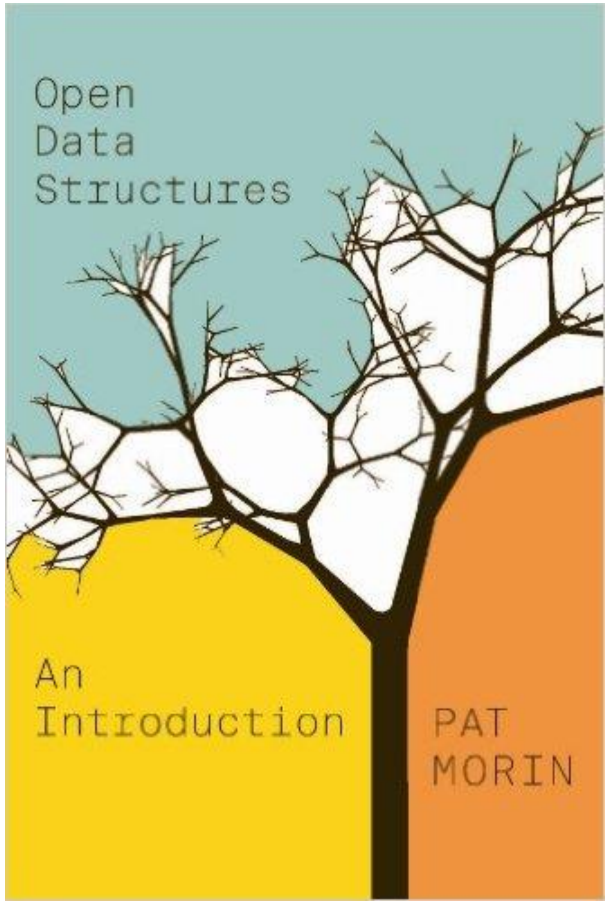
Graphs

COMP 2402
Abstract Data Types & Algorithms

Readings

Today's class

- Graphs
  - Chapter 12



Graphs

A **directed graph** G is a pair of sets

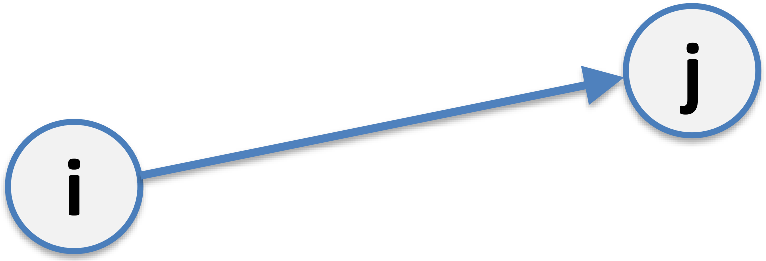
$$G = (V, E)$$

V is a non-empty set of **vertices** (or **nodes**)

E is a set **edges** which are pairs of vertices.

Graphs

The edge (i, j) is directed from vertex i to vertex j.



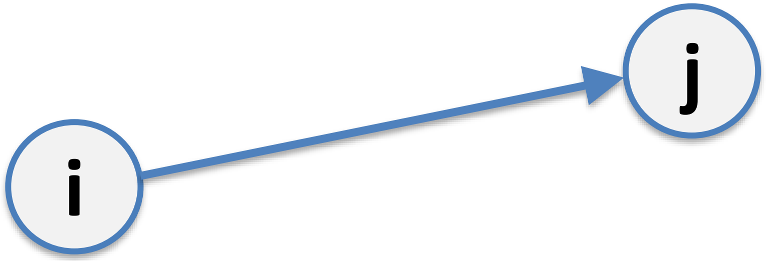
i is the **source** of the edge

j is the **target** of the edge

Adjacent vertices are connected by an edge.

Graphs

The edge (i, j) is directed from vertex i to vertex j.



Adjacent vertices are connected by an edge.

We say that

i is adjacent **to** j and j is adjacent **from** i

Graphs

A **path** in G is a sequence of vertices $v_0, v_1, \dots v_k$ such that for every $i \in \{1, \dots k\}$ the edge (v_{i-1}, v_i) is in E

The path **length** is the number of edges in the parth.

A path is a **cycle** if the edge (v_k, v_0) is also in E

Graphs

A path (or cycle) is **simple** if all of its vertices are unique.

If there is a path from v_i to v_j then we say that v_j is **reachable** from v_i

Graphs

The **degree** of a vertex v is the number of edges that v as one of its endpoints.

We can divide the degree into two components:

- **outDegree** (the number of edges leaving v)
- **inDegree** (the number of edges coming in to v).

Graphs

A **Graph** is also an ADT/interface.

- `addEdge(i,j)`–adds the edge (i,j) to E
- `removeEdge(i,j)`–removes the edge (i,j) from E
- `hasEdge(i,j)`-returns true if (i,j) is in E, false otherwise
- `outEdges(i)`-returns a list of all vertices j where (i,j) is in E
- `inEdges(i)`-returns a list of all vertices j where (j,i) is in E

Graphs

How do we implement the **Graph** interface?

- Consider the special case of undirected graphs in which there exists a unique path between every pair of vertices and there are no cycles

  This is a **tree**. We have already seen how to implement a special case of trees (binary trees).

Graphs

How do we implement the **Graph** interface?

We’ll consider two data structures, to store a directed graph, that are built on the notion of adjacency.

1. Adjacency Matrix

2. Adjacency Lists

Graphs

Consider a graph G with $|V| = n$ nodes and $|E| = m$ edges.

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$M[i][j] = \begin{cases} true & (i,j) \in E \\ false & (i,j) \notin E \end{cases}$$

Graphs

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$M[i][j] = \begin{cases} true/1 & (i,j) \in E \\ false/0 & (i,j) \notin E \end{cases}$$

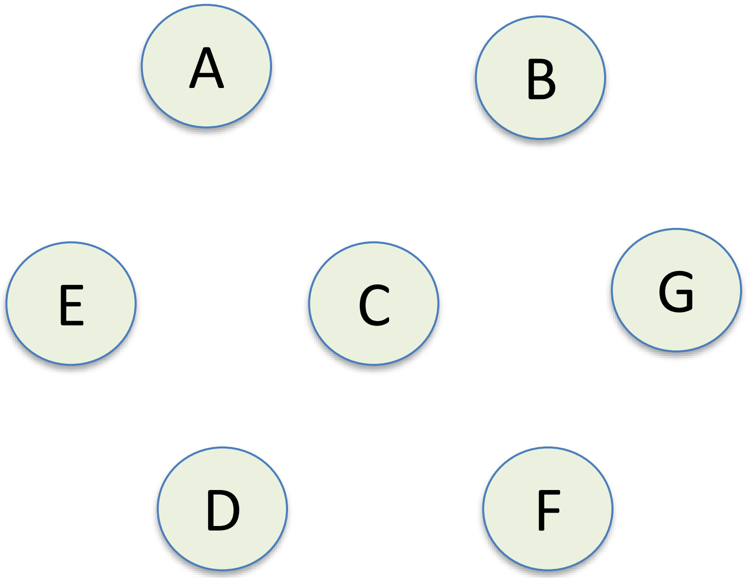
0	1	0	0	0	0	0
0	0	0	0	0	0	1
1	1	0	0	0	0	0
0	0	0	0	0	1	0
1	0	0	0	0	0	0
0	0	0	1	0	0	1
0	0	0	0	0	0	1

Graphs

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$M[i][j] = \begin{cases} true/1 & (i,j) \in E \\ false/0 & (i,j) \notin E \end{cases}$$

	A	B	C	D	E	F	G
A	0	1	0	0	0	0	0
B	0	0	0	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	0	0	0	0	0	1

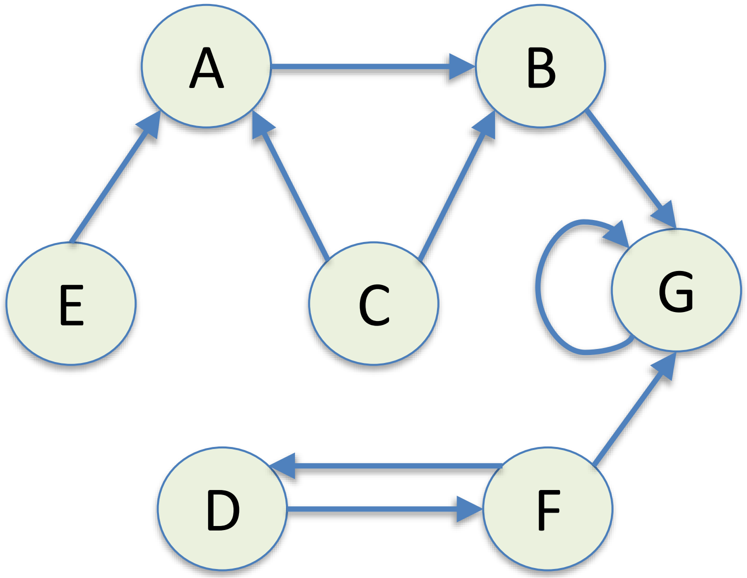


Graphs

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$M[i][j] = \begin{cases} true/1 & (i,j) \in E \\ false/0 & (i,j) \notin E \end{cases}$$

	A	B	C	D	E	F	G
A	0	1	0	0	0	0	0
B	0	0	0	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	0	0	0	0	0	1



Graphs

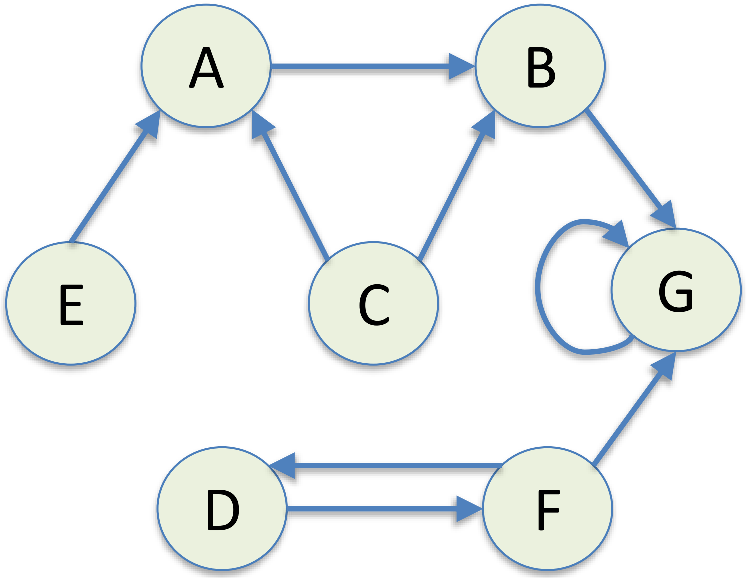
The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$M[i][j] = \begin{cases} true/1 & (i,j) \in E \\ false/0 & (i,j) \notin E \end{cases}$$

	A	B	C	D	E	F	G
A							
B							
C							
D							
E							
F							
G	0	0	0	0	0	0	1

G is called a **sink**
(outDegree is 0)

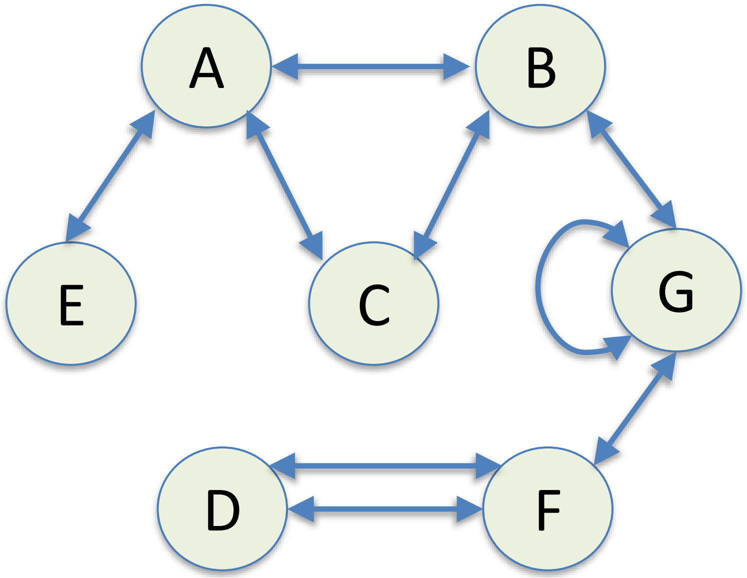
E and C are called **sources**
(indegree is 0)



Graphs

In an **undirected** graph, the matrix is symmetric.

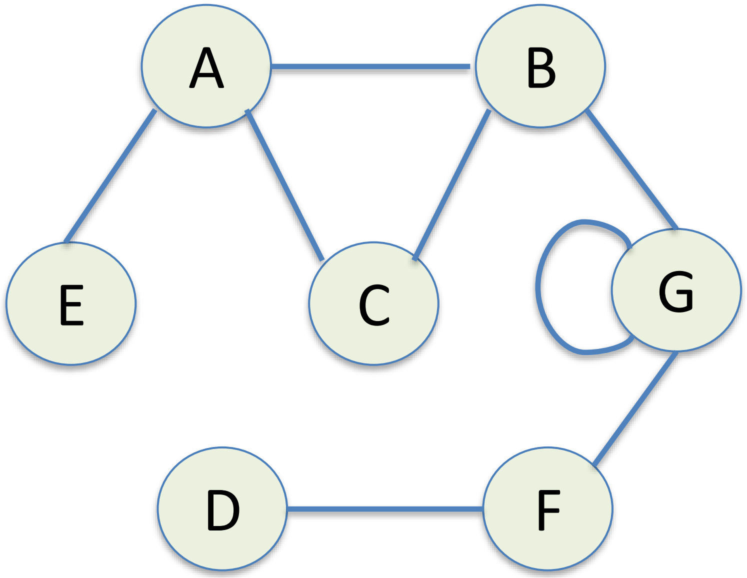
	A	B	C	D	E	F	G
A	0	1	1	0	1	0	0
B	1	0	1	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	1	0	0	0	1	1



Graphs

In an **undirected** graph, the matrix is symmetric.

	A	B	C	D	E	F	G
A	0	1	1	0	1	0	0
B	1	0	1	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	1	0	0	0	1	1



Graphs

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$a[i][j] = \begin{cases} true & (i,j) \in E \\ false & (i,j) \notin E \end{cases}$$

```
void addEdge(int i, int j) {  a[i][j] = true; }
void removeEdge(int i, int j) { a[i][j] = false; }
boolean hasEdge(int i, int j) { return a[i][j]; }
```

Graphs

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$a[i][j] = \begin{cases} true & (i,j) \in E \\ false & (i,j) \notin E \end{cases}$$

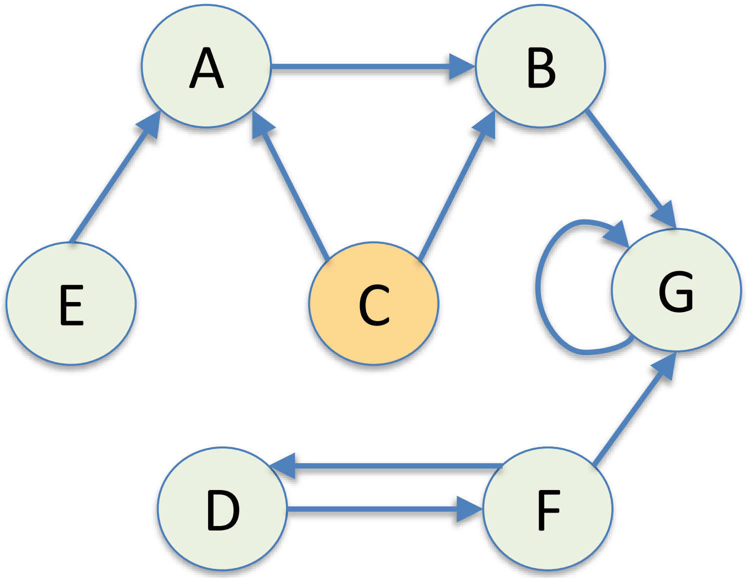
```
void addEdge(int i, int j) {  a[i][j] = true; }
void removeEdge(int i, int j) { a[i][j] = false; }
boolean hasEdge(int i, int j) { return a[i][j]; }
```

addEdge(i,j), removeEdge(i,j) and hasEdge(i,j) are all $O(1)$ time operations

Graphs

Notice that each row in the matrix corresponds to the outEdges of a node.

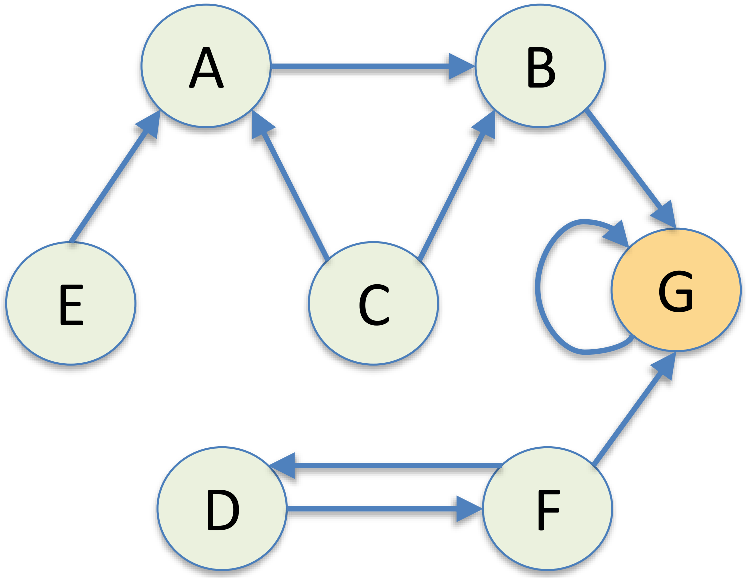
	A	B	C	D	E	F	G
A	0	1	0	0	0	0	0
B	0	0	0	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	0	0	0	0	0	1



Graphs

Notice that each column in the matrix corresponds to the inEdges of a node.

	A	B	C	D	E	F	G
A	0	1	0	0	0	0	0
B	0	0	0	0	0	0	1
C	1	1	0	0	0	0	0
D	0	0	0	0	0	1	0
E	1	0	0	0	0	0	0
F	0	0	0	1	0	0	1
G	0	0	0	0	0	0	1



Graphs

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$a[i][j] = \begin{cases} true & (i,j) \in E \\ false & (i,j) \notin E \end{cases}$$

```
List outEdges(int i) {
   List edges = new ArrayList();
   for (int j = 0; j < n; j++) {
      if (a[i][j]) {
         edges.add(j);
      }
   }
   return edges;
}
```

Graphs

The Adjacency Matrix of G is the $n \times n$ boolean matrix

$$a[i][j] = \begin{cases} true & (i,j) \in E \\ false & (i,j) \notin E \end{cases}$$

```
List outEdges(int i) {
   List edges = new ArrayList();
   for (int j = 0; j < n; j++) {
      if (a[i][j]) {
         edges.add(j);
      }
   }
   return edges;
}
```

outEdges(i) and inEdges(i) are both $O(n)$ operations

Graphs

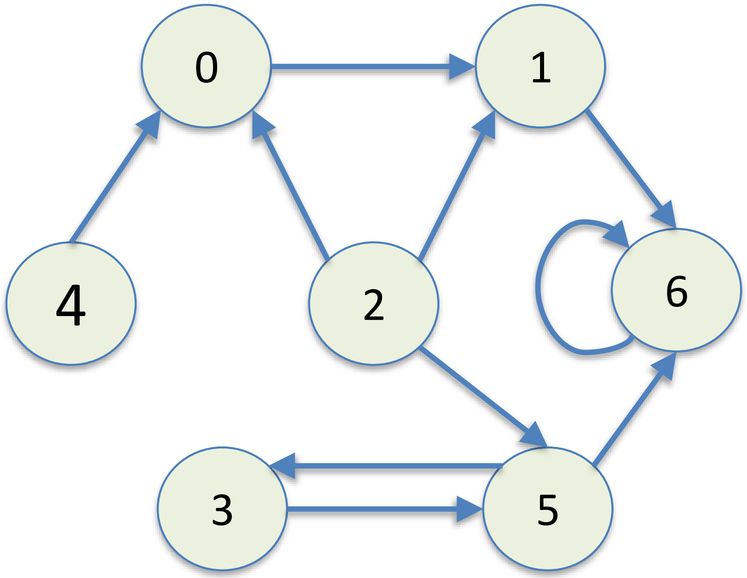
Consider a graph G with $|V| = n$ nodes and $|E| = m$ edges.

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

Graphs

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

adj	0	1	2	3	4	5	6
	↓	↓	↓	↓	↓	↓	↓
	1	6	0	5	0	6	6
			5			3	
			1				



Graphs

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

```
void addEdge(int i, int j) {
   adj[i].add(j);
}

List outEdges(int i) {
   return adj[i];
}
```

adj	0	1	2	3	4	5	6
	↓	↓	↓	↓	↓	↓	↓
	1	6	0	5	0	6	6
			5			3	
			1				

Graphs

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

```
void addEdge(int i, int j) {
   adj[i].add(j);
}

List outEdges(int i) {
   return adj[i];
}
```

addEdge(i,j) and outEdges(i) are both $O(1)$ operations

Graphs

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

What about

`removeEdge(i,j)`

`hasEdge(i,j)`

adj

0	1	2	3	4	5	6
↓	↓	↓	↓	↓	↓	↓
1	6	0	5	0	6	6
		5			3	
		1				

Graphs

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

What about

`removeEdge(i,j)`

`hasEdge(i,j)`

removeEdge(i,j) and hasEdge(i,j) are both $O(\text{outDegree}(i))$ time operations

Graphs

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

What about inEdges(i)?

adj	0	1	2	3	4	5	6
	↓	↓	↓	↓	↓	↓	↓
	1	6	0	5	0	6	6
			5			3	
			1				

Graphs

The **Adjacency List** representation of G is a length n array, adj, of lists. The list at index j contains all vertices that are incident from vertex j.

What about inEdges(i)?

inEdges(i) is a $O(n + m)$ time operation

Graphs

	Adjacency Matrix	Adjacency List
addEdge	$O(1)$	$O(1)$
removeEdge	$O(1)$	$O(\deg(i))$
hasEdge	$O(1)$	$O(\deg(i))$
outEdges	$O(n)$	$O(1)$
inEdges	$O(n)$	$O(n+m)$
Space used	$O(n^2)$	$O(n+m)$

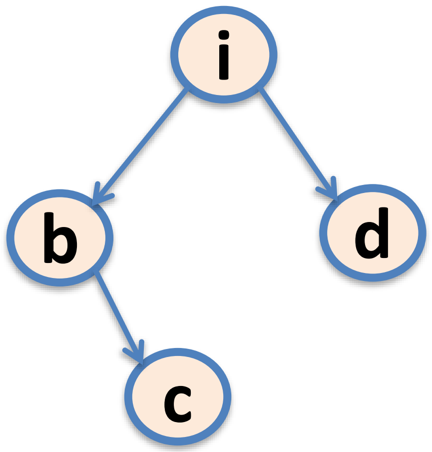
Graph Explorations

We have already seen **breadth-first** and **depth-first search** algorithms for binary trees.

We can use these (slightly modified) to explore graphs: starting with some vertex we find all vertices that are reachable from it.

Breadth-First Search

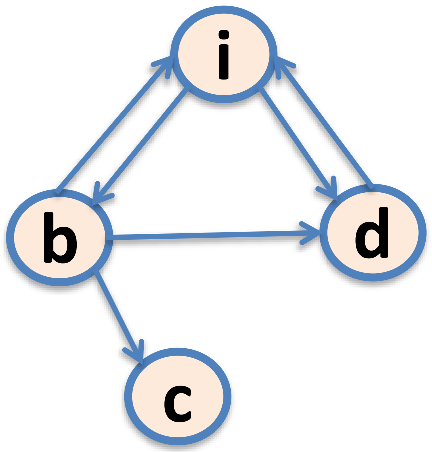
In a **breadth-first search** of a graph we start with a node **i**, and visit the neighbours of **i**, then visit all the neighbours of the neighbours of **i**, etc.



Use a **Queue**

Breadth-First Search

In a **breadth-first search** of a graph we start with a node **i**, and visit the neighbours of **i**, then visit all the neighbours of the neighbours of **i**, etc.

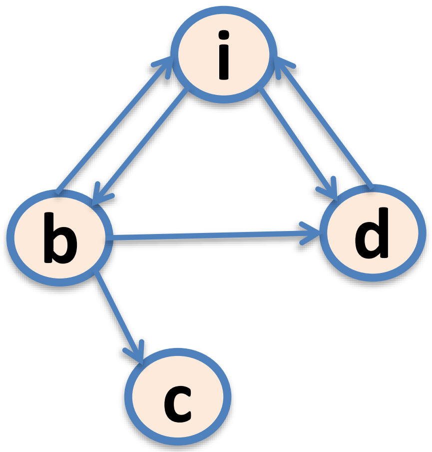


It's a bit more complicated with graphs.

We need to keep track of the nodes we have already discovered

Breadth-First Search

In a **breadth-first search** of a graph we start with a node **i**, and visit the neighbours of **i**, then visit all the neighbours of the neighbours of **i**, etc.



The execution of the BFS constructs a BFS **search tree**.

You will be asked about this

Breadth-First Search

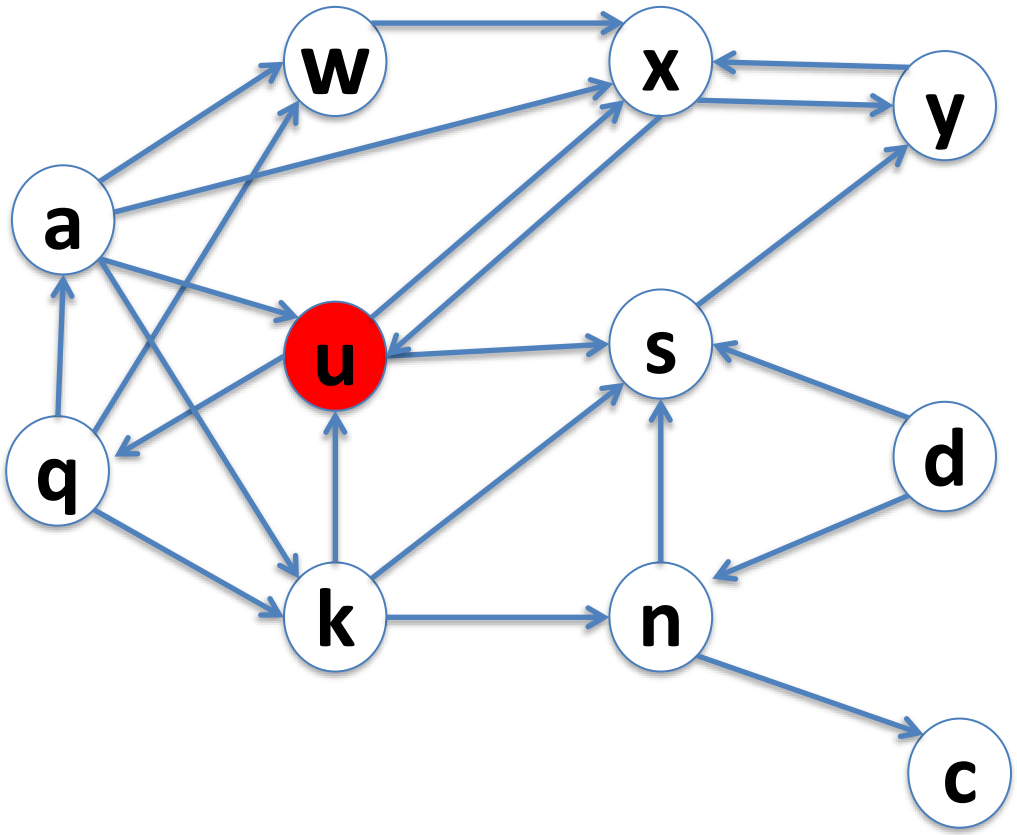
```
bfs(G, root)
   seen = boolean array of size n
   q = empty queue
   q.add(root)
   seen[root] = true
   while q is not empty do
      i = q.remove()
      for each vertex j in outEdges(i) do
         if seen[j] is false
            q.add(j)
            seen[j] = true
```

Breadth-First Search

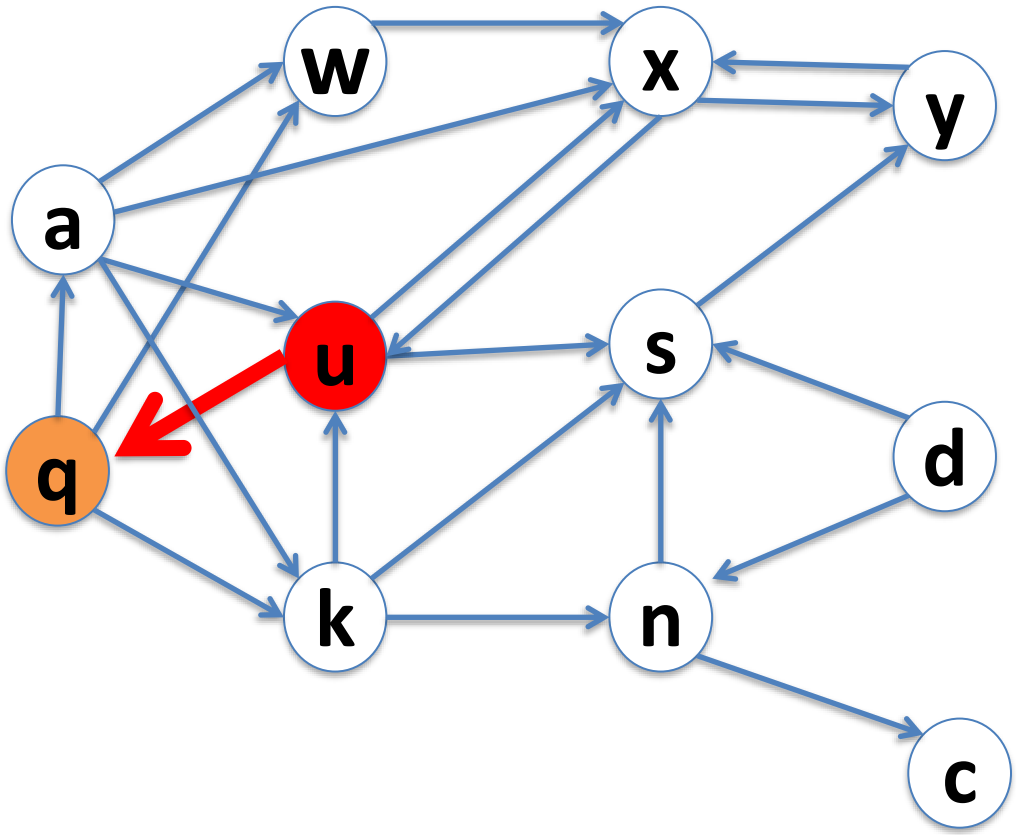
```
bfs(G, root)
    seen = boolean array of size n
    q = empty queue
    q.add(root)
    seen[root] = true
    while q is not empty do
        i = q.remove()
        for each vertex j in outEdges(i) do
            if seen[j] is false
                q.add(j)
                seen[j] = true
```

Order these in increasing order

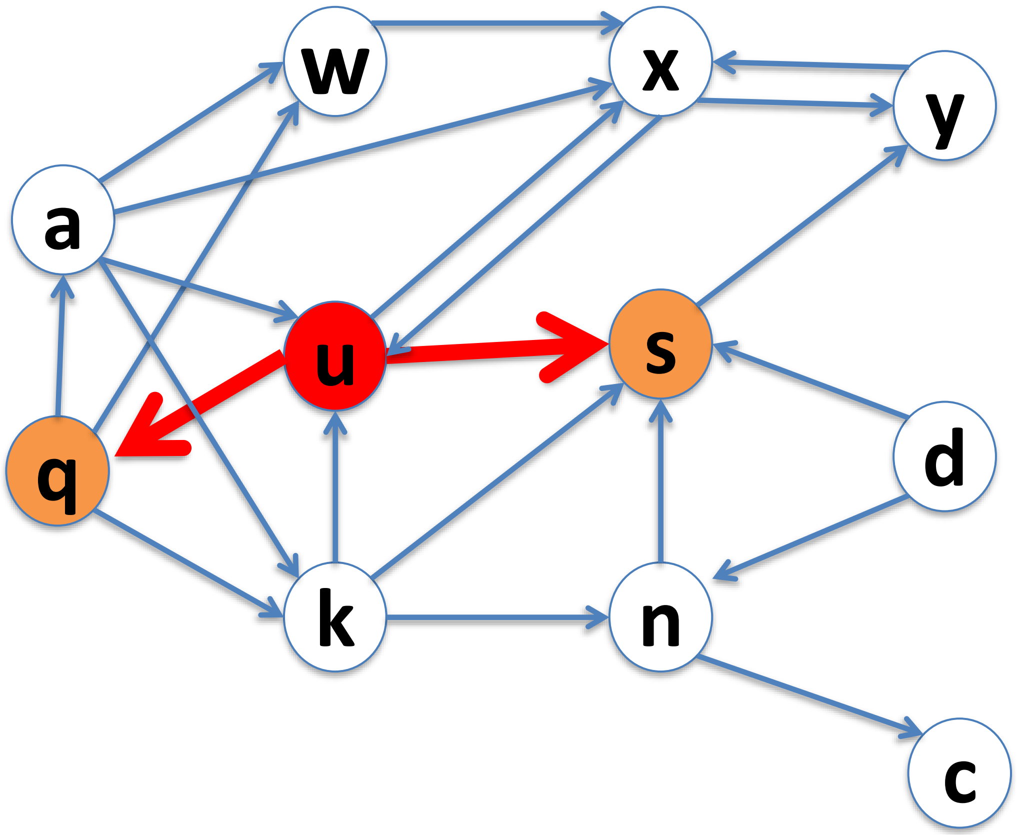
Breadth-First Search



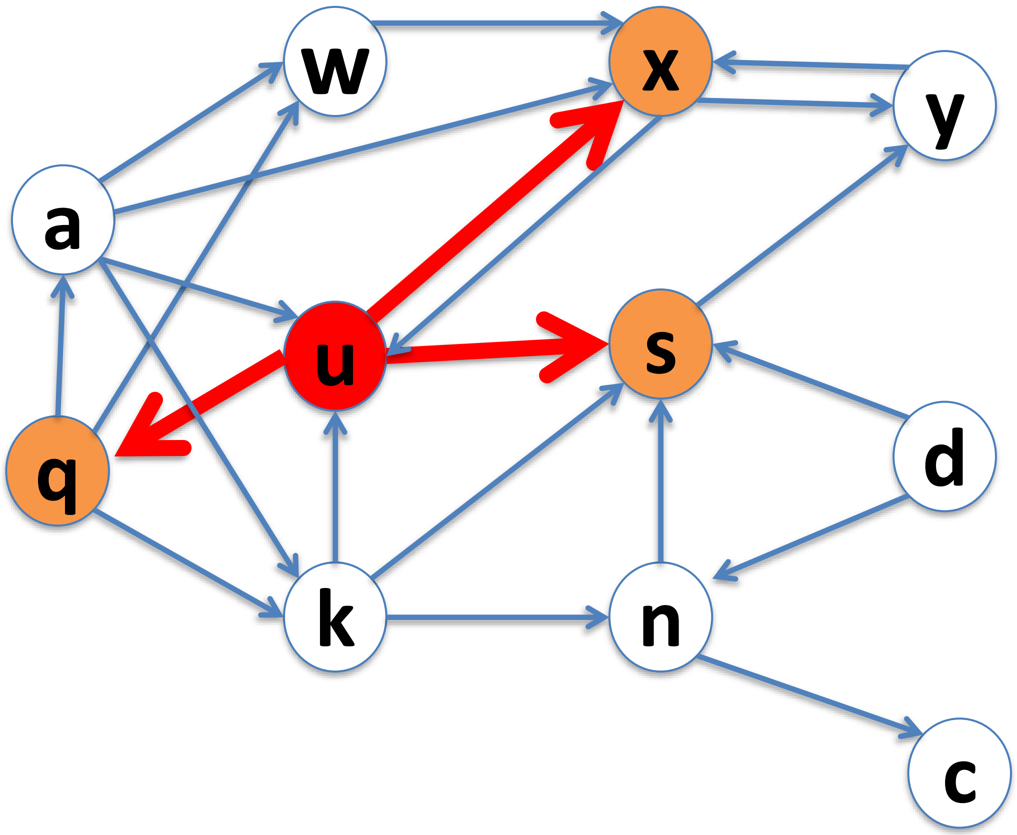
Breadth-First Search



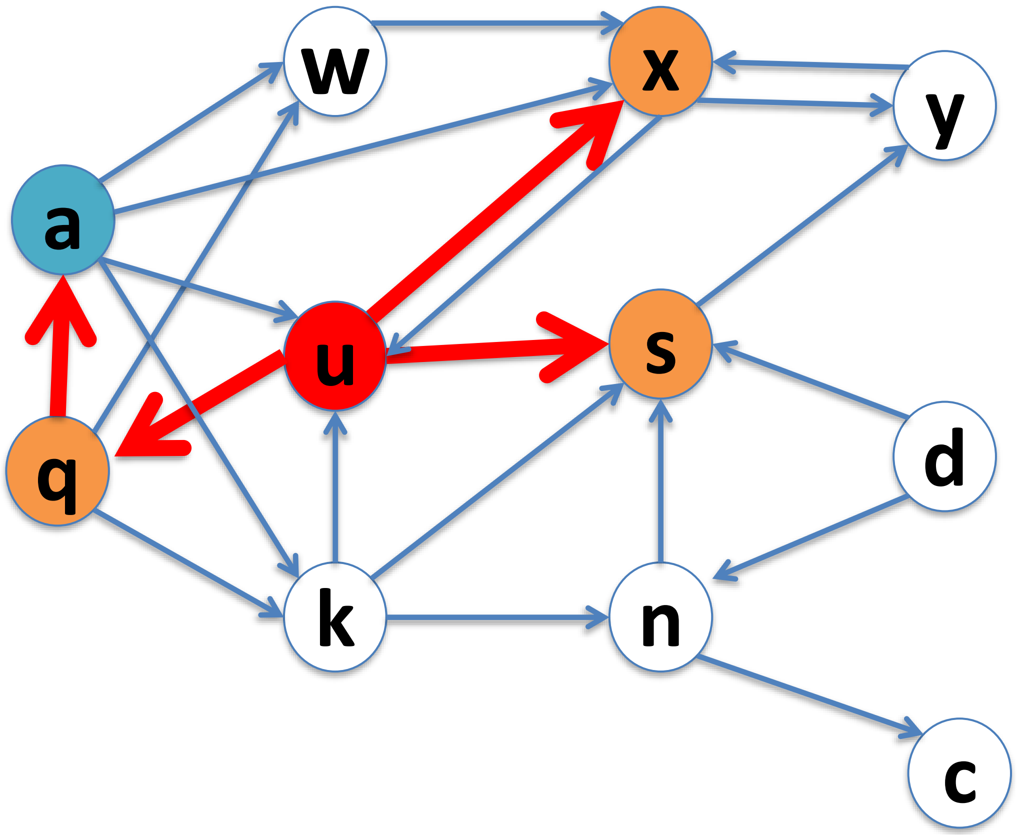
Breadth-First Search



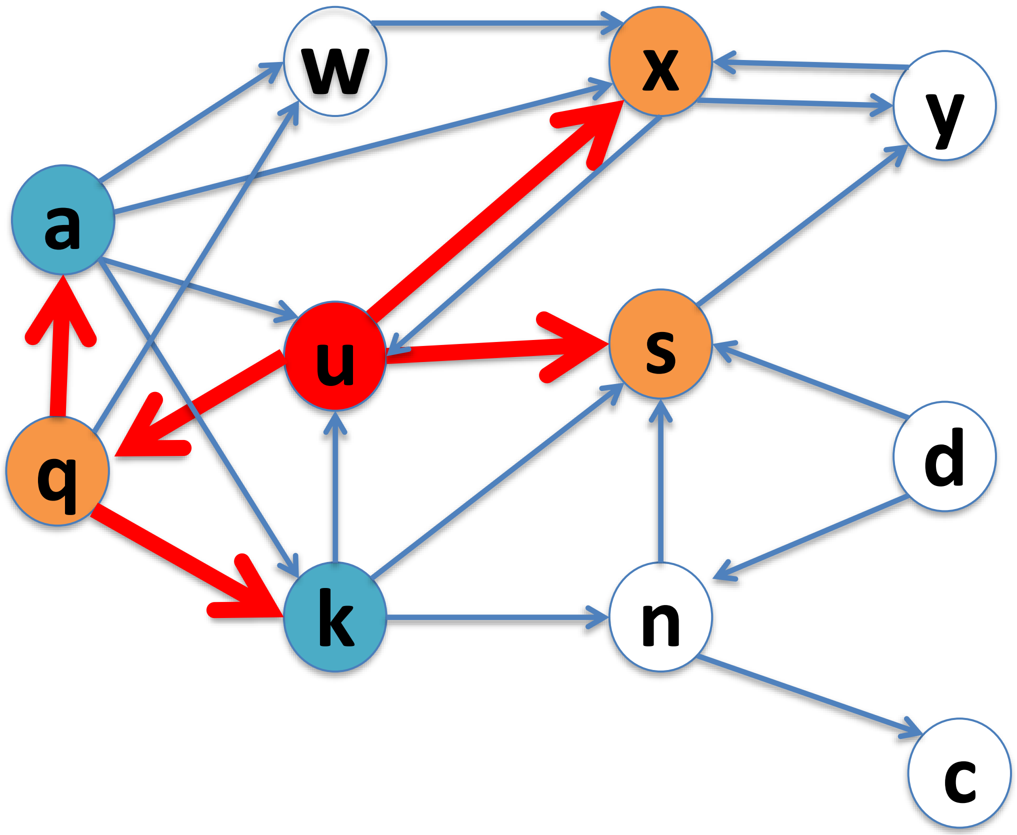
Breadth-First Search



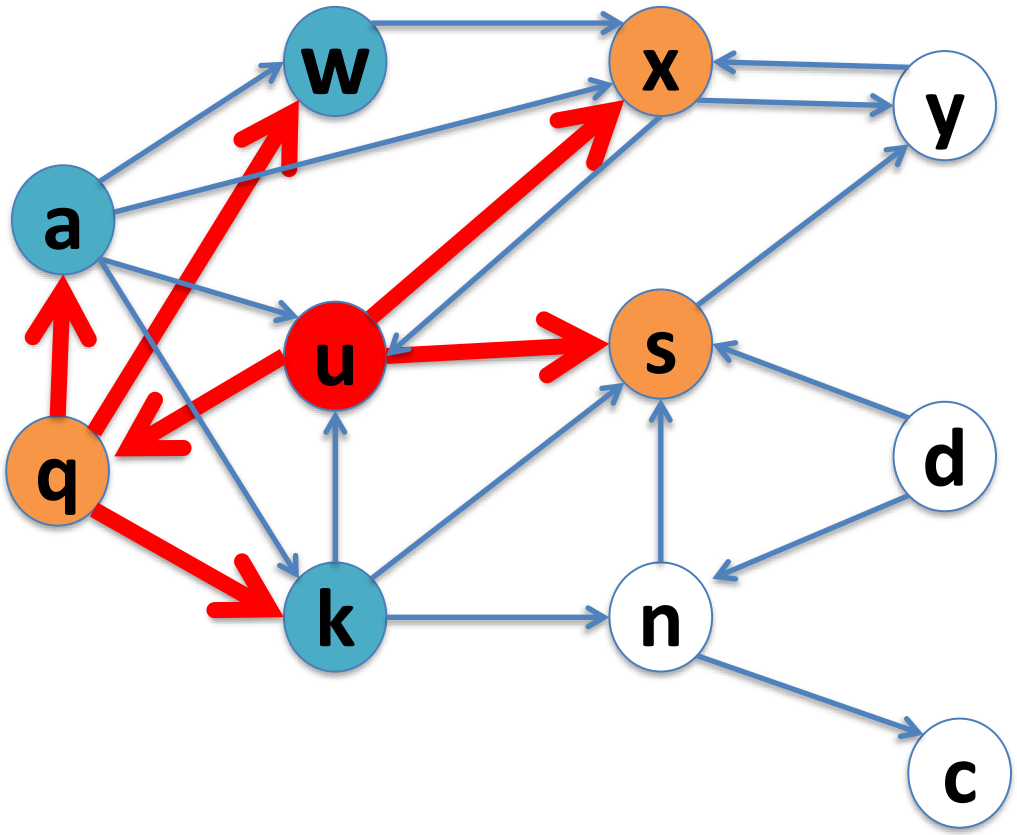
Breadth-First Search



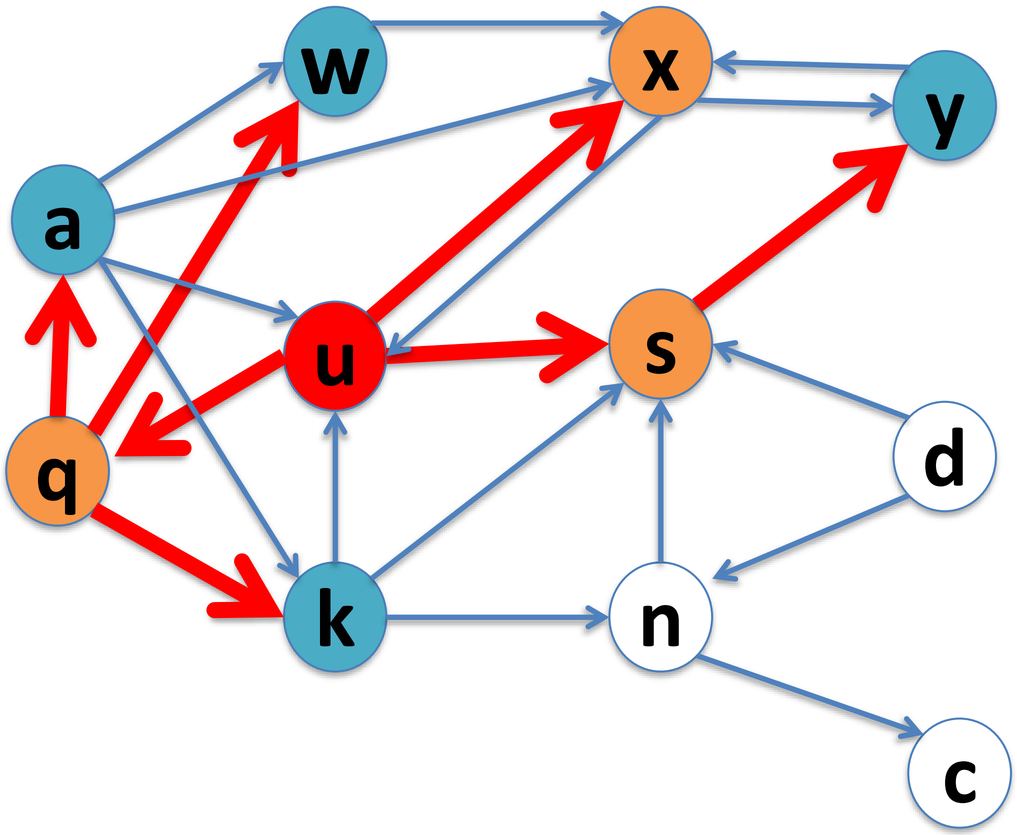
Breadth-First Search



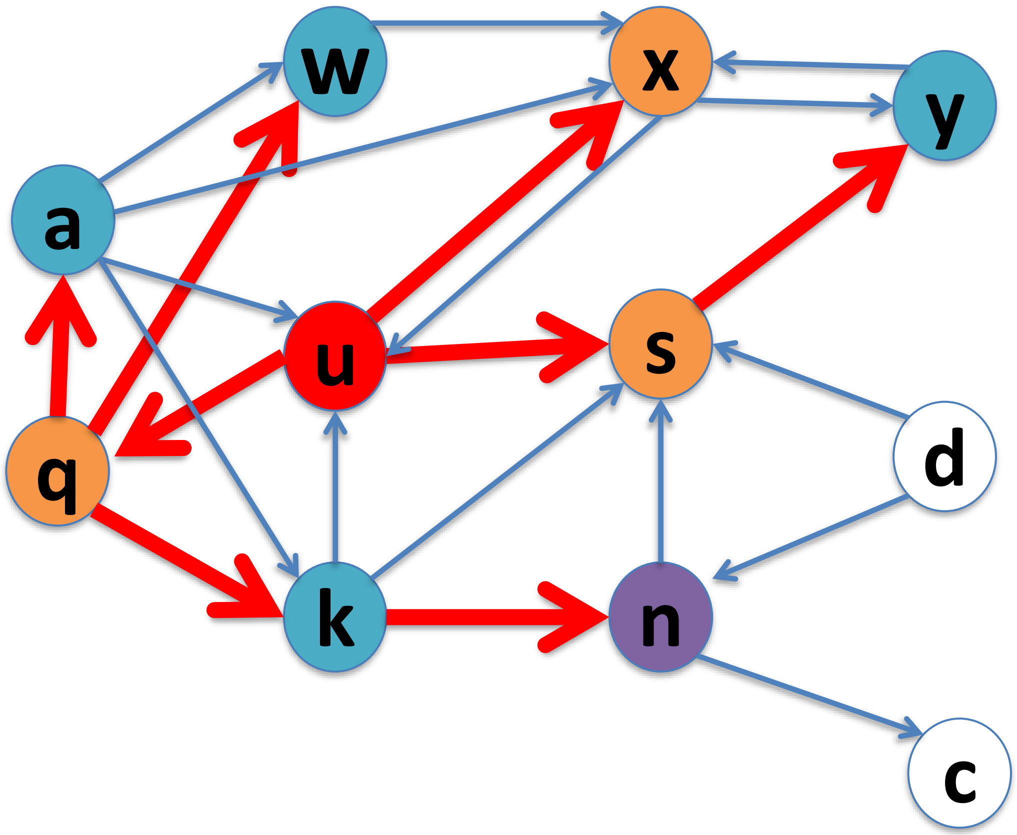
Breadth-First Search



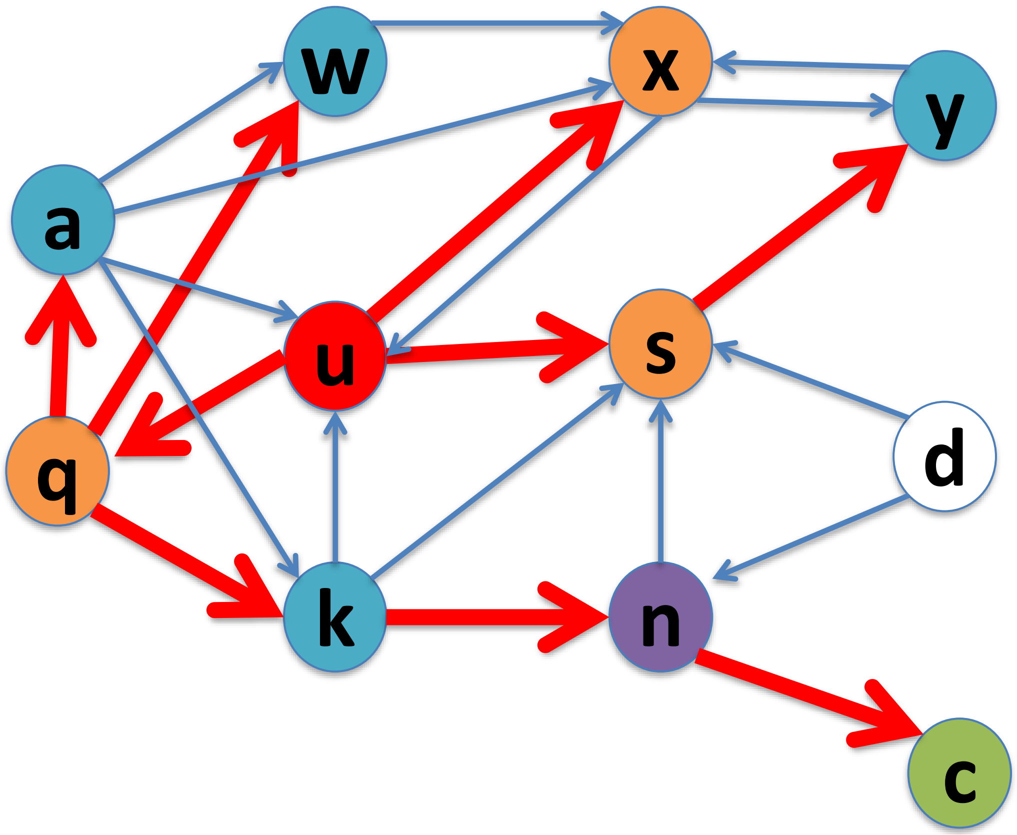
Breadth-First Search



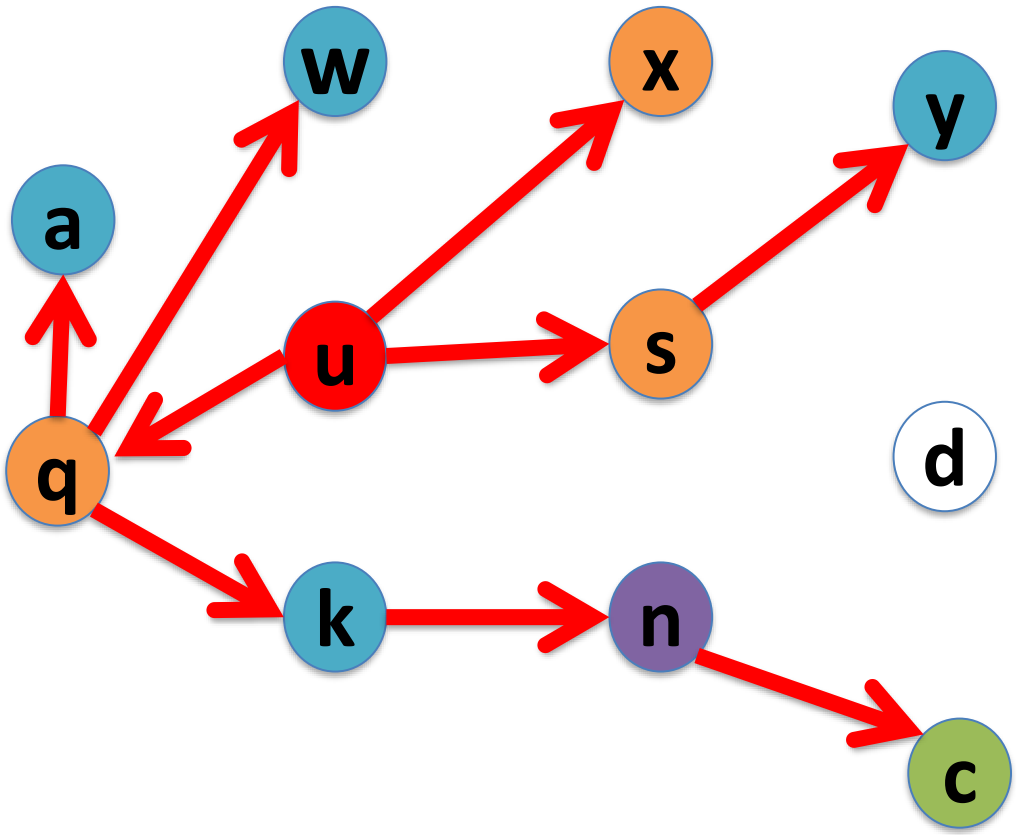
Breadth-First Search



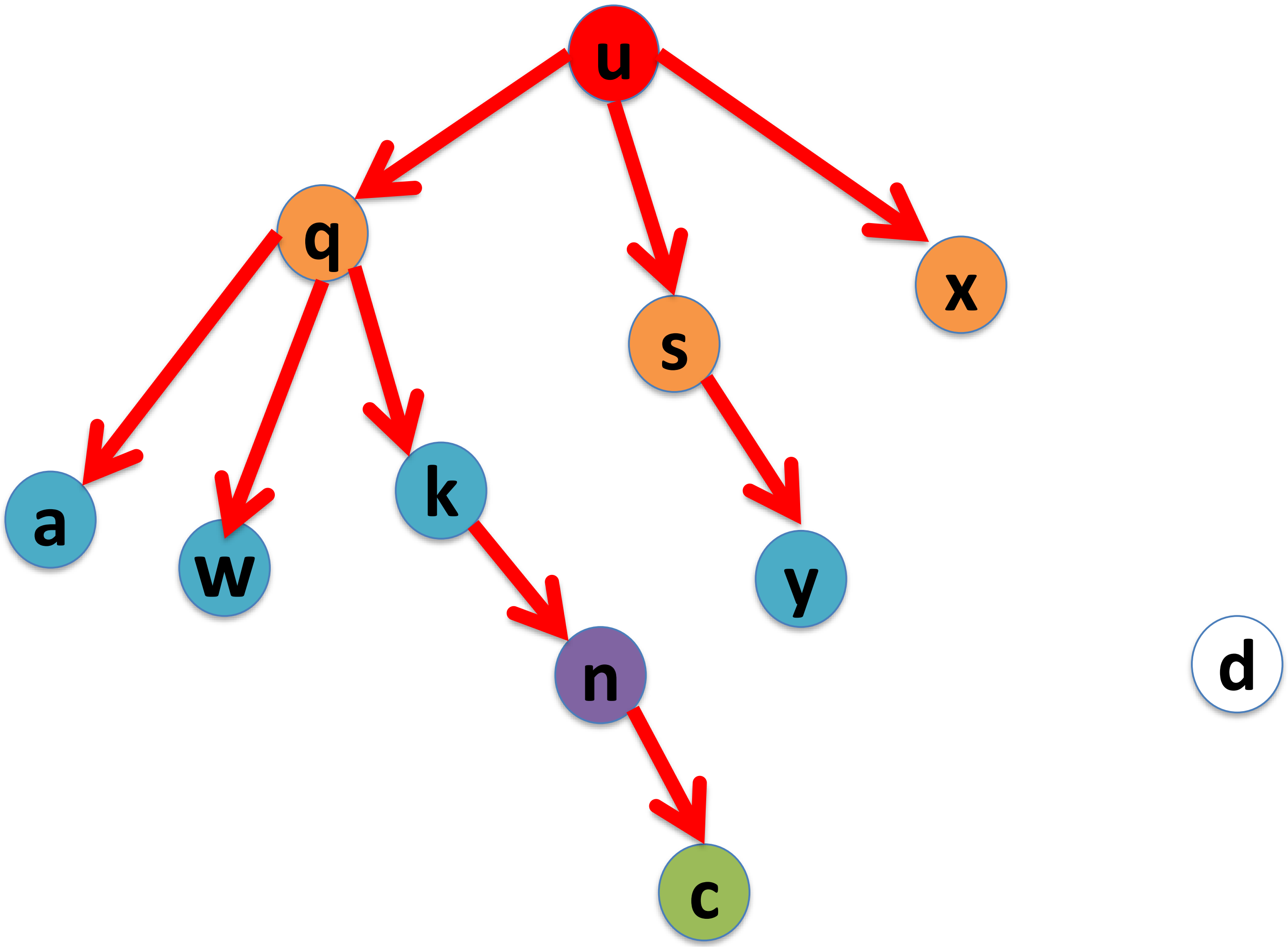
Breadth-First Search



Breadth-First Search



Breadth-First Search



Breadth-First Search

What is the runtime of **breadth-first search**?

- Adjacency Matrix

- Adjacency Lists

Breadth-First Search

```
bfs(G, root)
   seen = boolean array of size n
   q = empty queue
   q.add(root)
   seen[root] = true
   while q is not empty do
      i = q.remove()
      for each vertex j in outEdges(i) do
         if seen[j] is false
            q.add(j)
            seen[j] = true
```

Order these in increasing order

Breadth-First Search

What is the runtime of **breadth-first search**?

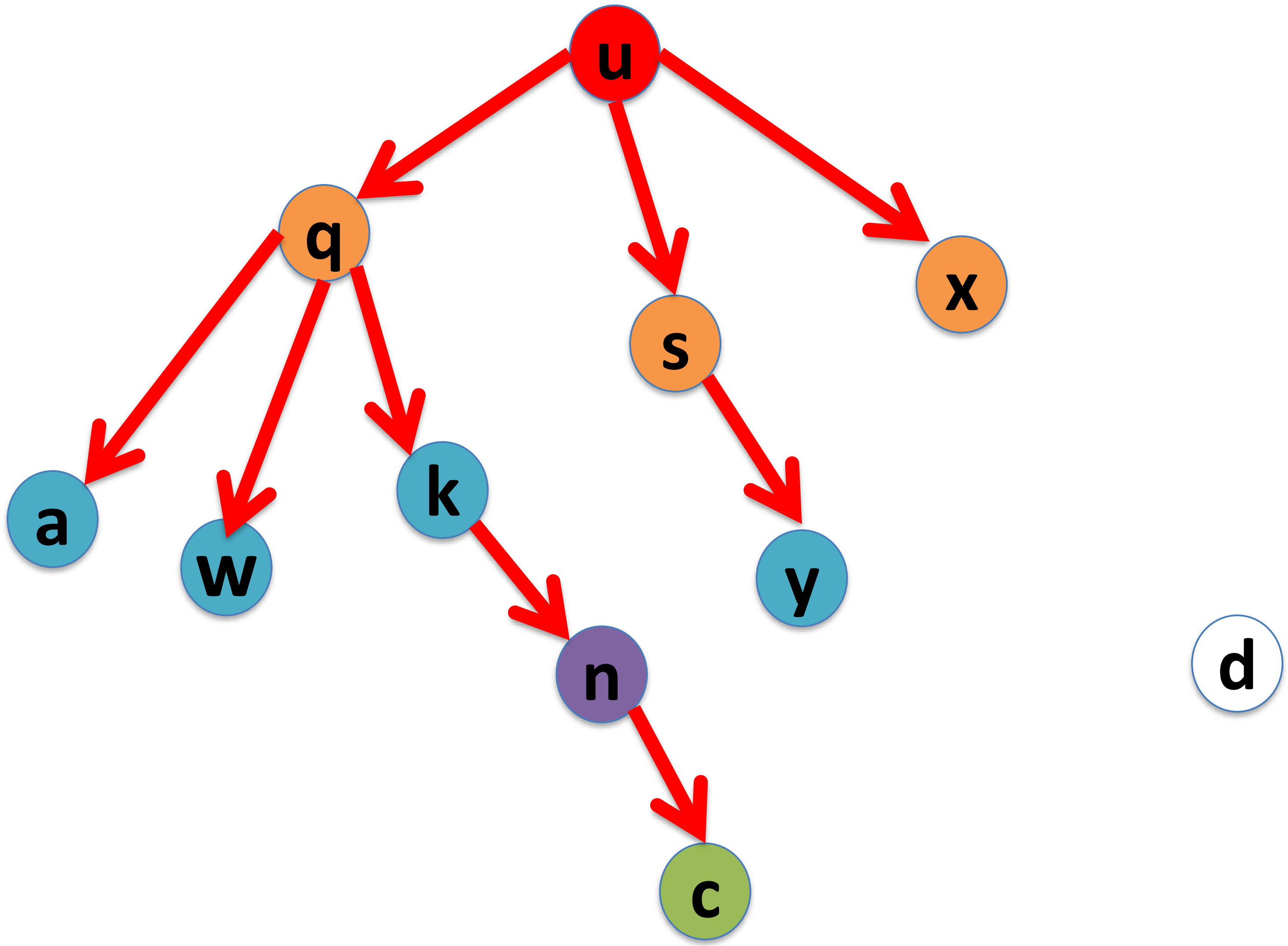
- Adjacency Matrix

$$O(n^2)$$

- Adjacency Lists

$$O(n + m)$$

Breadth-First Search



Breadth-First Search

In a **breadth-first search** of a graph starting with some vertex **u**, we find the **shortest path** from **u** to all other vertices reachable from **u**.

(More about this in COMP3804)

Depth-First Search

In a **depth-first search** we start with some node **r** and keep taking steps (following edges) while possible. Once we get stuck, we backtrack and try another path; repeating this until we have explored all the graph that is possible.

Like the BFS, we need to keep track of nodes: nodes we haven’t seen, nodes we have seen but are not done with and nodes we are done with.

Depth-First Search

In a **depth-first search** we start with some node **r** and keep taking steps (following edges) while possible. Once we get stuck, we backtrack and try another path; repeating this until we have explored all the graph that is possible.

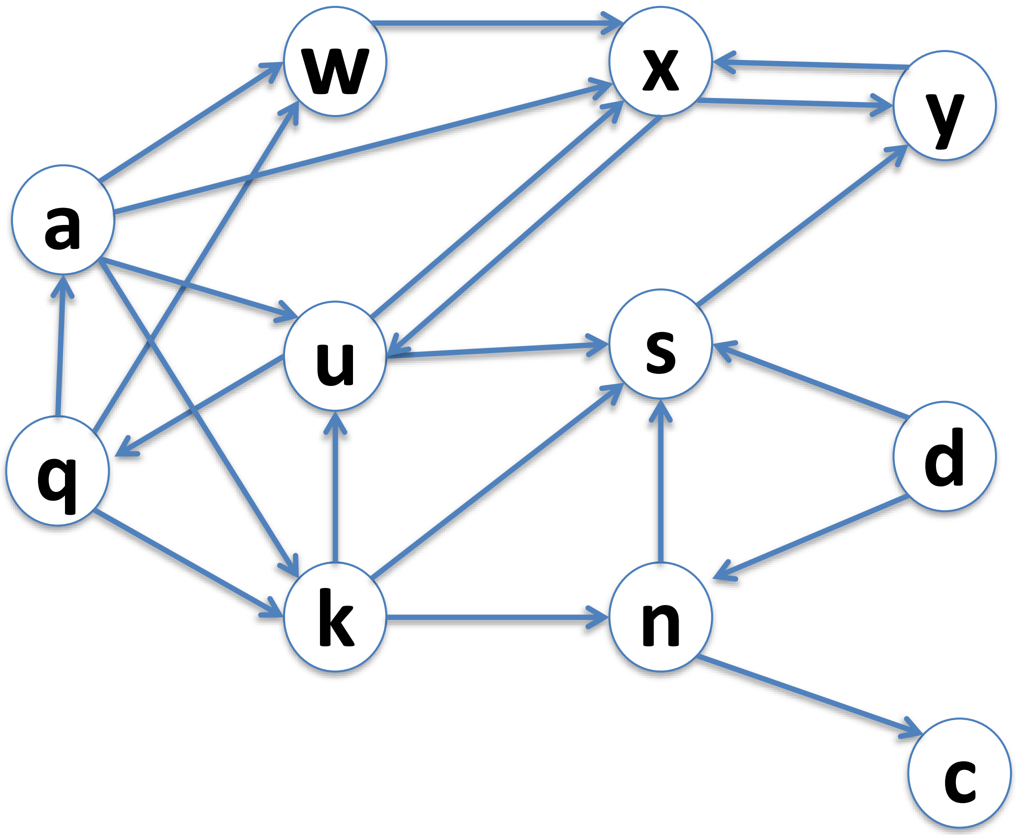
Like the BFS, the DFS algorithm constructs a DFS **search tree**. (When choosing which node to visit always choose the smallest first!)

Depth-First Search

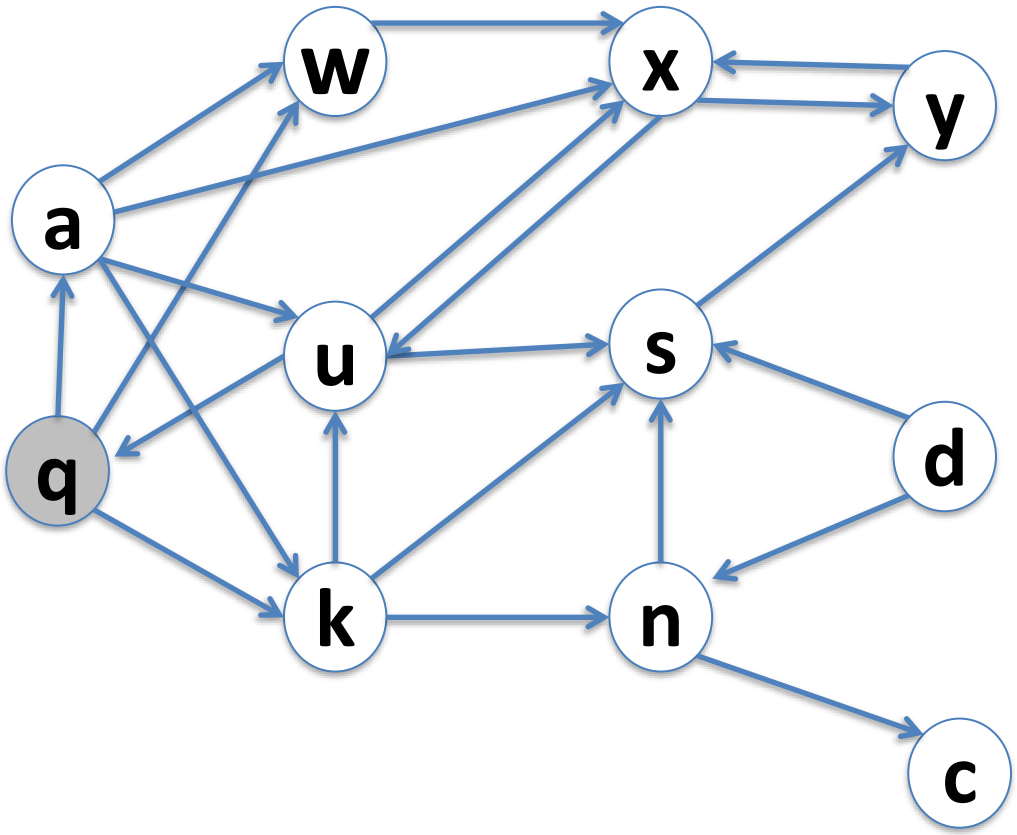
Just like we did for binary trees, we can do a **depth-first search** using a **Stack**.

We'll push elements onto the stack in **DECREASING** order, so that when they get popped of, they are in **INCREASING** order.

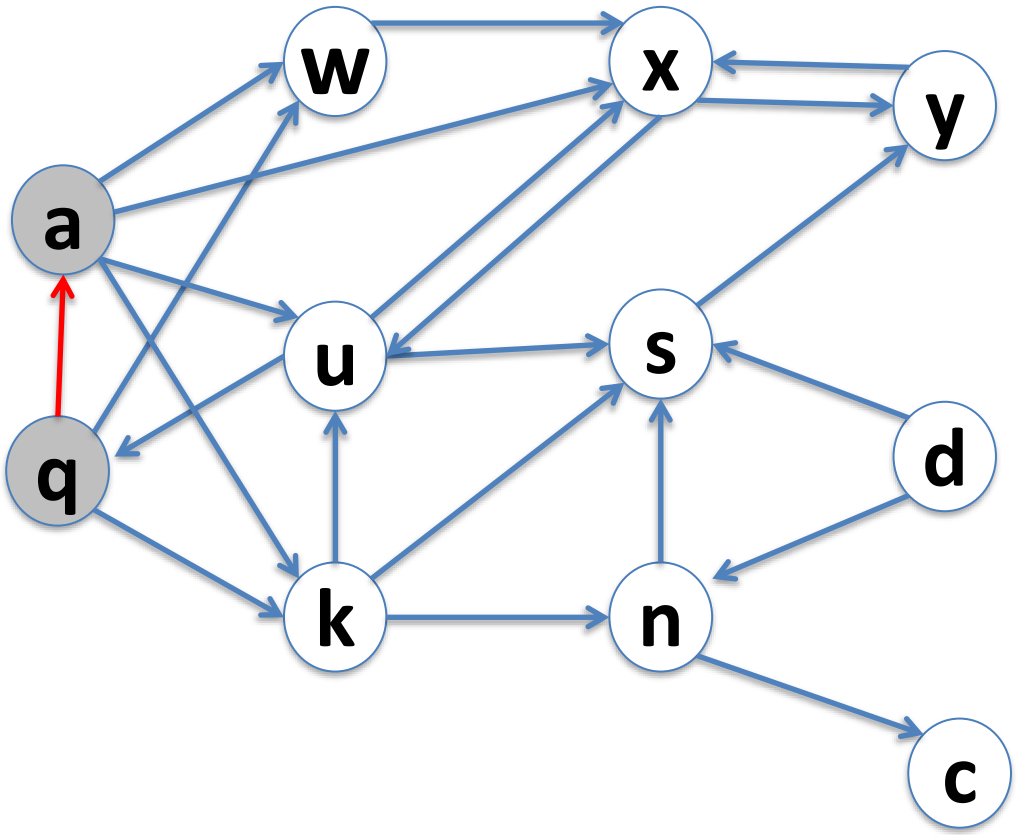
Depth-First Search



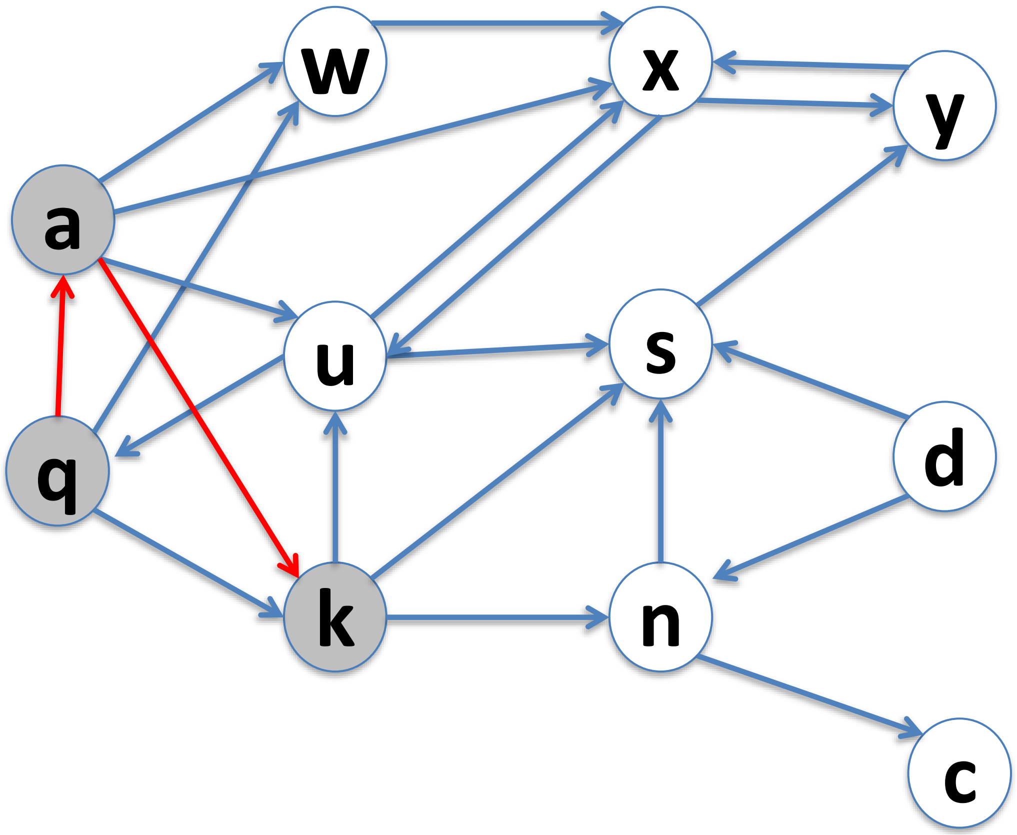
Depth-First Search



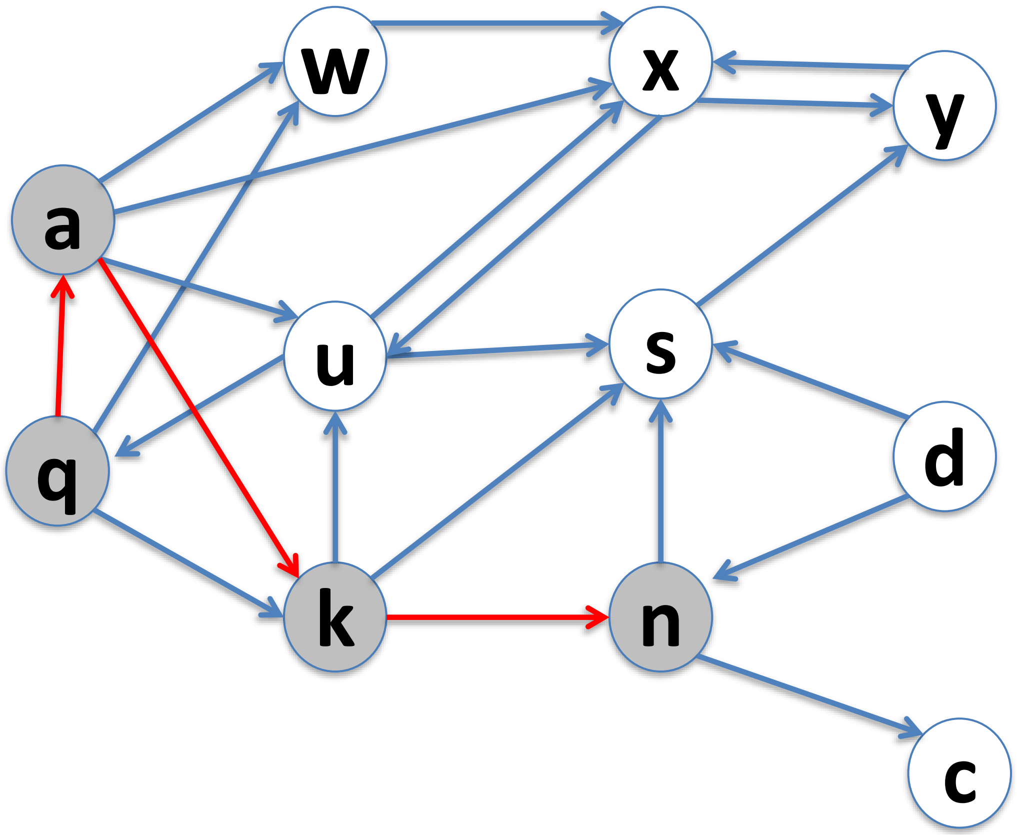
Depth-First Search



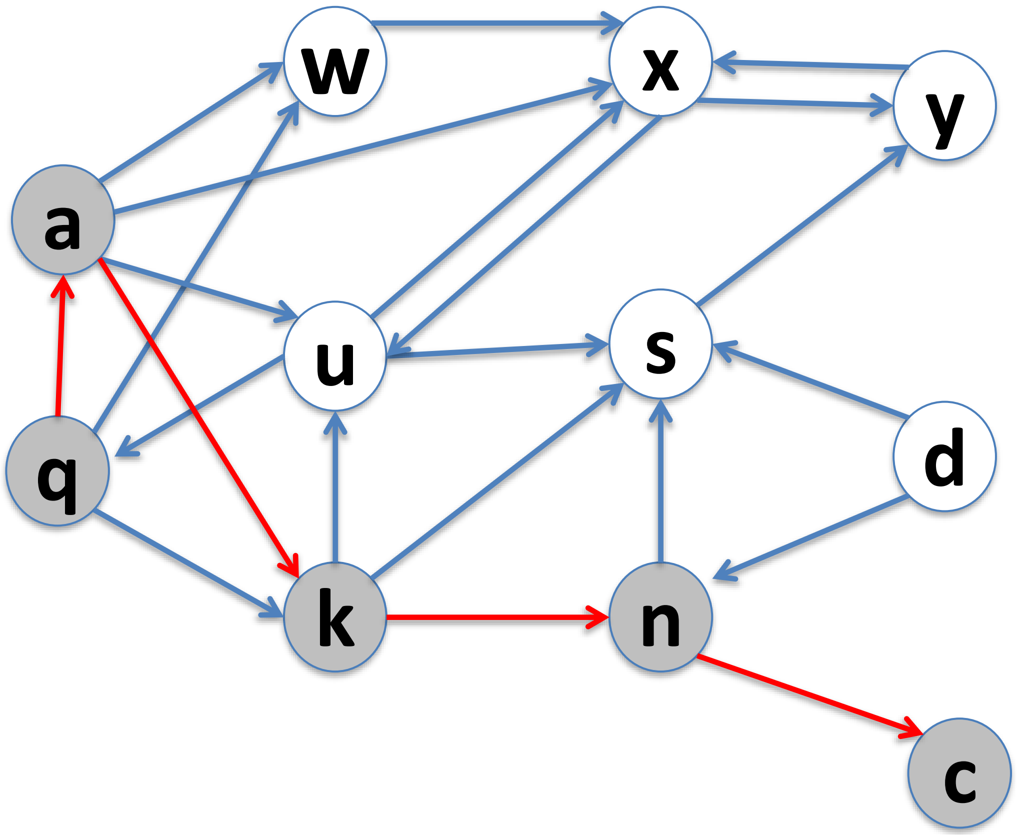
Depth-First Search



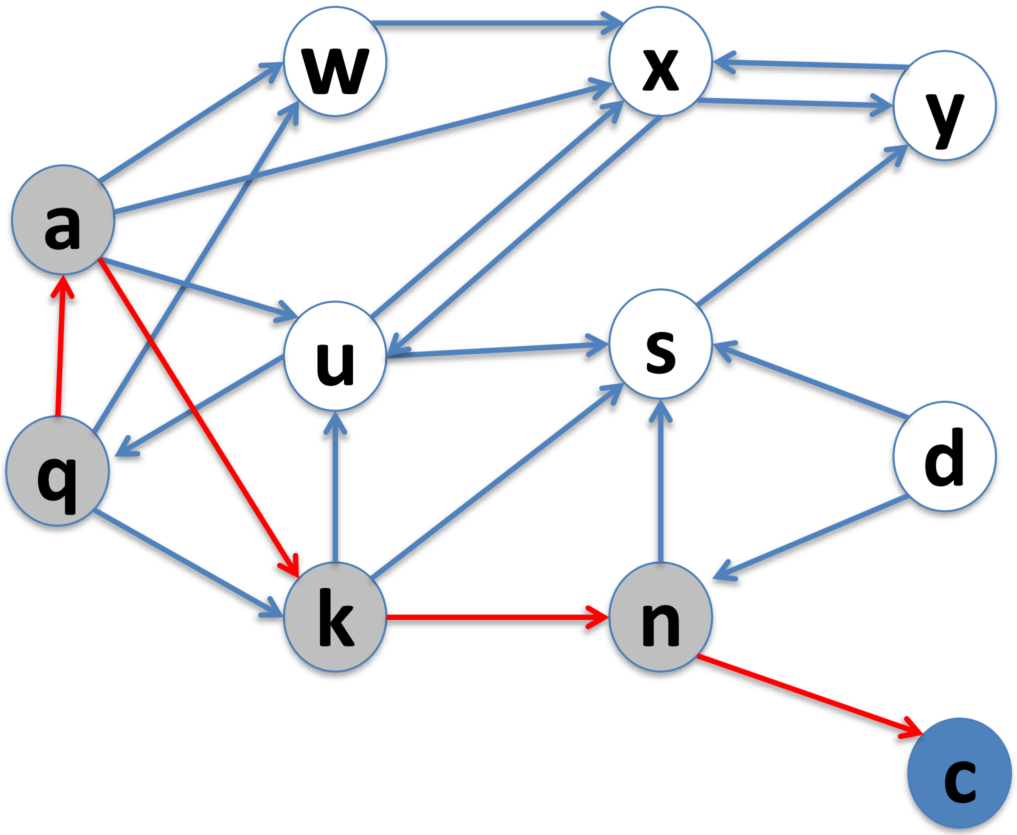
Depth-First Search



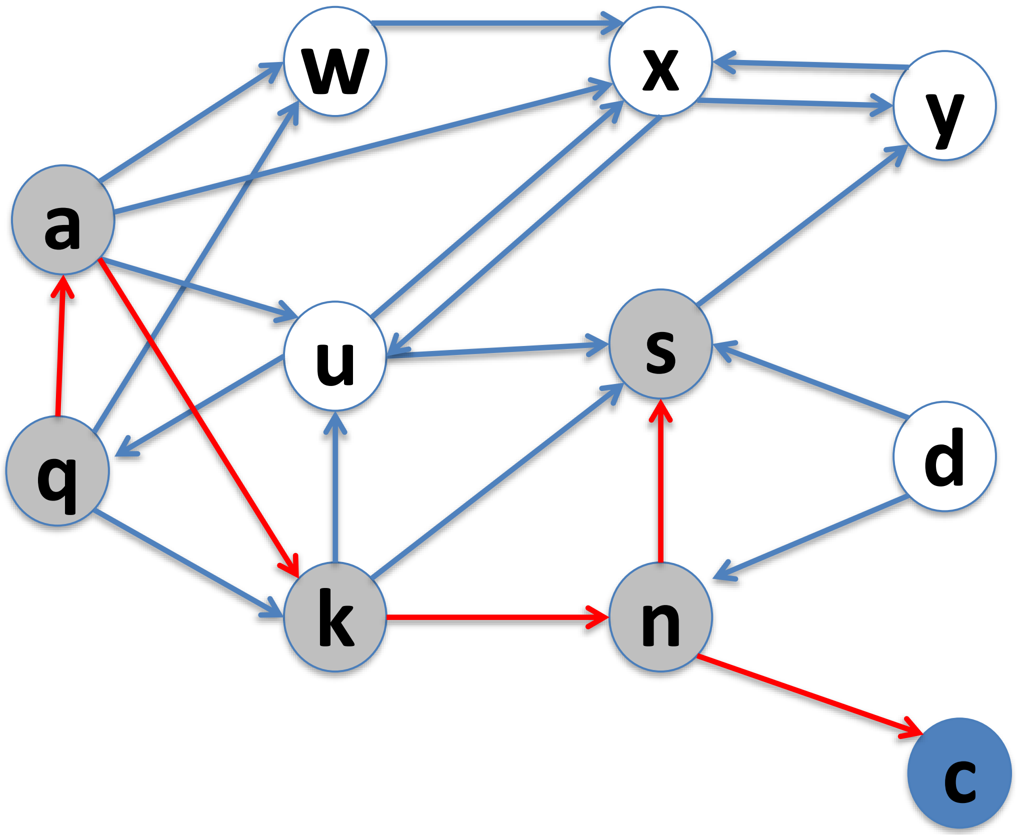
Depth-First Search



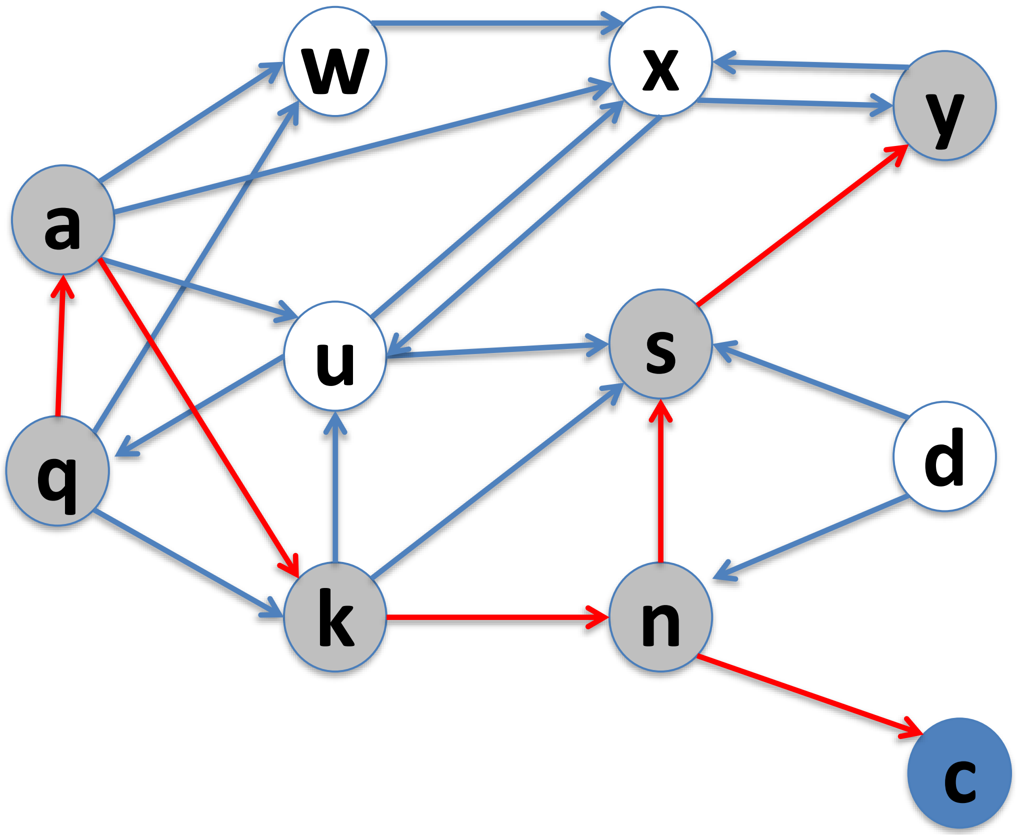
Depth-First Search



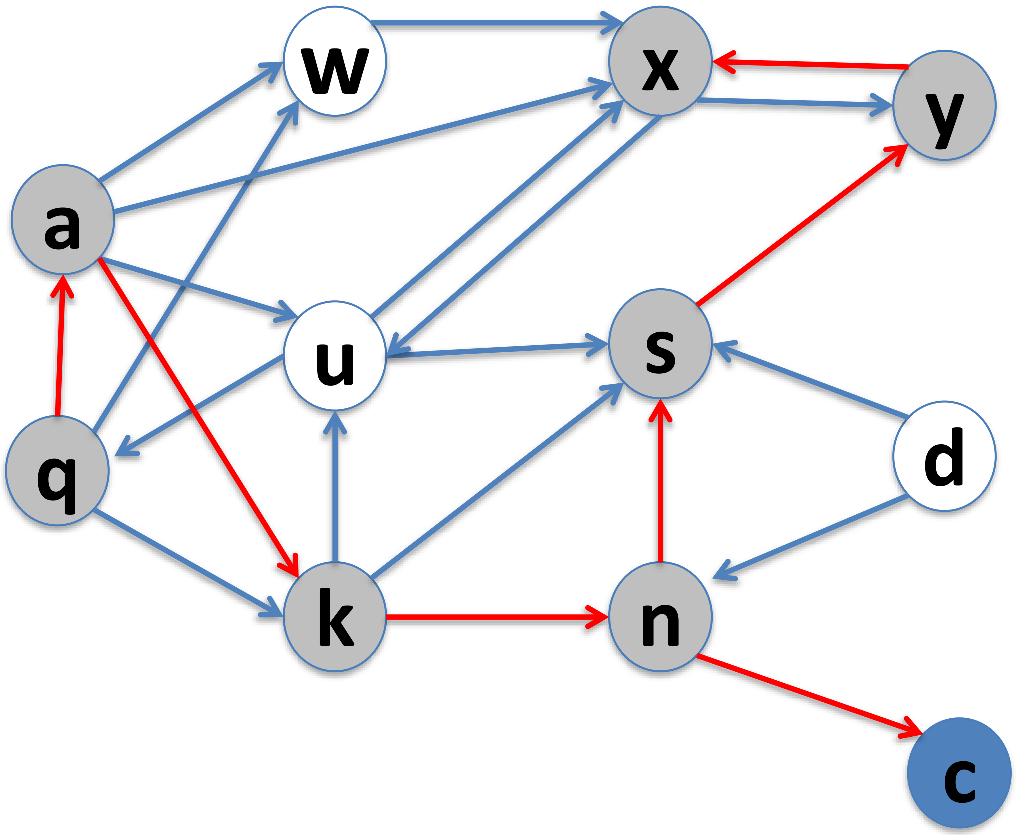
Depth-First Search



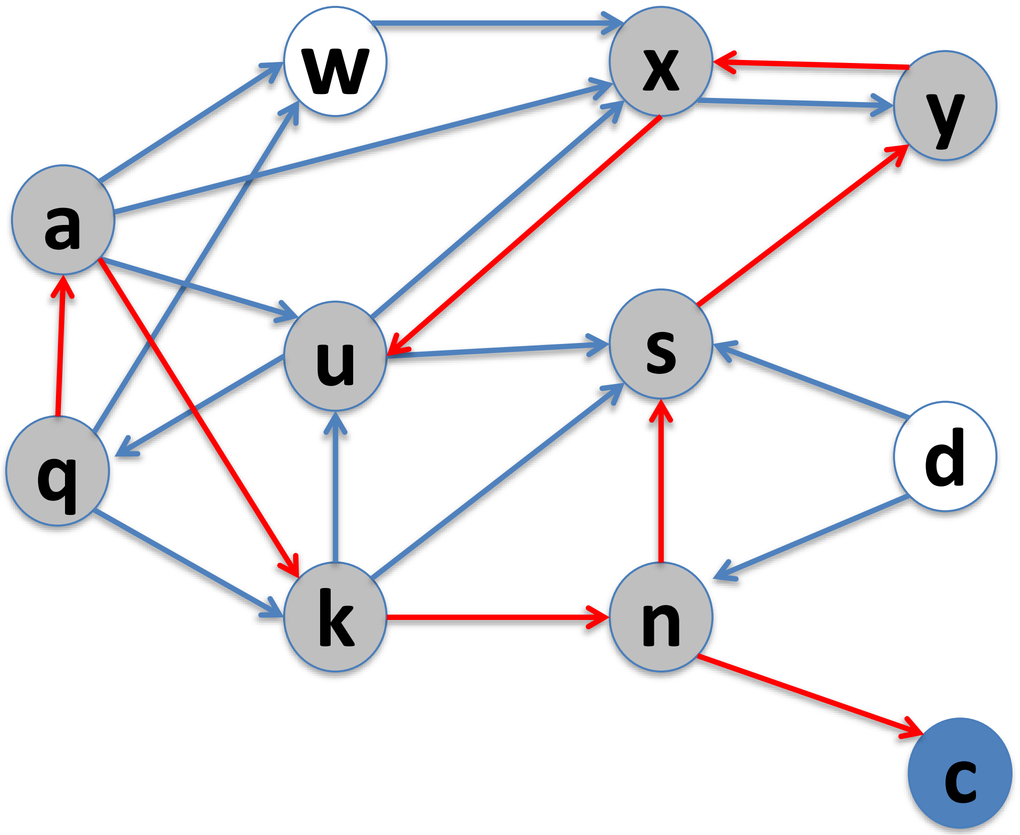
Depth-First Search



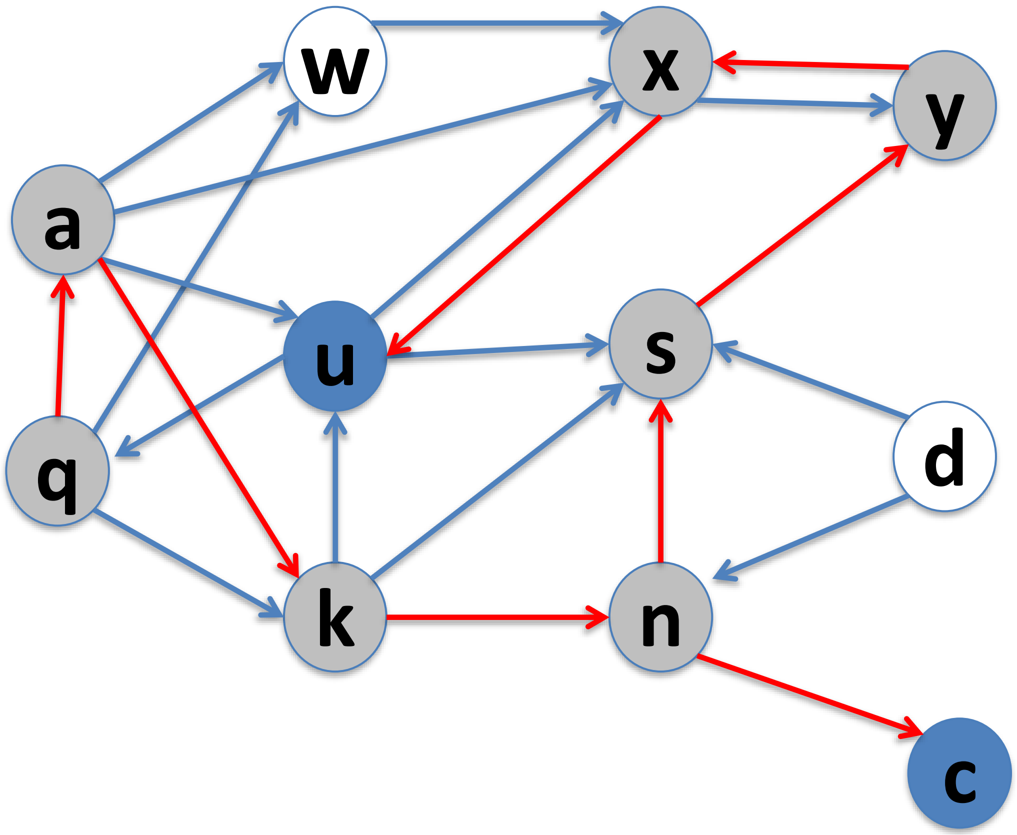
Depth-First Search



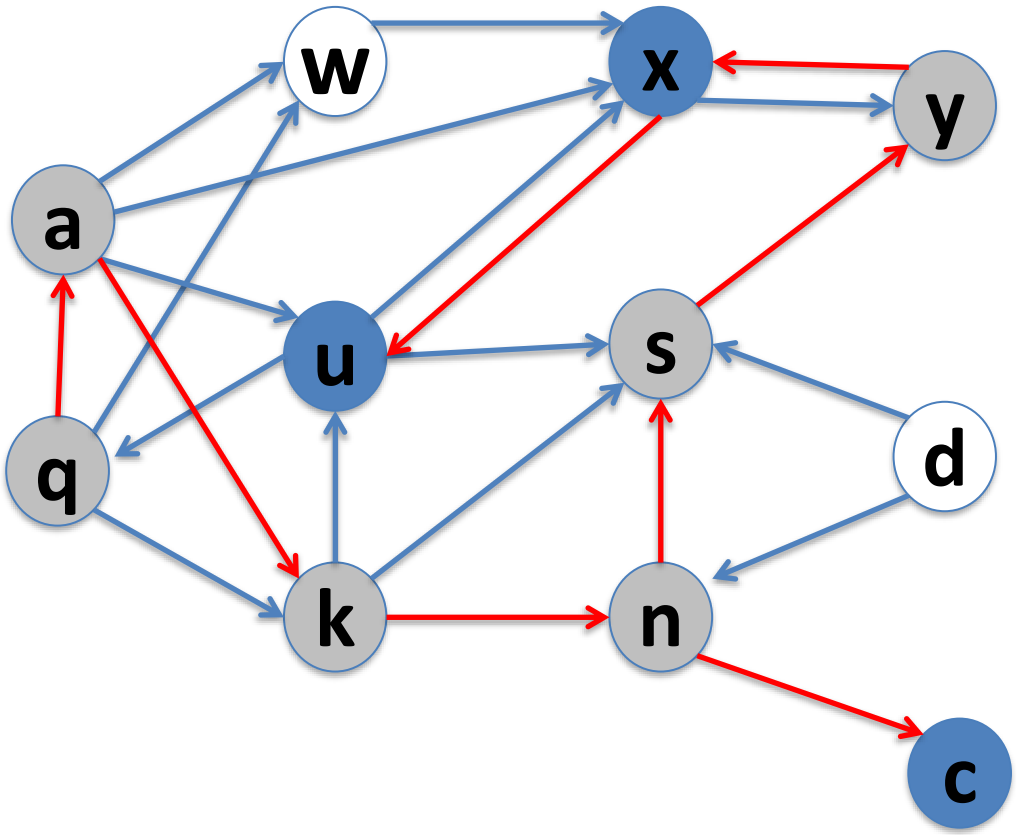
Depth-First Search



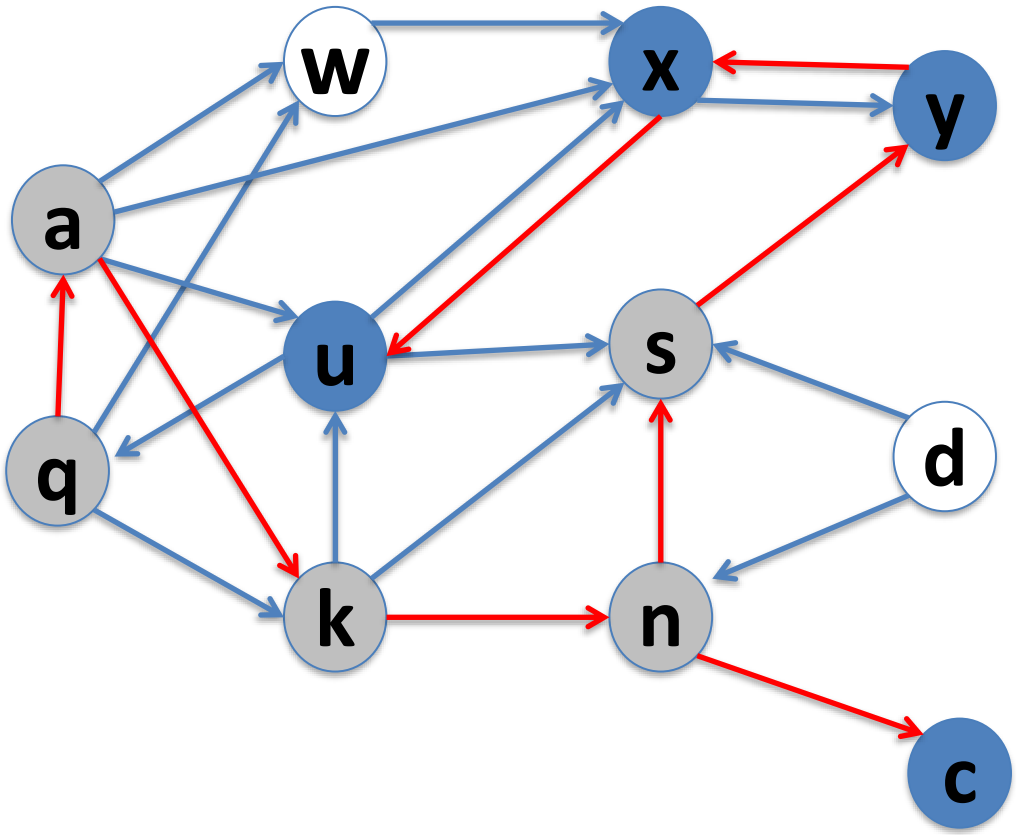
Depth-First Search



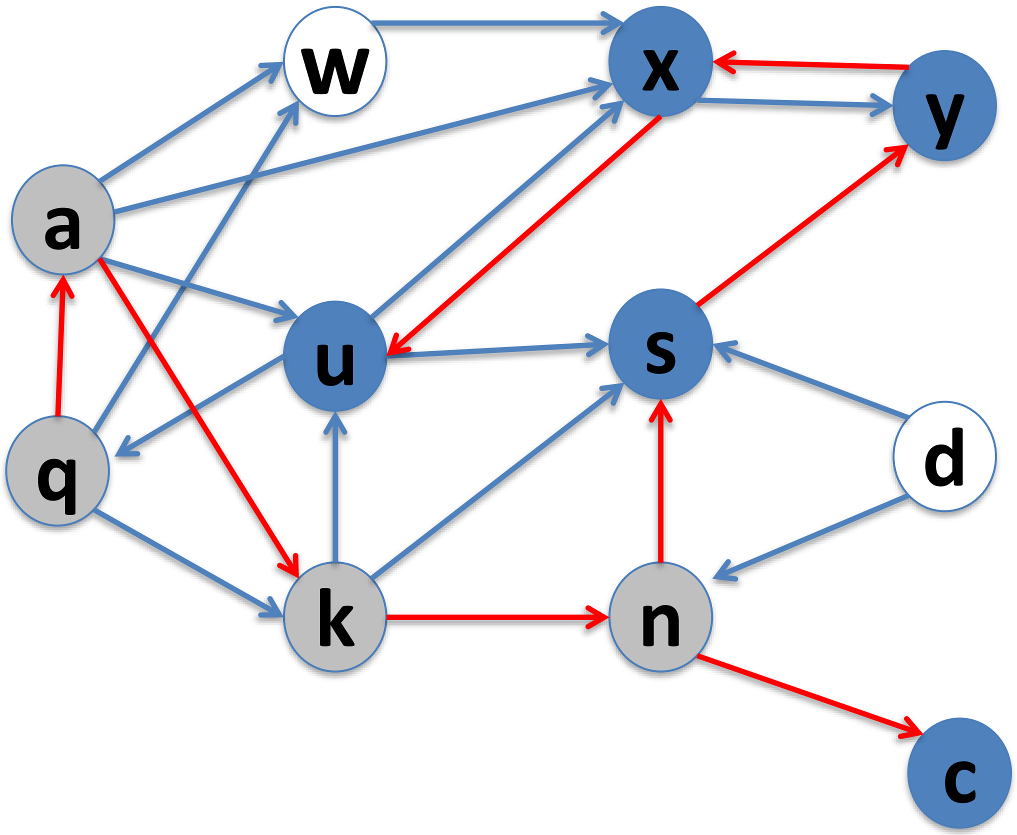
Depth-First Search



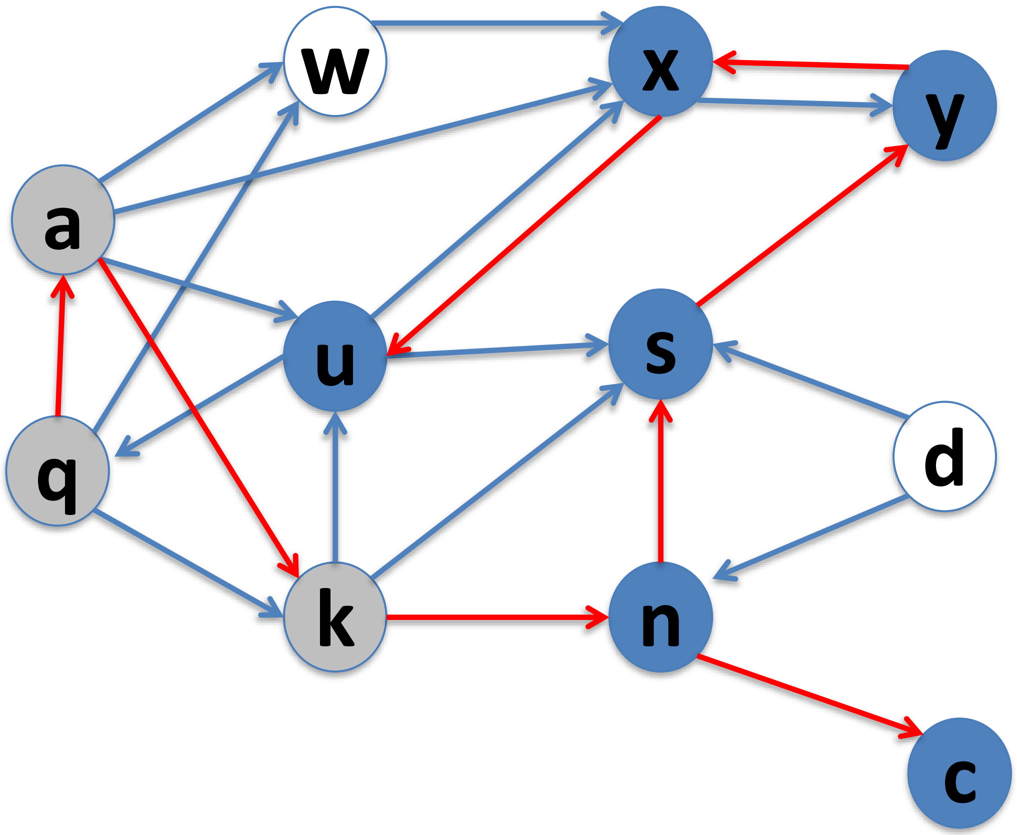
Depth-First Search



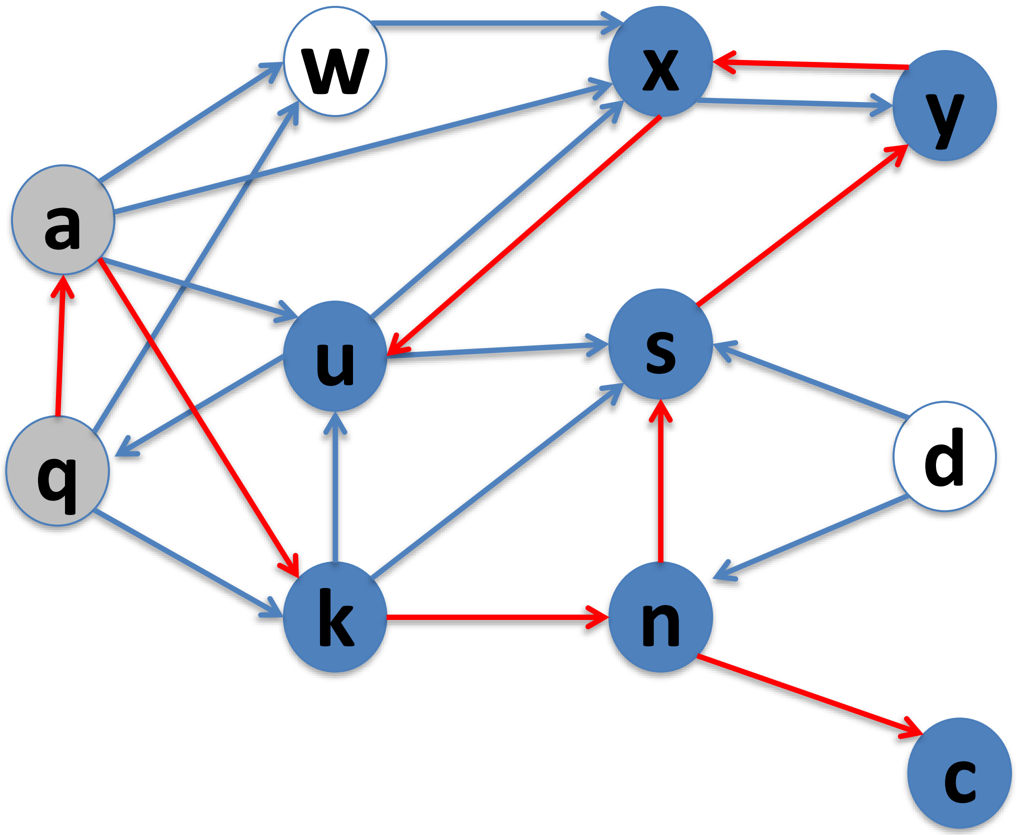
Depth-First Search



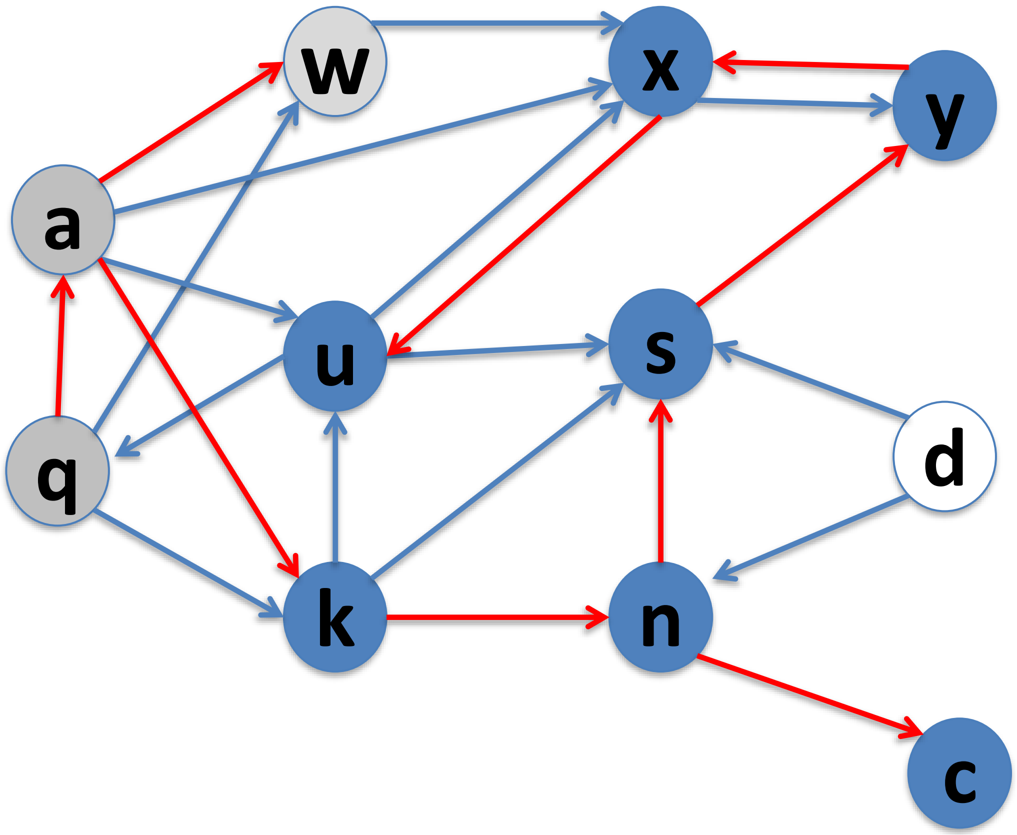
Depth-First Search



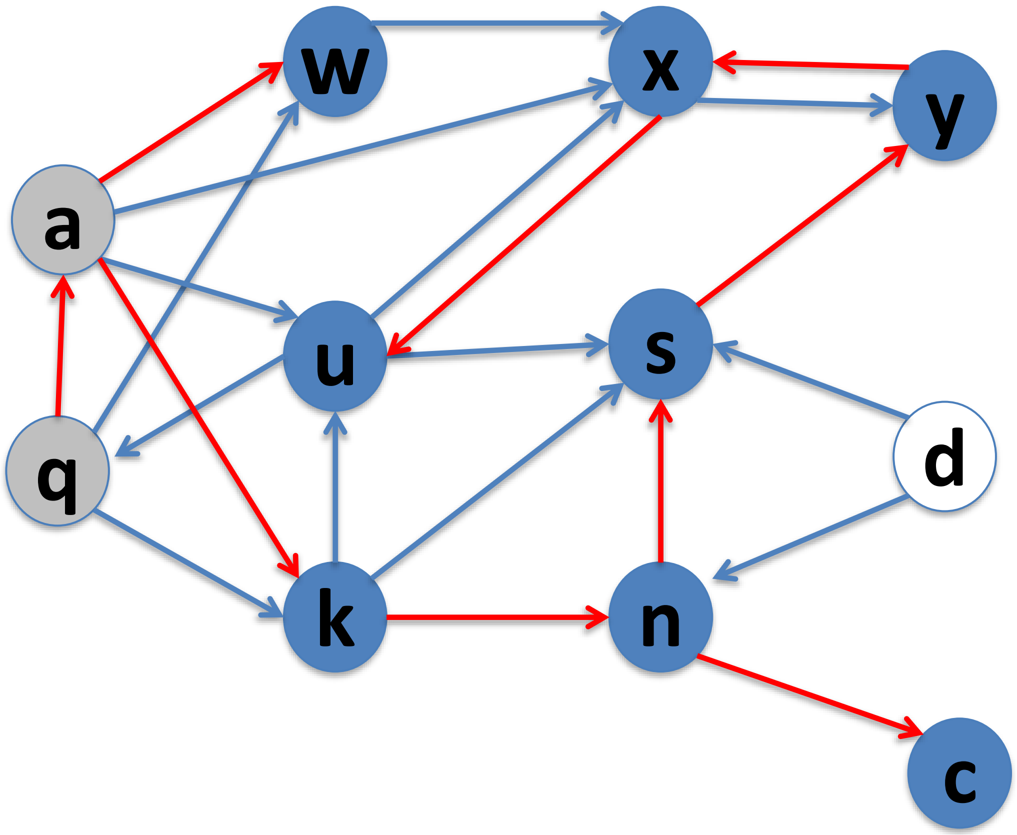
Depth-First Search



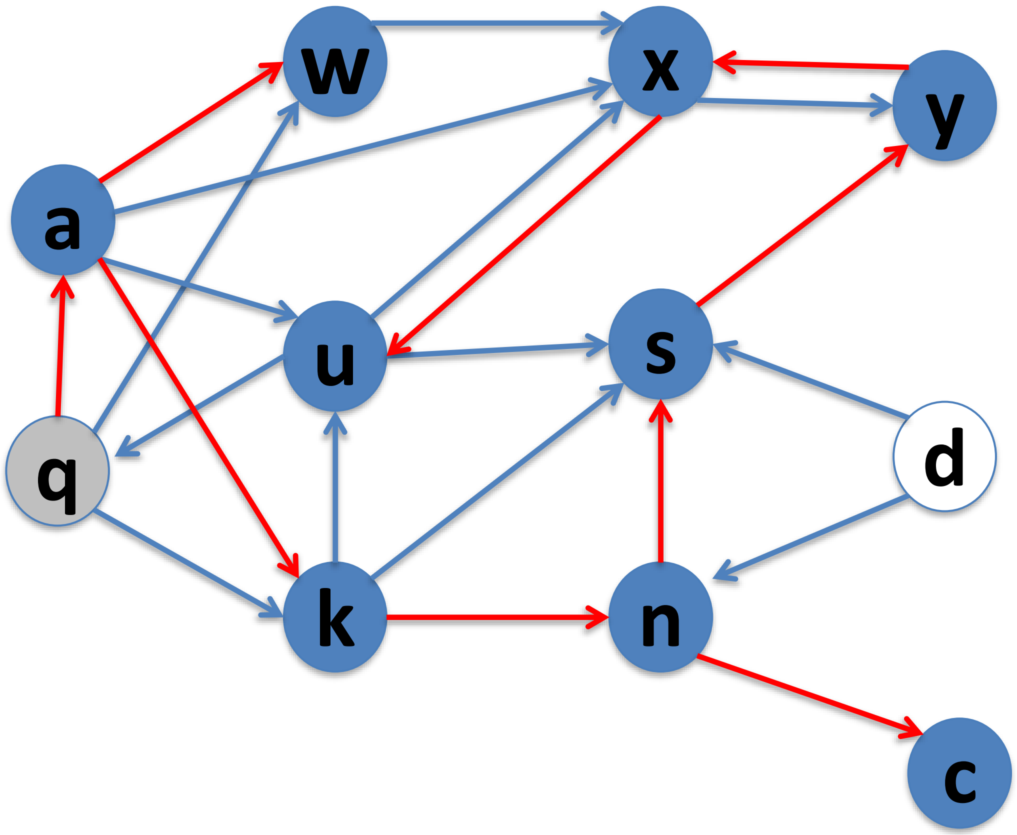
Depth-First Search



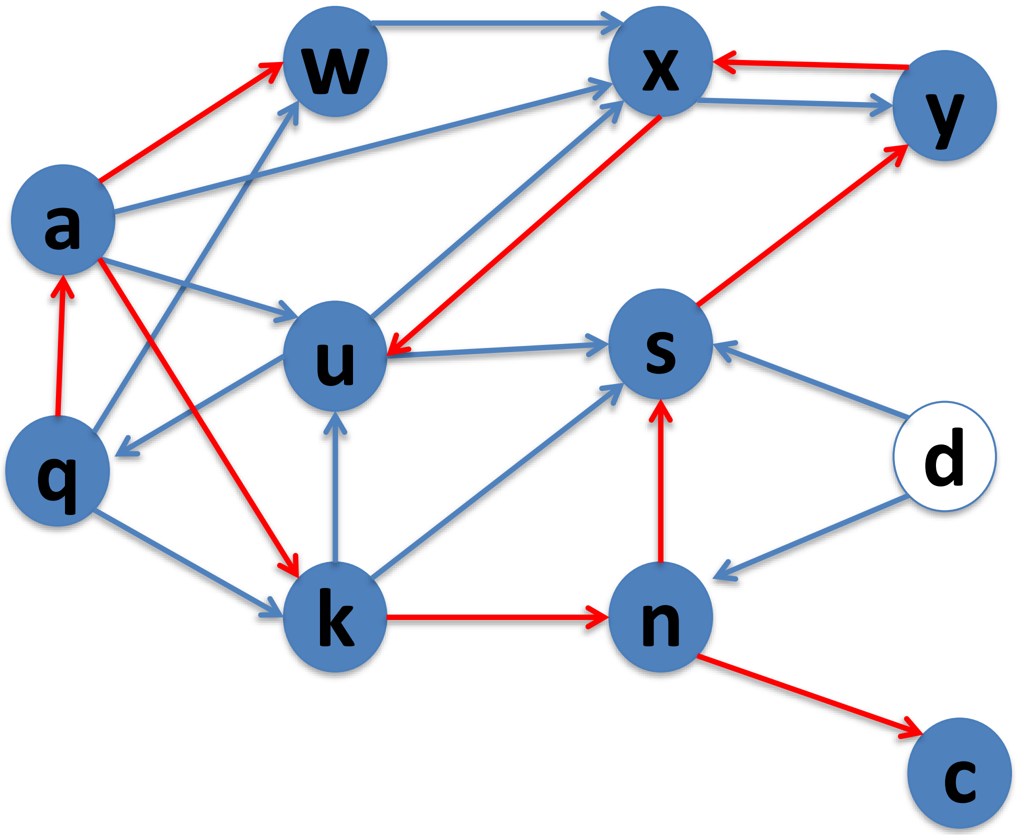
Depth-First Search



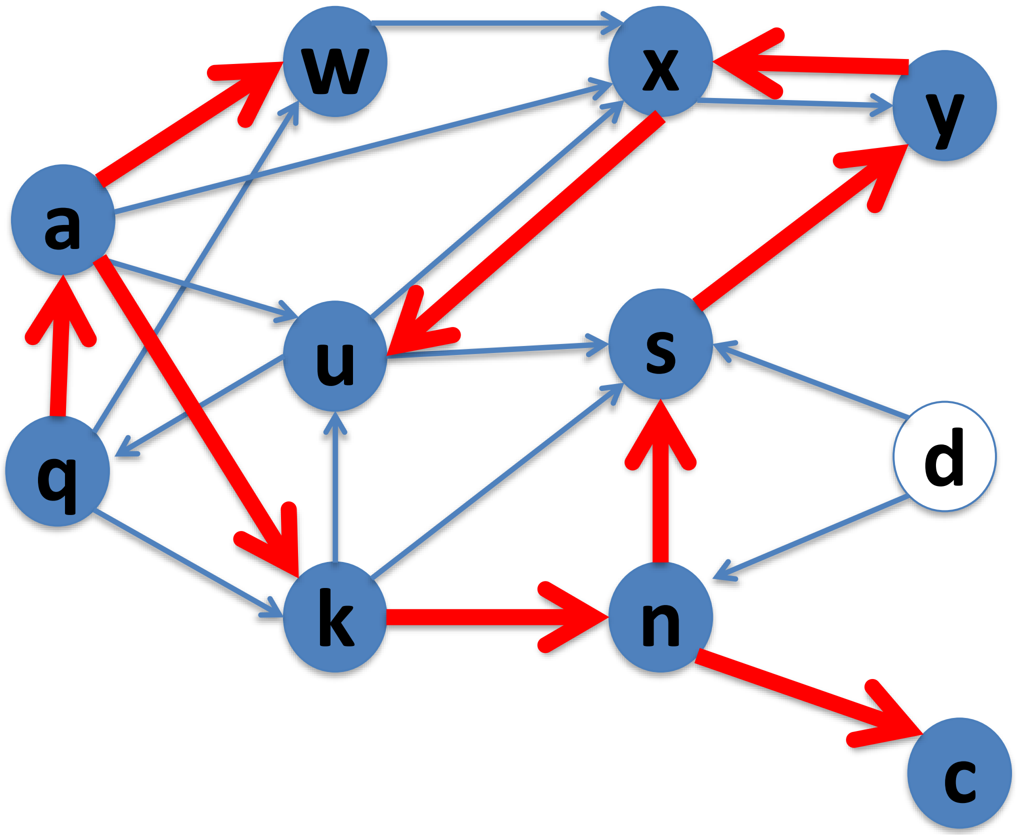
Depth-First Search



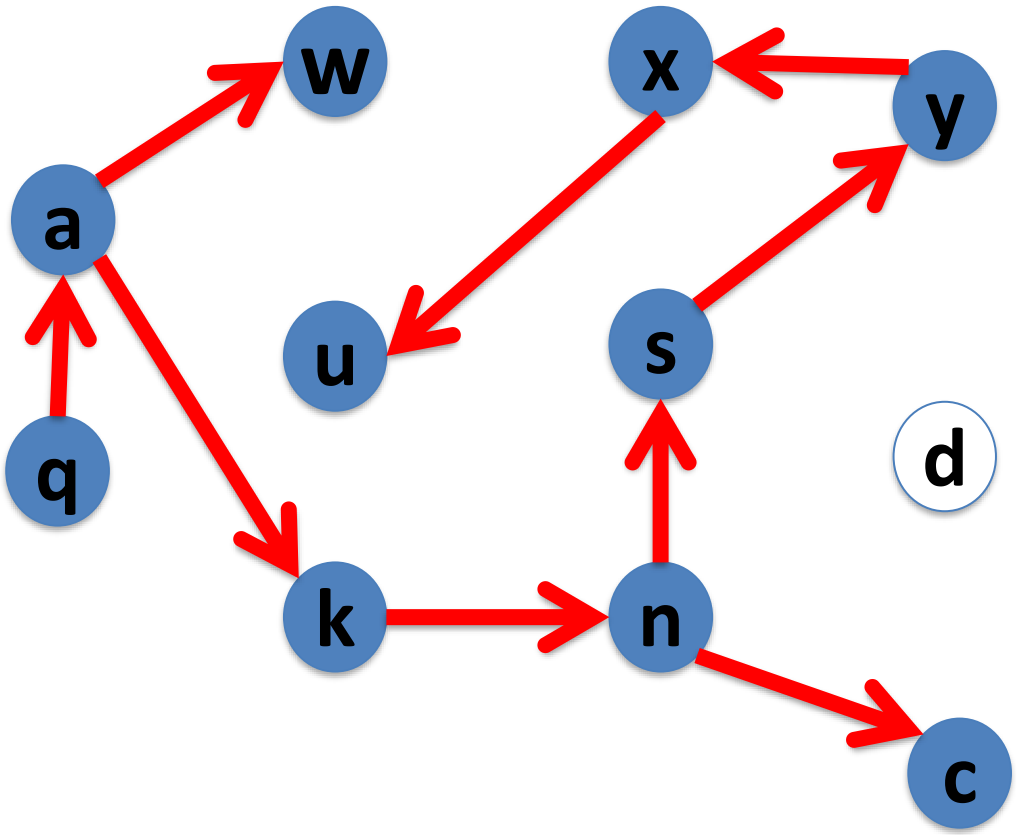
Depth-First Search



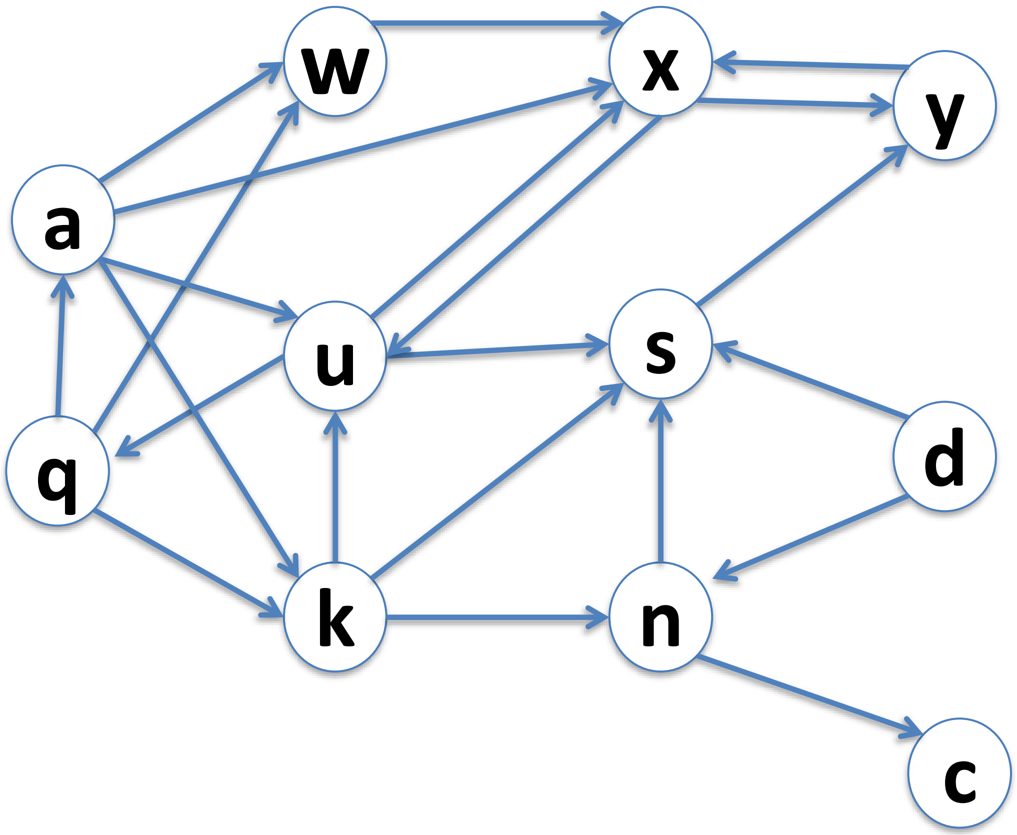
Depth-First Search



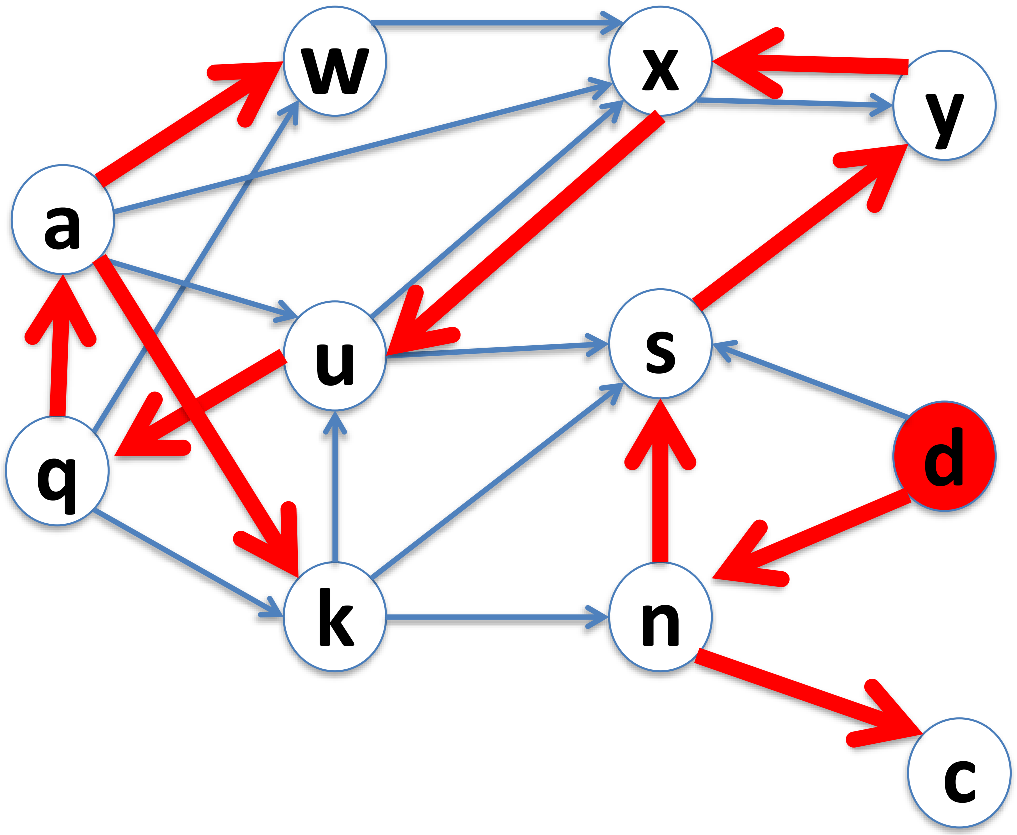
Depth-First Search



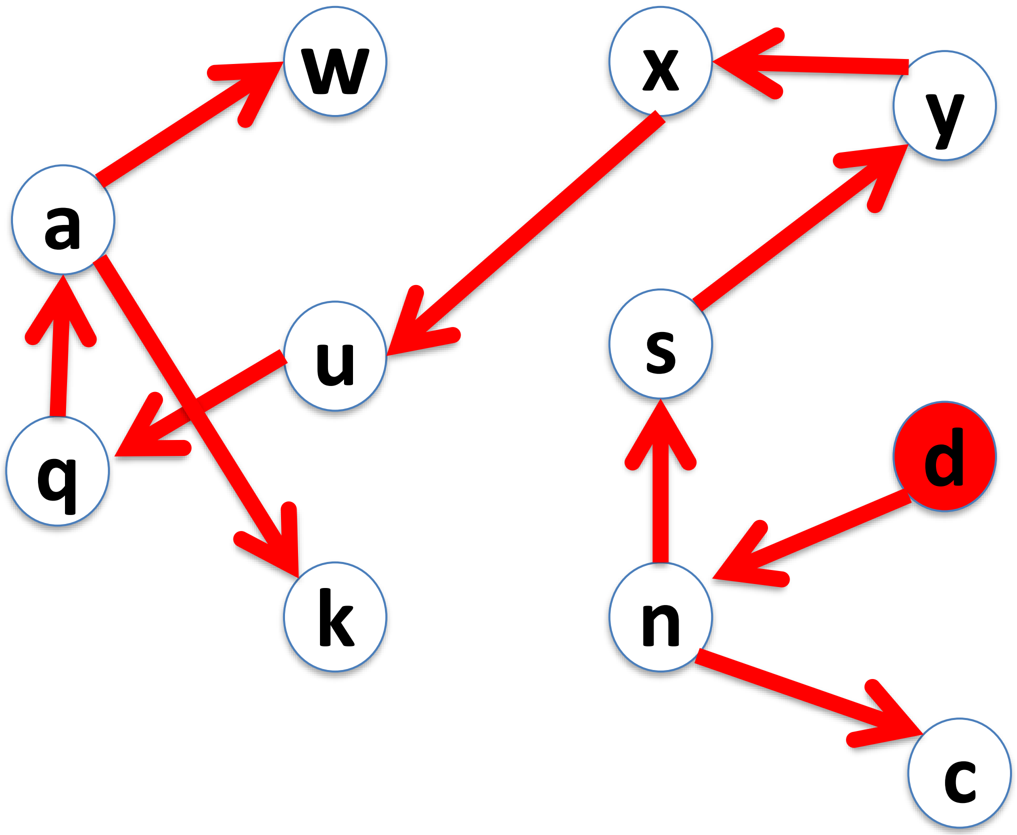
Depth-First Search



Depth-First Search



Depth-First Search



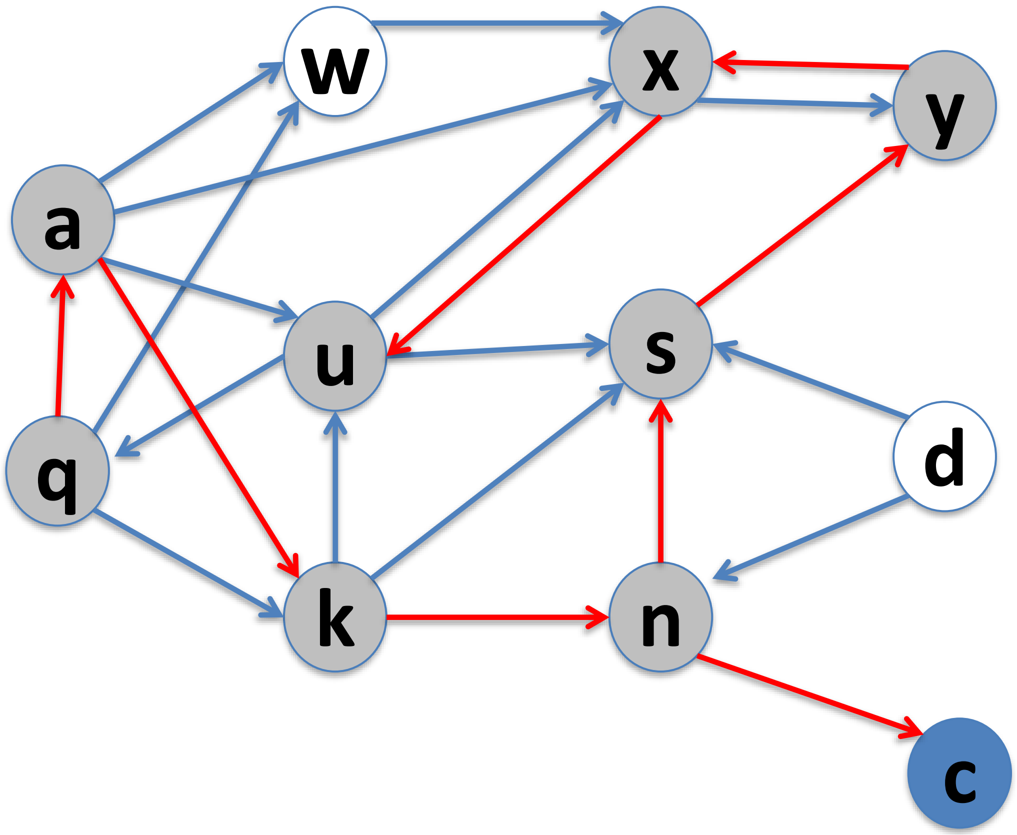
Depth-First Search

What is the runtime of DFS?

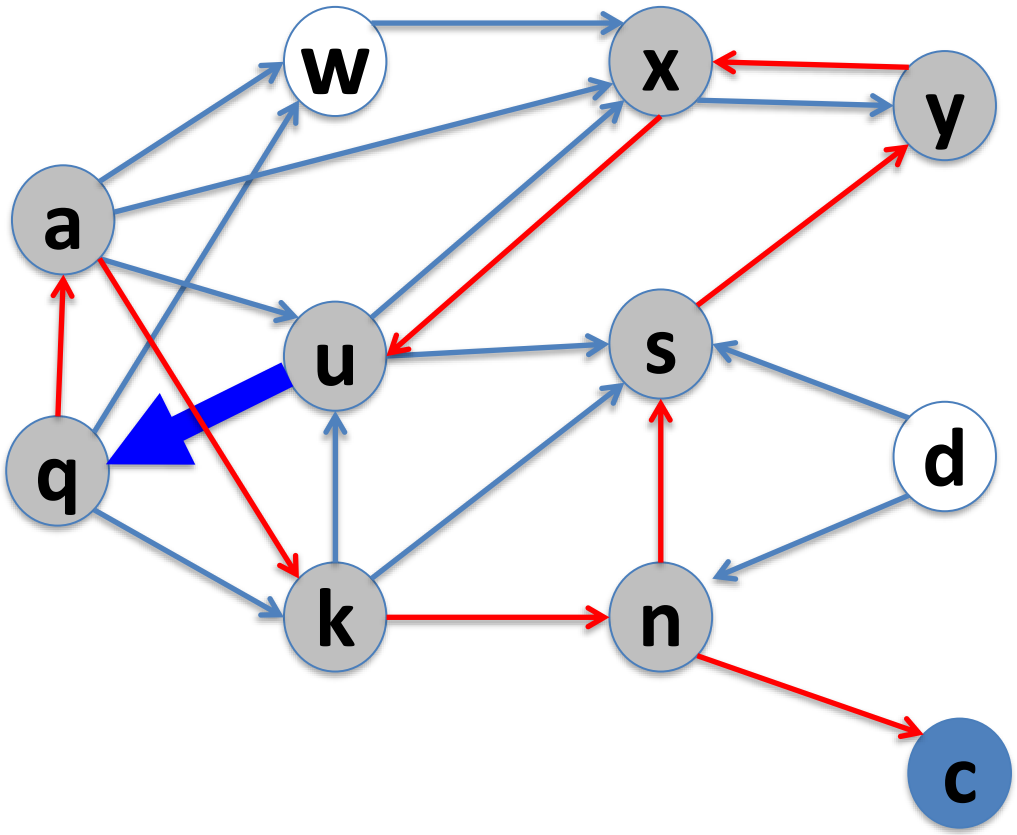
- Adjacency Matrix

- Adjacency Lists

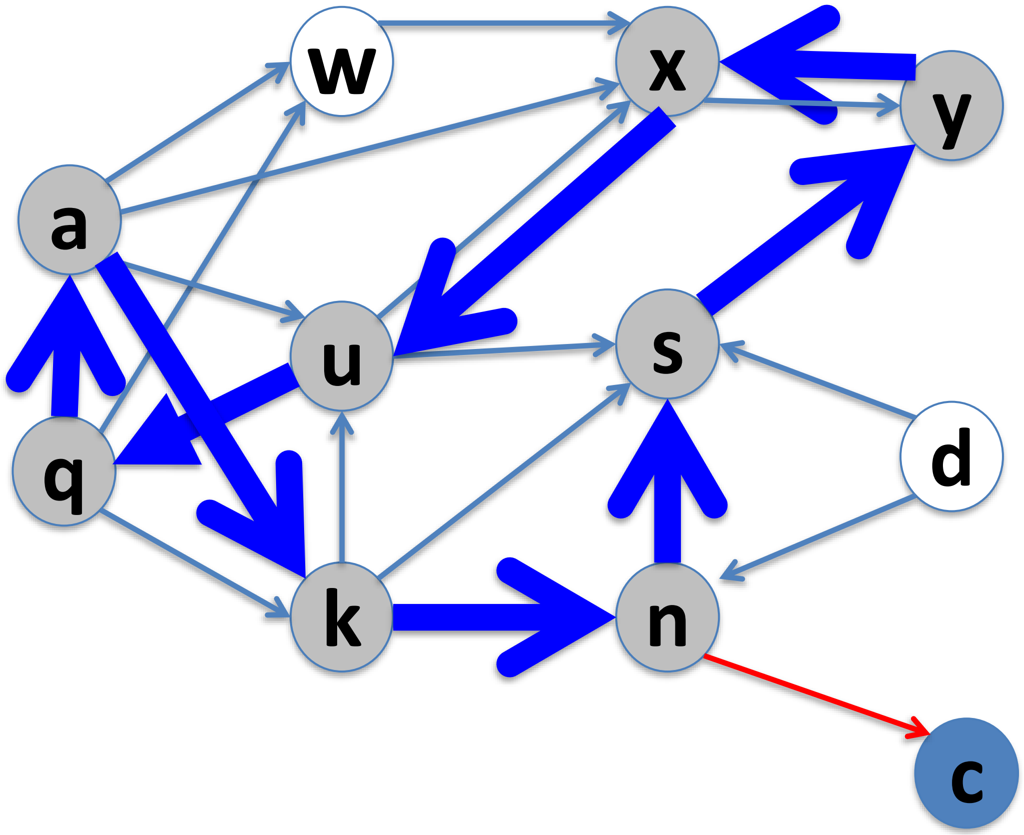
Depth-First Search



Depth-First Search



Depth-First Search



Depth-First Search

Depth-first search can be used to detect cycles in a graph.

(More about this in COMP3804)

Graph Exploration

What is the runtime of BFS and DFS?

- Adjacency Matrix

$$O(n^2)$$

- Adjacency Lists

$$O(n + m)$$