CSCI 104
Graph Representation and Traversals

Mark Redekopp
David Kempe
Sandra Batista

GRAPH REPRESENTATIONS

Graph Notation

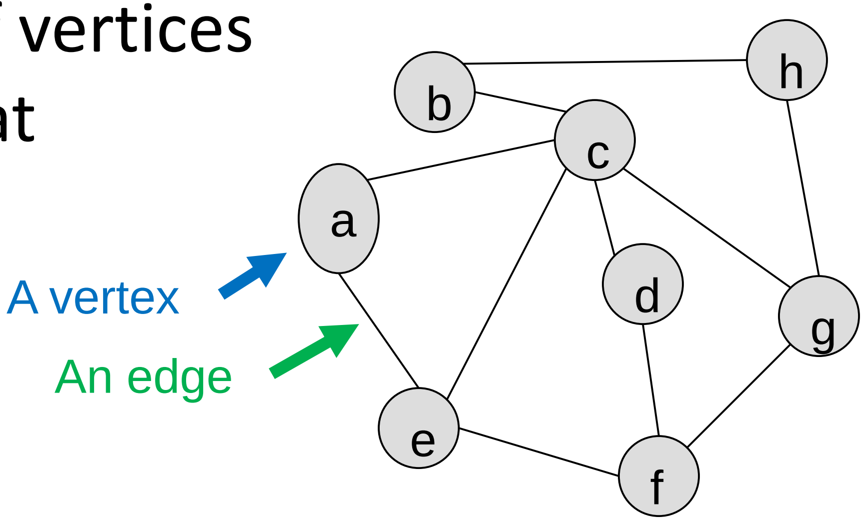
- A **graph** is a collection of vertices (or nodes) and edges that connect vertices

V
a
b
c
d
e
f
g
h

E
(a,c)
(a,e)
(b,h)
(b,c)
(c,e)
(c,d)
(c,g)
(d,f)
(e,f)
(f,g)
(g,h)

|V|=n=8

|E|=m=11



- Let **V** be the set of vertices
- Let **E** be the set of edges
- Let **|V| or n** refer to the number of vertices
- Let **|E| or m** refer to the number of edges

Graphs in the Real World

- Social networks
- Computer networks / Internet
- Path planning
- Interaction diagrams
- Bioinformatics

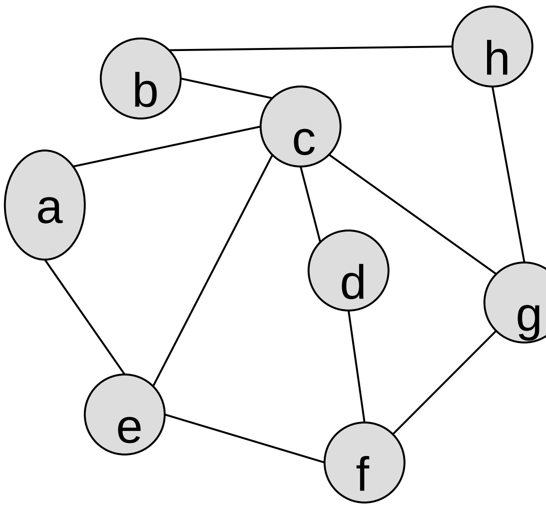
Basic Graph Representation

- Can simply store edges in list/array
  - Unsorted
  - Sorted

V	E
a	(a,c)
b	(a,e)
c	(b,h)
d	(b,c)
e	(c,e)
f	(c,d)
g	(c,g)
h	(d,f)
	(e,f)
	(f,g)
	(g,h)

|V|=n=8

|E|=m=11



Graph ADT

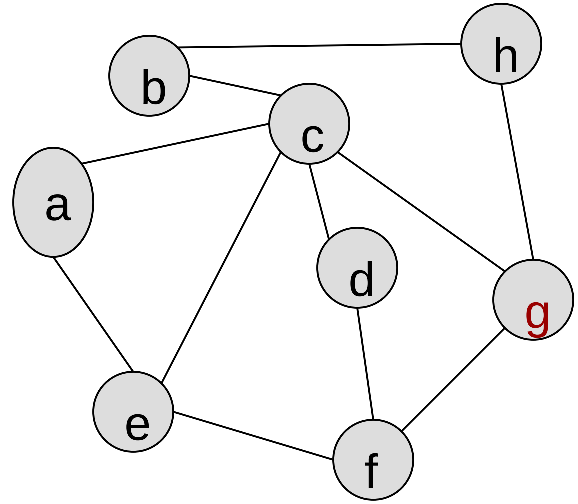
- What operations would you want to perform on a graph?
- `addVertex() : Vertex`
- `addEdge(v1, v2)`
- `getAdjacencies(v1) : List<Vertices>`
  - `Returns any vertex with an edge from v1 to itself`
- `removeVertex(v)`
- `removeEdge(v1, v2)`
- `edgeExists(v1, v2) : bool`

```
#include<iostream>
using namespace std;

template <typename V, typename E>
class Graph{

};
```

Perfect for templating the data associated with a vertex and edge as V and E

More Common Graph Representations

- Graphs are really just a list of lists
  - List of vertices each having their own list of adjacent vertices
- Alternatively, sometimes graphs are also represented with an adjacency matrix
  - Entry at (i,j) = 1 if there is an edge between vertex i and j, 0 otherwise



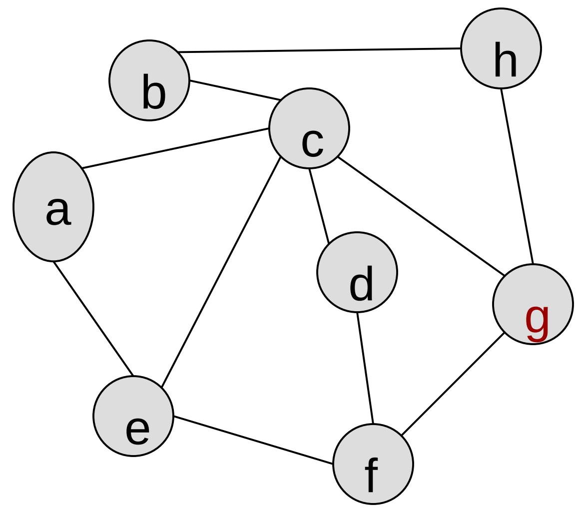
List of Vertices	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Let |V| = n = # of vertices and |E| = m = # of edges
- Adjacency List Representation
  - O(________________) memory storage
  - Existence of an edge requires searching adjacency list
- Adjacency Matrix Representation
  - O(________________) storage
  - Existence of an edge requires O(__________) lookup



List of Vertices

a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

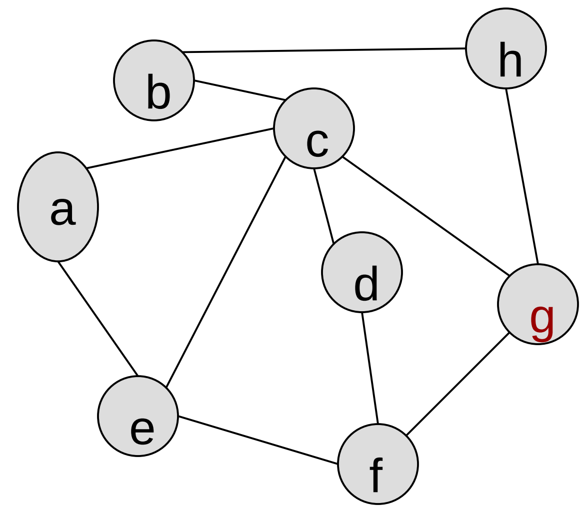
Adjacency Lists

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Let $|V| = n$ = # of vertices and $|E| = m$ = # of edges
- Adjacency List Representation
  - $O(|V| + |E|)$ memory storage
  - Existence of an edge requires searching adjacency list
  - Define **degree** to be the number of edges incident on a vertex (deg(a) = 2, deg(c) = 5, etc.
- Adjacency Matrix Representation
  - $O(|V|^2)$ storage
  - Existence of an edge requires O(1) lookup (e.g. matrix[i][j] == 1)



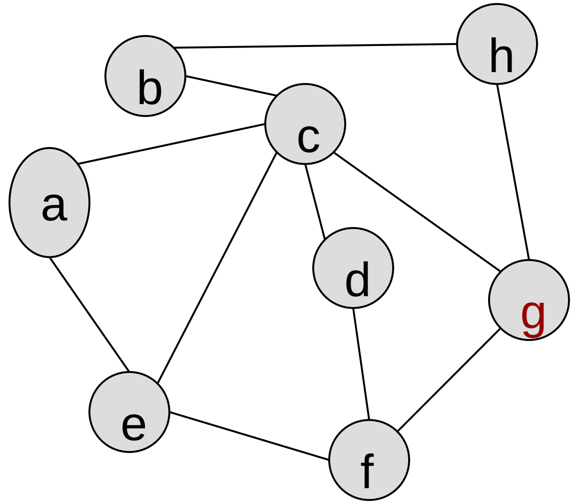
List of Vertices	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a…
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b



List of Vertices

a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

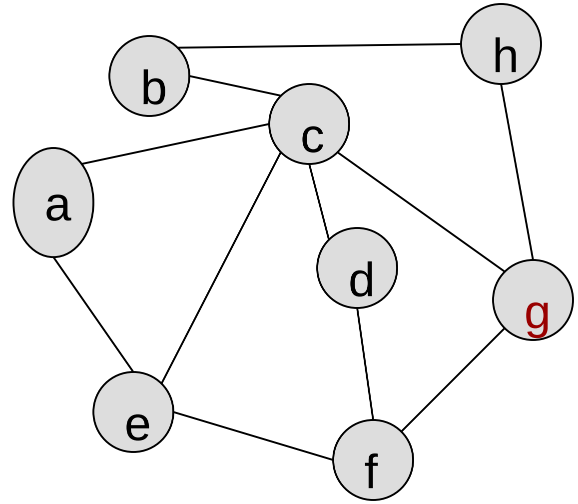
Adjacency Lists

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Representations

- Can 'a' get to 'b' in two hops?
- Adjacency List
  - For each neighbor of a…
  - Search that neighbor's list for b
- Adjacency Matrix
  - Take the dot product of row a & column b



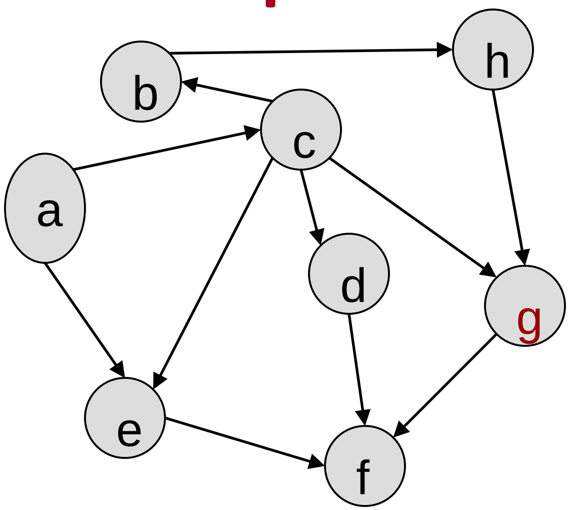
List of Vertices	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

	a	**b**	**c**	**d**	**e**	**f**	**g**	**h**
a	0	0	1	0	1	0	0	0
b	0	0	1	0	0	0	0	1
c	1	1	0	1	1	0	1	0
d	0	0	1	0	0	1	0	0
e	1	0	1	0	0	1	0	0
f	0	0	0	1	1	0	1	0
g	0	0	1	0	0	1	0	1
h	0	1	0	0	0	0	1	0

Adjacency Matrix Representation

Directed vs. Undirected Graphs

- In the previous graphs, edges were **undirected** (meaning edges are 'bidirectional' or 'reflexive')
  - An edge (u,v) implies (v,u)
- In **directed** graphs, links are unidirectional
  - An edge (u,v) does not imply (v,u)
  - For Edge (u,v): the **source** is u, **target** is v
- For adjacency list form, you may need 2 lists per vertex for both predecessors and successors



List of Vertices	Adjacency Lists
a	c,e
b	c,h
c	a,b,d,e,g
d	c,f
e	a,c,f
f	d,e,g
g	c,f,h
h	b,g

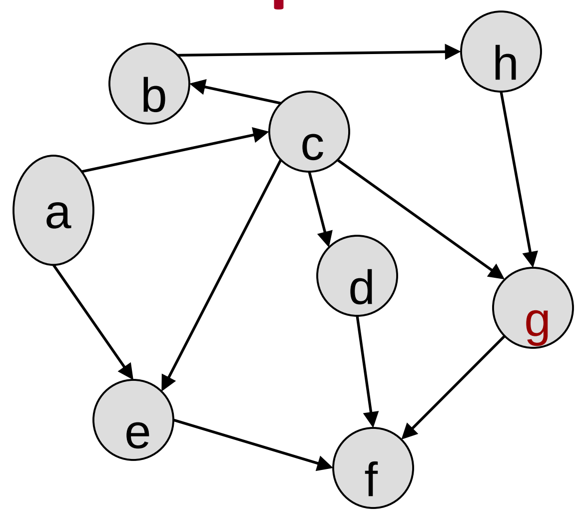
Target

Source	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	0	0	0	0	0	1
c	0	1	0	1	1	0	1	0
d	0	0	0	0	0	1	0	0
e	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	1	0	0
h	0	0	0	0	0	0	1	0

Adjacency Matrix Representation

Directed vs. Undirected Graphs

- In directed graph with edge (u,v) we define
  - Successor(u) = v
  - Predecessor(v) = u
- Using an adjacency list representation *may* warrant two lists predecessors and successors



List of Vertices

List of Vertices	Succs	Preds
a	c,e	
b	h	c
c	b,d,e,g	a
d	f	c
e	f	a,c
f		d, e, g
g	f	c,h
h	g	b

Target

Source

	a	b	c	d	e	f	g	h
a	0	0	1	0	1	0	0	0
b	0	0	0	0	0	0	0	1
c	0	1	0	1	1	0	1	0
d	0	0	0	0	0	1	0	0
e	0	0	0	0	0	1	0	0
f	0	0	0	0	0	0	0	0
g	0	0	0	0	0	1	0	0
h	0	0	0	0	0	0	1	0

Adjacency Matrix Representation

Graph Runtime, |V| = n, |E| =m

Operation vs Implementation for Edges	Add edge	Delete Edge	Test Edge	Enumerate edges for single vertex
Unsorted array or Linked List				
Sorted array				
Adjacency List				
Adjacency Matrix				

Graph Runtime, |V| = n, |E| =m

Operation vs Implementation for Edges	Add edge	Delete Edge	Test Edge	Enumerate edges for single vertex
Unsorted array or Linked List	Θ(1)	Θ(m)	Θ(m)	Θ(m)
Sorted array	Θ(m)	Θ(m)	Θ(log m) [if binary search used]	Θ(log m)+Θ(deg(v)) [if binary search used]
Adjacency List	Time to find List for a given vertex + Θ(1)	Time to find List for a given vertex + Θ(deg(v))	Time to find List for a given vertex + Θ(deg(v))	Time to find List for a given vertex + Θ(deg(v))
Adjacency Matrix	Θ(1)	Θ(1)	Θ(1)	Θ(deg(v))

Graph Memory Requirements

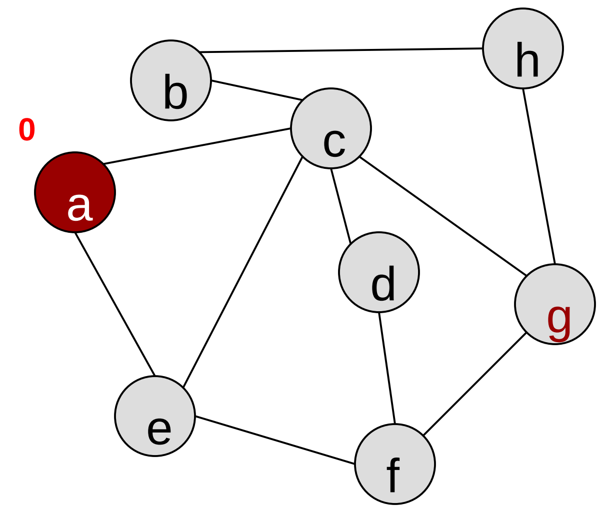
- For an adjacency list:
- For adjacency matrix:
- We call a graph ***sparse*** if |E| is O(n)
- We call a graph ***dense*** if |E| is Big_Omega(n^2)
- What representation is better for a sparse graph?
- What representation is better for a dense graph?

USC Viterbi School of Engineering

BREADTH-FIRST SEARCH

Breadth-First Search

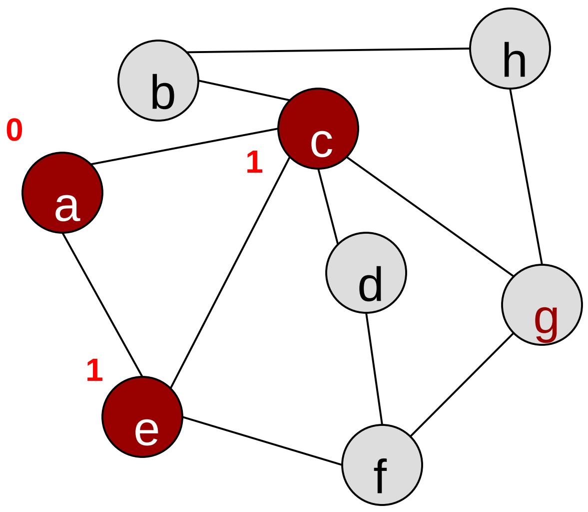
- Given a graph with vertices, V, and edges, E, and a starting vertex that we'll call u
- BFS starts at u ('a' in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex



Depth 0: a

Breadth-First Search

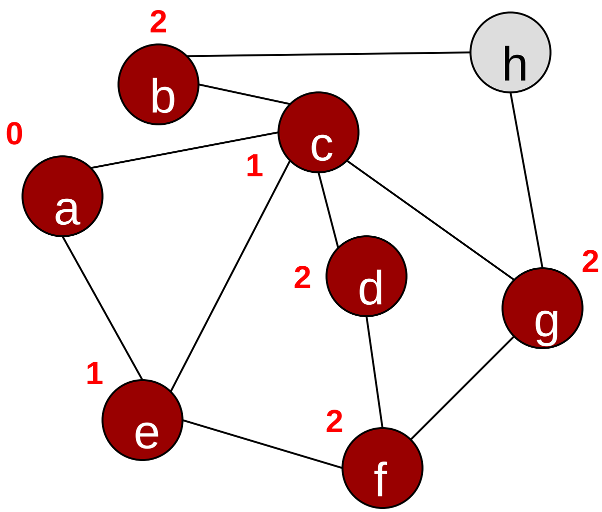
- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u ('a' in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex



Depth 0: a
Depth 1: c,e

Breadth-First Search

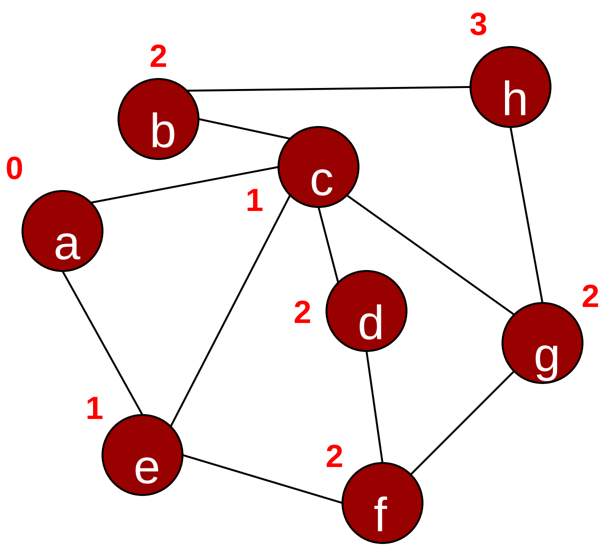
- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u ('a' in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex



Depth 0: a
Depth 1: c,e
Depth 2: b,d,f,g

Breadth-First Search

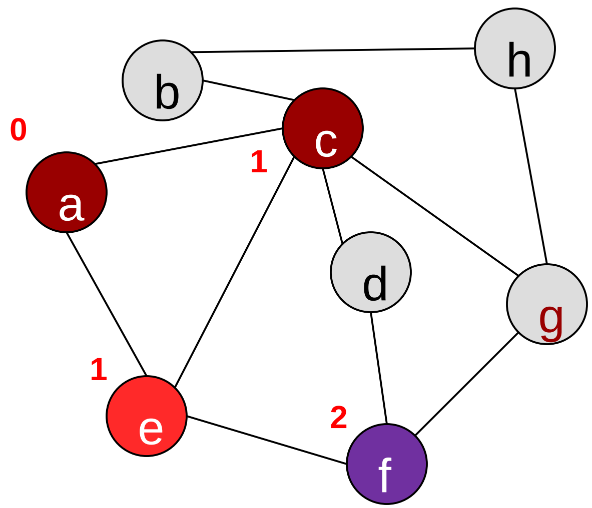
- Given a graph with vertices, V, and edges, E, and a starting vertex, u
- BFS starts at u ('a' in the diagram to the left) visits nearest neighbors, then to their neighbors and so on
- Goal: Find shortest paths from the start vertex to every other vertex



Depth 0: a
Depth 1: c,e
Depth 2: b,d,f,g
Depth 3: h

Developing the Algorithm

- Key idea: Must explore all nearer neighbors before exploring further-away neighbors
- From 'a' we find 'e' and 'c'
  - Must explore all vertices at depth i before any vertices at depth i+1
  - What data structure may help us?



Depth 0: a
Depth 1: c,e
Depth 2: b,d,f,g
Depth 3: h

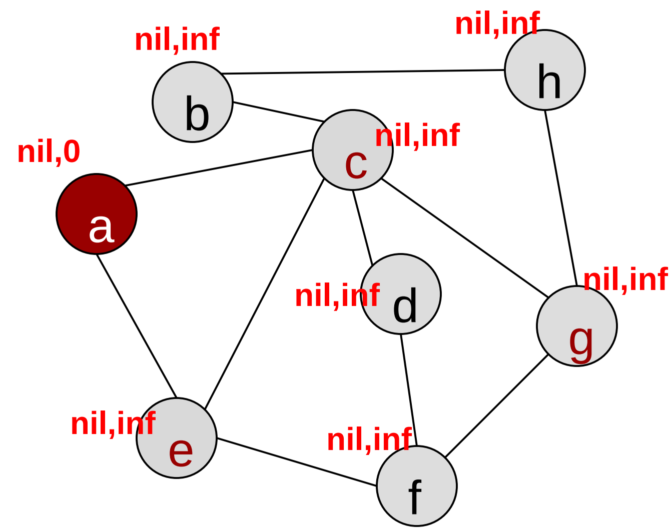
Developing the Algorithm

- Exploring all vertices in the order they are found implies we will explore all vertices at shallower depth before greater depth
  - Keep a first-in / first-out queue (FIFO) of neighbors found
- Put newly found vertices in the back and pull out a vertex from the front to explore next
- We don't want to put a vertex in the queue more than once…
  - 'mark' a vertex the first time we encounter it
  - only allow unmarked vertices to be put in the queue
- May also keep a 'predecessor' array: Allows us to find a shortest-path back to the start vertex

Breadth-First Search

Algorithm:

```
BFS(G,u)
 for each vertex v
      pred[v] = nil, d[v] = inf.
 Q = new Queue
 Q.enqueue(u),  d[u]=0
 while Q is not empty
    v = Q.front(); Q.dequeue()
    foreach neighbor, w, of v:
        if pred[w] == nil // w not found
            Q.enqueue(w)
            pred[w] = v,  d[w] = d[v] + 1
```



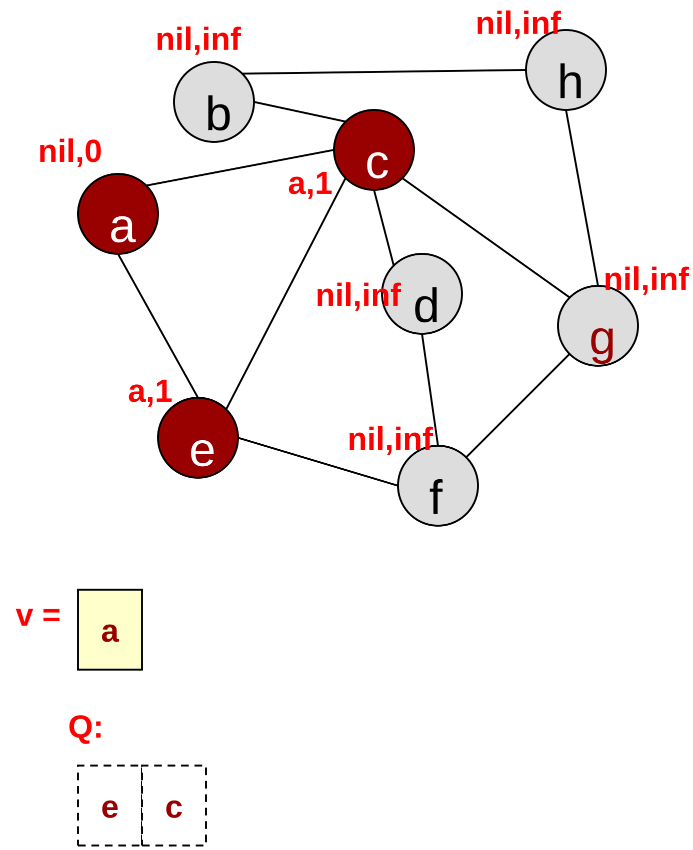
Q:

a

Breadth-First Search

Algorithm:

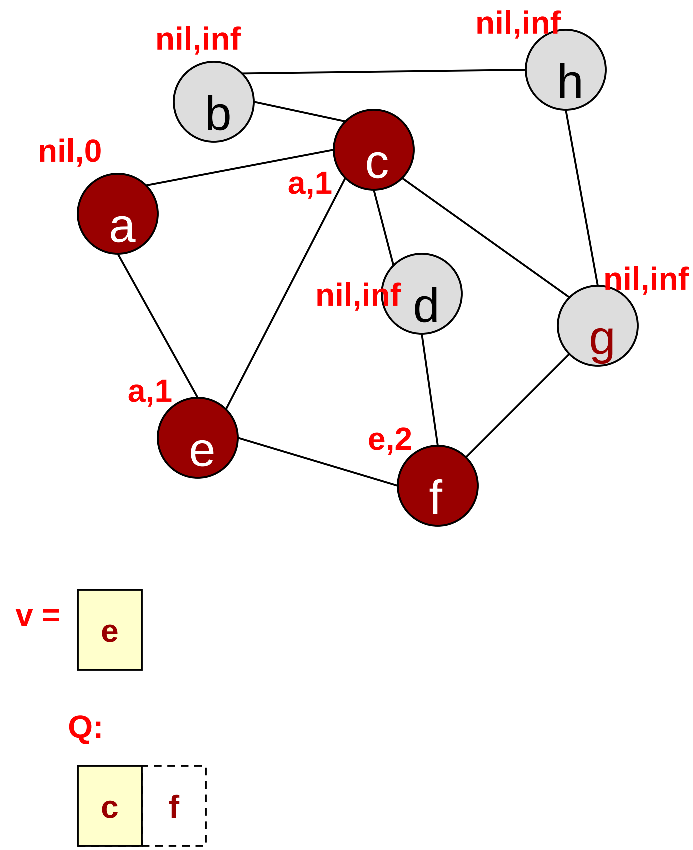
```
BFS(G,u)
for each vertex v
    pred[v] = nil, d[v] = inf.
Q = new Queue
Q.enqueue(u),  d[u]=0
while Q is not empty
   v = Q.front(); Q.dequeue()
   foreach neighbor, w, of v:
       if pred[w] == nil // w not found
           Q.enqueue(w)
           pred[w] = v,  d[w] = d[v] + 1
```



Breadth-First Search

Algorithm:

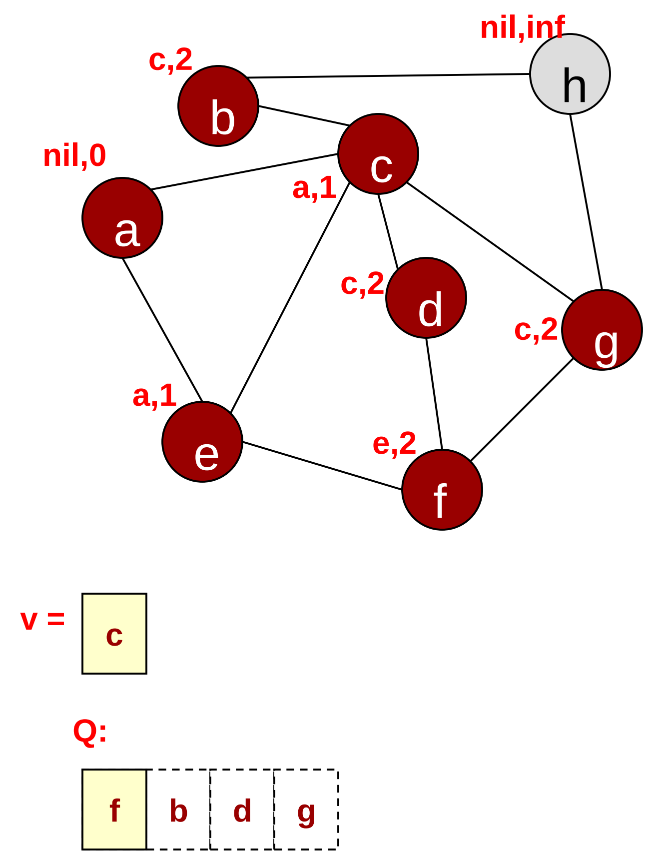
```
BFS(G,u)
  for each vertex v
      pred[v] = nil, d[v] = inf.
  Q = new Queue
  Q.enqueue(u),  d[u]=0
  while Q is not empty
     v = Q.front(); Q.dequeue()
     foreach neighbor, w, of v:
         if pred[w] == nil // w not found
             Q.enqueue(w)
             pred[w] = v,  d[w] = d[v] + 1
```



Breadth-First Search

Algorithm:

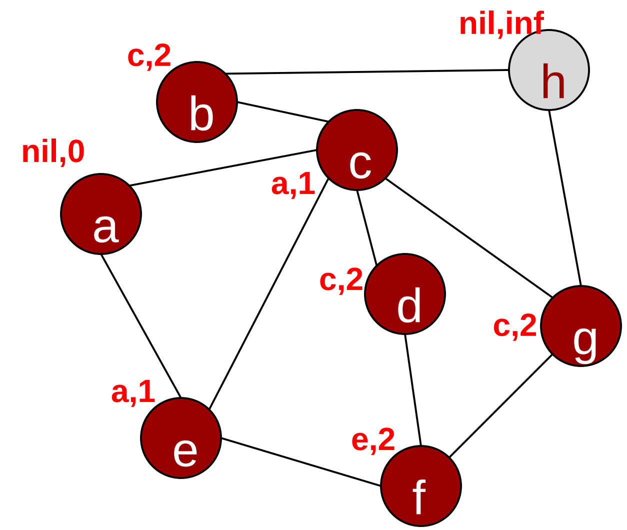
```
BFS(G,u)
for each vertex v
     pred[v] = nil, d[v] = inf.
Q = new Queue
Q.enqueue(u),  d[u]=0
while Q is not empty
   v = Q.front(); Q.dequeue()
   foreach neighbor, w, of v:
       if pred[w] == nil // w not found
           Q.enqueue(w)
           pred[w] = v,  d[w] = d[v] + 1
```



Breadth-First Search

Algorithm:

```
BFS(G,u)
  for each vertex v
       pred[v] = nil, d[v] = inf.
  Q = new Queue
  Q.enqueue(u),  d[u]=0
  while Q is not empty
     v = Q.front(); Q.dequeue()
     foreach neighbor, w, of v:
         if pred[w] == nil // w not found
             Q.enqueue(w)
             pred[w] = v,  d[w] = d[v] + 1
```



v = f

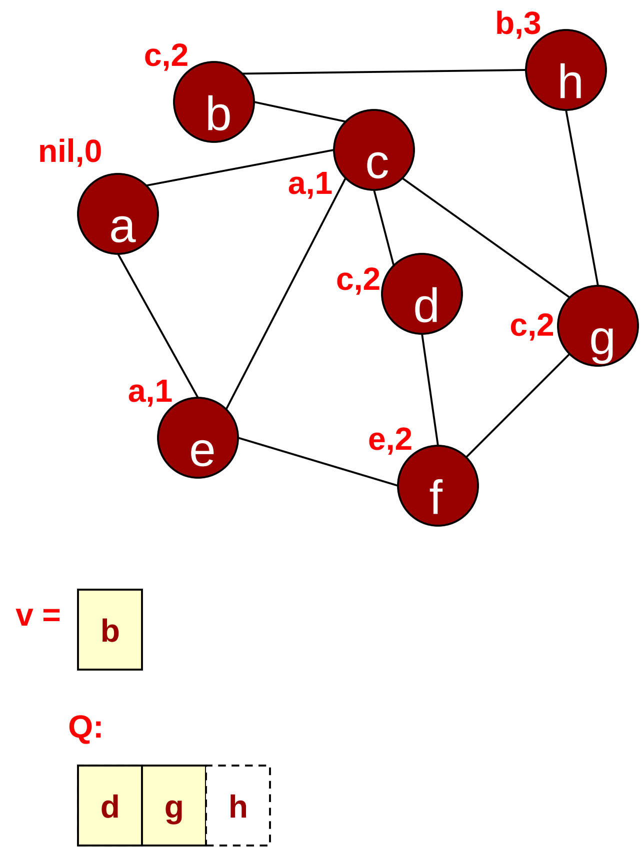
Q:

b	d	g

Breadth-First Search

Algorithm:

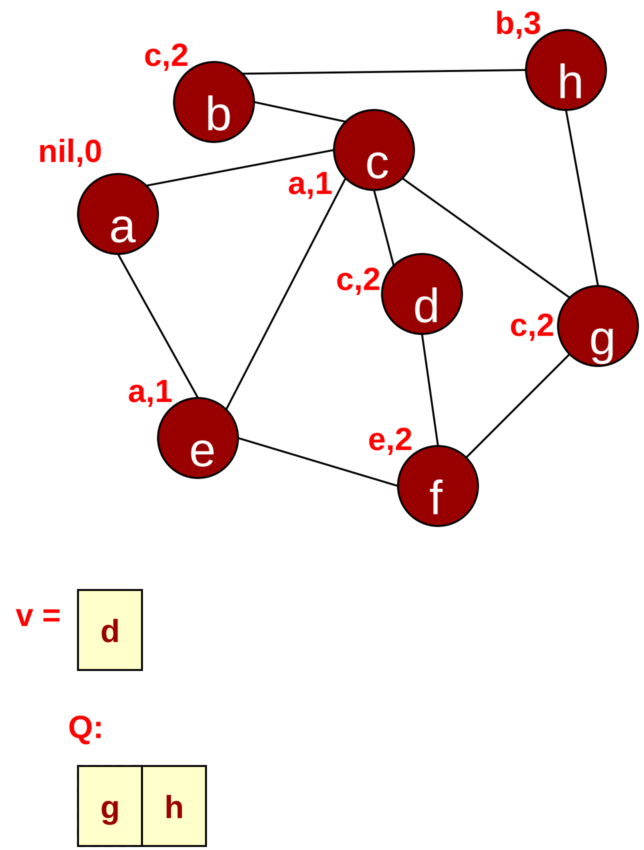
```
BFS(G,u)
  for each vertex v
      pred[v] = nil, d[v] = inf.
  Q = new Queue
  Q.enqueue(u),  d[u]=0
  while Q is not empty
     v = Q.front(); Q.dequeue()
     foreach neighbor, w, of v:
        if pred[w] == nil // w not found
           Q.enqueue(w)
           pred[w] = v,  d[w] = d[v] + 1
```



Breadth-First Search

Algorithm:

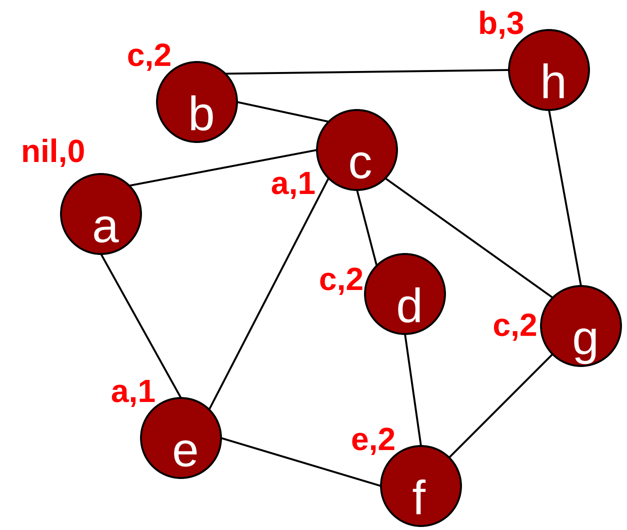
```
BFS(G,u)
for each vertex v
     pred[v] = nil, d[v] = inf.
Q = new Queue
Q.enqueue(u),  d[u]=0
while Q is not empty
   v = Q.front(); Q.dequeue()
   foreach neighbor, w, of v:
        if pred[w] == nil // w not found
             Q.enqueue(w)
             pred[w] = v,  d[w] = d[v] + 1
```



Breadth-First Search

Algorithm:

```
BFS(G,u)
for each vertex v
    pred[v] = nil, d[v] = inf.
Q = new Queue
Q.enqueue(u),  d[u]=0
while Q is not empty
   v = Q.front(); Q.dequeue()
   foreach neighbor, w, of v:
       if pred[w] == nil // w not found
           Q.enqueue(w)
           pred[w] = v,  d[w] = d[v] + 1
```



v = g

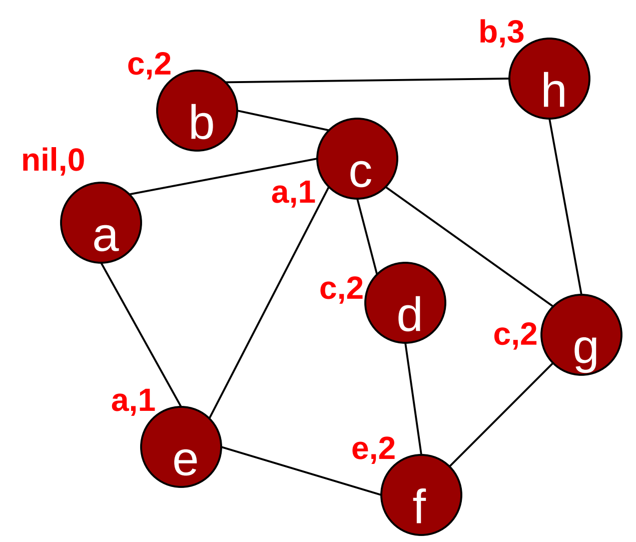
Q:

h

Breadth-First Search

Algorithm:

```
BFS(G,u)
  for each vertex v
      pred[v] = nil, d[v] = inf.
  Q = new Queue
  Q.enqueue(u),  d[u]=0
  while Q is not empty
     v = Q.front(); Q.dequeue()
     foreach neighbor, w, of v:
         if pred[w] == nil // w not found
             Q.enqueue(w)
             pred[w] = v,  d[w] = d[v] + 1
```

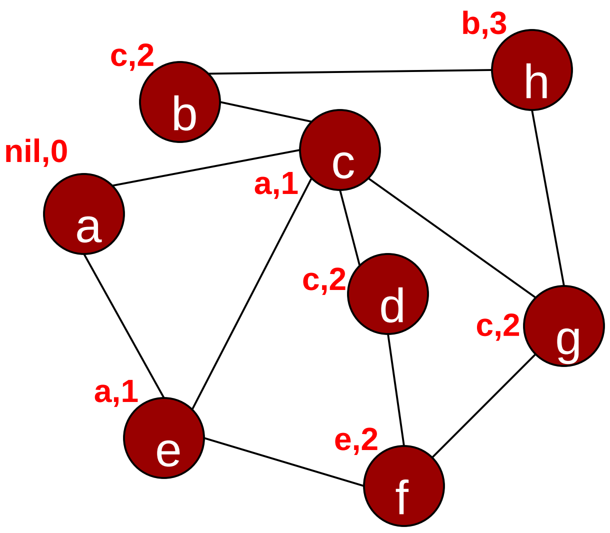


v = h

Q:

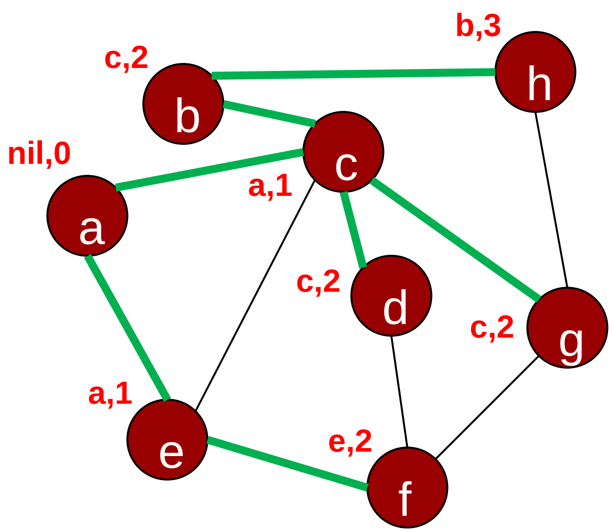
Breadth-First Search

- Shortest paths can be found be walking predecessor value from any node backward
- Example:
  - Shortest path from a to h
  - Start at h
  - Pred[h] = b (so walk back to b)
  - Pred[b] = c (so walk back to c)
  - Pred[c] = a (so walk back to a)
  - Pred[a] = nil … no predecessor, Done!!

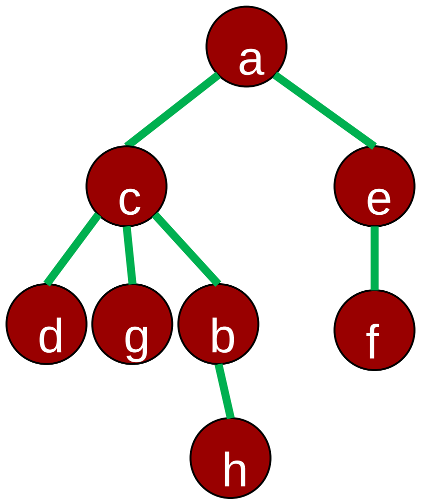


Breadth-First Search Trees

- BFS (and later DFS) will induce a tree subgraph (i.e. acyclic, one parent each) from the original graph
  - BFS is tree of shortest paths from the source to all other vertices (in connected component)



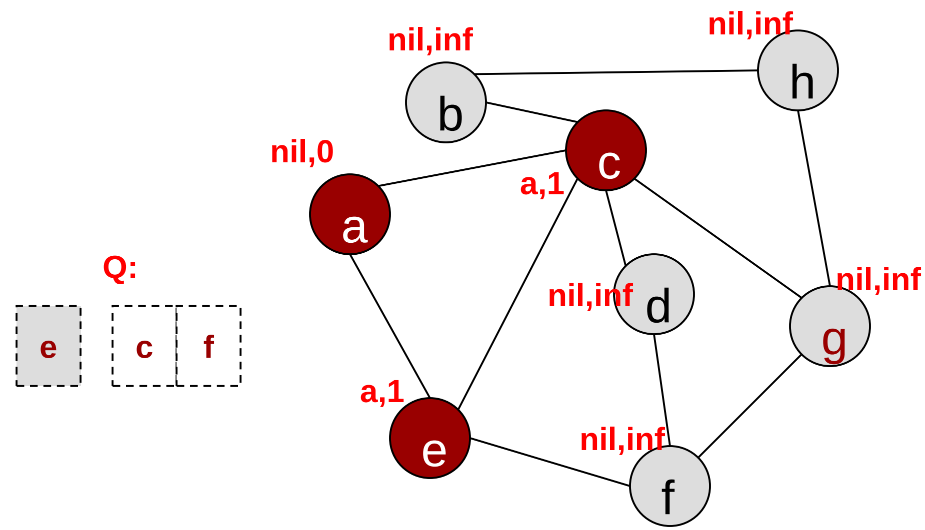
Original graph, G



BFS Induced Tree

Correctness

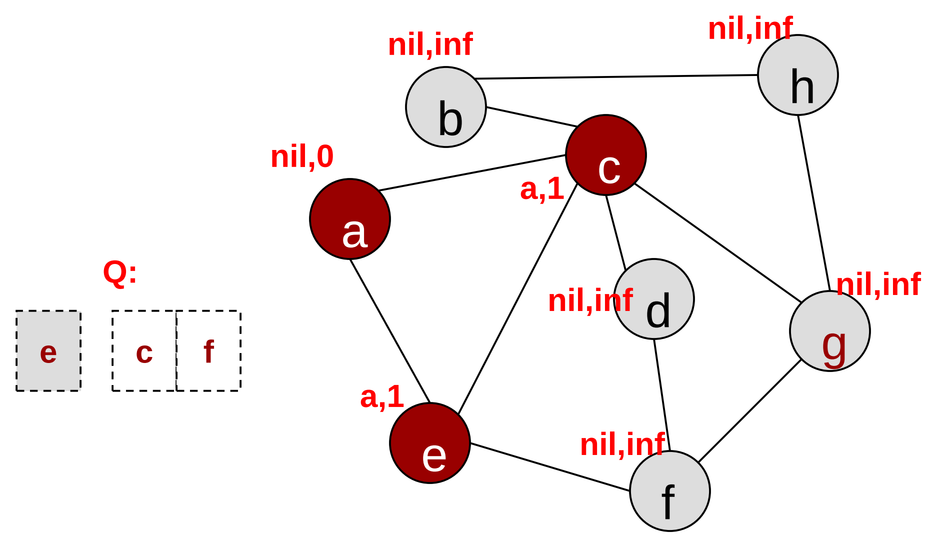
- Define
  - dist(s,v) = correct shortest distance
  - d[v] = BFS computed distance
  - p[v] = predecessor of v
- Loop invariant
  - What can we say about the nodes in the queue, their d[v] values, relationship between d[v] and dist[v], etc.?




```
BFS(G,u)
for each vertex v
    pred[v] = nil, d[v] = inf.
Q = new Queue
Q.enqueue(u),  d[u]=0
while Q is not empty
   v = Q.front(); Q.dequeue()
   foreach neighbor, w, of v:
      if pred[w] == nil // w not found
         Q.enqueue(w)
         pred[w] = v,  d[w] = d[v] + 1
```

Correctness

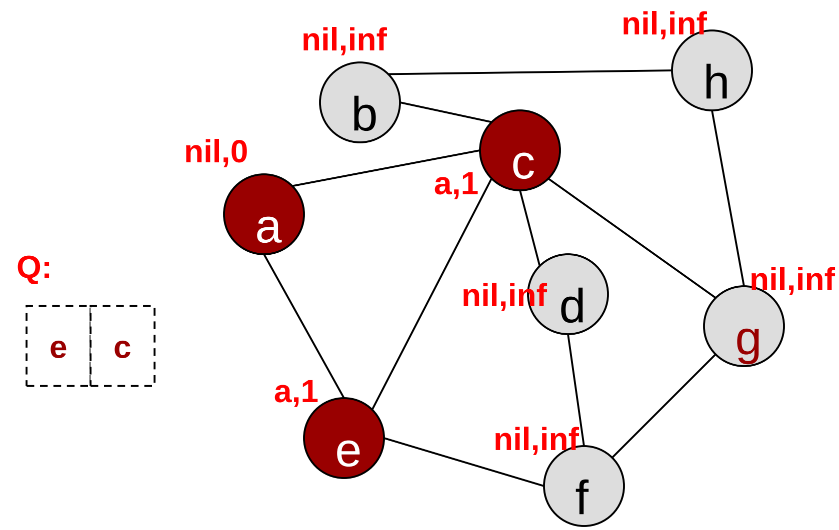
- Define
  - dist(s,v) = correct shortest distance
  - d[v] = BFS computed distance
  - p[v] = predecessor of v
- Loop invariant
  - All vertices with p[v] != nil (i.e. already in the queue or popped from queue) have d[v] = dist(s,v)
  - The distance of the nodes in the queue are sorted
    - If Q = {v_1, v_2, …, v_r} then d[v_1] <= d[v_2] <= … <= d[v_r]
  - The nodes in the queue are from 2 adjacent layers/levels
    - i.e. d[v_k] <= d[v_1] + 1
    - Suppose there is a node from a 3rd level (d[v_1] + 2), it must have been found by some, vi, where d[v_i] = d[v_1]+1




```
BFS(G,u)
for each vertex v
    pred[v] = nil, d[v] = inf.
Q = new Queue
Q.enqueue(u),  d[u]=0
while Q is not empty
   v = Q.front(); Q.dequeue()
   foreach neighbor, w, of v:
      if pred[w] == nil // w not found
         Q.enqueue(w)
         pred[w] = v,  d[w] = d[v] + 1
```

Breadth-First Search

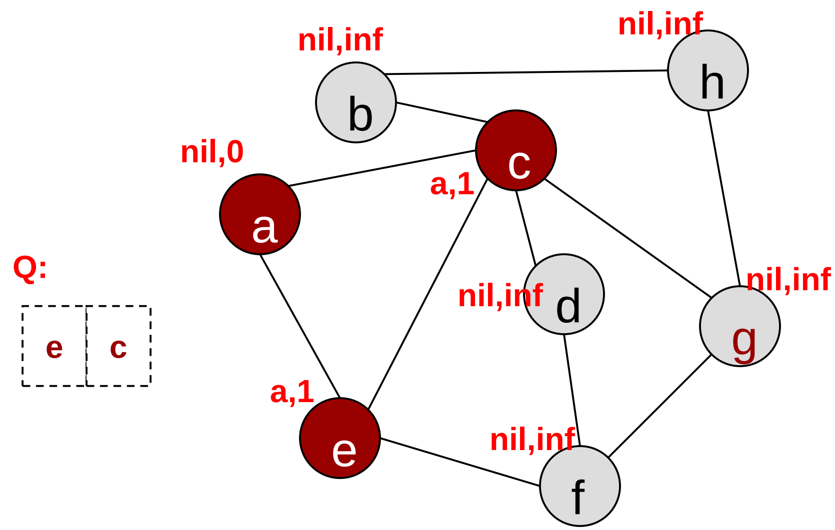
- Analyze the run time of BFS for a graph with n vertices and m edges
  - Find T(n,m)
- How many times does loop on line 5 iterate?
- How many times loop on line 7 iterate?




```
BFS(G,u)
1  for each vertex v
2      pred[v] = nil, d[v] = inf.
3  Q = new Queue
4  Q.enqueue(u),  d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9             Q.enqueue(w)
10            pred[w] = v,  d[w] = d[v] + 1
```

Breadth-First Search

- Analyze the run time of BFS for a graph with n vertices and m edges
  - Find T(n)
- How many times does loop on line 5 iterate?
  - N times (one iteration per vertex)
- How many times loop on line 7 iterate?
  - For each vertex, v, the loop executes deg(v) times
  - $= \sum_{v \in V} \theta[1 + \deg(v)]$
  - $= \theta(\sum_v 1) + \theta(\sum_v \deg(v))$
  - = Θ (n) + Θ (m)
- Total = Θ(n+m)




```
BFS(G,u)
1  for each vertex v
2      pred[v] = nil, d[v] = inf.
3  Q = new Queue
4  Q.enqueue(u),  d[u]=0
5  while Q is not empty
6     v = Q.front(); Q.dequeue()
7     foreach neighbor, w, of v:
8         if pred[w] == nil // w not found
9             Q.enqueue(w)
10            pred[w] = v,  d[w] = d[v] + 1
```

DFS Algorithm

- DFS visits and completes all children before completing (and going on to a sibling)
- Process:
  - Visit a node
  - Mark as visited (started)
  - For each visited neighbor, visit it and perform DFS on all of their children
  - Only then, mark as finished
- Let's trace recursive DFS!
- If cycles in the graph, mark nodes so we know to stop examining them:
  - White = unvisited,
  - Gray = visited but not finished
  - Black = finished

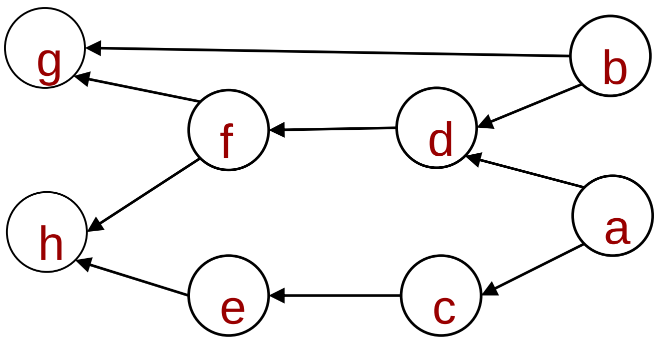
```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
       DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

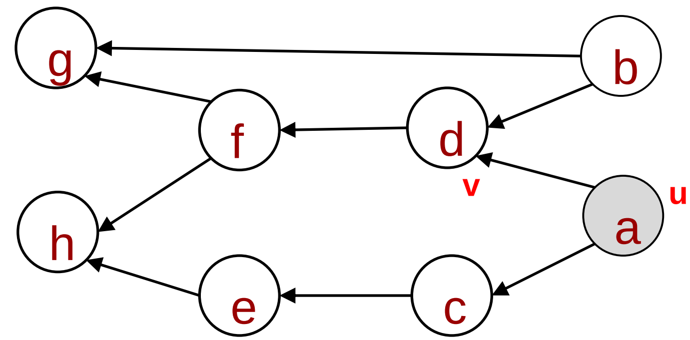
```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u,l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```

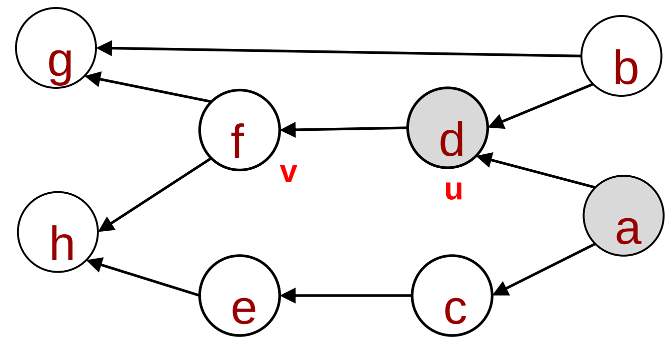


DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
       DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```

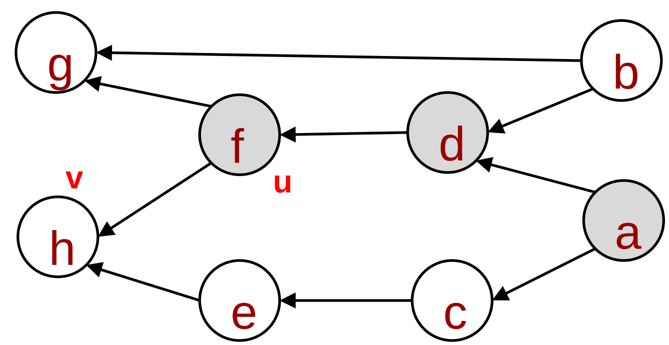


DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```

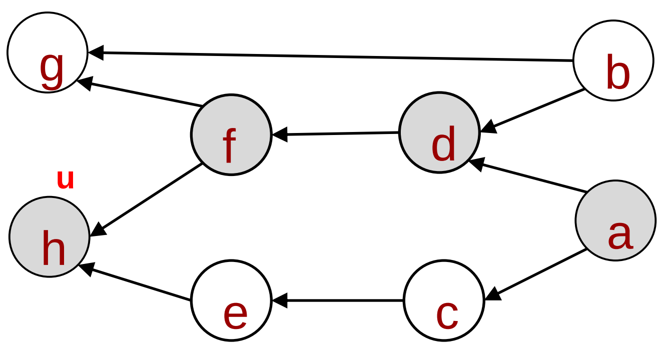


DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



DFS-Visit(G,h,l):

DFS-Visit(G,f,l):

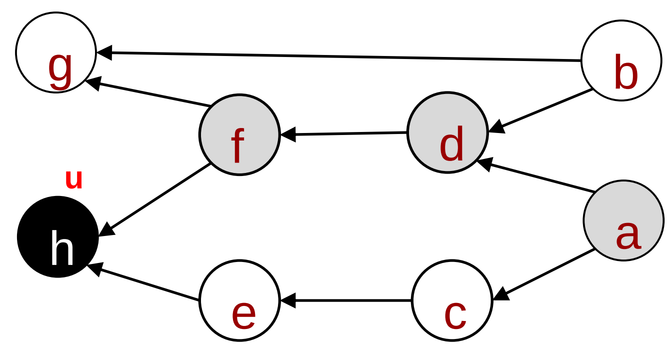
DFS-Visit(G,d,l):

DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
  for each vertex u
    u.color = WHITE
  finish_list = empty_list
  for each vertex u do
     if u.color == WHITE then
        DFS-Visit (G, u, finish_list)
  return finish_list
```

```
DFS-Visit (G, u, l)
    u.color = GRAY
    for each vertex v in Adj(u) do
      if v.color = WHITE then
          DFS-Visit (G, v)
    u.color = BLACK
    l.append(u)
```



Finish_list:

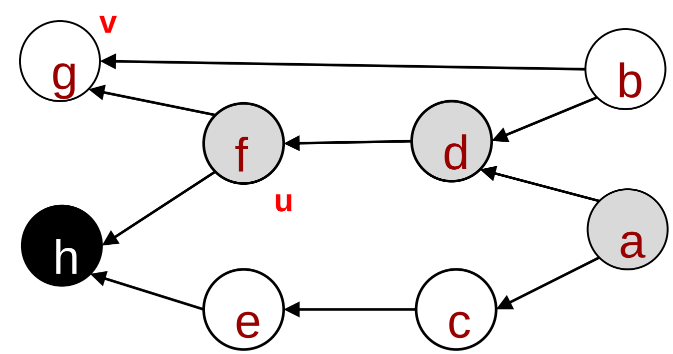
h

DFS-Visit(G,h,l):
DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

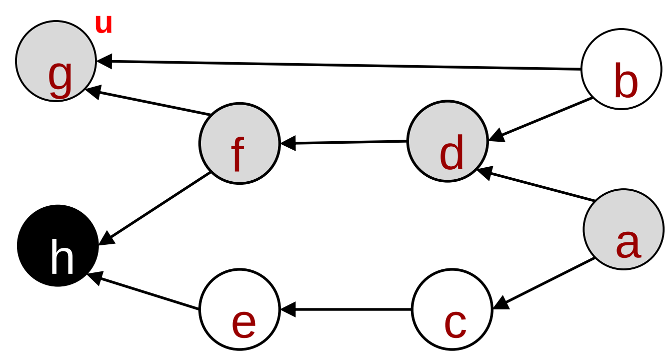
h

DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
      DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u,l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

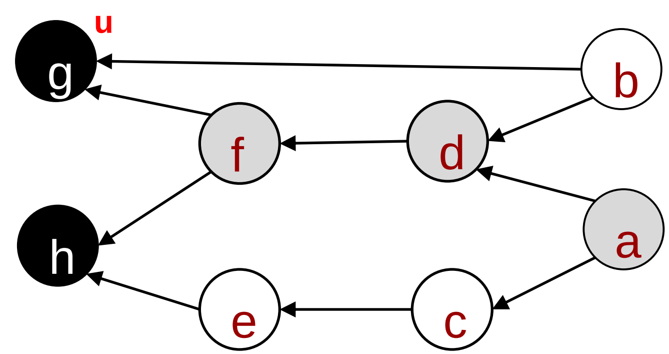
h

DFS-Visit(G,g,l):
DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
       DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

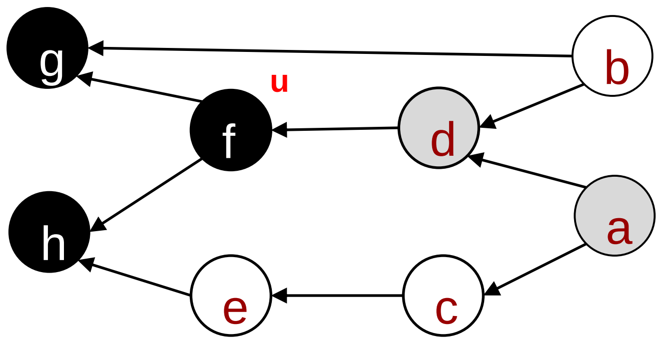
h,
g

DFS-Visit(G,g,l):
DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
  for each vertex u
    u.color = WHITE
  finish_list = empty_list
  for each vertex u do
     if u.color == WHITE then
       DFS-Visit (G, u, finish_list)
  return finish_list
```

```
DFS-Visit (G, u, l)
    u.color = GRAY
    for each vertex v in Adj(u) do
      if v.color = WHITE then
          DFS-Visit (G, v)
    u.color = BLACK
    l.append(u)
```



Finish_list:

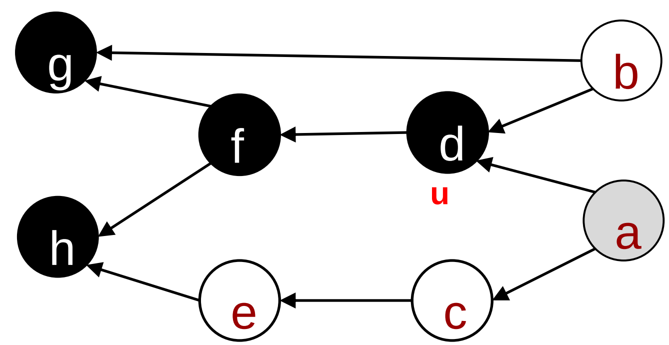
h,
g,
f

DFS-Visit(G,f,l):
DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

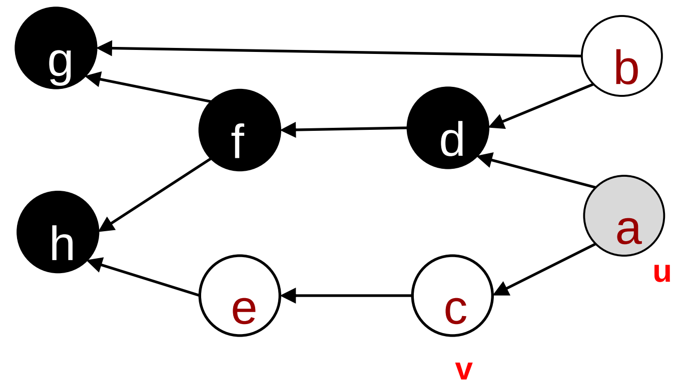
h,
g,
f,
d

DFS-Visit(G,d,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
1  for each vertex u
2    u.color = WHITE
3  finish_list = empty_list
4  for each vertex u do
5     if u.color == WHITE then
6        DFS-Visit (G, u, finish_list)
7  return finish_list
```

```
DFS-Visit (G, u, l)
1    u.color = GRAY
2    for each vertex v in Adj(u) do
3      if v.color = WHITE then
4          DFS-Visit (G, v)
5    u.color = BLACK
6    l.append(u)
```



Finish_list:

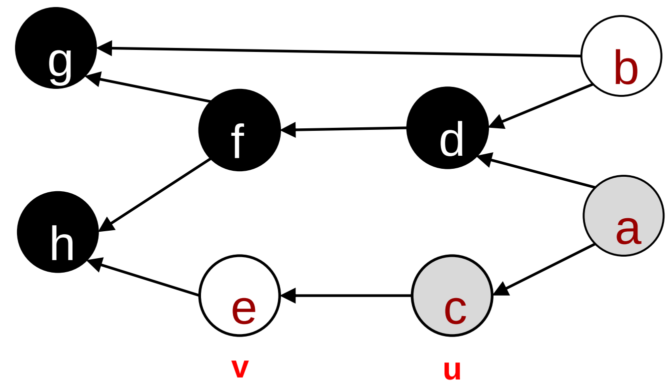
h,
g,
f,
d

DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

h,
g,
f,
d

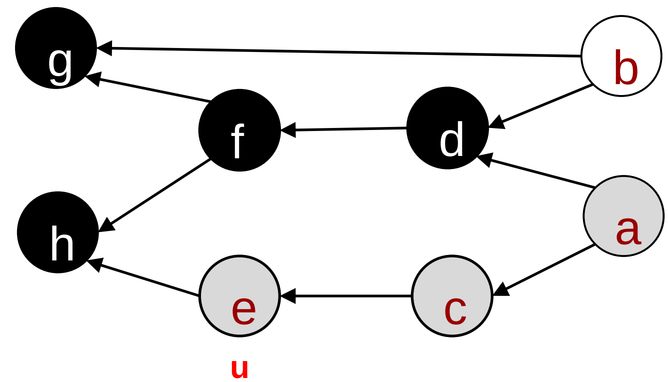
DFS-Visit(G,c,l):

DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

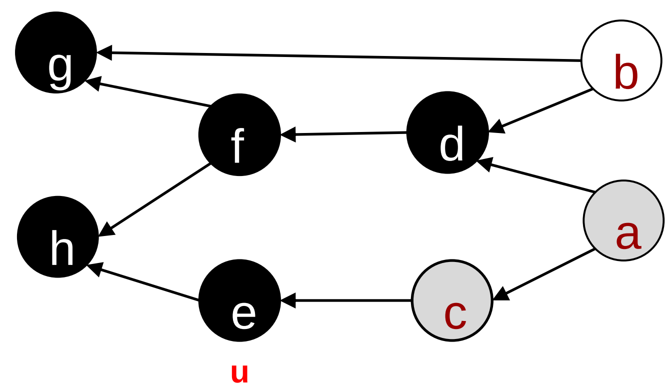
h, g, f, d

DFS-Visit(G,e,l):
DFS-Visit(G,c,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

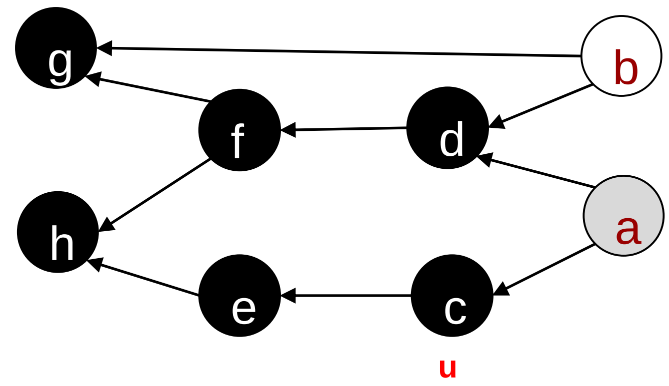
h,
g,
f,
d.
e

DFS-Visit(G,e,l):
DFS-Visit(G,c,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

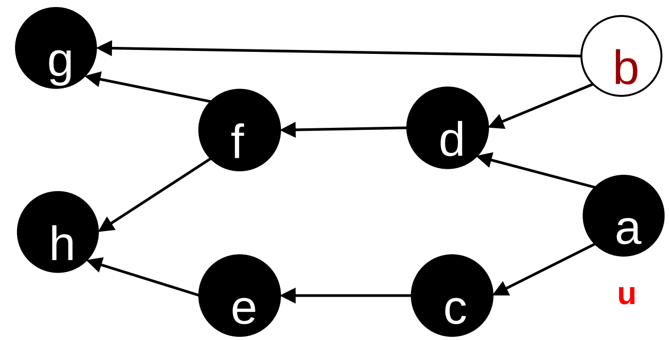
h, g, f, d. e, c

DFS-Visit(G,c,l):
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

h,
g,
f,
d.
e,
c,
a

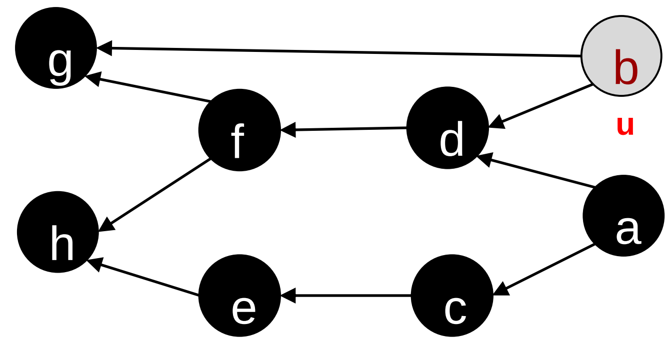
DFS-Visit(G,a,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
      DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```

May iterate through many complete vertices before finding b to launch a new search from



Finish_list:

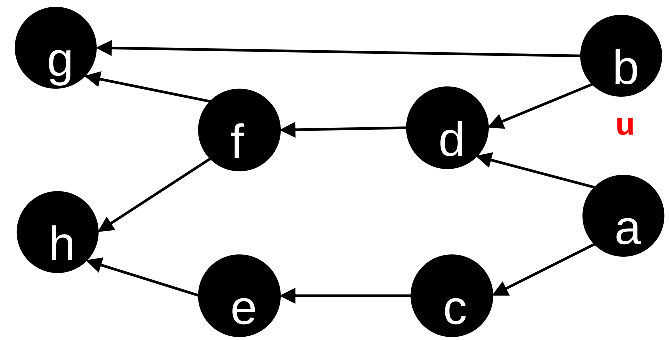
h, g, f, d. e, c, a

DFS-Visit(G,b,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
     DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



Finish_list:

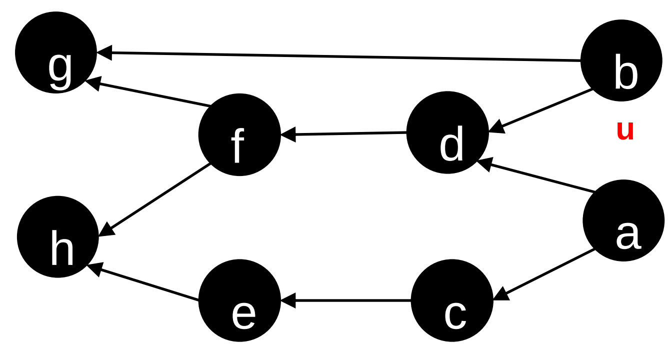
h,
g,
f,
d.
e,
c,
a,
b

DFS-Visit(G,b,l):

Depth First-Search

```
DFS-All (G)
for each vertex u
  u.color = WHITE
finish_list = empty_list
for each vertex u do
   if u.color == WHITE then
      DFS-Visit (G, u, finish_list)
return finish_list
```

```
DFS-Visit (G, u, l)
  u.color = GRAY
  for each vertex v in Adj(u) do
    if v.color = WHITE then
        DFS-Visit (G, v)
  u.color = BLACK
  l.append(u)
```



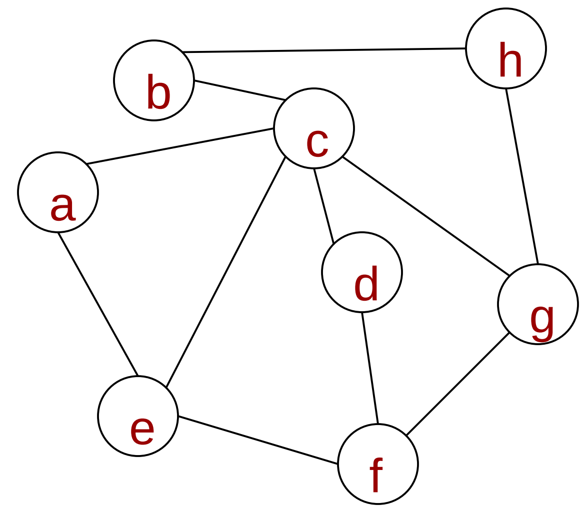
Finish_list:

h,
g,
f,
d.
e,
c,
a,
b

ITERATIVE VERSION

Depth First-Search

```
DFS (G,s)
for each vertex u
  u.color = WHITE
st = new Stack
st.push_back(s)
while st not empty
   u = st.back()
   if u.color == WHITE then
     u.color = GRAY
     foreach vertex v in Adj(u) do
        if v.color == WHITE
           st.push_back(v)
   else if u.color != WHITE
     u.color = BLACK
     st.pop_back()
```



st: | a |

BFS vs. DFS Algorithm

- BFS and DFS are more similar than you think
  - Do we use a FIFO/Queue (BFS) or LIFO/Stack (DFS) to store vertices as we find them

```
BFS-Visit (G, start_node)
  for each vertex u
    u.color = WHITE
    u.pred = nil
  bfsq = new Queue
  bfsq.push_back(start_node)
  while bfsq not empty
     u = bfsq.pop_front()
     if u.color == WHITE
       u.color = GRAY
       foreach vertex v in Adj(u) do
         bfsq.push_back(v)
```

```
DFS-Visit (G, start_node)
  for each vertex u
    u.color = WHITE
    u.pred = nil
  st = new Stack
  st.push_back(start_node)
  while st not empty
     u = st.top(); st.pop()
     if u.color == WHITE
       u.color = GRAY
       foreach vertex v in Adj(u) do
          st.push_back(v)
```