CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
Capen 208

Lecture 19-20
Graph Traversals

Announcements

This Week

- Recitation meets as normal
- My office hours are scheduled as normal
- No assignments
- No TA office hours

Next Week

- PA2 releases
- All OH back to normal

Graph Connectivity

Some Common Questions

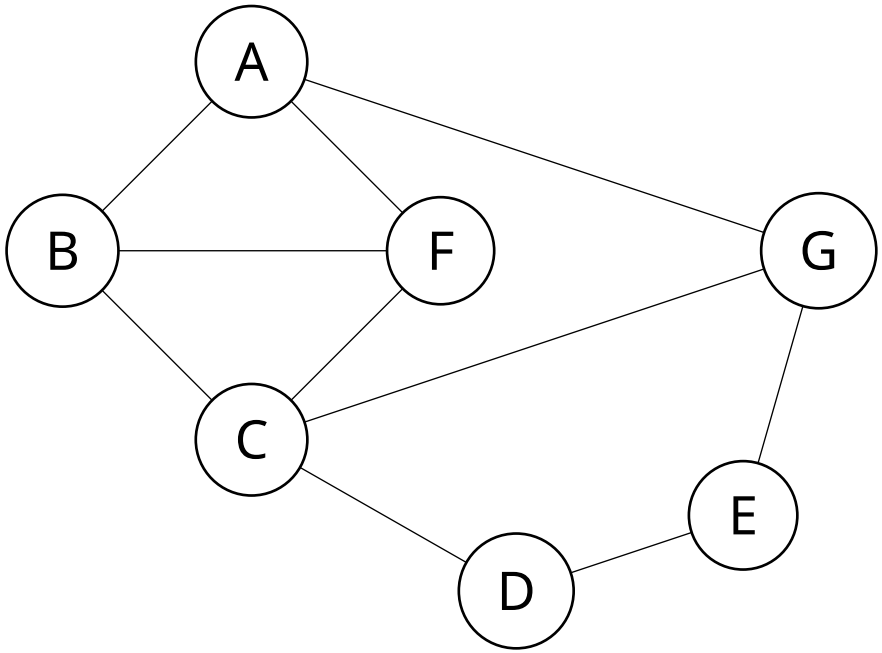
Given a Graph, G:

- Is vertex **u** adjacent to vertex **v**?
- Is vertex **u** connected to vertex **v** via some path?
- Which vertices are connected to vertex **v**?
- What is the *shortest path* from vertex **u** to vertex **v**?

Subgraphs

Subgraph:

A subgraph of graph $G = (V, E)$ is a graph, $G' = (V', E')$ where $V' \subset V$ and $E' \subset E$

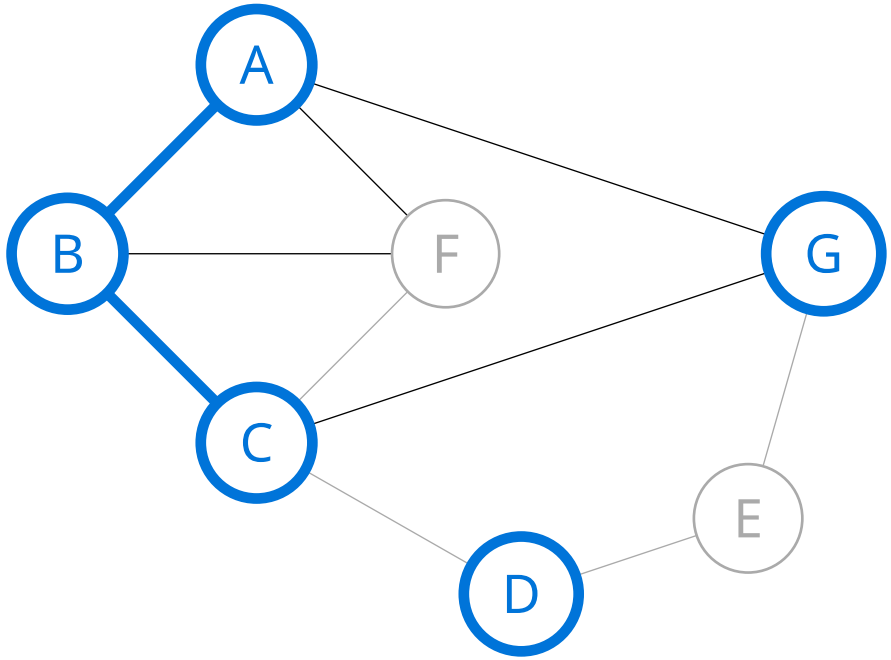


Subgraphs

Subgraph:

A subgraph of graph $G = (V, E)$ is a graph, $G' = (V', E')$ where $V' \subset V$ and $E' \subset E$

$$V' = \{A, B, C, D, G\}$$
$$E' = \{(A, B), (B, C), (A, G), (C, G)\}$$

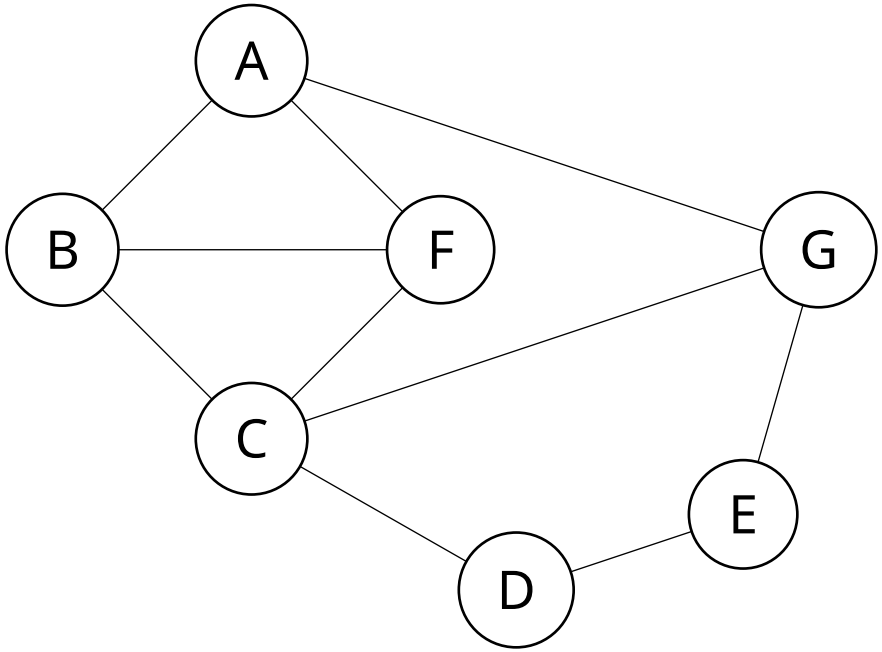


Subgraphs

Subgraph:

A subgraph of graph $G = (V, E)$ is a graph, $G' = (V', E')$ where $V' \subset V$ and $E' \subset E$

$$V' = \{A, B, C, D, G\}$$
$$E' = \{(A, B), (B, C), (A, G), (C, G)\}$$



Spanning Subgraph:

A spanning subgraph of G is a subgraph of G that contains ALL the vertices of G.

$$V' = \{A, B, C, D, E, F, G\}$$
$$E' = \{(A, B), (B, C), (A, G), (C, G), (D, E)\}$$

Subgraphs

Subgraph:

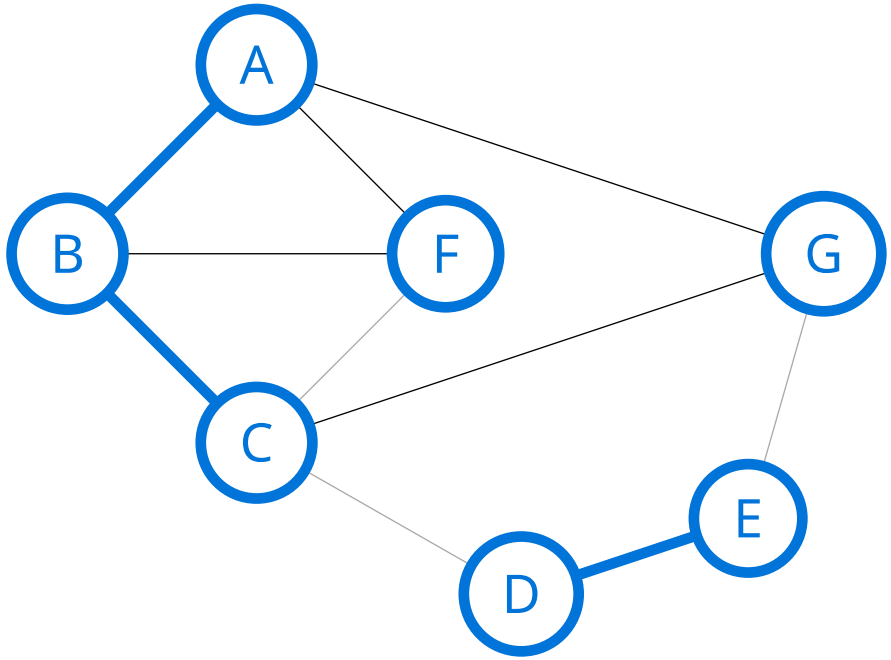
A subgraph of graph $G = (V, E)$ is a graph, $G' = (V', E')$ where $V' \subset V$ and $E' \subset E$

$$V' = \{A, B, C, D, G\}$$
$$E' = \{(A, B), (B, C), (A, G), (C, G)\}$$

Spanning Subgraph:

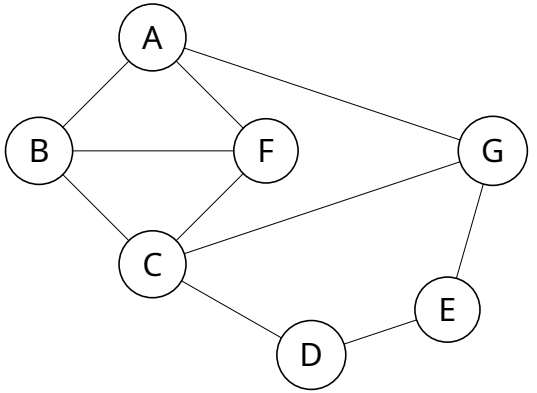
A spanning subgraph of G is a subgraph of G that contains ALL the vertices of G.

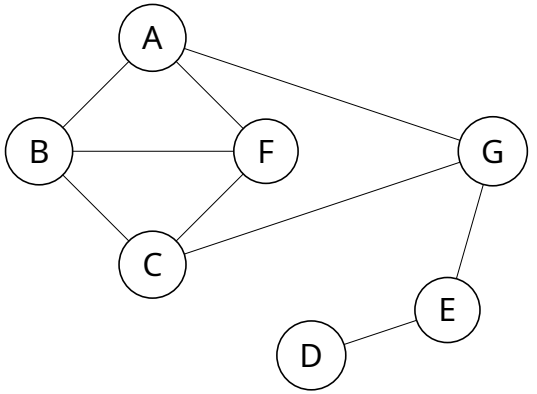
$$V' = \{A, B, C, D, E, F, G\}$$
$$E' = \{(A, B), (B, C), (A, G), (C, G), (D, E)\}$$



Connected Components

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

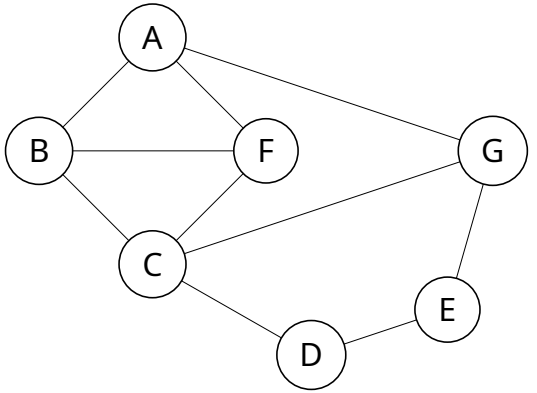


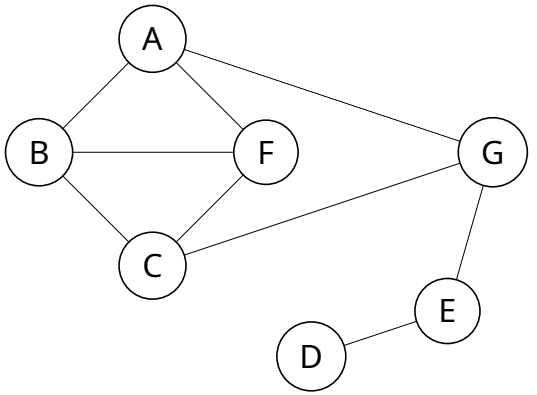


Connected Components

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

The top graph is connected, bottom graph is not





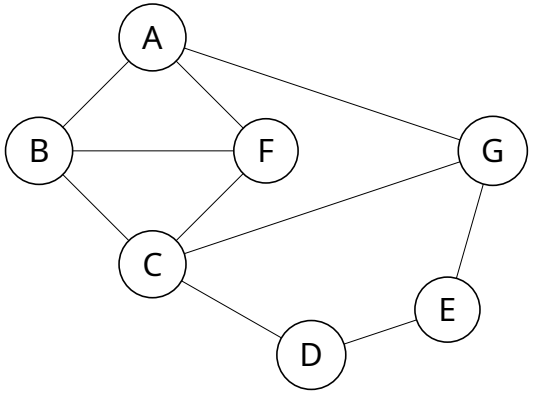
Connected Components

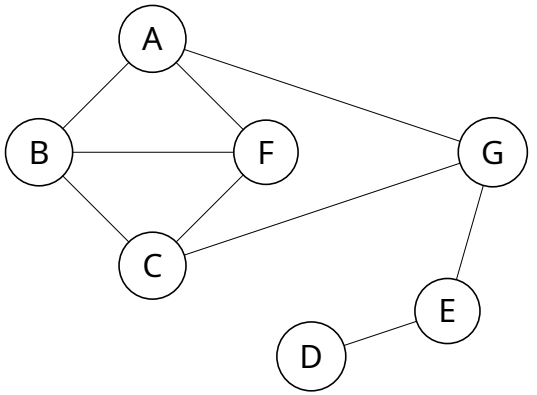
An undirected graph is **connected** if there is a path between every pair of vertices.

The top graph is connected, bottom graph is not

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

- Maximal means you cannot add anything more to it without breaking the property
- A connected graph by definition has one connected component





Connected Components

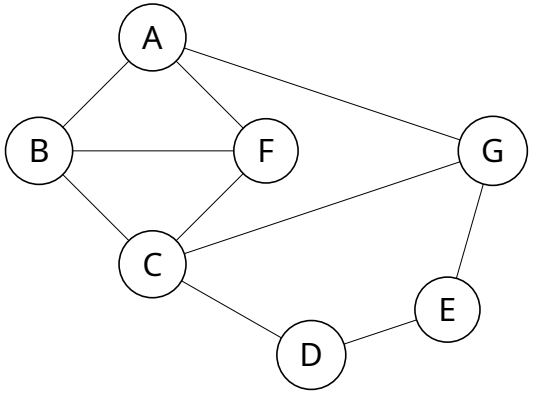
An undirected graph is **connected** if there is a path between every pair of vertices.

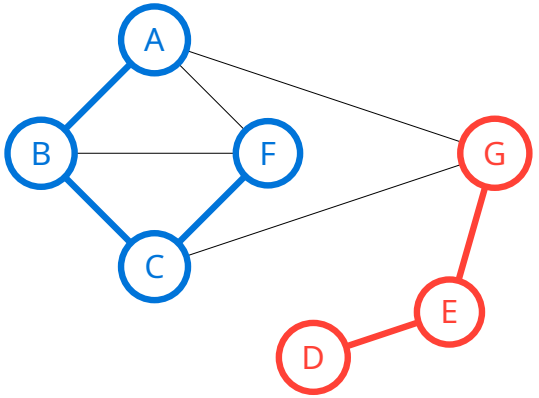
The top graph is connected, bottom graph is not

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

- Maximal means you cannot add anything more to it without breaking the property
- A connected graph by definition has one connected component

The bottom graph has two connected components





Trees

A **free tree** is an undirected graph, T, such that...

- There is **exactly** one simple path between each pair of nodes
  - This means that T is connected and T has no cycles

Trees

A **free tree** is an undirected graph, T, such that...

- There is **exactly** one simple path between each pair of nodes
  - This means that T is connected and T has no cycles

A **rooted tree** is a directed graph T such that...

- One vertex of T is the *root*
- There is exactly one simple path from the root to every other vertex

Trees

A **free tree** is an undirected graph, T, such that...

- There is **exactly** one simple path between each pair of nodes
  - This means that T is connected and T has no cycles

A **rooted tree** is a directed graph T such that...

- One vertex of T is the *root*
- There is exactly one simple path from the root to every other vertex

A **(free/rooted) forest** is a graph F such that...

- Every connected component is a (free/rooted) tree

Spanning Trees

A **Spanning Tree** of a connected graph is a *spanning subgraph* that is also a tree

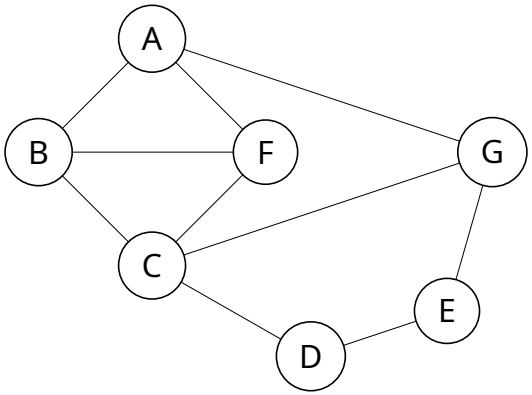
- It is not unique unless the graph itself is also a tree

Spanning Trees

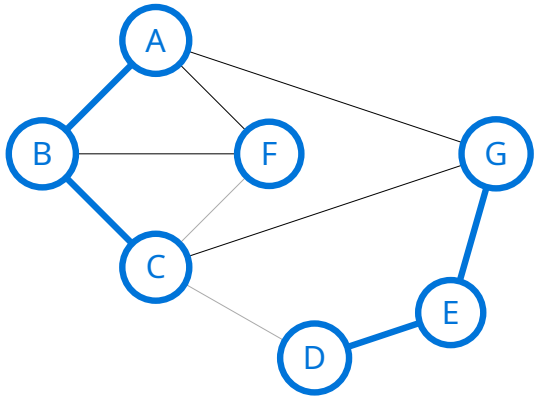
A **Spanning Tree** of a connected graph is a *spanning subgraph* that is also a tree

- It is not unique unless the graph itself is also a tree

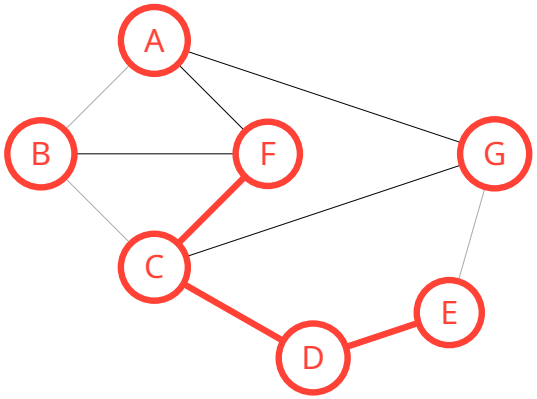
Graph G



A Spanning Tree of G



Another Spanning Tree of G



Depth-First Search (Recursive)

Depth-First Search Goals

When we perform DFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G
- Construct a **spanning tree** for every connected component of G

Depth-First Search Goals

When we perform DFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected

Depth-First Search Goals

When we perform DFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components

Depth-First Search Goals

When we perform DFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components
  - **Side Effect:** Compute paths between all connected vertices

Depth-First Search Goals

When we perform DFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components
  - **Side Effect:** Compute paths between all connected vertices
  - **Side Effect:** Identify cycles

Depth-First Search Goals

When we perform DFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components
  - **Side Effect:** Compute paths between all connected vertices
  - **Side Effect:** Identify cycles

Secondary Goal

- Perform the traversal in linear time, $O(|V| + |E|)$

Basic Design

Top-Level Driver Function: `DFS(G)`

Input: Graph $G = (V, E)$

Output: A labeling of every edge as either:

- **Spanning Edge:** Part of the spanning tree
- **Back Edge:** Part of a cycle

Basic Design

Top-Level Driver Function: `DFS(G)`

Input: Graph $G = (V, E)$

Output: A labeling of every edge as either:

- **Spanning Edge:** Part of the spanning tree
- **Back Edge:** Part of a cycle

Helper Function: `DFSOne(G, v)`

Input: Graph $G = (V, E)$, starting vertex $v \in V$

Output: A labeling of every edge in the connected component of v

DFS Algorithm

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          DFSOne(G, v)
```

DFS Algorithm

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          DFSOne(G, v)
```

Start with everything unexplored

DFS Algorithm

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          DFSOne(G, v)
```

Make sure to check every vertex

DFS Algorithm

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          DFSOne(G, v)
```

Explore the connected component of unvisited vertices

Remember: **`DFSOne`** explores the entire connected component of **`v`**

It will mark vertices as **`VISITED`** and label edges as **`SPANNING`** or **`BACK`**

`DFS` ensures we visit every vertex

DFSOne Algorithm

```
 1 Input: Graph G, Vertex v
 2
 3 v.label = VISITED
 4 for e in v.outgoingEdges():
 5     if e.label == UNEXPLORED:
 6         w = e.dest
 7         if w.label == VISITED:
 8             e.label = BACK
 9         else:
10             e.label = SPANNING
11             DFSOne(G, w)
```

DFSOne Algorithm

```
1  Input: Graph G, Vertex v
2
3  v.label = VISITED
4  for e in v.outgoingEdges():
5      if e.label == UNEXPLORED:
6          w = e.dest
7          if w.label == VISITED:
8              e.label = BACK
9          else:
10             e.label = SPANNING
11             DFSOne(G, w)
```

Mark the vertex visited
(so we won't explore it again later)

DFSOne Algorithm

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

Check every edge leaving **v**

DFSOne Algorithm

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

Follow unexplored edges

DFSOne Algorithm

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

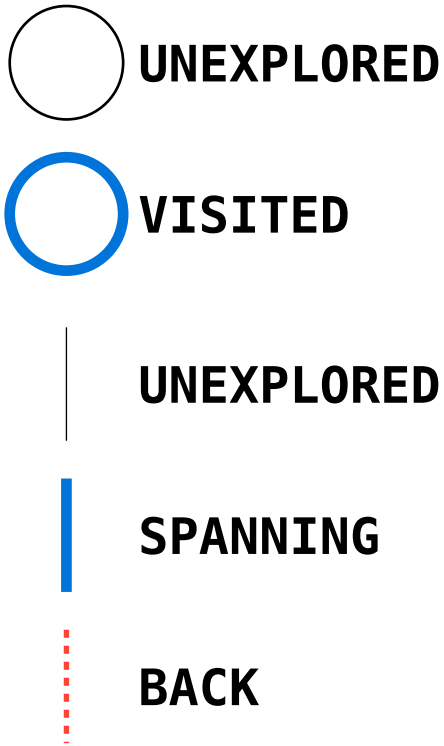
If they lead to a vertex we've already seen, it's a **BACK** edge... we just found a cycle

DFSOne Algorithm

```
1  Input: Graph G, Vertex v
2
3  v.label = VISITED
4  for e in v.outgoingEdges():
5      if e.label == UNEXPLORED:
6          w = e.dest
7          if w.label == VISITED:
8              e.label = BACK
9          else:
10             e.label = SPANNING
11             DFSOne(G, w)
```

If they lead to a new vertex, it's a **SPANNING** edge... **Immediately** explore the new vertex

DFS Example (Recursive Version)



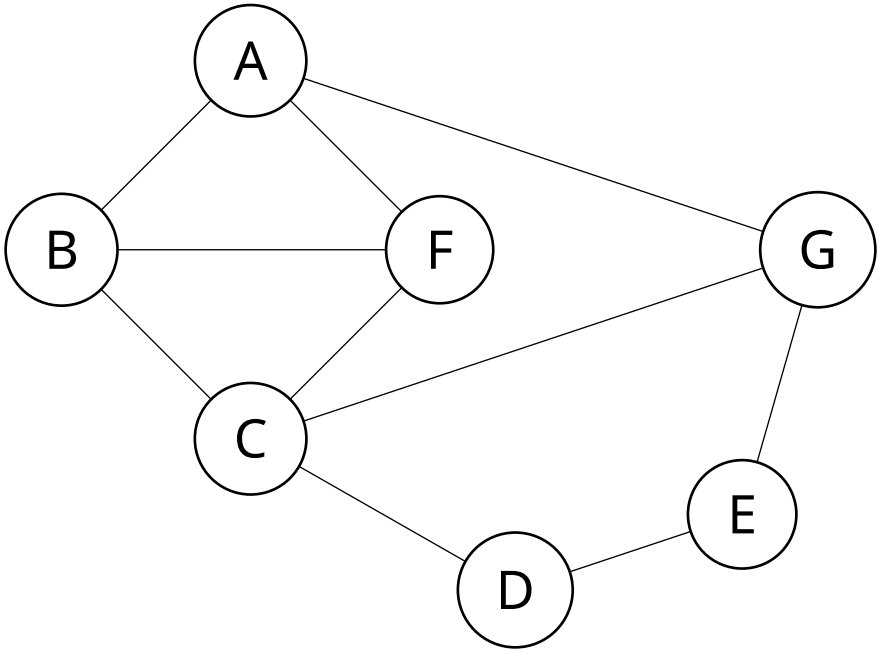
Current Vertex:

Edges To Check

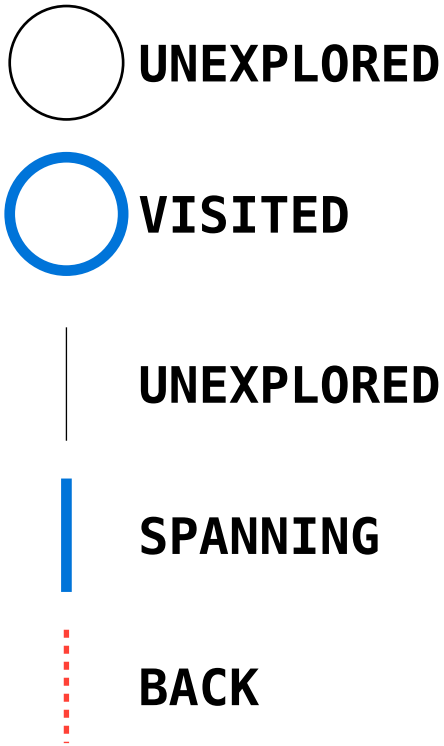
`[]`

Call Stack

`DFS(G)`



DFS Example (Recursive Version)



Current Vertex: A

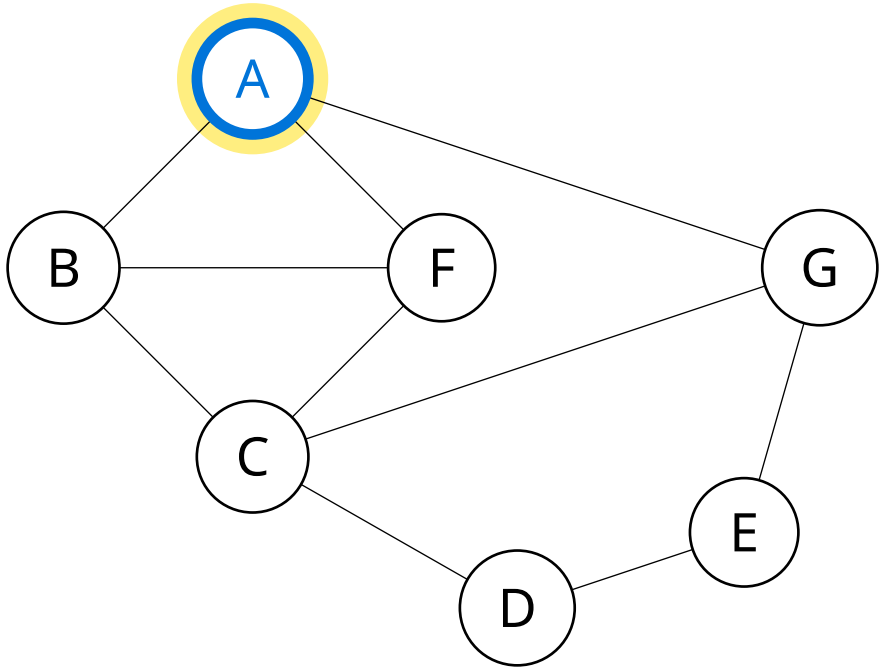
Edges To Check

`[B,F,G]`

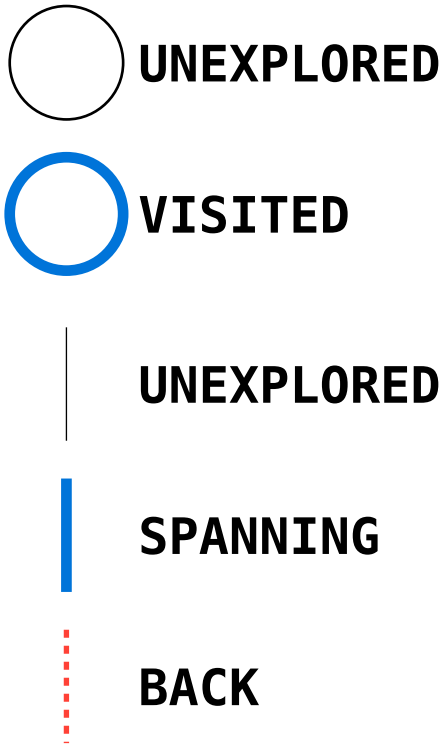
Call Stack

`DFS(G)`

`DFSOne(G,A)`



DFS Example (Recursive Version)



Current Vertex: B

Edges To Check

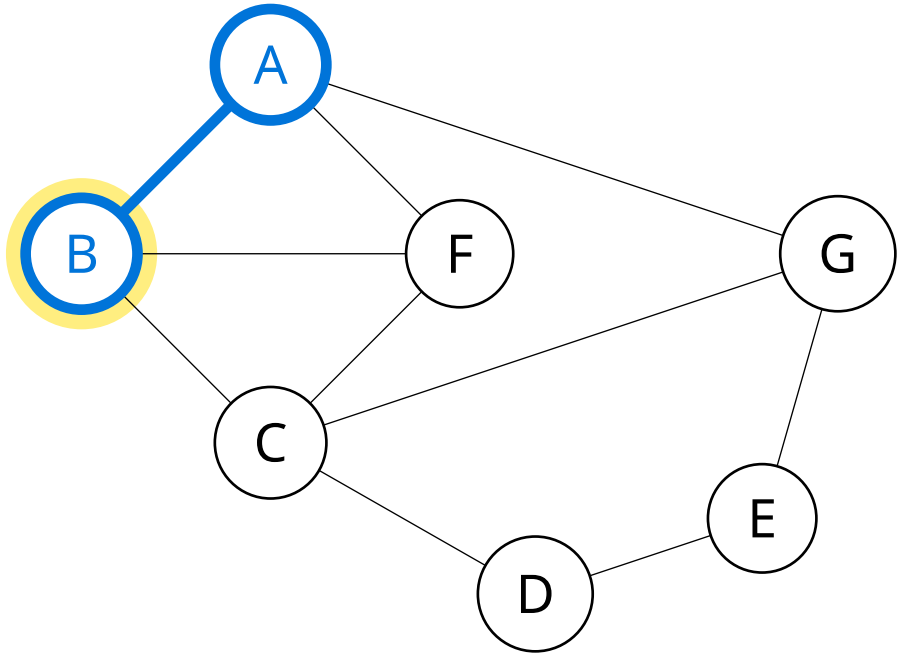
`[C,F]`

Call Stack

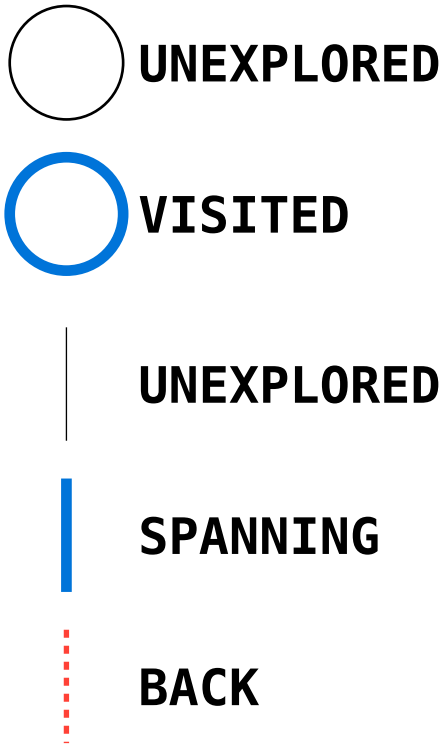
`DFS(G)`

`DFSOne(G,A)`

`DFSOne(G,B)`



DFS Example (Recursive Version)



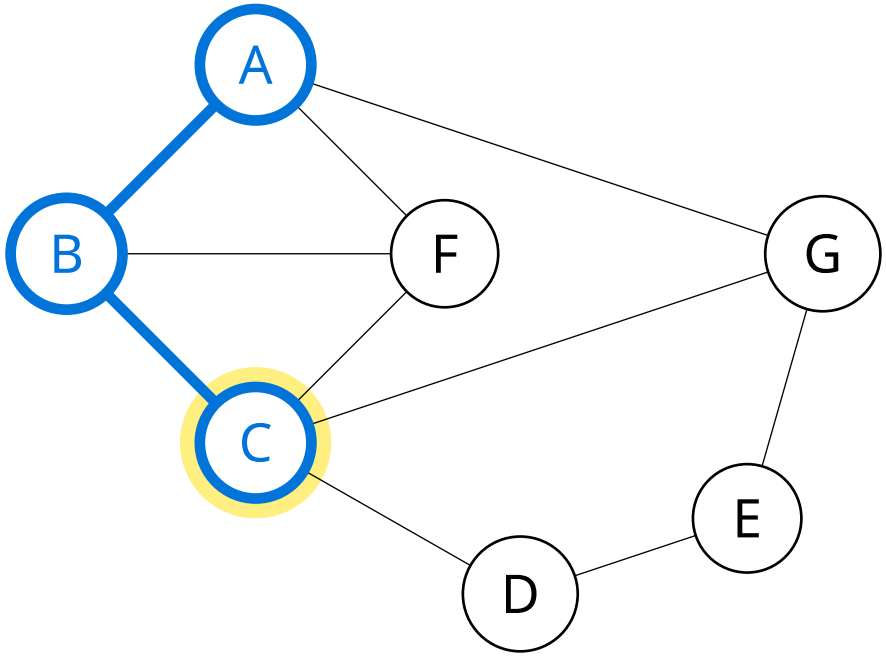
Current Vertex: C

Edges To Check

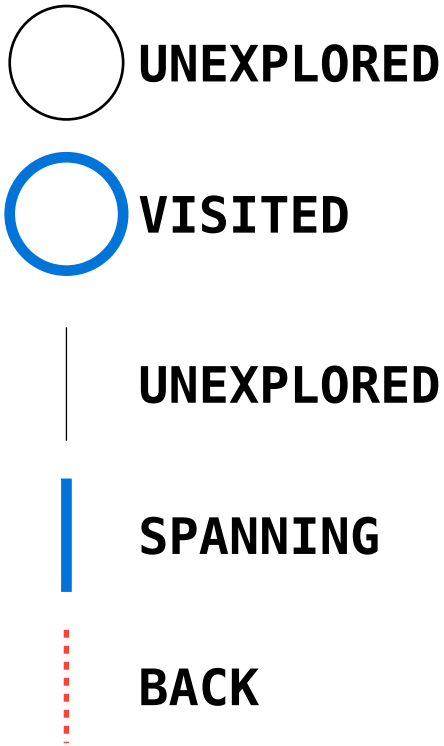
[D,F,G]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)



DFS Example (Recursive Version)



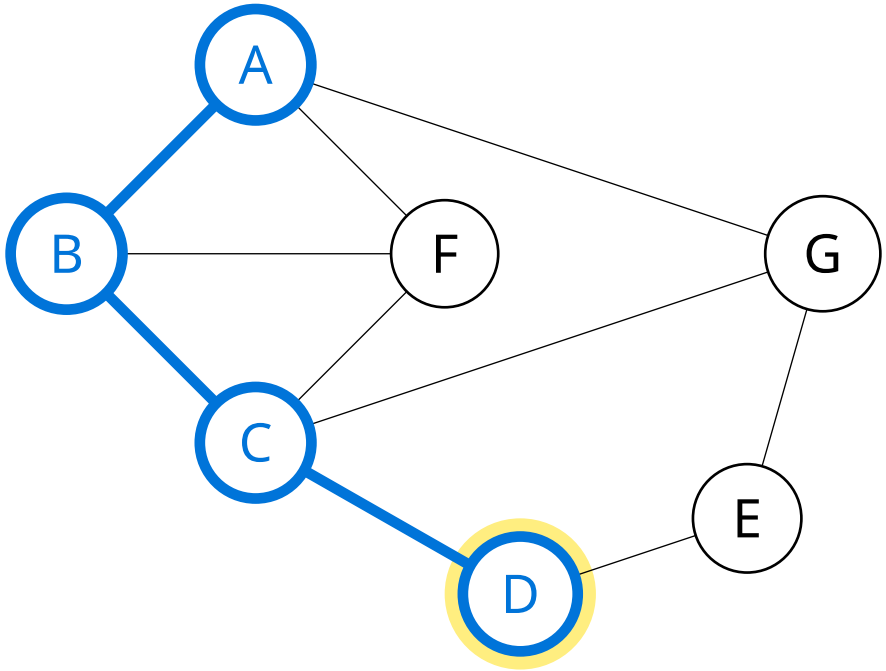
Current Vertex: D

Edges To Check

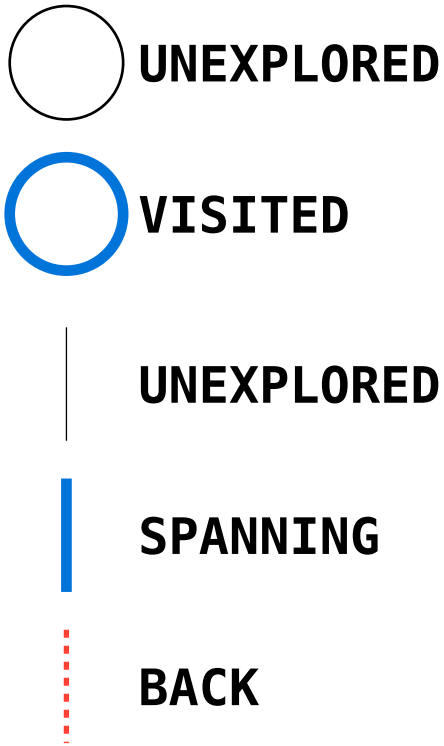
[E]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,D)



DFS Example (Recursive Version)



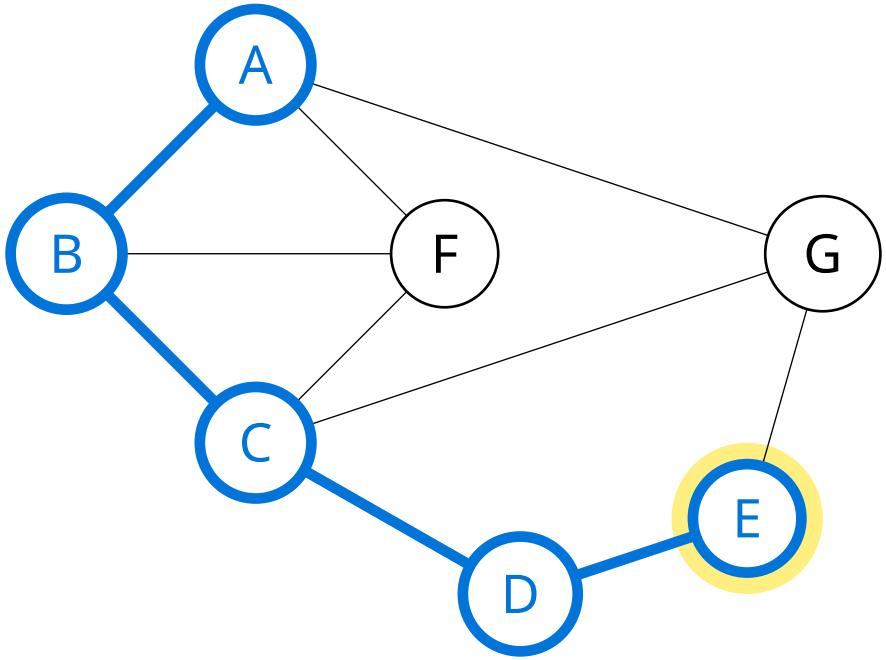
Current Vertex: E

Edges To Check

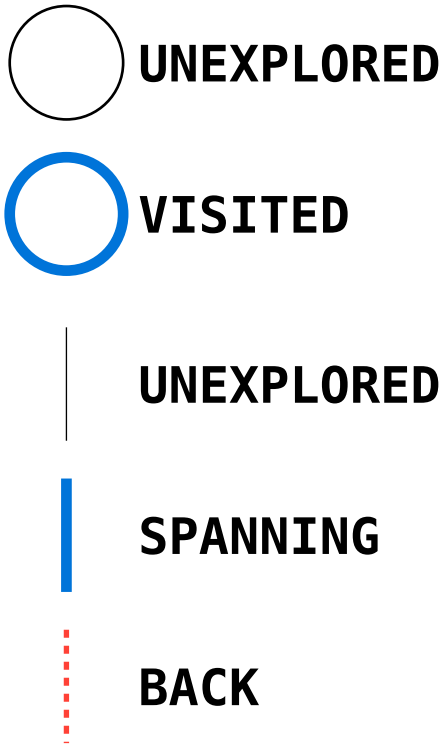
`[G]`

Call Stack

```
DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,D)
DFSOne(G,E)
```



DFS Example (Recursive Version)



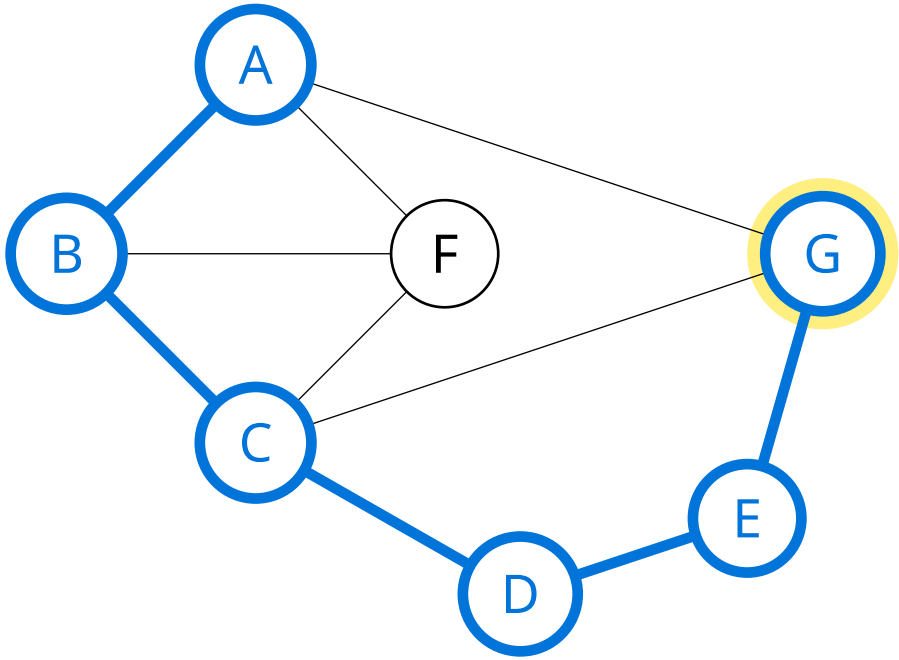
Current Vertex: G

Edges To Check

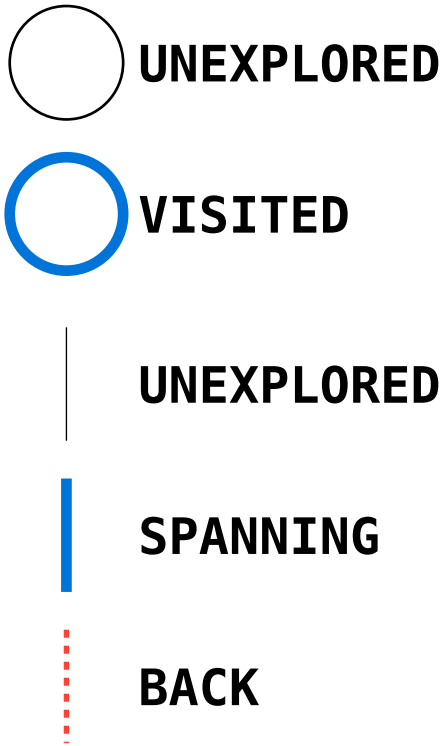
[A,C]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,D)
DFSOne(G,E)
DFSOne(G,G)



DFS Example (Recursive Version)



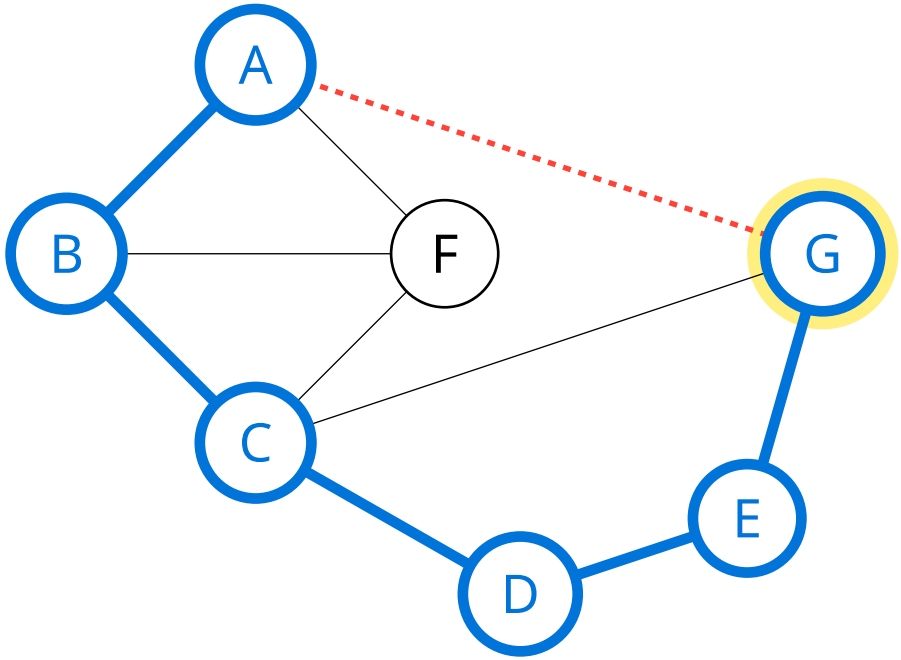
Current Vertex: G

Edges To Check

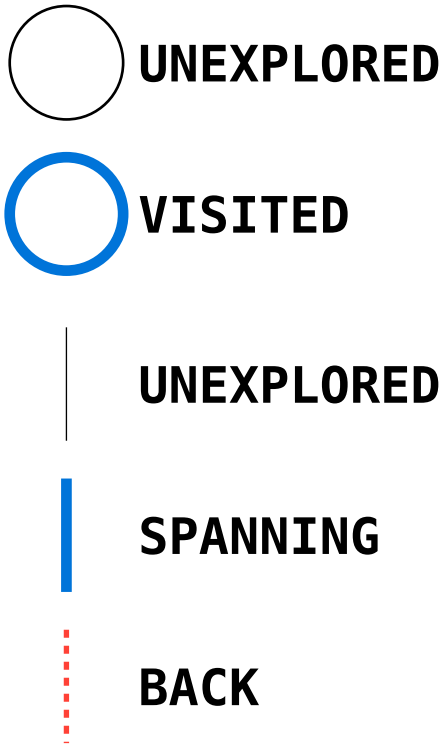
[C]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,D)
DFSOne(G,E)
DFSOne(G,G)



DFS Example (Recursive Version)



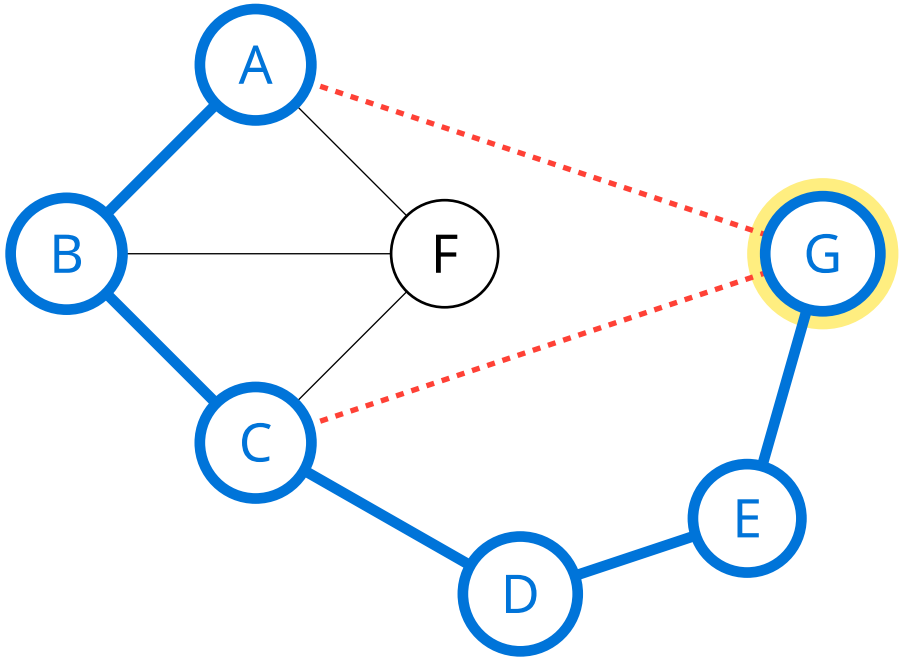
Current Vertex: G

Edges To Check

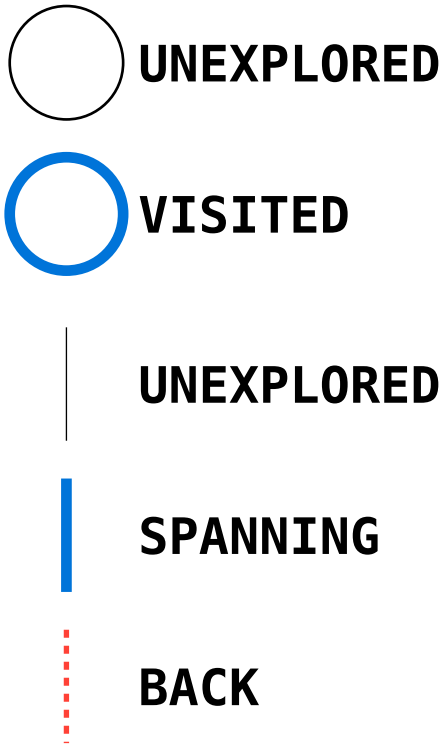
[]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,D)
DFSOne(G,E)
DFSOne(G,G)



DFS Example (Recursive Version)



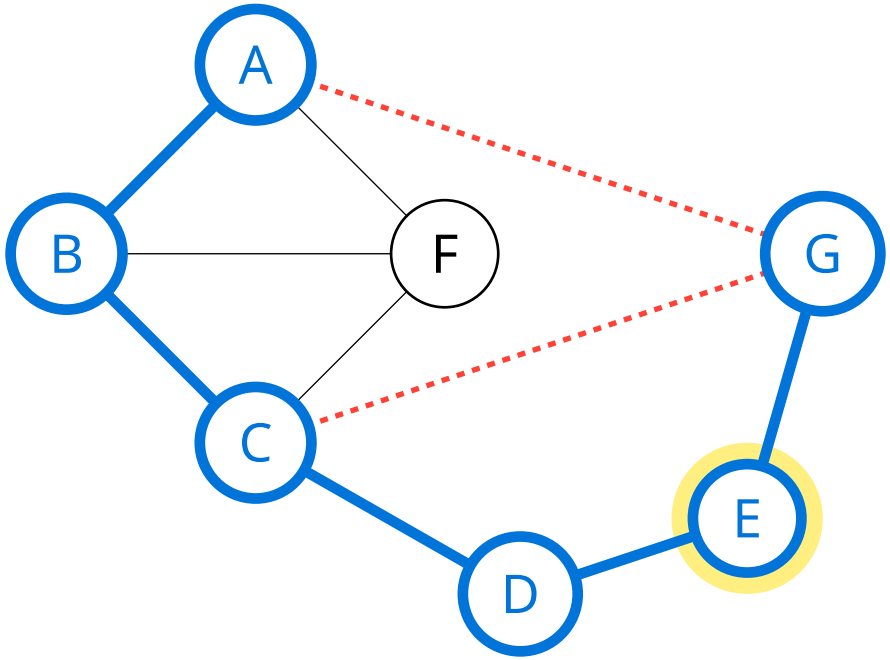
Current Vertex: E

Edges To Check

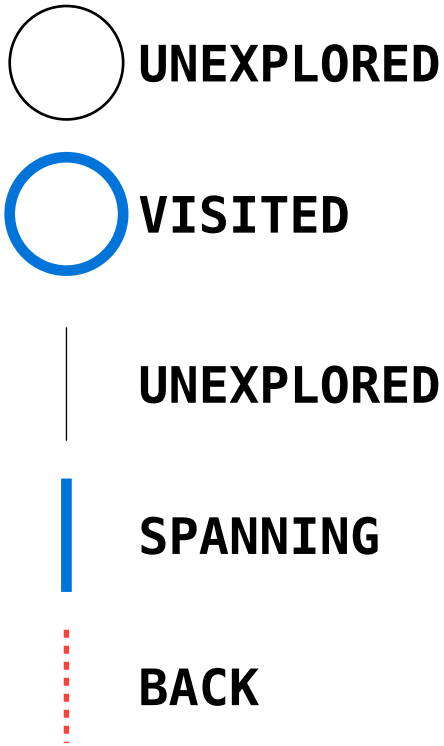
`[]`

Call Stack

```
DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,D)
DFSOne(G,E)
```



DFS Example (Recursive Version)



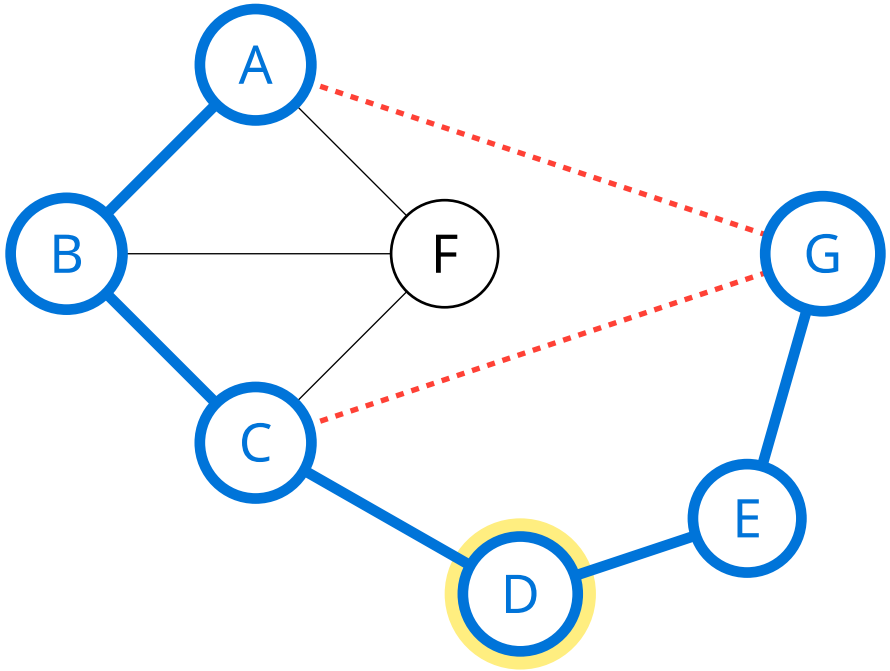
Current Vertex: D

Edges To Check

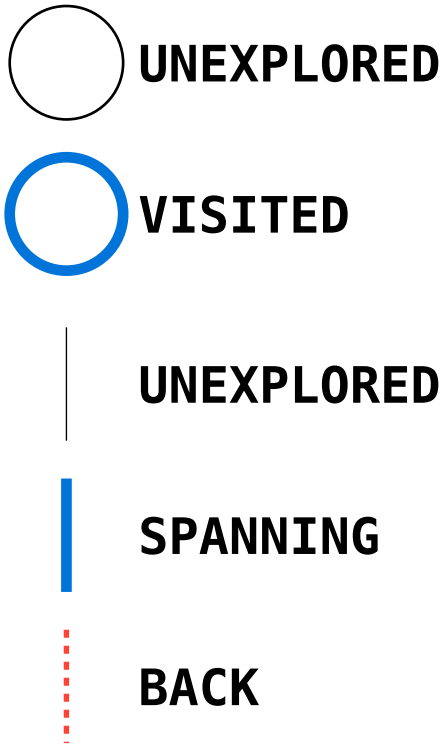
[]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,D)



DFS Example (Recursive Version)



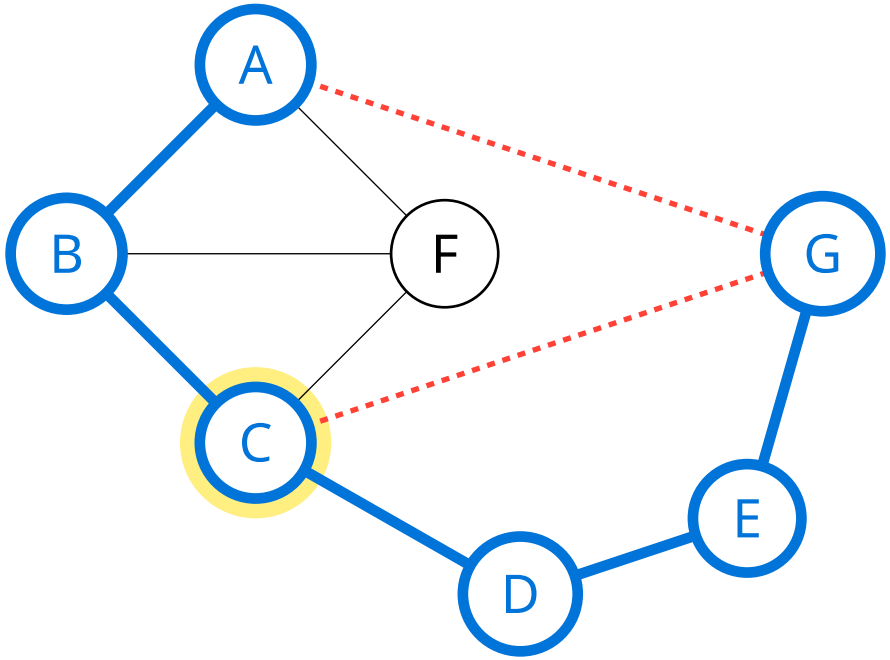
Current Vertex: C

Edges To Check

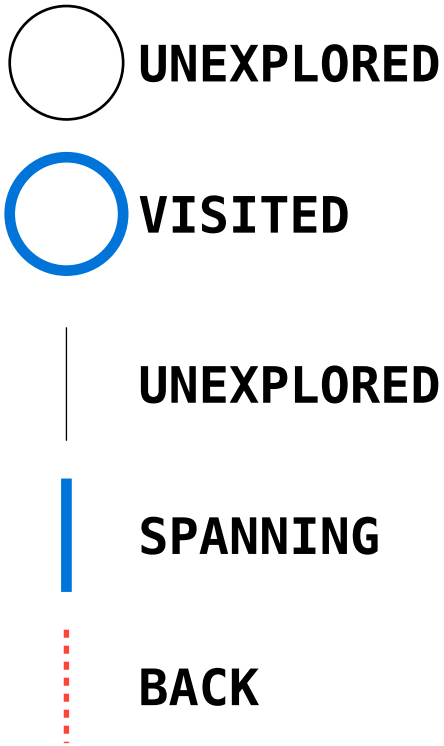
[F]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)



DFS Example (Recursive Version)



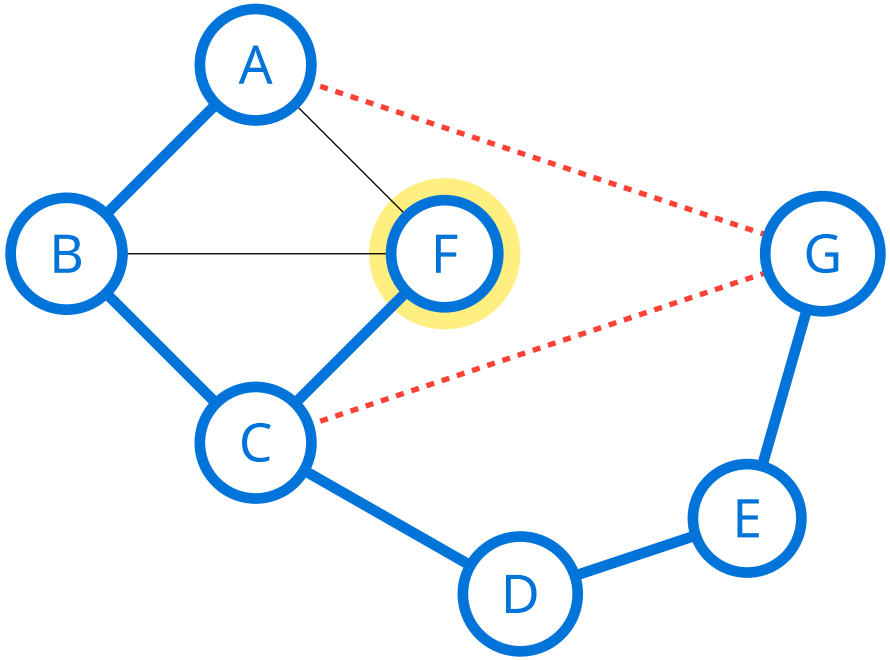
Current Vertex: F

Edges To Check

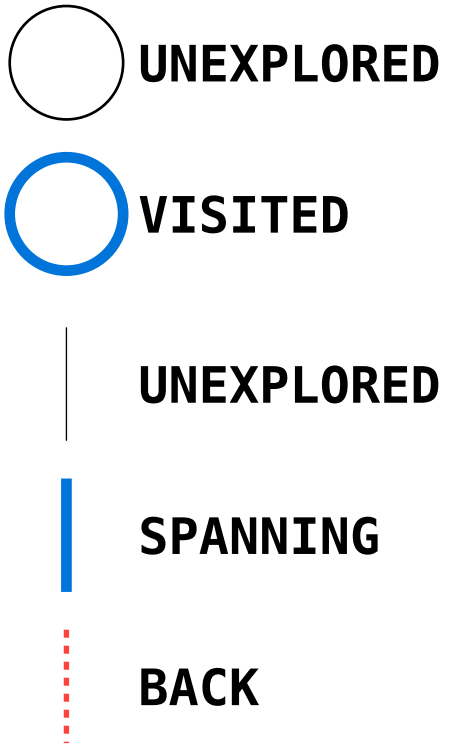
`[A,B]`

Call Stack

```
DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,F)
```



DFS Example (Recursive Version)



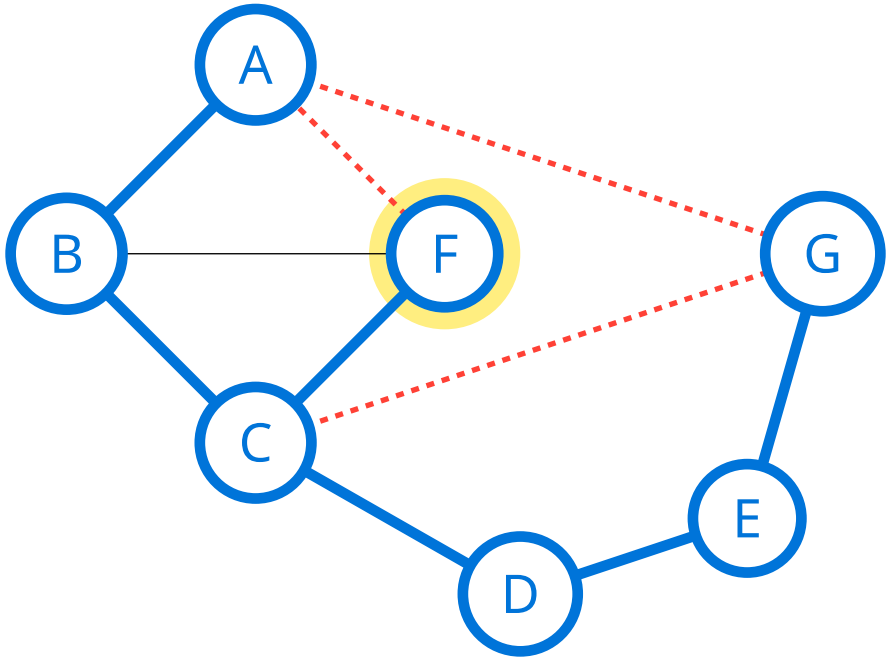
Current Vertex: F

Edges To Check

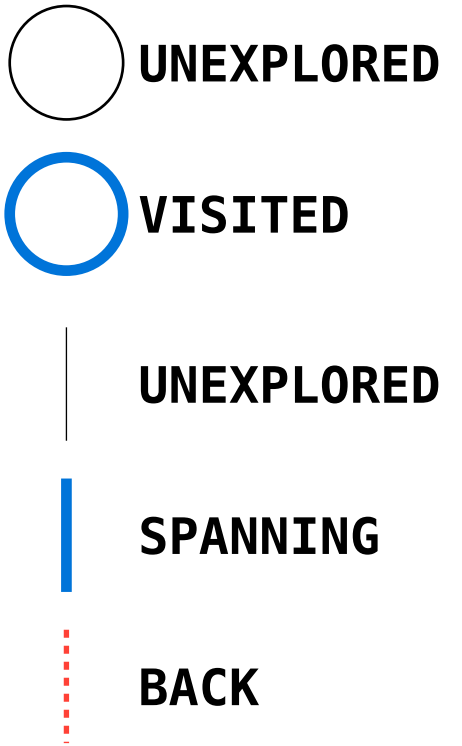
`[B]`

Call Stack

```
DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,F)
```



DFS Example (Recursive Version)



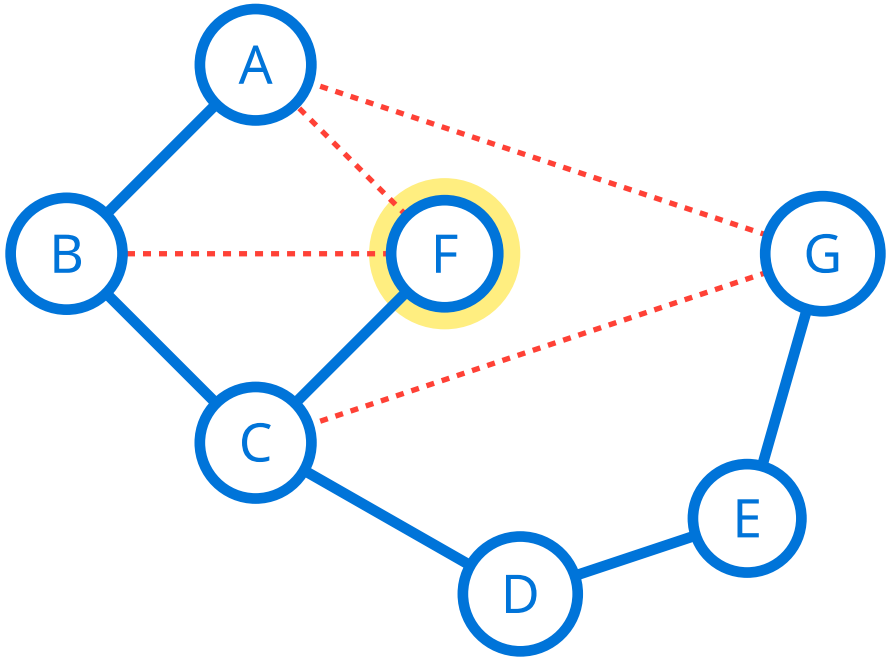
Current Vertex: F

Edges To Check

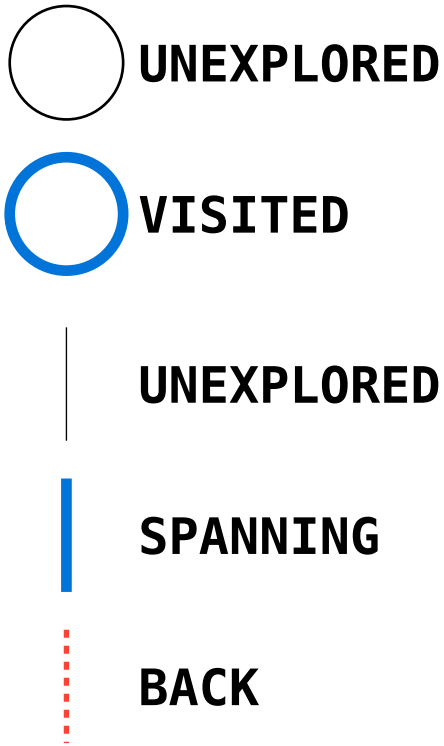
`[]`

Call Stack

```
DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)
DFSOne(G,F)
```



DFS Example (Recursive Version)



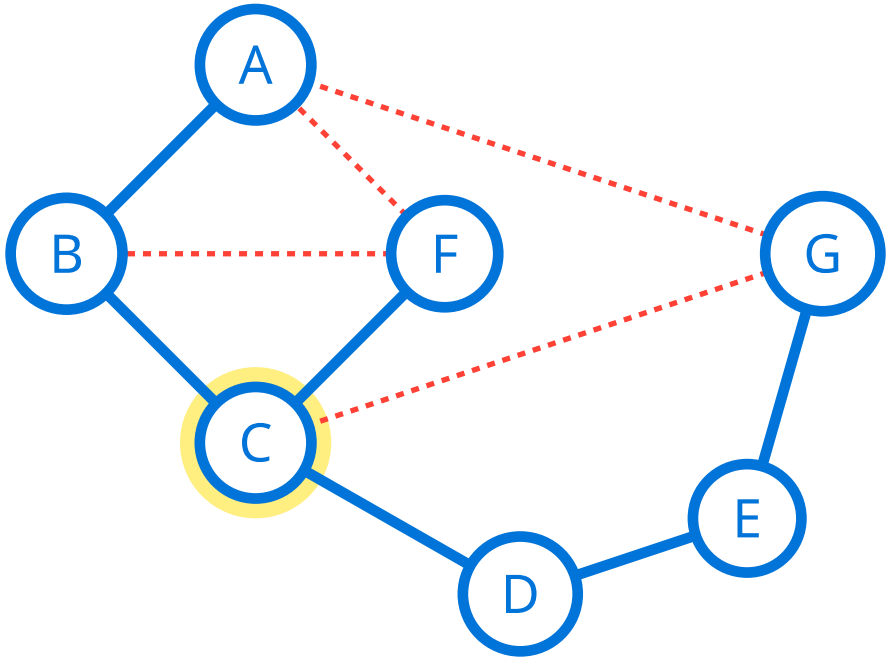
Current Vertex: C

Edges To Check

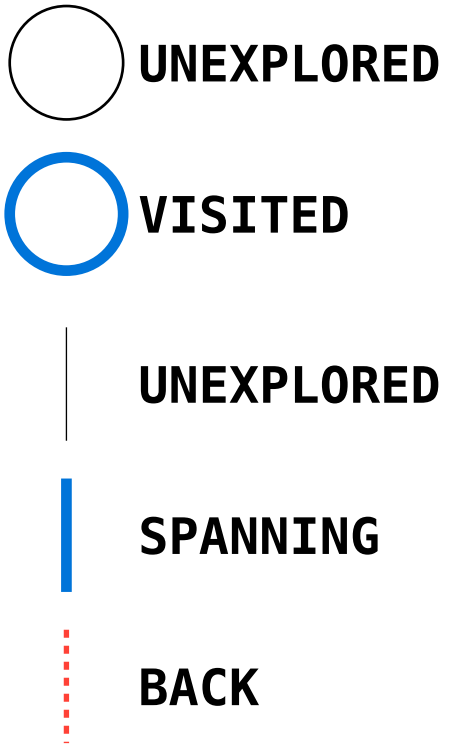
[]

Call Stack

DFS(G)
DFSOne(G,A)
DFSOne(G,B)
DFSOne(G,C)



DFS Example (Recursive Version)



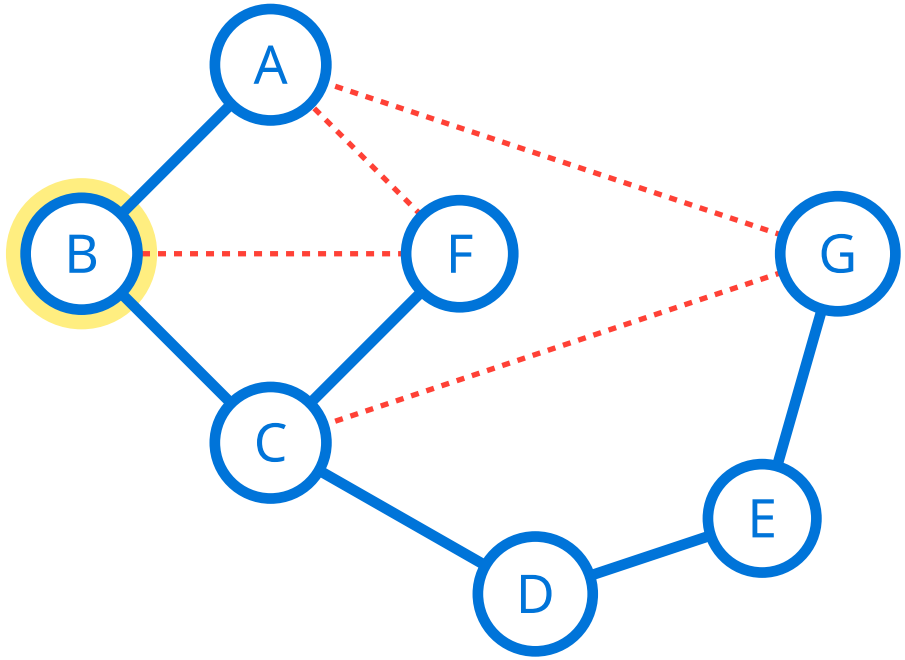
Current Vertex: B

Edges To Check

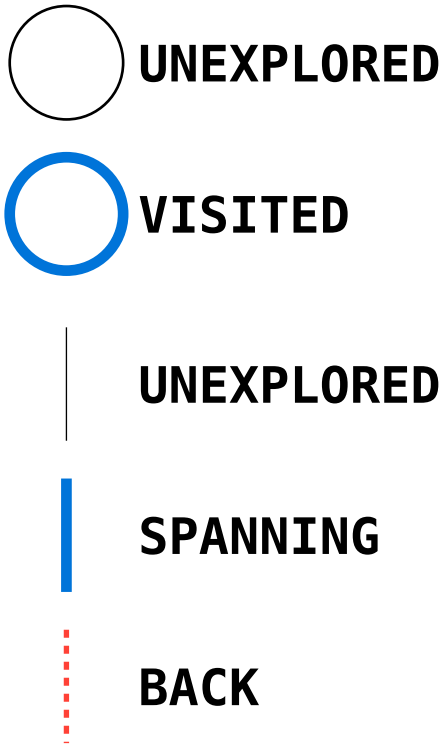
```
[]
```

Call Stack

```
DFS(G)
DFSOne(G,A)
DFSOne(G,B)
```



DFS Example (Recursive Version)



Current Vertex: A

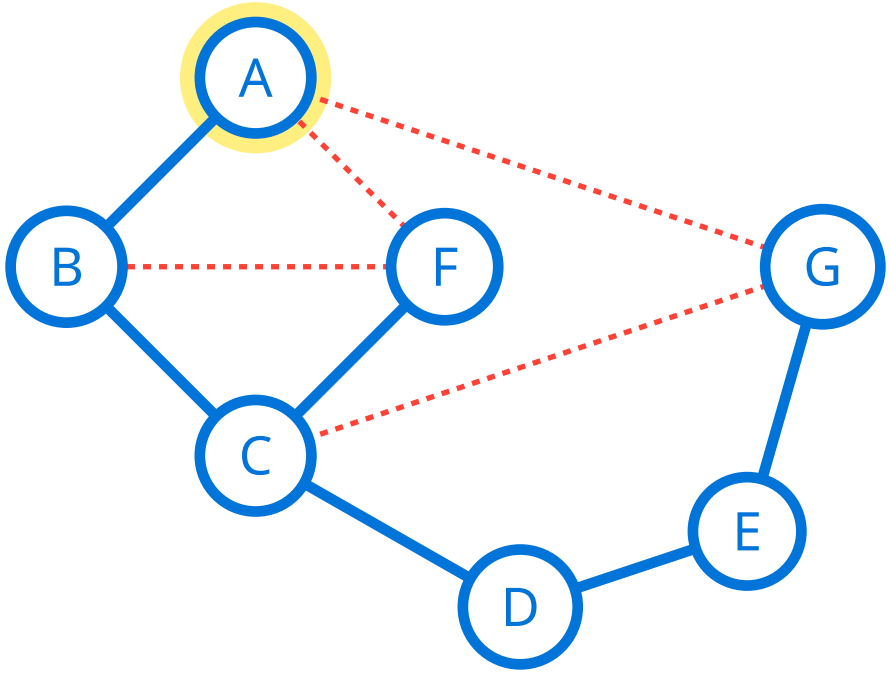
Edges To Check

`[]`

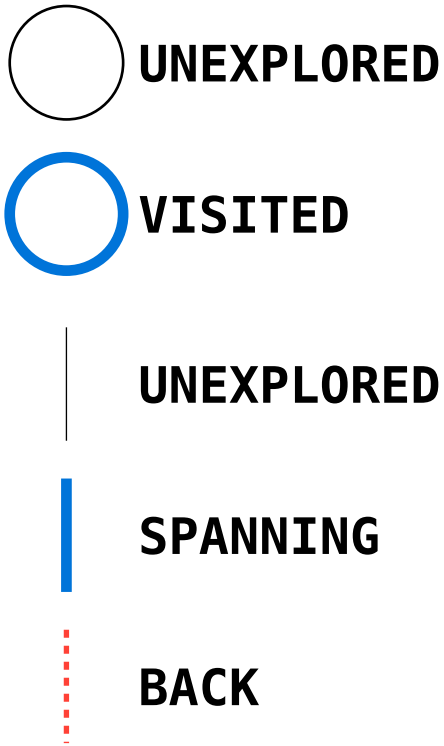
Call Stack

`DFS(G)`

`DFSOne(G,A)`



DFS Example (Recursive Version)



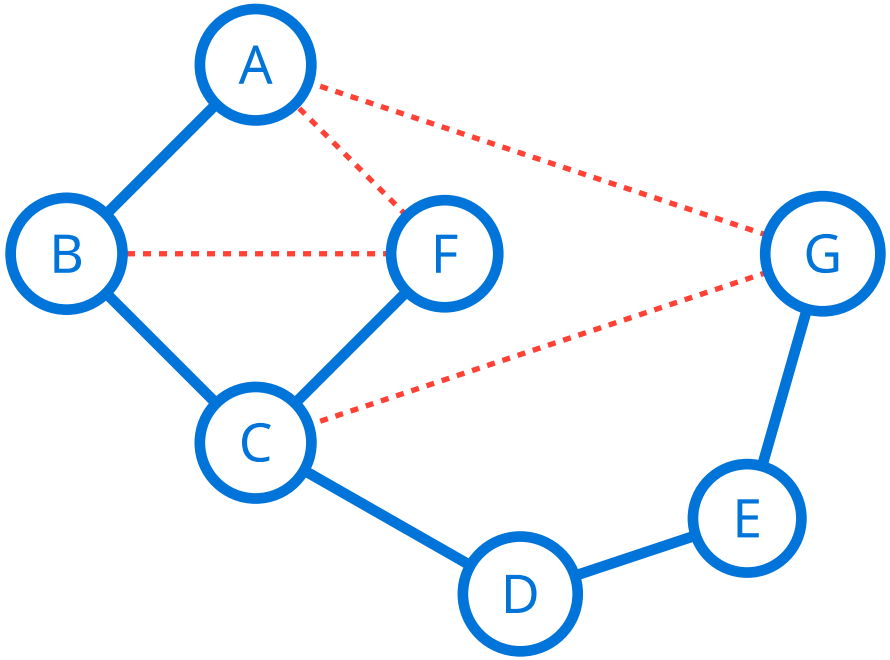
Current Vertex:

Edges To Check

`[]`

Call Stack

`DFS(G)`



DFS Runtime (Recursive Version)

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          DFSOne(G, v)
```

What is the growth function for the runtime of DFS?

DFS Runtime (Recursive Version)

```
1 Input: G = (V,E)
2
3 for e in E: e.label == UNEXPLORED
4 for v in V: v.label == UNEXPLORED
5 for v in V:
6     if v.label != VISITED:
7         DFSOne(G, v)
```

What is the growth function for the runtime of DFS?

$$\Theta(m) + \Theta(n) + \sum_{v \in V} T_{\text{DFSOne}}(G, v)$$

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

What is the growth function for the runtime of DFS?

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

What is the growth function for the runtime of DFS?

All lines but 4 and 11 are $\Theta(1)$...

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

What is the growth function for the runtime of DFS?

All lines but 4 and 11 are $\Theta(1)$...

Line 4: How long does it take to get **`outgoingEdges`**?

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

What is the growth function for the runtime of DFS?

All lines but 4 and 11 are $\Theta(1)$...

Line 4: How long does it take to get **`outgoingEdges`**?

$\Theta(1)$ (if using **`AdjacencyList`**)

How many loop iterations?

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

What is the growth function for the runtime of DFS?

All lines but 4 and 11 are $\Theta(1)$...

Line 4: How long does it take to get **`outgoingEdges`**?

$\Theta(1)$ (if using **`AdjacencyList`**)

How many loop iterations? $\deg(v)$

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

What is the growth function for the runtime of DFS?

All lines but 4 and 11 are $\Theta(1)$...

Line 11: Another call to **DFSOne**...

We are stuck...

Direct code analysis is tricky...

(we'll learn techniques for analyzing recursion after Spring Break)

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**?

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**? Exactly once

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**? Exactly once
- How much time is spent in **`DFSOne`** processing that **`Vertex`**?

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

How much time in `DFSOne(G,v)` is spent processing v?

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
11              DFSOne(G, w)
```

How much time in `DFSOne(G,v)` is spent processing v?

Line 11 is time spent processing **another vertex** so let's ignore it

DFSOne Runtime (Recursive Version)

```
 1  Input: Graph G, Vertex v
 2
 3  v.label = VISITED
 4  for e in v.outgoingEdges():
 5      if e.label == UNEXPLORED:
 6          w = e.dest
 7          if w.label == VISITED:
 8              e.label = BACK
 9          else:
10              e.label = SPANNING
…                   …
```

How much time in `DFSOne(G,v)` is spent processing v?

Line 11 is time spent processing **another vertex** so let's ignore it

Everything but the for loop is $\Theta(1)$

DFSOne Runtime (Recursive Version)

```
4  for e in v.outgoingEdges():
…      0(1)
```

How much time in `DFSOne(G,v)` is spent processing v?

Line 11 is time spent processing **another vertex** so let's ignore it

Everything but the for loop is $\Theta(1)$

$\Theta(1)$ time to get **`outgoingEdges`**

The loop runs $\deg(v)$ iterations

Total time spent on **`Vertex v`**:

DFSOne Runtime (Recursive Version)

```
4  for e in v.outgoingEdges():
…      Θ(1)
```

How much time in `DFSOne(G,v)` is spent processing v?

Line 11 is time spent processing **another vertex** so let's ignore it

Everything but the for loop is $\Theta(1)$

$\Theta(1)$ time to get **`outgoingEdges`**

The loop runs $\deg(v)$ iterations

Total time spent on **`Vertex v`**: $\Theta(\deg(v))$

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**? Exactly once
- How much time is spent in **`DFSOne`** processing that **`Vertex`**?

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**? Exactly once
- How much time is spent in **`DFSOne`** processing that **`Vertex`**? $\deg(v)$

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**? Exactly once
- How much time is spent in **`DFSOne`** processing that **`Vertex`**? $\deg(v)$
- What is the total cost of all **`DFSOne`** calls?

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**? Exactly once
- How much time is spent in **`DFSOne`** processing that **`Vertex`**? $\deg(v)$
- What is the total cost of all **`DFSOne`** calls?

$$\sum_{v \in V} \deg(v)$$

DFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is **`DFSOne`** called on each **`Vertex`**? Exactly once
- How much time is spent in **`DFSOne`** processing that **`Vertex`**? $\deg(v)$
- What is the total cost of all **`DFSOne`** calls?

$$\sum_{v \in V} \deg(v) = 2|E|$$

DFS Runtime Summary

1. Initialize all vertex labels	$\Theta(n)$
2. Initialize all edge labels	$\Theta(m)$
3. Iterate over every vertex in **`DFS`**	$\Theta(n)$
4. Process every vertex in **`DFSOne`**	$\sum_{v \in V} \deg(v) = 2\|E\| \in \Theta(m)$

Total: $\Theta(n + m)$

The order of exploration was controlled by the call **stack**

- As soon as we found a new vertex,`w`, we immediately called **`DFSOne(G,w)`**
- We backtracked to previous nodes only after **`DFSOne`** calls completed
- **Note:** We could also implement DFS non-recursively by managing a **`Stack`** of vertices to explore ourselves

DFS explores a Graph in LIFO order! Can we explore in FIFO order instead?

Breadth-First Search

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)
- Construct a **spanning tree** for every connected component of G

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components
  - **Side Effect:** Compute paths between all connected vertices

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components
  - **Side Effect:** Compute paths between all connected vertices
  - **Side Effect:** Identify cycles

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components
  - **Side Effect:** Compute paths between all connected vertices
  - **Side Effect:** Identify cycles
  - **Side Effect:** Computed paths will have the fewest edges from start vertex

Breadth-First Search Goals

When we perform BFS on a graph, $G = (V, E)$:

Primary Goals

- Visit **every** vertex in graph G (**in order of number of edges from start**)
- Construct a **spanning tree** for every connected component of G
  - **Side Effect:** Determine if the graph is connected
  - **Side Effect:** Compute the connected components
  - **Side Effect:** Compute paths between all connected vertices
  - **Side Effect:** Identify cycles
  - **Side Effect:** Computed paths will have the fewest edges from start vertex

Secondary Goal

- Perform the traversal in linear time, $O(|V| + |E|)$

Basic Design

Top-Level Driver Function: `BFS(G)`

Input: Graph $G = (V, E)$

Output: A labeling of every edge as either:

- **Spanning Edge:** Part of the spanning tree
- **Cross Edge:** Part of a cycle

Basic Design

Top-Level Driver Function: `BFS(G)`

Input: Graph $G = (V, E)$

Output: A labeling of every edge as either:

- **Spanning Edge:** Part of the spanning tree
- **Cross Edge:** Part of a cycle

Helper Function: `BFSOne(G, v)`

Input: Graph $G = (V, E)$, starting vertex $v \in V$

Output: A labeling of every edge in the connected component of v

BFS Algorithm

```
1 Input: G = (V,E)
2
3 for e in E: e.label == UNEXPLORED
4 for v in V: v.label == UNEXPLORED
5 for v in V:
6     if v.label != VISITED:
7         BFSOne(G, v)
```

BFS Algorithm

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          BFSOne(G, v)
```

Start with everything unexplored

BFS Algorithm

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          BFSOne(G, v)
```

Make sure to check every vertex

BFS Algorithm

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          BFSOne(G, v)
```

Explore the connected component of unvisited vertices

Remember: **BFSOne** explores the entire connected component of **v**

It will mark vertices as **VISITED** and label edges as **SPANNING** or **CROSS**

BFS ensures we visit every vertex

BFS0ne Algorithm

```
 1 Input: G = (V,E), Vertex v
 2 v.label = VISITED
 3 todo = Queue(v)
 4 while not todo.empty():
 5     curr = todo.dequeue()
 6     for e in curr.outgoingEdges():
 7         if e.label == UNEXPLORED:
 8             if e.dest.label == VISITED:
 9                 e.label = CROSS
10             else:
11                 e.label = SPANNING
12                 e.dest.label = VISITED
13                 todo.enqueue(e.dest)
```

BFS0ne Algorithm

```
1  Input: G = (V,E), Vertex v
2  v.label = VISITED
3  todo = Queue(v)
4  while not todo.empty():
5      curr = todo.dequeue()
6      for e in curr.outgoingEdges():
7          if e.label == UNEXPLORED:
8              if e.dest.label == VISITED:
9                  e.label = CROSS
10             else:
11                 e.label = SPANNING
12                 e.dest.label = VISITED
13                 todo.enqueue(e.dest)
```

Mark the vertex visited and create our TODO list:
a **Queue** containing **v**

BFS0ne Algorithm

```
1  Input: G = (V,E), Vertex v
2  v.label = VISITED
3  todo = Queue(v)
4  while not todo.empty():
5      curr = todo.dequeue()
6      for e in curr.outgoingEdges():
7          if e.label == UNEXPLORED:
8              if e.dest.label == VISITED:
9                  e.label = CROSS
10             else:
11                 e.label = SPANNING
12                 e.dest.label = VISITED
13                 todo.enqueue(e.dest)
```

Keep searching as long as we have vertices to explore

BFS0ne Algorithm

```
1  Input: G = (V,E), Vertex v
2  v.label = VISITED
3  todo = Queue(v)
4  while not todo.empty():
5      curr = todo.dequeue()
6      for e in curr.outgoingEdges():
7          if e.label == UNEXPLORED:
8              if e.dest.label == VISITED:
9                  e.label = CROSS
10             else:
11                 e.label = SPANNING
12                 e.dest.label = VISITED
13                 todo.enqueue(e.dest)
```

Dequeue the next vertex in our TODO list...
We explore in FIFO order!

BFS0ne Algorithm

```
 1  Input: G = (V,E), Vertex v
 2  v.label = VISITED
 3  todo = Queue(v)
 4  while not todo.empty():
 5      curr = todo.dequeue()
 6      for e in curr.outgoingEdges():
 7          if e.label == UNEXPLORED:
 8              if e.dest.label == VISITED:
 9                  e.label = CROSS
10              else:
11                  e.label = SPANNING
12                  e.dest.label = VISITED
13                  todo.enqueue(e.dest)
```

Check every edge leaving our current vertex

BFS0ne Algorithm

```
 1  Input: G = (V,E), Vertex v
 2  v.label = VISITED
 3  todo = Queue(v)
 4  while not todo.empty():
 5      curr = todo.dequeue()
 6      for e in curr.outgoingEdges():
 7          if e.label == UNEXPLORED:
 8              if e.dest.label == VISITED:
 9                  e.label = CROSS
10              else:
11                  e.label = SPANNING
12                  e.dest.label = VISITED
13                  todo.enqueue(e.dest)
```

Follow unexplored edges

BFS0ne Algorithm

```
1  Input: G = (V,E), Vertex v
2  v.label = VISITED
3  todo = Queue(v)
4  while not todo.empty():
5      curr = todo.dequeue()
6      for e in curr.outgoingEdges():
7          if e.label == UNEXPLORED:
8              if e.dest.label == VISITED:
9                  e.label = CROSS
10             else:
11                 e.label = SPANNING
12                 e.dest.label = VISITED
13                 todo.enqueue(e.dest)
```

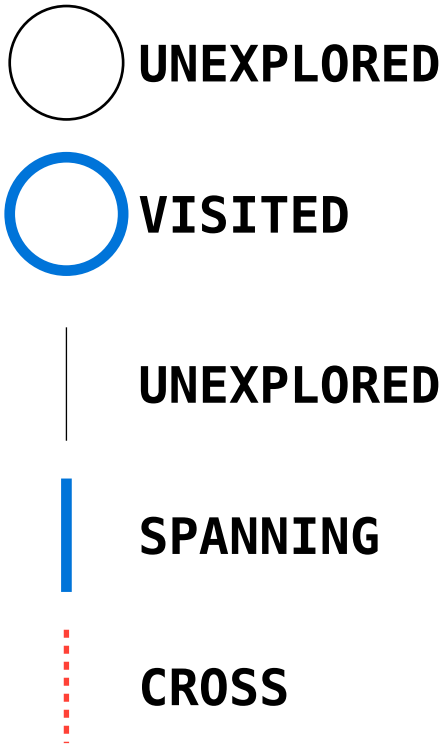
If they lead to a vertex we've already seen, it's a **CROSS** edge... we just found a cycle

BFS0ne Algorithm

```
1  Input: G = (V,E), Vertex v
2  v.label = VISITED
3  todo = Queue(v)
4  while not todo.empty():
5      curr = todo.dequeue()
6      for e in curr.outgoingEdges():
7          if e.label == UNEXPLORED:
8              if e.dest.label == VISITED:
9                  e.label = CROSS
10             else:
11                 e.label = SPANNING
12                 e.dest.label = VISITED
13                 todo.enqueue(e.dest)
```

If they lead to a new vertex, it's a **SPANNING** edge...then mark the vertex **VISITED** and add it to our TODO list to explore **later**

BFS Example



Current Vertex:

TODO (Queue)

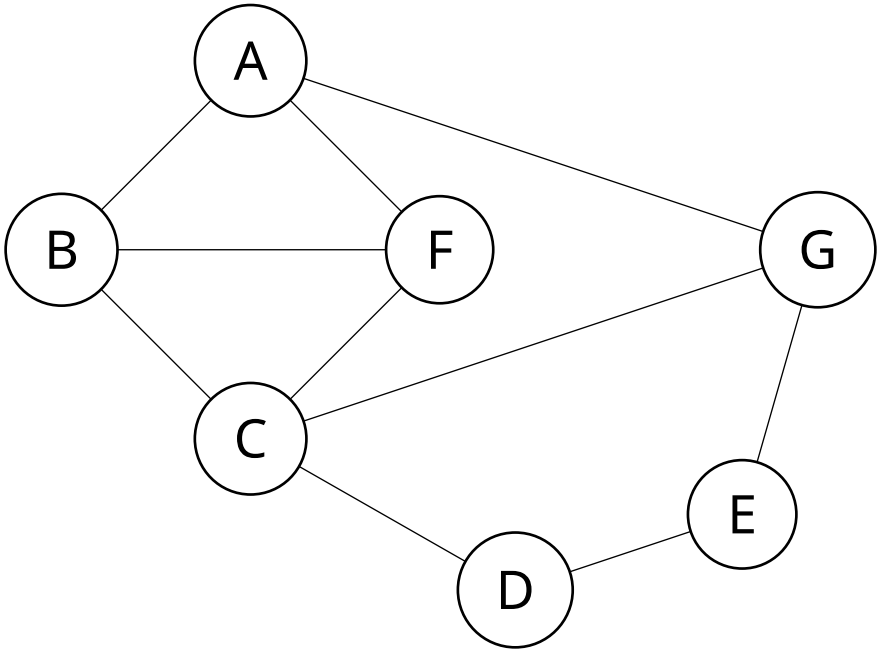
`[]`

Edges To Check

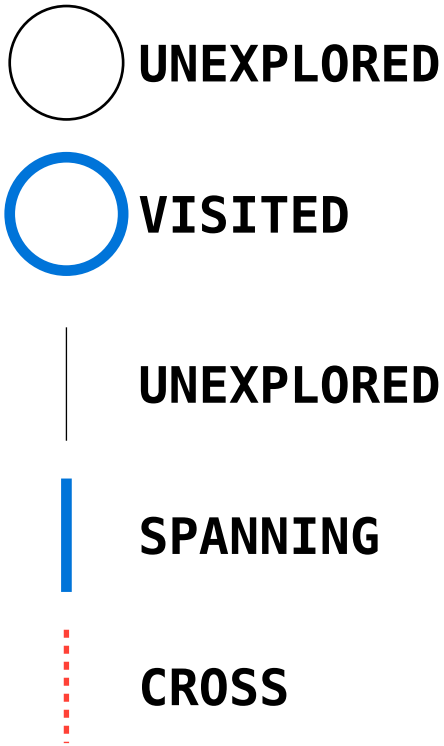
`[]`

Call Stack

`BFS(G)`



BFS Example



Current Vertex:

TODO (Queue)

`[A]`

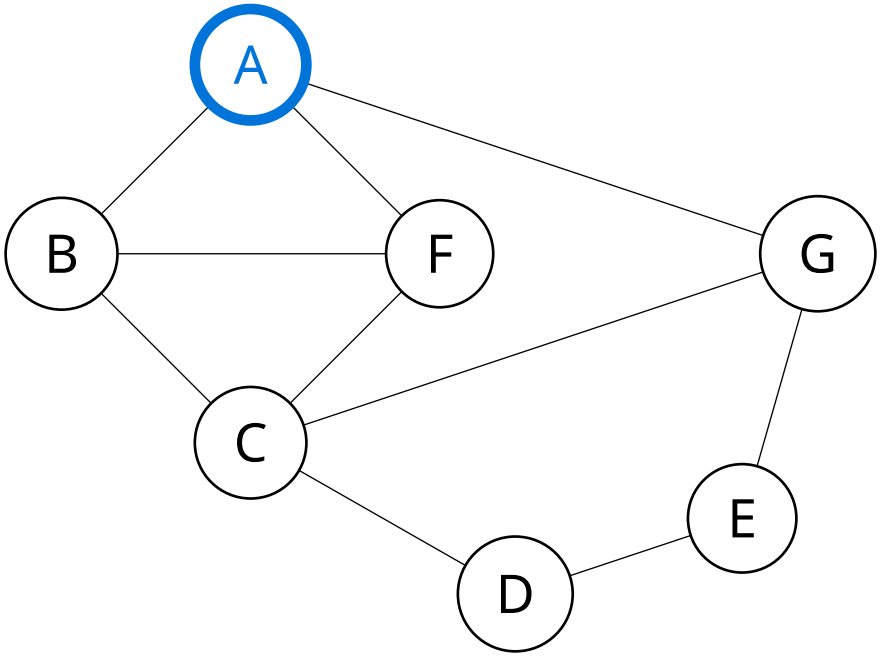
Edges To Check

`[]`

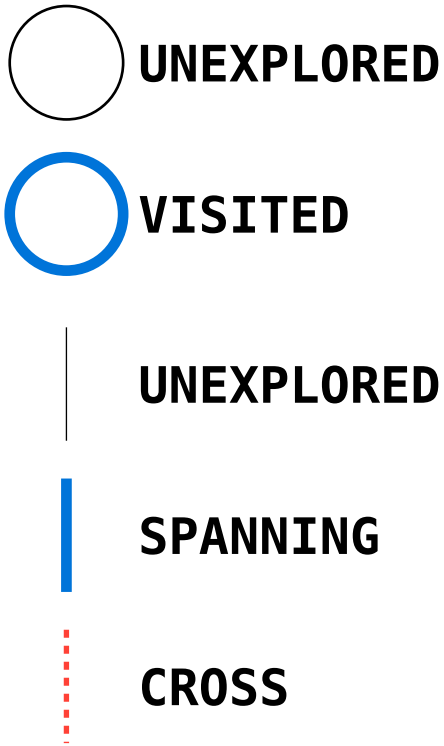
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



Current Vertex: A

TODO (Queue)

`[]`

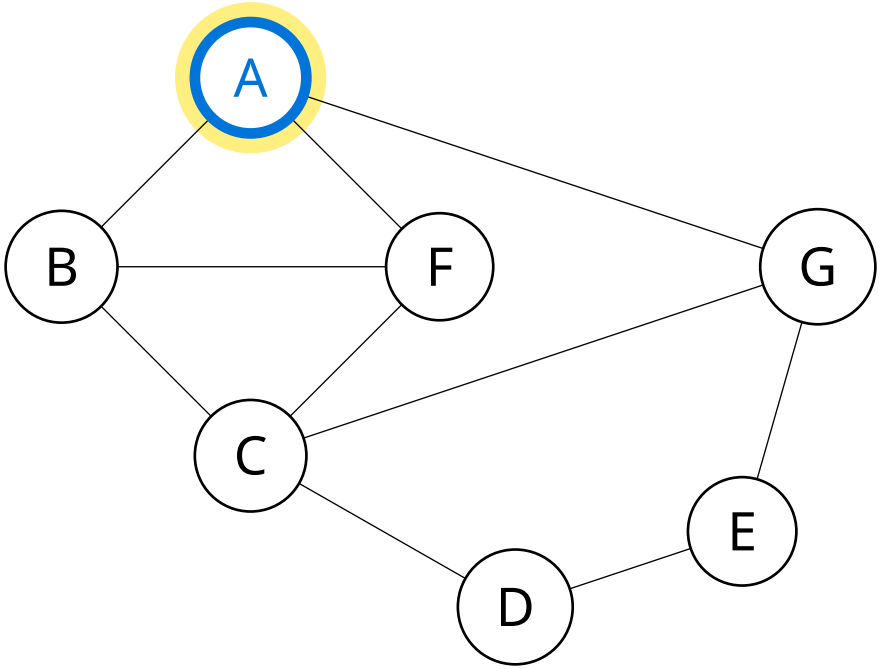
Edges To Check

`[B,F,G]`

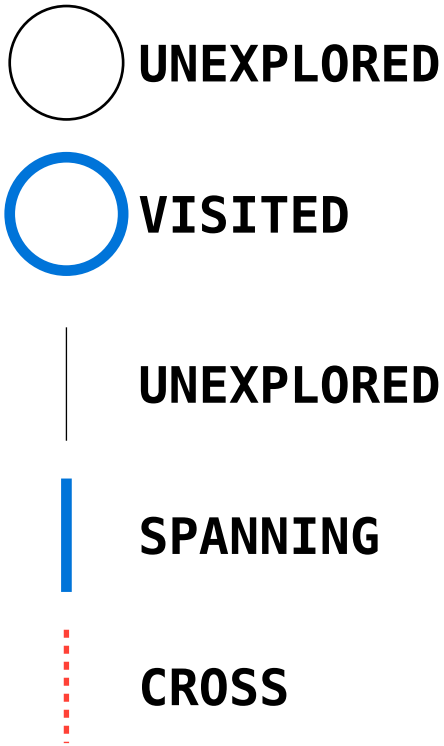
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



Current Vertex: A

TODO (Queue)

`[B]`

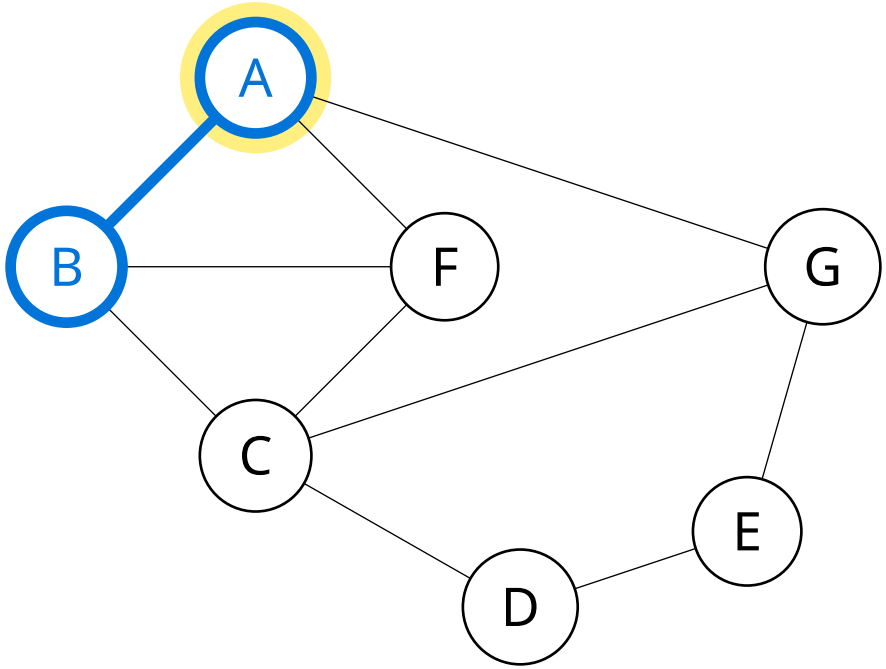
Edges To Check

`[F,G]`

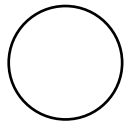
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: A

TODO (Queue)

`[B,F]`

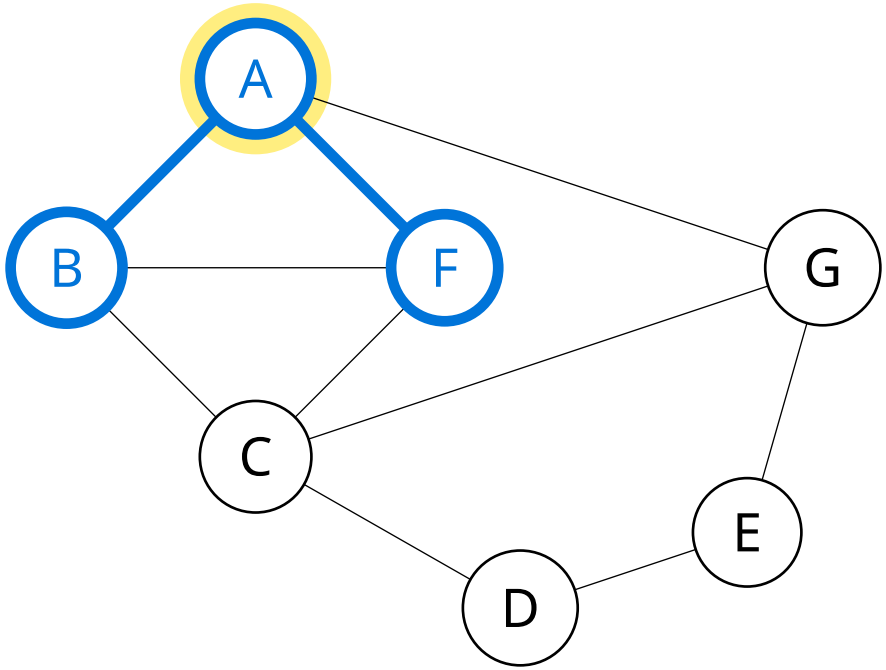
Edges To Check

`[G]`

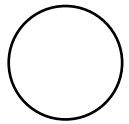
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: A

TODO (Queue)

`[B,F,G]`

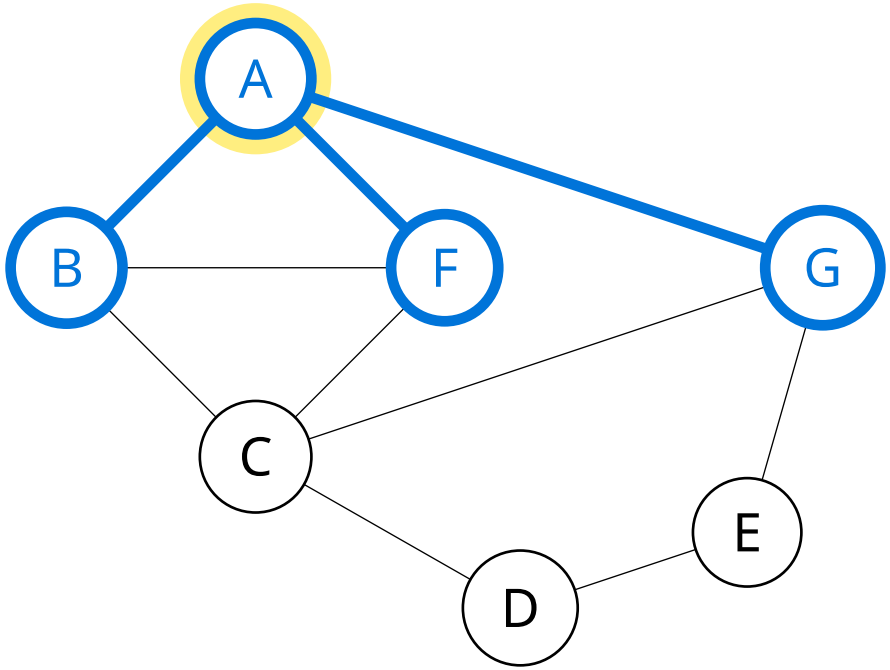
Edges To Check

`[]`

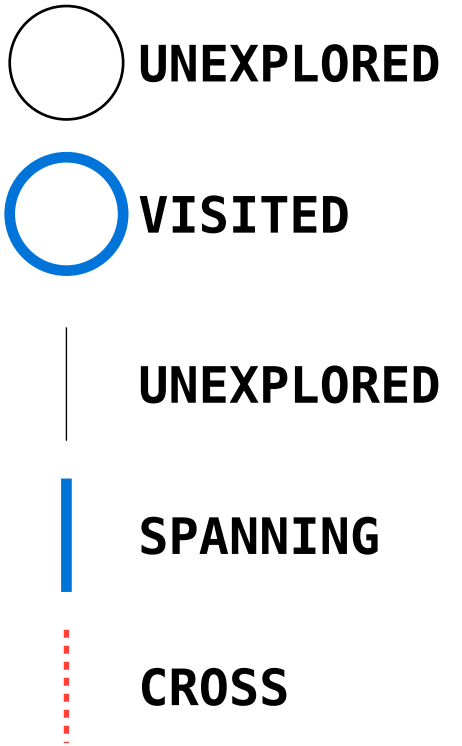
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



Current Vertex: B

TODO (Queue)

`[F,G]`

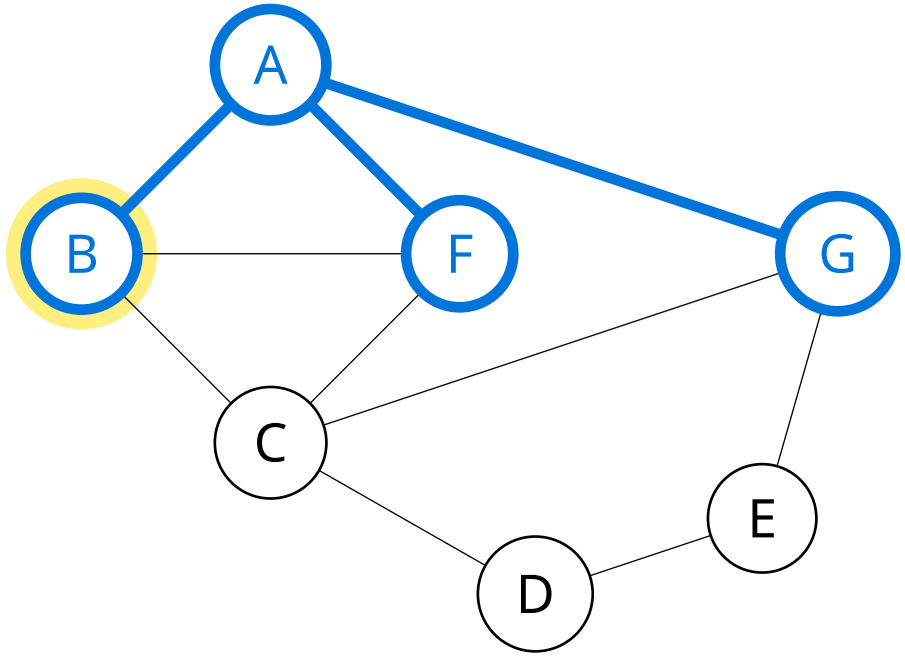
Edges To Check

`[C,F]`

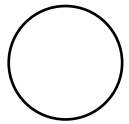
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: B

TODO (Queue)

`[F,G,C]`

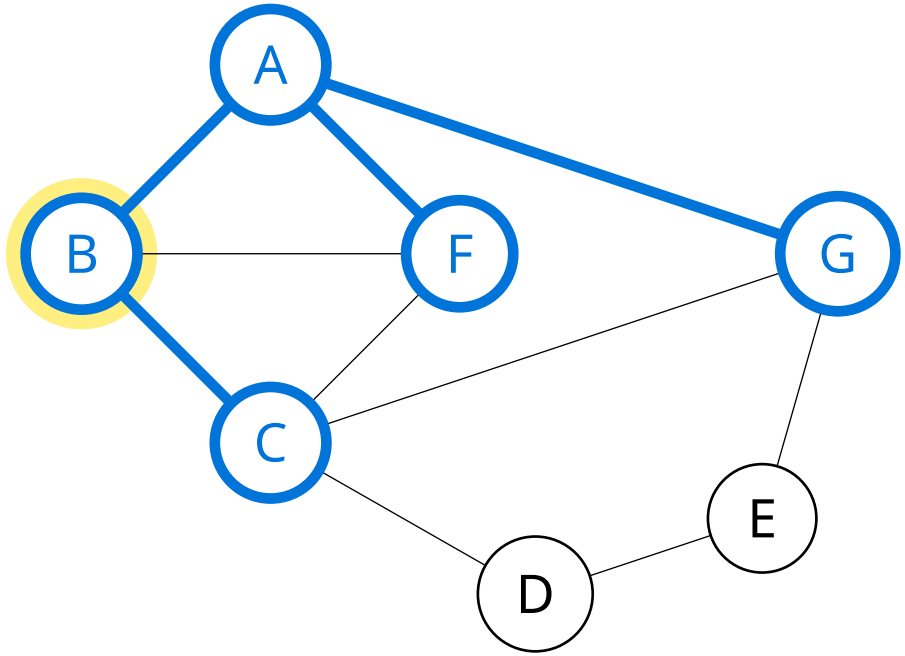
Edges To Check

`[F]`

Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example

UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: B

TODO (Queue)

`[F,G,C]`

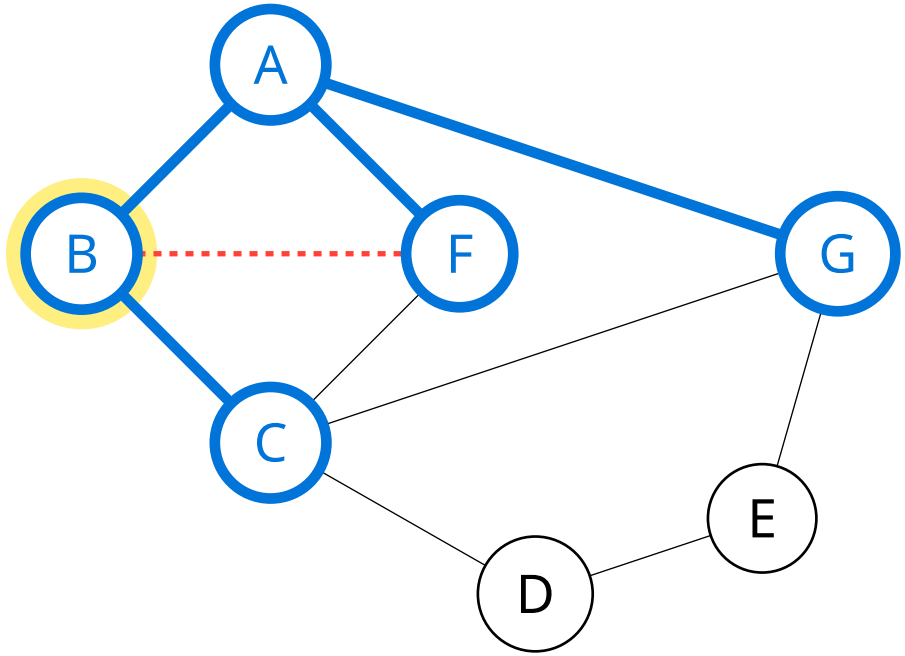
Edges To Check

`[]`

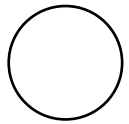
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: F

TODO (Queue)

`[G,C]`

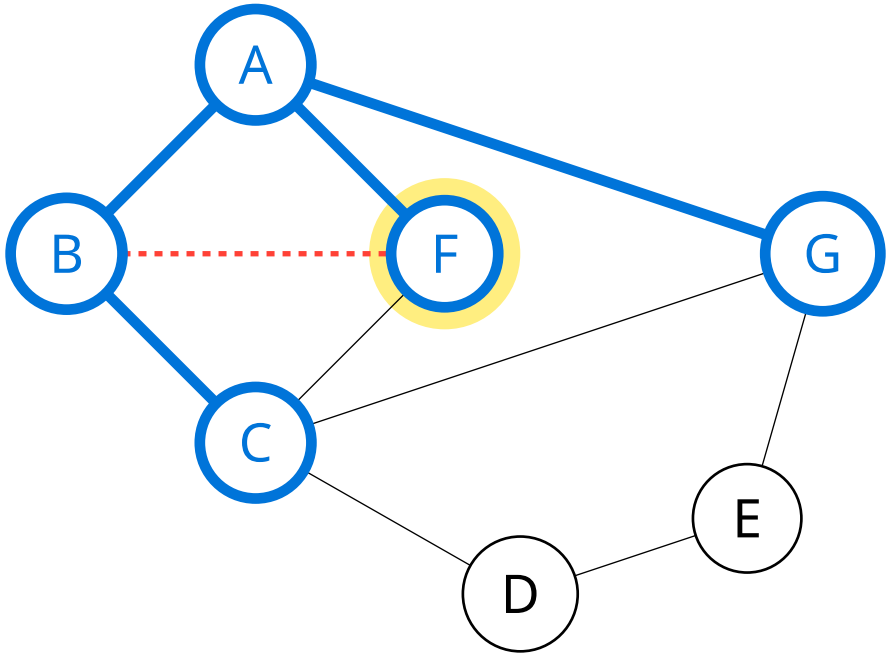
Edges To Check

`[C]`

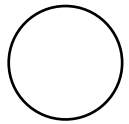
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: F

TODO (Queue)

`[G,C]`

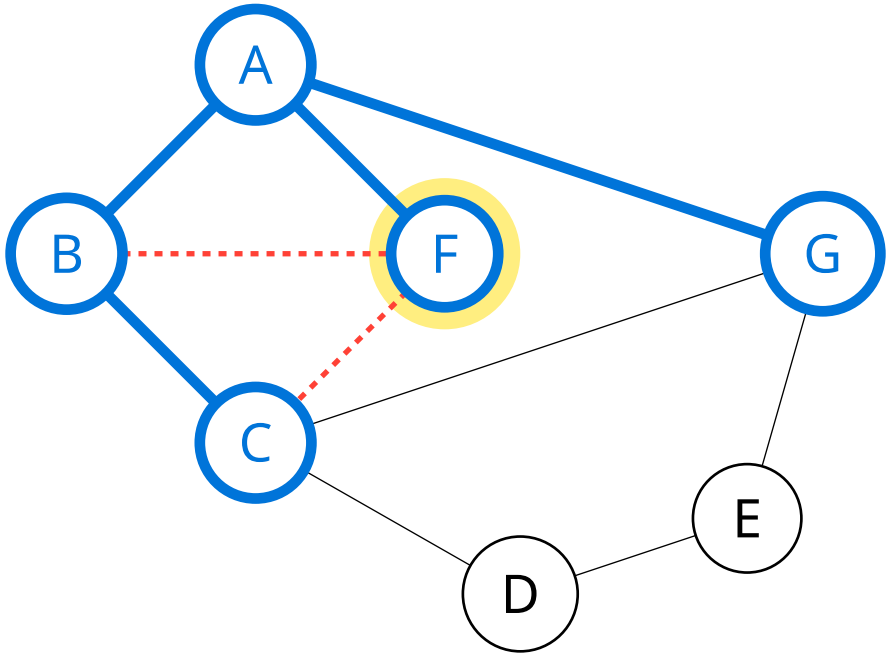
Edges To Check

`[]`

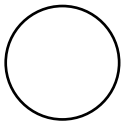
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: `G`

TODO (Queue)

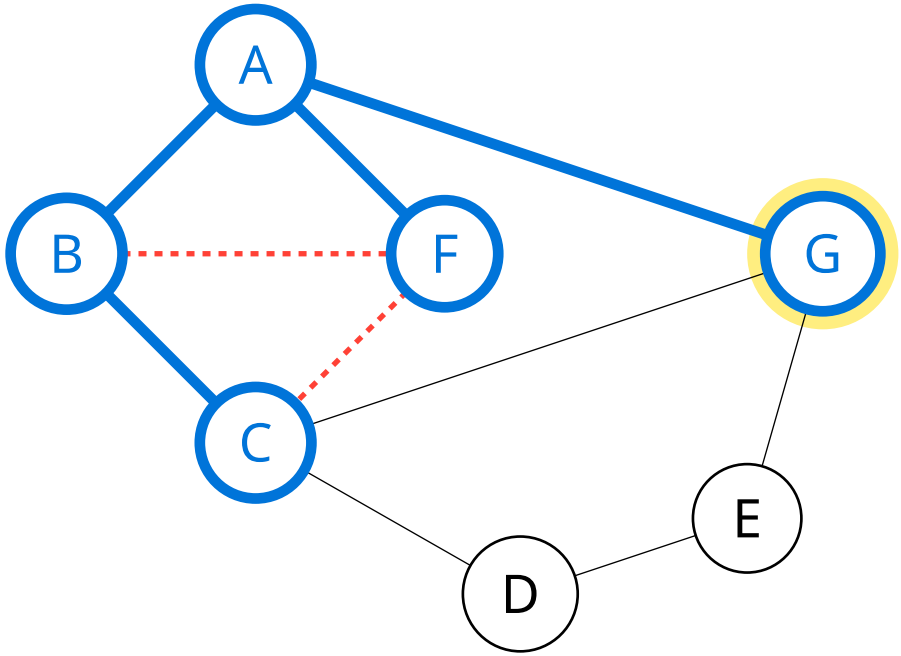
`[C]`

Edges To Check

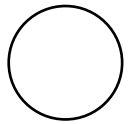
`[C,E]`

Call Stack

```
BFS(G)
BFSOne(G,A)
```



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: `G`

TODO (Queue)

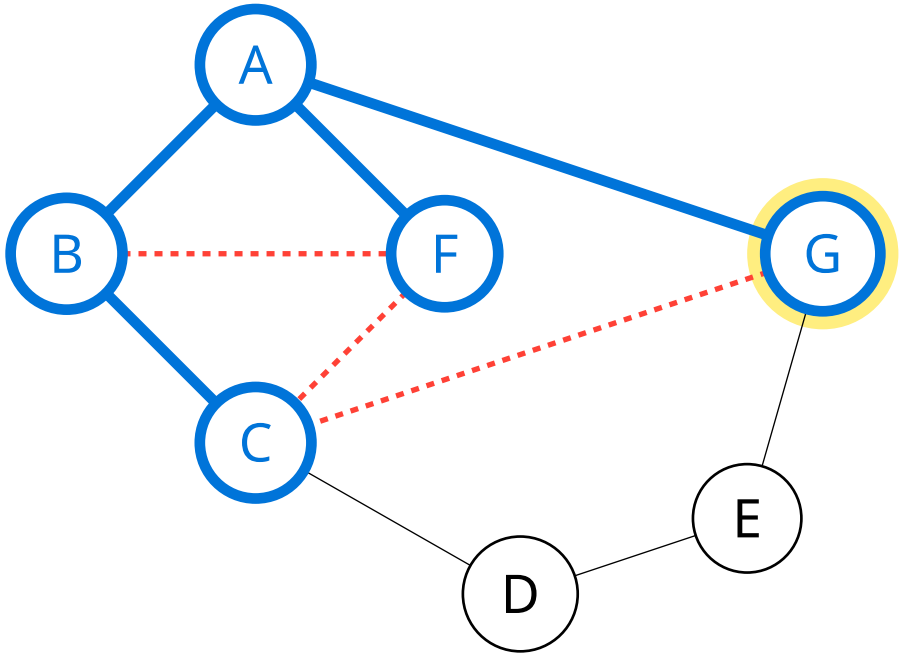
`[C]`

Edges To Check

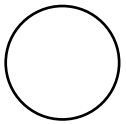
`[E]`

Call Stack

```
BFS(G)
BFSOne(G,A)
```



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: `G`

TODO (Queue)

`[C,E]`

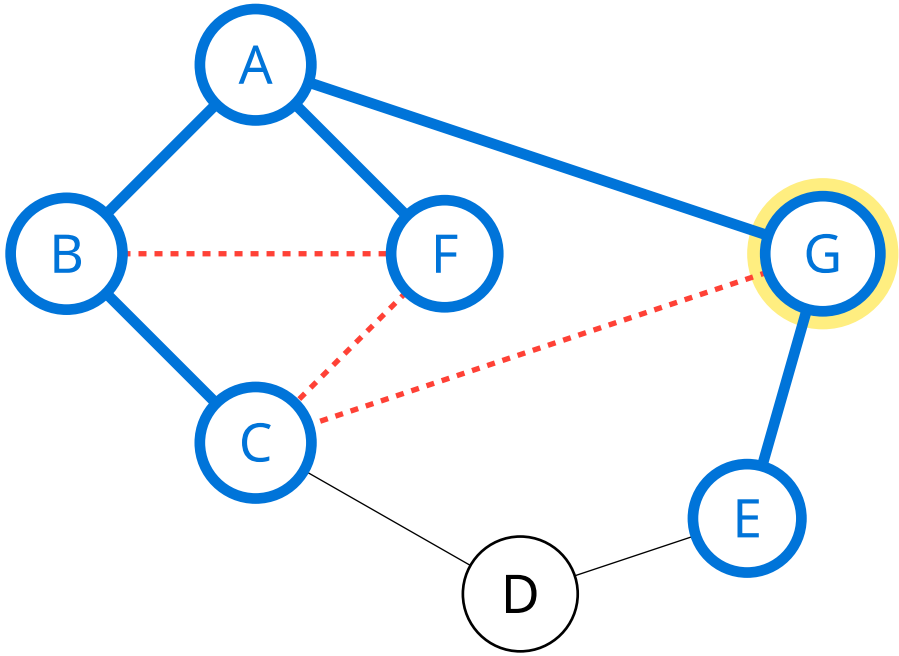
Edges To Check

`[]`

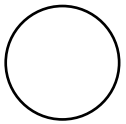
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: C

TODO (Queue)

[E]

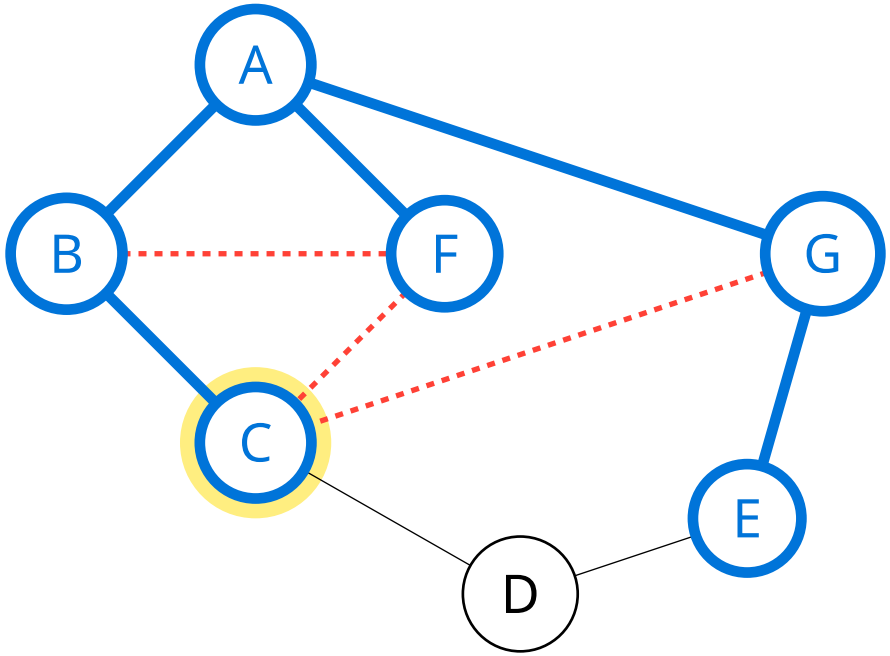
Edges To Check

[D]

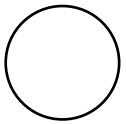
Call Stack

BFS(G)

BFSOne(G,A)



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: C

TODO (Queue)

[E,D]

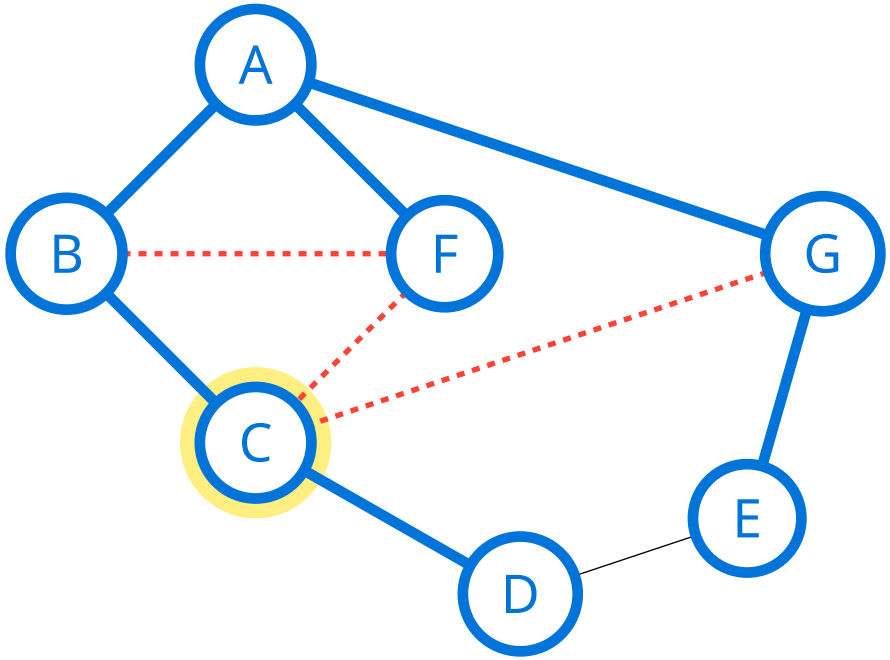
Edges To Check

[]

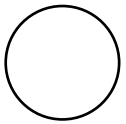
Call Stack

BFS(G)

BFSOne(G,A)



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: E

TODO (Queue)

`[D]`

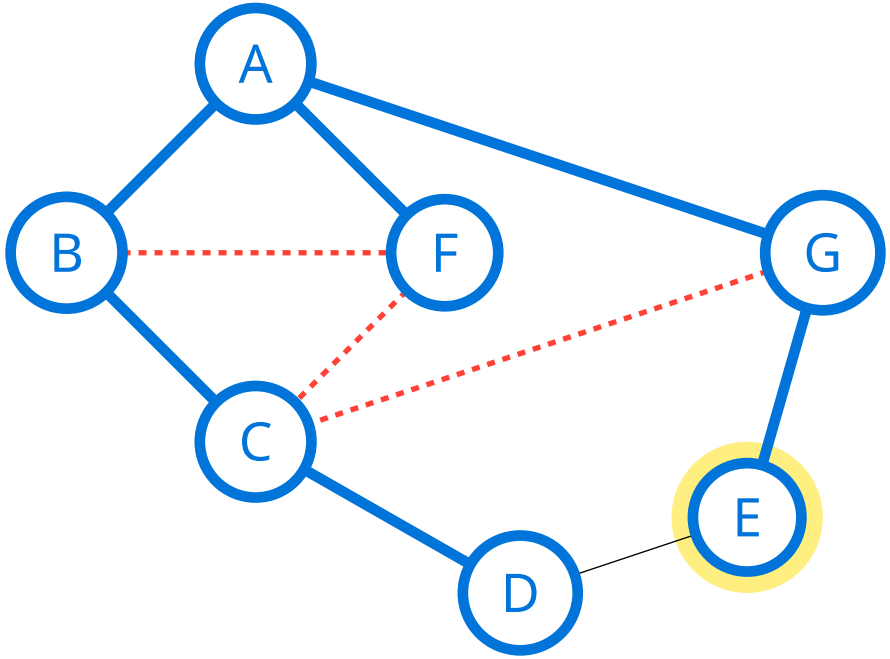
Edges To Check

`[D]`

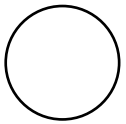
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: E

TODO (Queue)

`[D]`

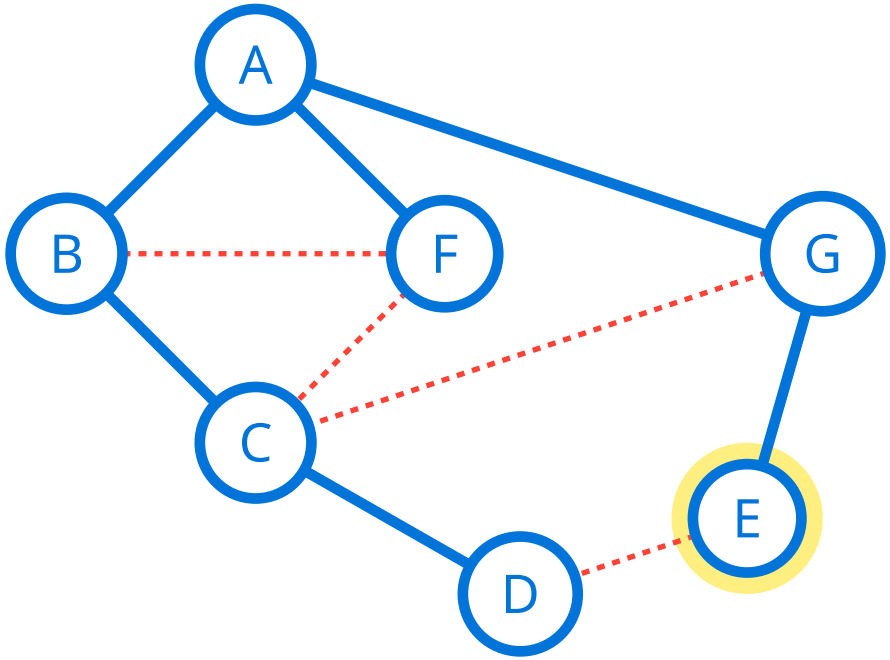
Edges To Check

`[]`

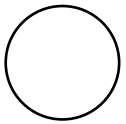
Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex: D

TODO (Queue)

[]

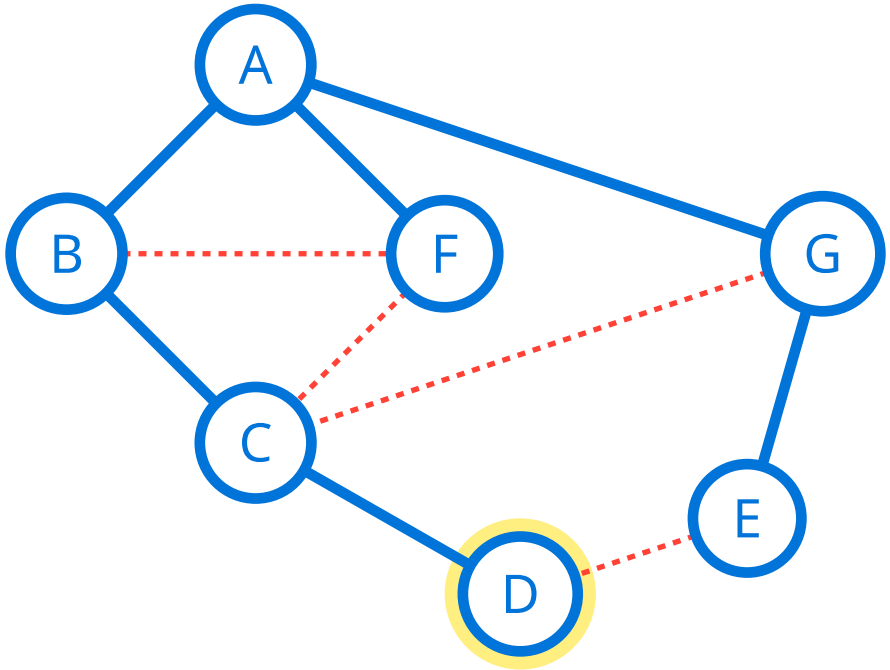
Edges To Check

[]

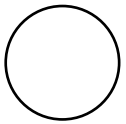
Call Stack

BFS(G)

BFSOne(G,A)



BFS Example



UNEXPLORED

VISITED

UNEXPLORED

SPANNING

CROSS

Current Vertex:

TODO (Queue)

`[]`

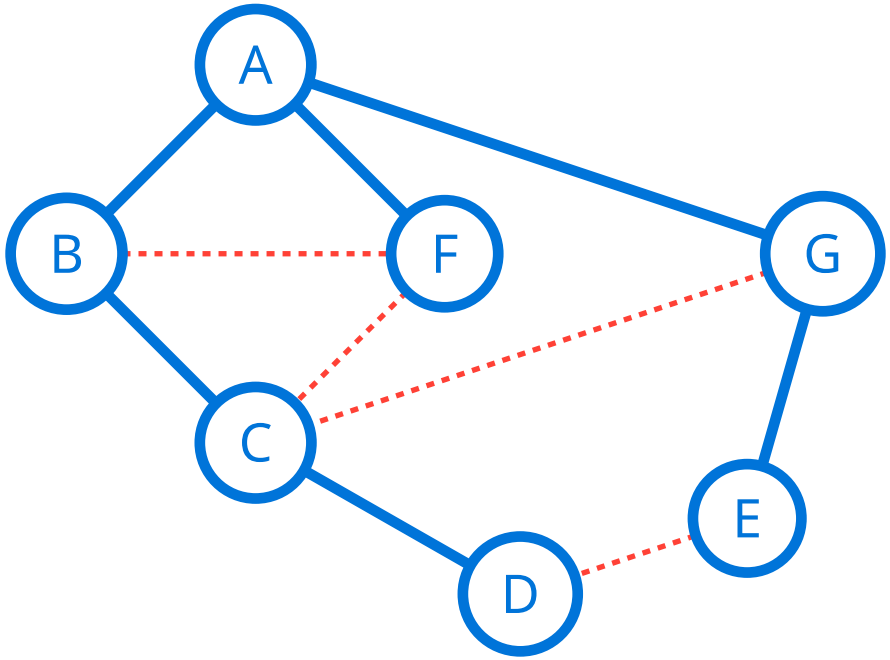
Edges To Check

`[]`

Call Stack

`BFS(G)`

`BFSOne(G,A)`



BFS Runtime

```
1  Input: G = (V,E)
2
3  for e in E: e.label == UNEXPLORED
4  for v in V: v.label == UNEXPLORED
5  for v in V:
6      if v.label != VISITED:
7          BFSOne(G, v)
```

What is the growth function for the runtime of BFS?

BFS Runtime

```
1 Input: G = (V,E)
2
3 for e in E: e.label == UNEXPLORED
4 for v in V: v.label == UNEXPLORED
5 for v in V:
6     if v.label != VISITED:
7         BFSOne(G, v)
```

What is the growth function for the runtime of BFS?

$$\Theta(m) + \Theta(n) + \sum_{v \in V} T_{\text{BFSOne}}(G, v)$$

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list?

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list? Exactly once

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list? Exactly once
- How many times is each **`Vertex`** removed from our TODO list?

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list? Exactly once
- How many times is each **`Vertex`** removed from our TODO list? Exactly once

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list? Exactly once
- How many times is each **`Vertex`** removed from our TODO list? Exactly once
- How much time is spent in **`BFSOne`** processing that **`Vertex`**?

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list? Exactly once
- How many times is each **`Vertex`** removed from our TODO list? Exactly once
- How much time is spent in **`BFSOne`** processing that **`Vertex`**? $\deg(v)$

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list? Exactly once
- How many times is each **`Vertex`** removed from our TODO list? Exactly once
- How much time is spent in **`BFSOne`** processing that **`Vertex`**? $\deg(v)$
- How much time is spent processing all vertices in total?

$$\sum_{v \in V} \deg(v)$$

BFS Analysis – The Big Picture

Think about the runtime in terms of the graph itself rather than the code

- How many times is each **`Vertex`** added to our TODO list? Exactly once
- How many times is each **`Vertex`** removed from our TODO list? Exactly once
- How much time is spent in **`BFSOne`** processing that **`Vertex`**? $\deg(v)$
- How much time is spent processing all vertices in total?

$$\sum_{v \in V} \deg(v) = 2|E|$$

BFS Runtime Summary

1. Initialize all vertex labels	$\Theta(n)$
2. Initialize all edge labels	$\Theta(m)$
3. Enqueue and Dequeue each vertex once	$\Theta(n)$
4. Process every vertex	$\sum_{v \in V} \deg(v) = 2\|E\| \in \Theta(m)$

Total: $\Theta(n + m)$

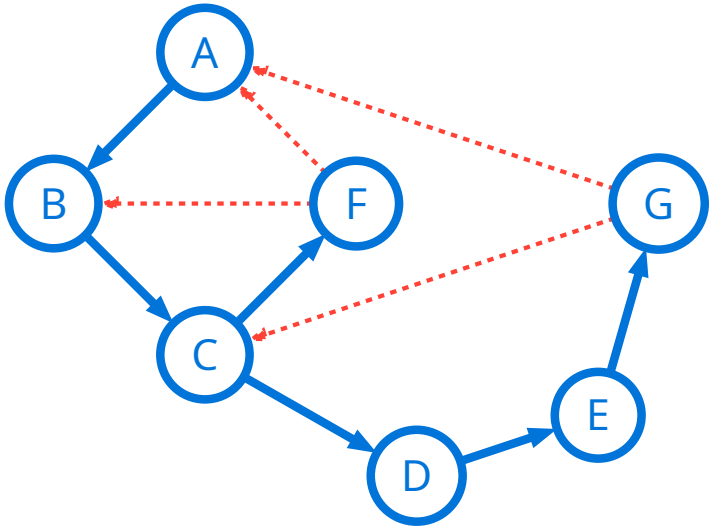
The order of exploration was controlled by the TODO list (a **`Queue`**)

- When we found a new vertex, we added it to the BACK of our TODO list
- We explore new vertices in the order that we found them
- **Note:** Switching the **`Queue`** to a **`Stack`** would be an implementation of **`DFS`**

DFS explores a Graph in FIFO order! This is why the paths it finds have the fewest edges!

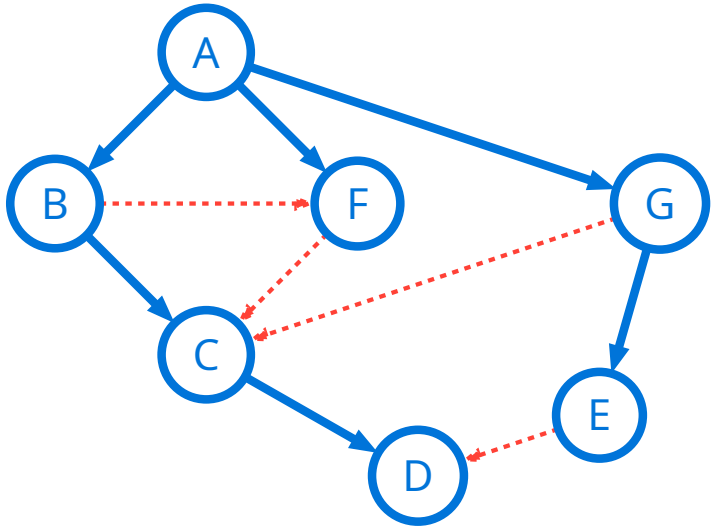
DFS vs BFS Cycles

DFS Traversal



BACK edges point **back** to something that was previously discovered

BFS Traversal



CROSS edges point “across”, to nodes in the same level or one lower

DFS vs BFS Common Uses

Finds...	DFS	BFS
Spanning Trees	✓	✓
Connected Components	✓	✓
Paths	✓	✓
Cycles	✓	✓
Shortest Paths*		✓
Articulation Points	✓	

* *not necessarily true for all graphs...more on this next time*