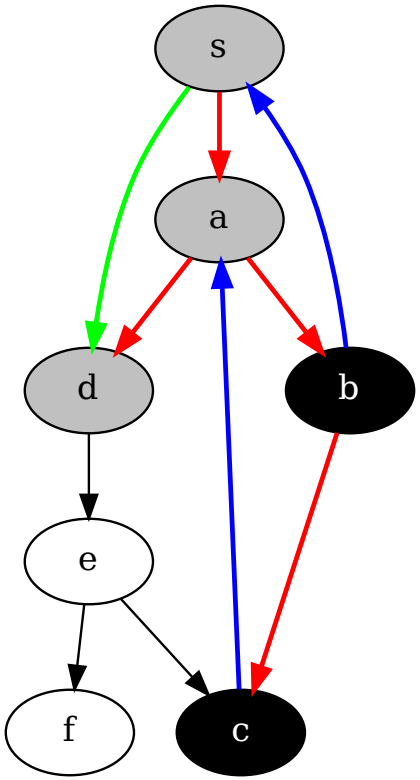
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

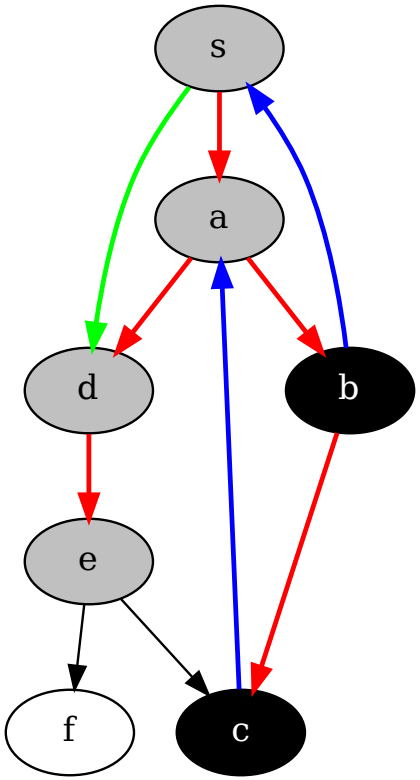
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

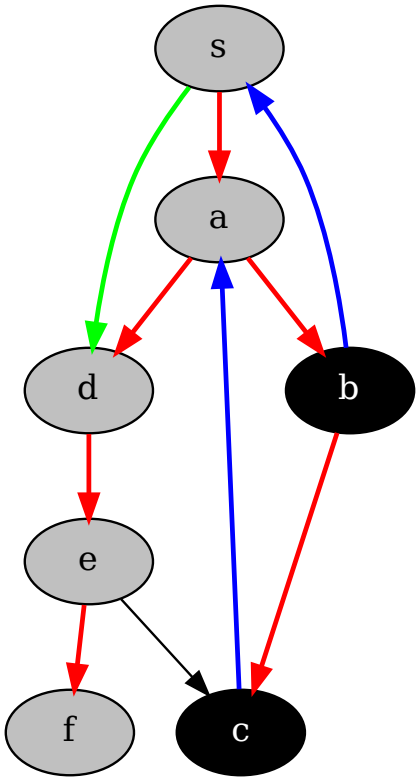
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

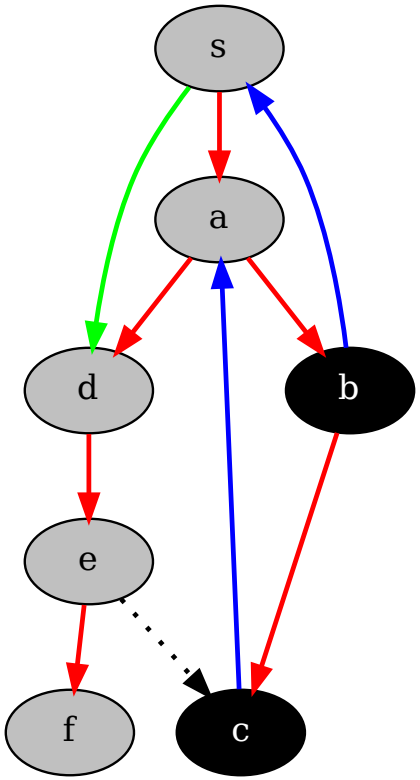
Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

Example – Digraph



Legend:

- Tree edge: Red
- Back edge: Blue
- Forward edge: Green
- Cross edge: Dotted

Edge classification

DFS partitions the edges of a (di)graph into four types:

- Tree edges: edges which connect a vertex to its parent
- Back edges: edges which connect a vertex to its ancestor
- Forward edges: edges which connect a vertex to its descendant
- Cross edges: edges which connect two vertices without a descendant-ancestor relation

If at time when uv was considered:

- v was White $\rightarrow$ Tree edge
- v was Gray $\rightarrow$ Back edge
- v was Black *to* Forward or Cross edge
  - Depending on whether v is descendant of u, which can be decided by comparing their time intervals.

Undirected DFS classification

Theorem

If G is undirected, then all edges are Tree or Back edges.

Undirected DFS classification

Theorem

If G is undirected, then all edges are Tree or Back edges.

Proof.

- Distinction between forward and back edges is not significant in undirected graphs.

Undirected DFS classification

Theorem

If G is undirected, then all edges are Tree or Back edges.

Proof.

- Distinction between forward and back edges is not significant in undirected graphs.
- For uv to be Cross, v must be Black when u was discovered.
- But, since vu is also a valid edge, before finishing v we would have visited u, so vu would have been a tree edge.

Space Considerations

Memory Usage

- BFS and DFS are **optimal** in time complexity
  - $O(n + m)$ time
  - Input has size $O(n + m)$, and we need to read all of it!
- Both algorithms use space $\Theta(n)$
  - Need to remember color of each vertex (White, Gray, Black)
- Optimal space?

Memory Usage

- BFS and DFS are **optimal** in time complexity
  - $O(n+m)$ time
  - Input has size $O(n+m)$, and we need to read all of it!
- Both algorithms use space $\Theta(n)$
  - Need to remember color of each vertex (White, Gray, Black)
- Optimal space?
- Possible argument: input already takes $\Theta(n+m)$ space, so using $O(n)$ extra space is insignificant.

Memory Usage

- BFS and DFS are **optimal** in time complexity
  - $O(n+m)$ time
  - Input has size $O(n+m)$, and we need to read all of it!
- Both algorithms use space $\Theta(n)$
  - Need to remember color of each vertex (White, Gray, Black)
- Optimal space?
- Possible argument: input already takes $\Theta(n+m)$ space, so using $O(n)$ extra space is insignificant.
  - **Not convincing!** Input could be given in some implicit form.
  - We care about minimizing the **working** space we use.

Reachability

Problem (Reachability)

Given (di)graph G, two vertices s, t, is there a path from s to t in G?

Reachability

Problem (Reachability)

Given (di)graph G, two vertices s, t, is there a path from s to t in G?

Can reachability be solved using less than $\Theta(n)$ space?

Savitch's theorem

```
1:  procedure REACH-IN-k(G,s,t,k)
2:      if k < 0 then
3:          Return No
4:      end if
5:      if s = t or (st ∈ E and k > 0) then
6:          Return Yes
7:      end if
8:      for v ∈ V \ {s, t} do
9:          if Reach-in-k(G,s,v,⌈k/2⌉) then
10:             if Reach-in-k(G,v,t,⌊k/2⌋) then
11:                 Return Yes
12:             end if
13:         end if
14:     end for
15:     Return No
16: end procedure
```

Savitch's theorem – Analysis

- Correctness is straightforward (induction on k).
- Space complexity $S(k)$:
  - Total space needed for local variables: $O(\log n)$
  - Recursive calls: $S(k/2)$
  - $\Rightarrow S(k) \leq O(\log n) + S(k/2) \leq \ldots O(\log k \log n)$
  - $k \leq n \Rightarrow S(k) = O(\log^2 n)$
  - Great!

Savitch's theorem – Analysis

- Correctness is straightforward (induction on k).
- Space complexity $S(k)$:
  - Total space needed for local variables: $O(\log n)$
  - Recursive calls: $S(k/2)$
  - $\Rightarrow S(k) \leq O(\log n) + S(k/2) \leq \ldots O(\log k \log n)$
  - $k \leq n \Rightarrow S(k) = O(\log^2 n)$
  - Great!
- Time complexity $T(k)$:
  - Iterations of local loop: n
  - Recursive calls per iteration: $2\,T(k/2)$
  - $\Rightarrow T(k) \leq 2nT(k/2) \leq \ldots (2n)^{\log k}$
  - $k \leq n \Rightarrow T(k) = 2^{O(\log^2 n)} = n^{O(\log n)}$
  - Ouch!

L=NL?

- (Di)graph reachability can be solved in linear time and space.
- It can be solved in much less (poly-logarithmic) space, but only if we accept super-polynomial time (as far as we know so far).
- Can poly-time and poly-log space be achieved simultaneously?
  - One of the most notorious open problems in TCS!
  - L=NL?
  - (Compare with more famous P=NP problem...)

L=NL?

- (Di)graph reachability can be solved in linear time and space.
- It can be solved in much less (poly-logarithmic) space, but only if we accept super-polynomial time (as far as we know so far).
- Can poly-time and poly-log space be achieved simultaneously?
  - One of the most notorious open problems in TCS!
  - L=NL?
  - (Compare with more famous P=NP problem...)
- What if we make the problem a little easier?
  - Reachability of **undirected** graphs, can be decided in $O(\log n)$ space and $n^{O(1)}$ (**randomized**) time.

A random walk procedure

```
1:  procedure REACH(G, s, t)
2:      count←0
3:      cur←s
4:      while count≤ n^4 do
5:          if cur=t then
6:              Output Yes
7:          end if
8:          count++
9:          cur← Rand(N(cur))
10:     end while
11:     Output No
12: end procedure
```

(Partial) analysis

Random walk procedure:

- Is always correct if no path exists.
- Is correct with high probability $(1 - o(1))$ if a path exists.
  - Proof beyond the scope of this course.
  - Idea: expected cover time: $O(n^3)$ for any undirected graph.
- Runs in time $O(n^4)$ and space $O(\log n)$.

(Partial) analysis

Random walk procedure:

- Is always correct if no path exists.
- Is correct with high probability $(1 - o(1))$ if a path exists.
  - Proof beyond the scope of this course.
  - Idea: expected cover time: $O(n^3)$ for any undirected graph.
- Runs in time $O(n^4)$ and space $O(\log n)$.
- Does **NOT** work for directed graphs!
  - L=NL problem still open!