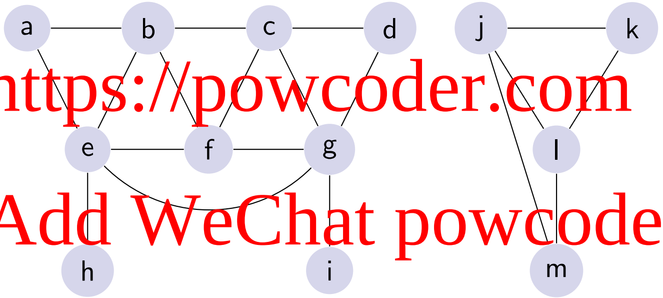
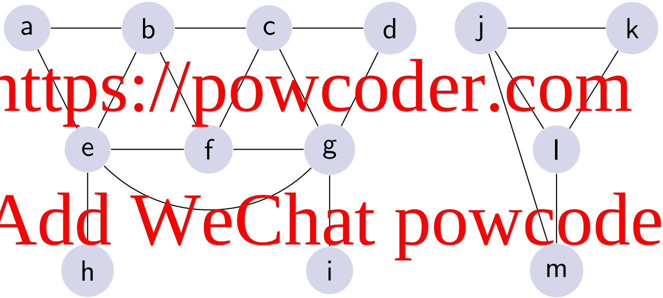
Algorithms Week 8

Ljubomir Perković, DePaul University

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.

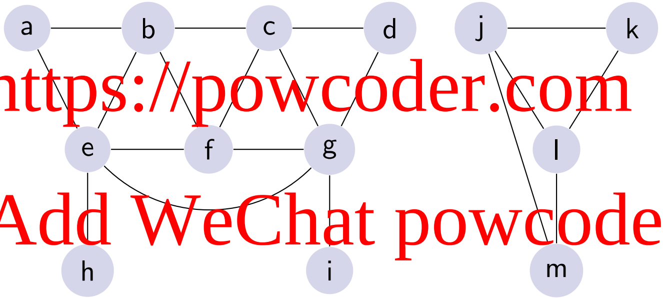


A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



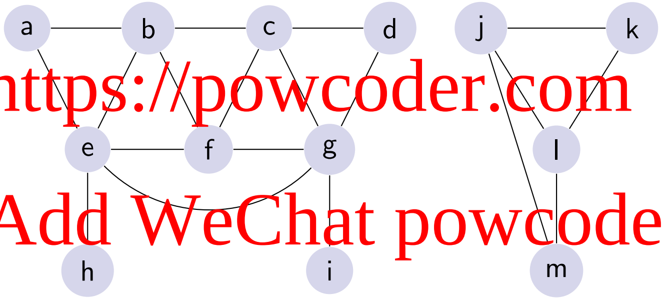
undirected graph

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



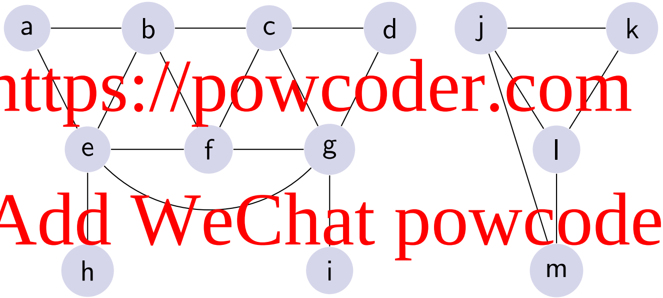
directed graph

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



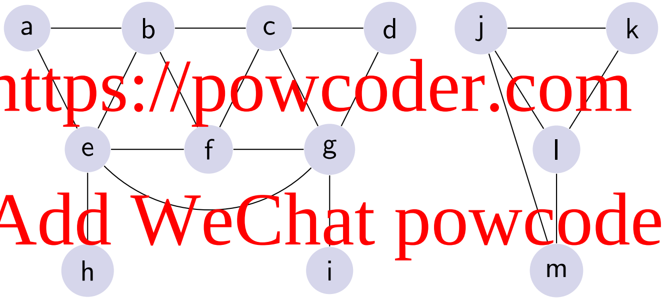
endpoints

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



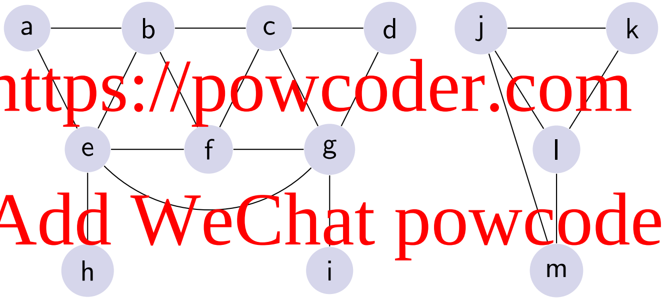
tail and head

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



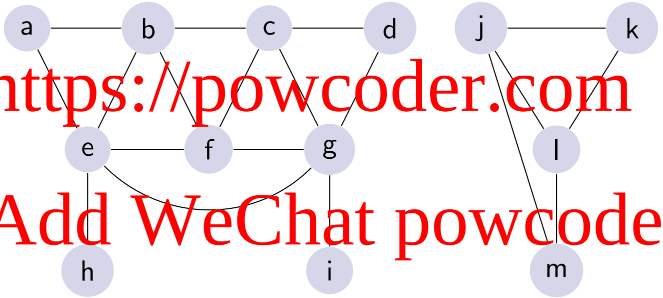
neighbor

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



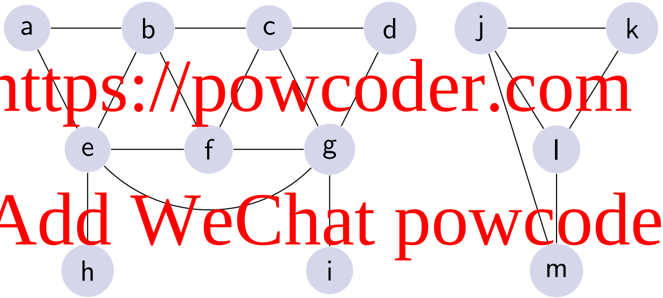
adjacent

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



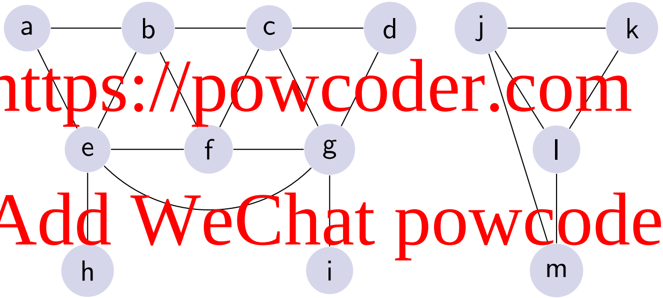
degree

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



in-degree and out-degree

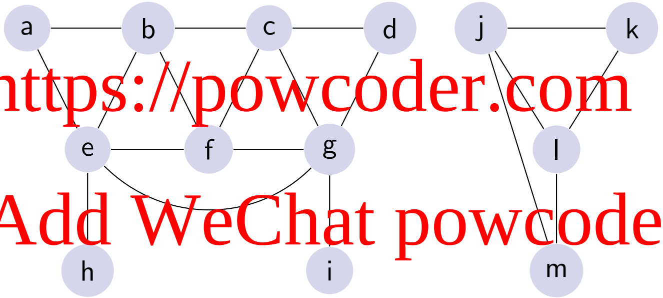
A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



subgraph

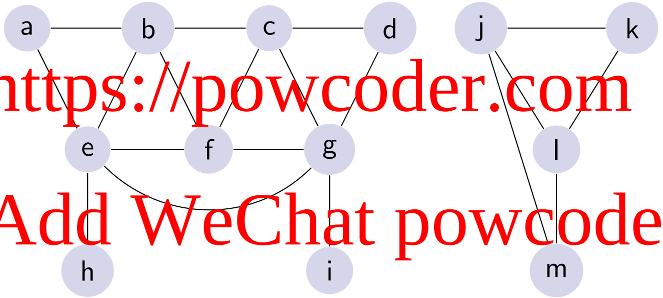
Definitions

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



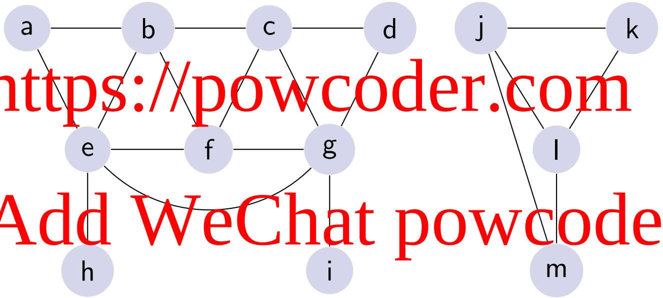
walk

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



path

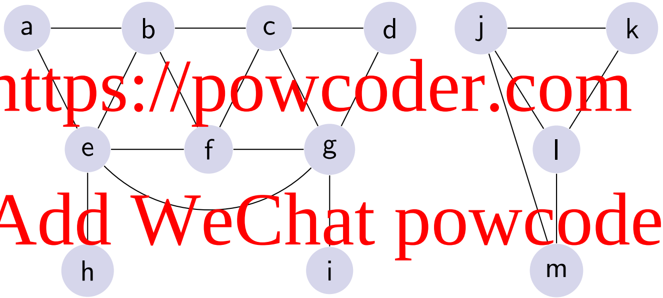
A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



reachable

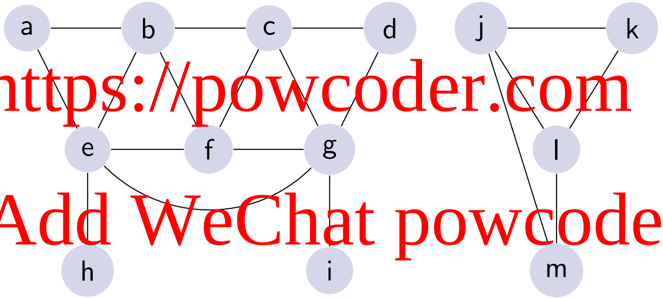
Definitions

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



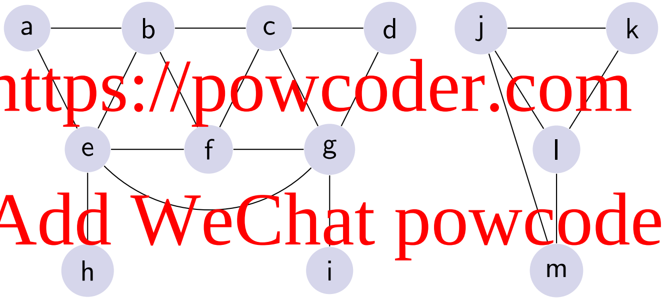
connected

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



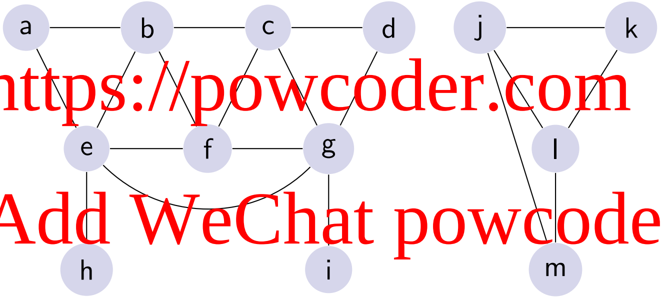
conected components

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



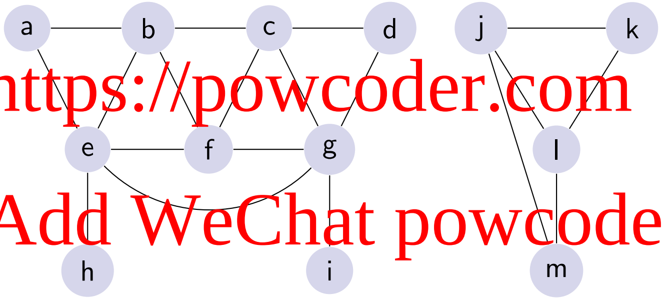
closed walk

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



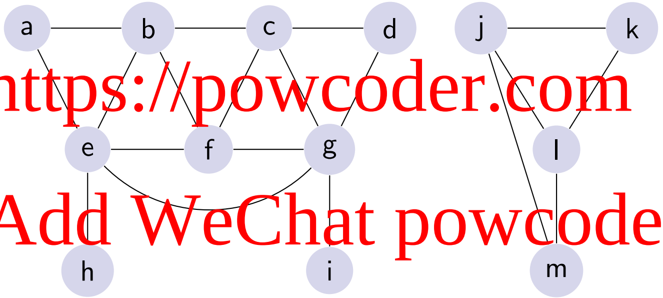
cycle

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



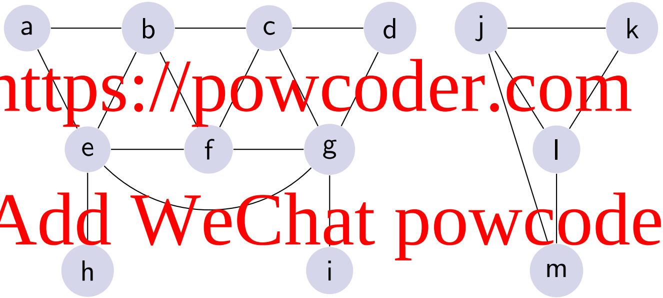
acyclic

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



tree

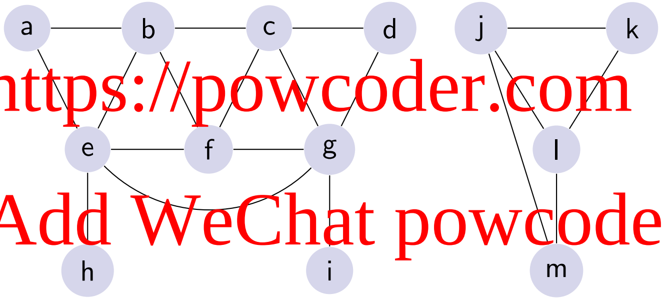
A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



forest

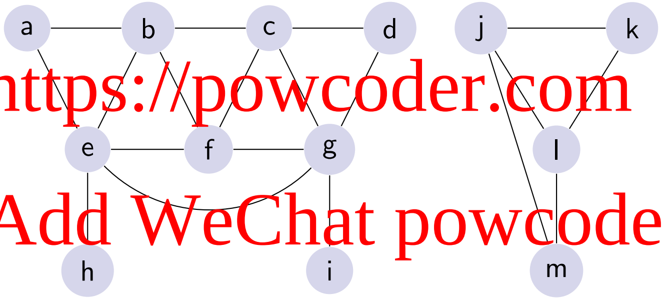
Definitions

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



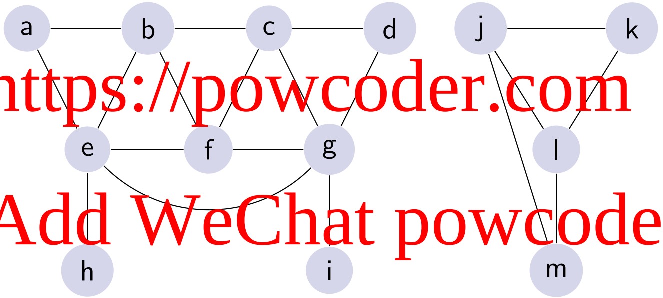
directed walk

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



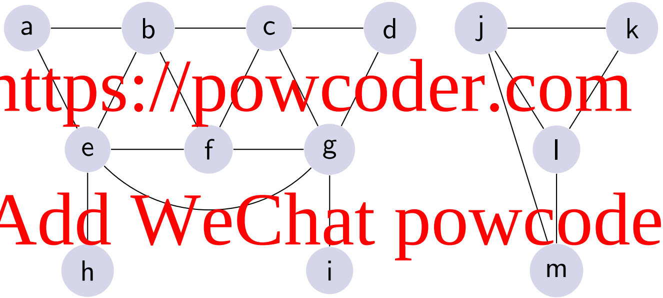
reachable

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



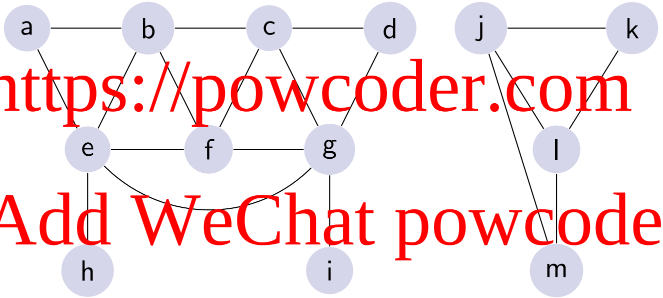
strongly connected

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



acyclic

A graph is a pair of sets (V, E) where V is a set of elements called vertices (or nodes) and E is a set of pairs of elements of V called edges.



dag

Graphs can be used to model many natural and social phenomena:

Graphs can be used to model many natural and social phenomena:

- map with towns and roads between them

Graphs can be used to model many natural and social phenomena:

- map with towns and roads between them
- molecule

Graphs can be used to model many natural and social phenomena:

- map with towns and roads between them
- molecule
- social network

Graphs can be used to model many natural and social phenomena:

- map with towns and roads between them
- molecule
- social network
- web graph

Graphs can be used to model many natural and social phenomena:

- map with towns and roads between them
- molecule
- social network
- web graph
- dependency graph

Graphs can be used to model many natural and social phenomena:

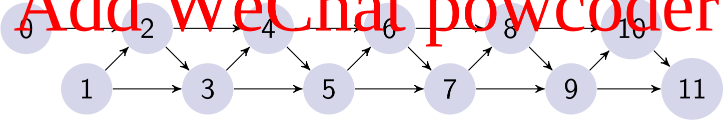
- map with towns and roads between them
- molecule
- social network
- web graph
- dependency graph
  - resource allocation graph in operating systems

Graphs can be used to model many natural and social phenomena:

- map with towns and roads between them
- molecule
- social network
- web graph
- dependency graph
  - resource allocation graph in operating systems
  - recurrence relation graph

$$Fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ Fib(n-1) + Fib(n-2), & \text{otherwise} \end{cases}$$

$$Fib(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ Fib(n-1) + Fib(n-2), & \text{otherwise} \end{cases}$$



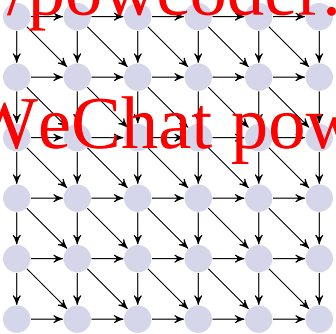
Longest Common Subsequence recurrence graph

If $LCS(i,j)$ is the length of the LCS of $A[1..i]$ and $B[1..j]$ then

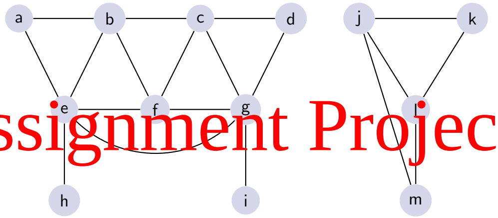
$$LCS(i,j) = \begin{cases} 0, & \text{if } j \text{ or } i = 0 \\ LCS(i-1,j-1)+1, & \text{if } A[i] = B[j] \\ \max\{LCS(i,j-1), LCS(i-1,j)\} & \text{otherwise} \end{cases}$$

If $LCS(i,j)$ is the length of the LCS of $A[1..i]$ and $B[1..j]$ then

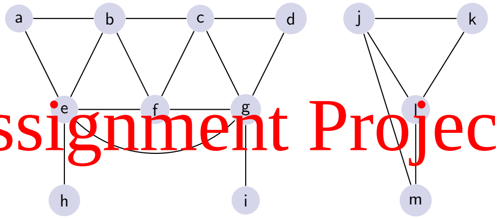
$$LCS(i,j) = \begin{cases} 0, & \text{if } j \text{ or } i = 0 \\ LCS(i-1,j-1)+1, & \text{if } A[i] = B[j] \\ \max\{LCS(i,j-1), LCS(i-1,j)\} & \text{otherwise} \end{cases}$$



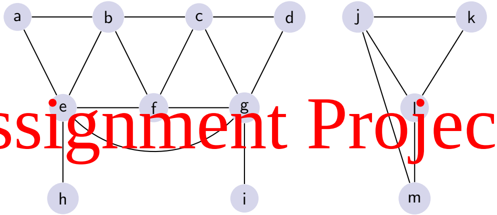
Graph data structure: adjacency list



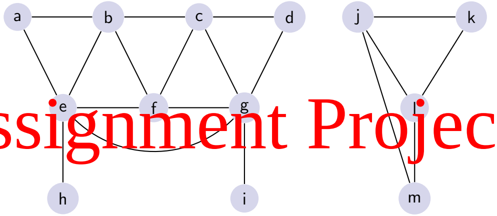
Graph data structure: adjacency list



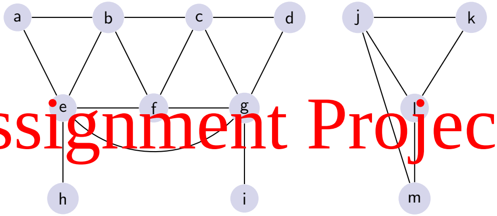
- Test $uv \in E$



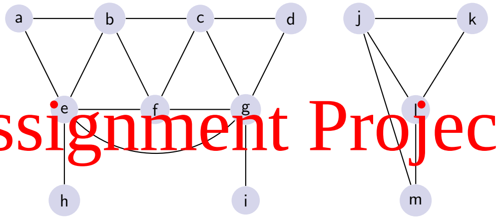
- Test $uv \in E$
- List v's neighbors



- Test $uv \in E$
- List v's neighbors
- List all edges

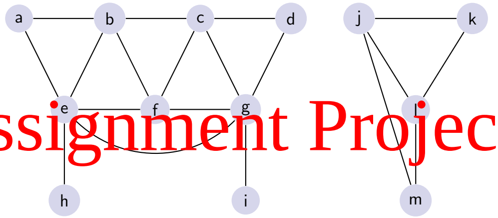


- Test $uv \in E$
- List v's neighbors
- List all edges
- Insert edge uv

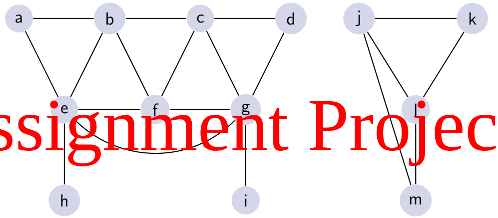


- Test $uv \in E$
- List v's neighbors
- List all edges
- Insert edge uv
- Delete edge uv

Graph data structure: adjacency matrix

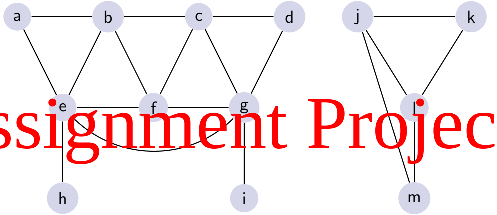


Graph data structure: adjacency matrix



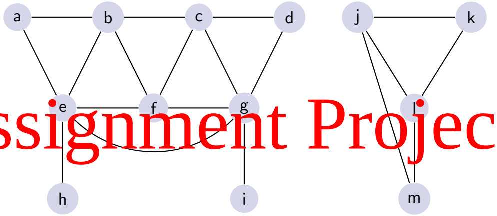
- Test $uv \in E$

Graph data structure: adjacency matrix

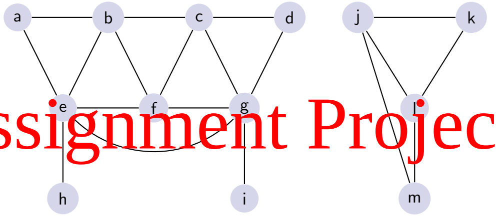


- Test $uv \in E$
- List v's neighbors

Graph data structure: adjacency matrix

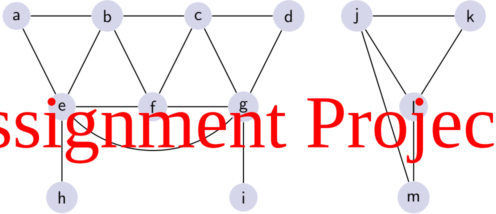


- Test $uv \in E$
- List v's neighbors
- List all edges



- Test $uv \in E$
- List v's neighbors
- List all edges
- Insert edge uv

Graph data structure: adjacency matrix



- Test $uv \in E$
- List v's neighbors
- List all edges
- Insert edge uv
- Delete edge uv

	Adjacency list	Adjacency matrix
Space	$\Theta(V+E)$	$\Theta(V^2)$
Test $uv \in E$	$O(\min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$
List v's neighbors	$O(\deg(u)) = O(V)$	$\Theta(V)$
List all edges	$\Theta(V+E)$	$\Theta(V^2)$
Insert edge uv	$O(1)$	$O(1)$
Delete edge uv	$O(\deg(u) + \deg(v)) = O(V)$	$O(1)$

Iterating over all items of a data structure to search for something is a fundamental computing task.

Iterating over all items of a data structure to search for something is a fundamental computing task.

Obvious ways to systematically search arrays,

Iterating over all items of a data structure to search for something is a fundamental computing task.

Obvious ways to systematically search arrays, lists...

Iterating over all items of a data structure to search for something is a fundamental computing task.

Obvious ways to systematically search arrays, lists...

Less obvious how to do it for non-linear data structures:

Iterating over all items of a data structure to search for something is a fundamental computing task.

Obvious ways to systematically search arrays, lists...

Less obvious how to do it for non-linear data structures:

- Rooted trees:

Iterating over all items of a data structure to search for something is a fundamental computing task.

Obvious ways to systematically search arrays, lists...

Less obvious how to do it for non-linear data structures:

- Rooted trees: use pre-order, in-order, post-order, or level order traversal

Iterating over all items of a data structure to search for something is a fundamental computing task.

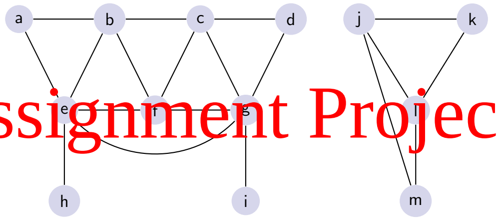
Obvious ways to systematically search arrays, lists...

Less obvious how to do it for non-linear data structures:

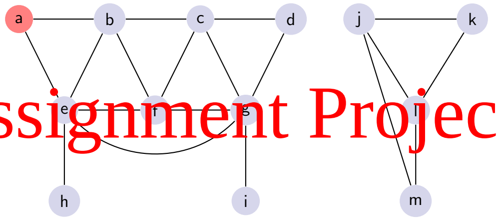
- Rooted trees: use pre-order, in-order, post-order, or level order traversal

- Graphs: Depth-First traversal/Search (DFS), Breadth-First traversal/Search (BFS)...

Depth-First Search (Recursive)

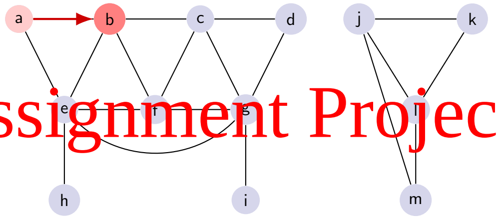



```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

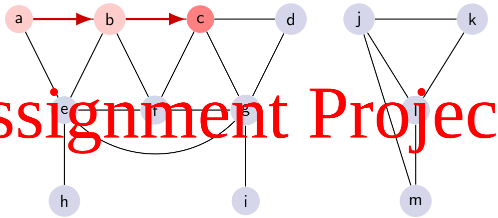



```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

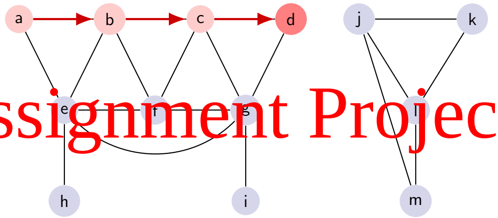
Depth-First Search (Recursive)



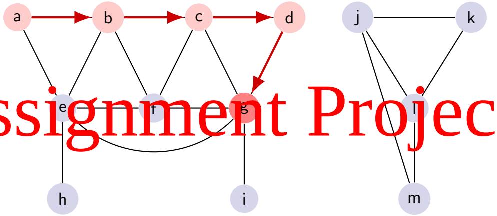

```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```



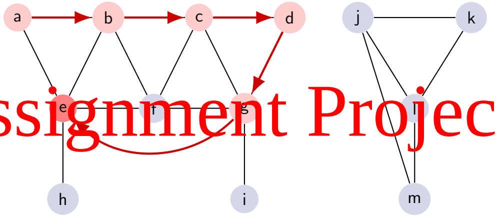

```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```




```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

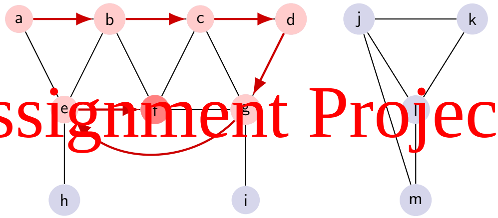



```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

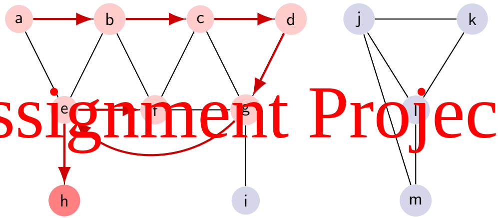



```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

Depth-First Search (Recursive)

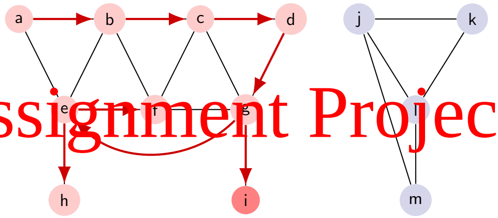



```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

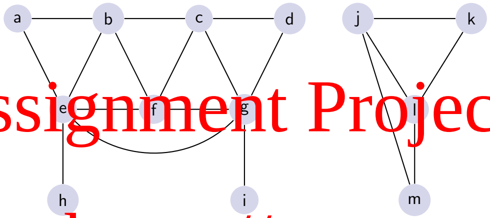



```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

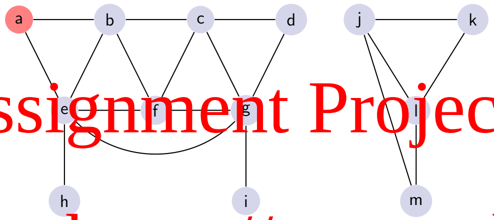
Depth-First Search (Recursive)




```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

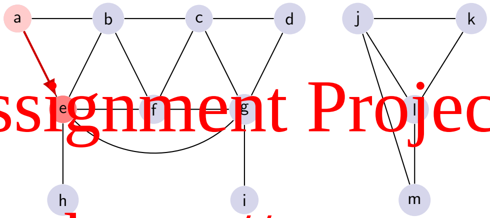



```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

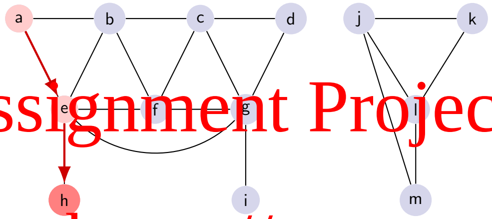



```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

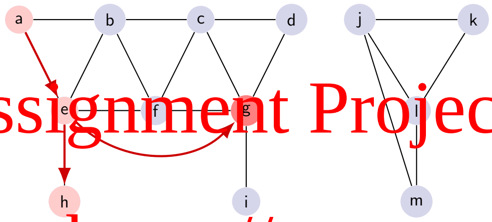
Depth-First Search (Iterative)




```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

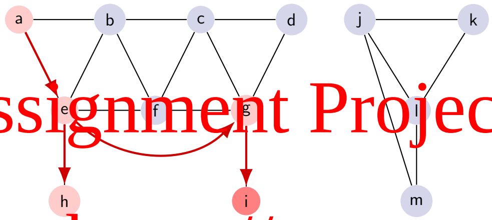



```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

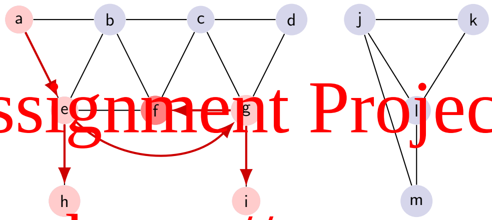



```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ← Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

Depth-First Search (Iterative)

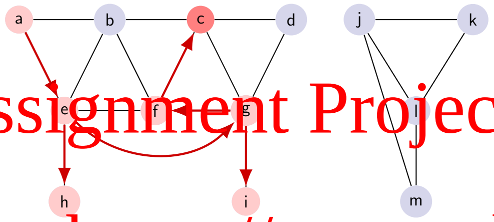



```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

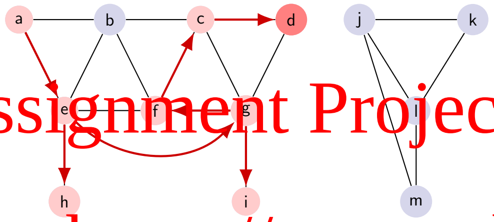



```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

Depth-First Search (Iterative)

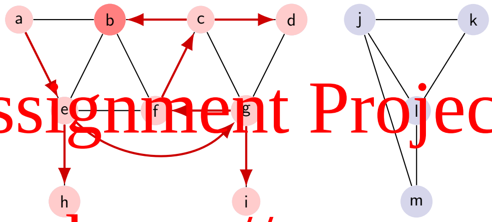



```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```




```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

Depth-First Search (Iterative)




```
IterativeDFS(s)
  Push(s)
  while the stack is not empty
    v ⇐ Pop
    if v is unmarked
      mark v
      for each edge vw
        Push(w)
```

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
    take v from the bag
    if v is unmarked
      mark v
      for each edge vw
        put w into the bag
```

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
    take v from the bag
    if v is unmarked
      mark v
      for each edge vw
        put w into the bag
```


If T is the time required to insert a single item into the bag or delete a single item from the bag then the running time of WFS is

```
WhateverFirstSearch(s):
  put s into the bag
  while the bag is not empty
    take v from the bag
    if v is unmarked
      mark v
      for each edge vw
        put w into the bag
```


If T is the time required to insert a single item into the bag or delete a single item from the bag then the running time of WFS is

- $O(V + ET)$ if G stored in adjacency list
- $O(V^2 + ET)$ if G stored in an adjacency matrix

To construct explicitly the spanning search tree:

Whatever-First Search spanning tree

To construct explicitly the spanning search tree:

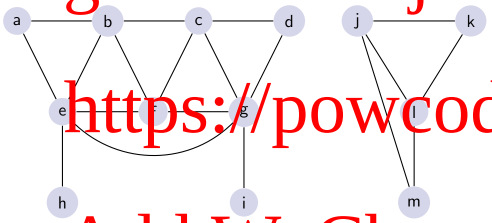

```
WhateverFirstSearch(s):
  put (-, s) in bag
    while the bag is not empty
    take (p, v) from the bag
    if v is unmarked
      mark v
      parent(v) ← p
      for each edge vw
        put (v, w) into the bag
```

If Bag == Stack then WFS == DFS.

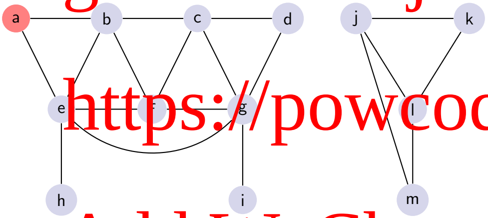
If Bag == Stack then WFS == DFS.

Since Stack operations push and pop run in $T = O(1)$ time DFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.

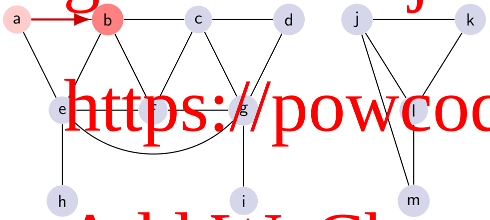


If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



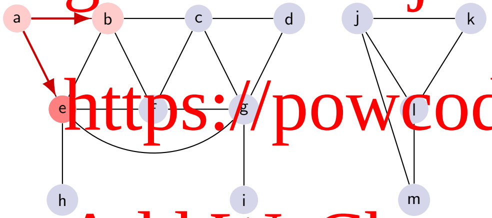
Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



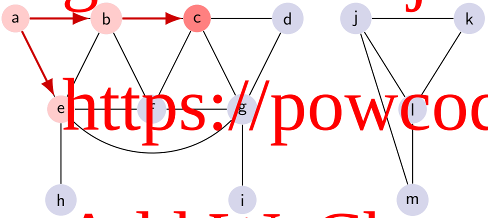
Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



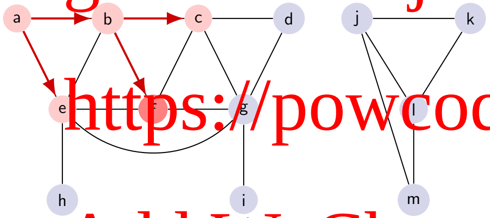
Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



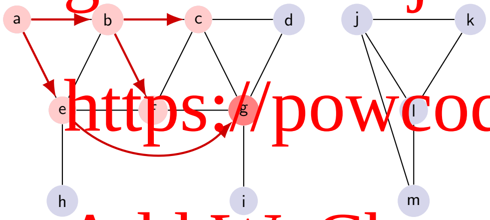
Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



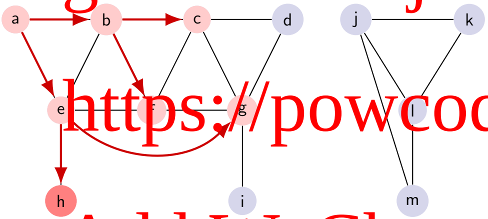
Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



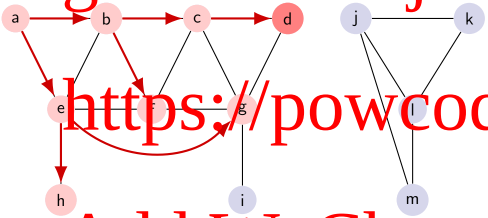
Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



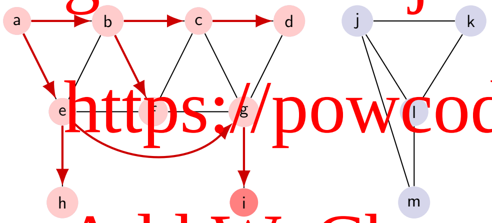
Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag == Queue then WFS == BFS, i.e. Breadth-First Search.



Since Queue operations enqueue and dequeue run in $T = O(1)$ time BFS runs in $O(V + E)$ time if G is represented using an adjacency list.

If Bag $==$ Priority then WFS is the family of algorithms Best-First Search.

If Bag == Priority then WFS is the family of algorithms Best-First Search.

By assigning edge priorities in specific ways Best-First Search can be used to compute:

If Bag == Priority then WFS is the family of algorithms Best-First Search.

By assigning edge priorities in specific ways Best-First Search can be used to compute:

- the minimum spanning tree

If Bag == Priority then WFS is the family of algorithms Best-First Search.

By assigning edge priorities in specific ways Best-First Search can be used to compute:

- the minimum spanning tree
- the tree containing the *shortest* paths from a specific vertex to all vertices reachable from it

If Bag == Priority then WFS is the family of algorithms Best-First Search.

By assigning edge priorities in specific ways Best-First Search can be used to compute:

- the minimum spanning tree
- the tree containing the *shortest* paths from a specific vertex to all vertices reachable from it
- the tree containing the *widest* paths from a specific vertex to all vertices reachable from it

If Bag == Priority then WFS is the family of algorithms Best-First Search.

By assigning edge priorities in specific ways Best-First Search can be used to compute:

- the minimum spanning tree
- the tree containing the *shortest* paths from a specific vertex to all vertices reachable from it
- the tree containing the *widest* paths from a specific vertex to all vertices reachable from it

Since PriorityQueue operations insert and extractMin (or extractMax) run in $T = O(\log E)$ time Best-First Search runs in $O(V + E \log E)$ time if G is represented using an adjacency list.

Disconnected graphs

Traverses the graph component by component.


```
WFSAll(G):
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      WhateverFirstSearch(v)
```

Traverses the graph component by component. Can be extended to count the number of connected components ...

```
CountComponents(G):
  count ← 0
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      count ← count + 1
      WhateverFirstSearch(v)
  return count
```

Traverses the graph component by component. Can be extended to count the number of connected components ... or record which component contains each vertex.


```
CountAndLabel(G):
  count ← 0
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      count ← count + 1
      LabelOne(v, count)
  return count
```

Traverses the graph component by component. Can be extended to count the number of connected components ... or record which component contains each vertex.


```
LabelOne(v, count):
  while the bag is not empty
    take v from the bag
    if v is unmarked
      mark v
      comp(v) ← count
      for each edge vw
        put w into the bag
```

```
RecursiveDFS(v):
  if v is unmarked
    mark v
      for each edge vw
        RecursiveDFS(w)
```

```
RecursiveDFS(v):
  if v is unmarked
    mark v
    for each edge vw
      RecursiveDFS(w)
```



```
DFS(v):
  mark v
  for each edge vw
    if w is unmarked
      parent(w) ← v
      DFS(w)
```


Slightly faster (in practice) by checking whether a node is marked before we recursively explore it.

```
DFSAll(G):
  clock ← 0
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      clock ←DFS(v,clock)
```

```
DFS(v, clock):
  mark v
  clock← clock+1
  v.pre← clock
  for each edge v→w
    if w is unmarked
      w.parent ← v
      clock ← DFS(w,clock)
  clock← clock+1
  v.post← clock
  return clock
```


Add counter (clock) that will be helpful in terms of understanding the graph structure.

This DFS algorithm assigns assigns

- *v.pre* (and advances the clock) just after pushing v onto the recursion stack

This DFS algorithm assigns assigns

- *v.pre* (and advances the clock) just after pushing v onto the recursion stack
- *v.post* (and advances the clock) just before popping v off the recursion stack.

This DFS algorithm assigns assigns

- *v.pre* (and advances the clock) just after pushing v onto the recursion stack

- *v.post* (and advances the clock) just before popping v off the recursion stack.

For any two vertices u and v, the intervals $[u.pre, u.post]$ and $[v.pre, v.post]$ are either disjoint or nested.

This DFS algorithm assigns assigns

- $v.pre$ (and advances the clock) just after pushing v onto the recursion stack
- $v.post$ (and advances the clock) just before popping v off the recursion stack.

For any two vertices u and v, the intervals $[u.pre, u.post]$ and $[v.pre, v.post]$ are either disjoint or nested.

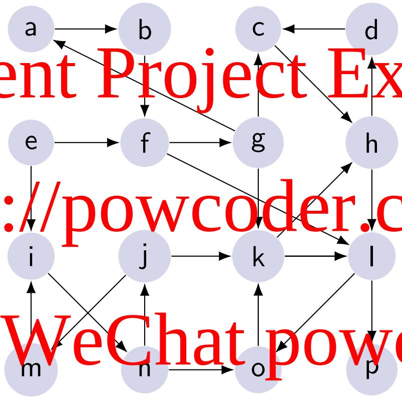
Moreover, $[u.pre, u.post]$ contains $[v.pre, v.post]$ if and only if $DFS(v)$ is called during the execution of $DFS(u)$, ...

This DFS algorithm assigns assigns

- $v.pre$ (and advances the clock) just after pushing v onto the recursion stack
- $v.post$ (and advances the clock) just before popping v off the recursion stack.

For any two vertices u and v, the intervals $[u.pre, u.post]$ and $[v.pre, v.post]$ are either disjoint or nested.

Moreover, $[u.pre, u.post]$ contains $[v.pre, v.post]$ if and only if $DFS(v)$ is called during the execution of $DFS(u)$, ... or equivalently, if and only if u is an ancestor of v in the DFS tree.


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p

- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$

- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest,

- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.

- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If $DFS(u)$ calls $DFS(v)$ directly then $u \to v$ is a tree edge

- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If $DFS(u)$ calls $DFS(v)$ directly then $u \rightarrow v$ is a tree edge
  - Otherwise, $u \rightarrow v$ is called a forward edge

- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If $DFS(u)$ calls $DFS(v)$ directly then $u \to v$ is a tree edge
  - Otherwise, $u \to v$ is called a forward edge

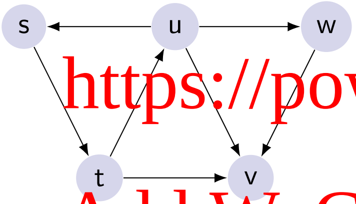
- If $DFS(v)$ has already been called and is still active when $DFS(u)$ begins then v is already on the recursion stack which implies $v.pre < u.pre < u.post < v.post$.

- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If $DFS(u)$ calls $DFS(v)$ directly then $u \rightarrow v$ is a tree edge
  - Otherwise, $u \rightarrow v$ is called a forward edge
- If $DFS(v)$ has already been called and is still active when $DFS(u)$ begins then v is already on the recursion stack which implies $v.pre < u.pre < u.post < v.post$. G must contain a directed path from v to u. $u \rightarrow v$ is then a back edge

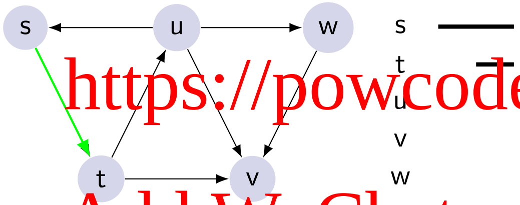
Classifying edge $u \to v$

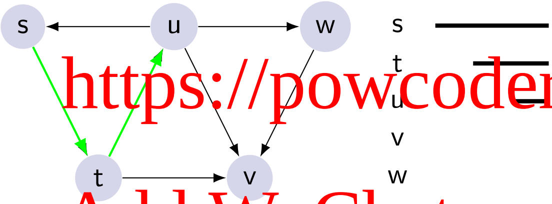
- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If $DFS(u)$ calls $DFS(v)$ directly then $u \to v$ is a tree edge
  - Otherwise, $u \to v$ is called a forward edge
- If $DFS(v)$ has already been called and is still active when $DFS(u)$ begins then v is already on the recursion stack which implies $v.pre < u.pre < u.post < v.post$. G must contain a directed path from v to u. $u \to v$ is then a back edge
- If $DFS(v)$ is finished when $DFS(u)$ begins, we immediately have $v.post < u.pre$.

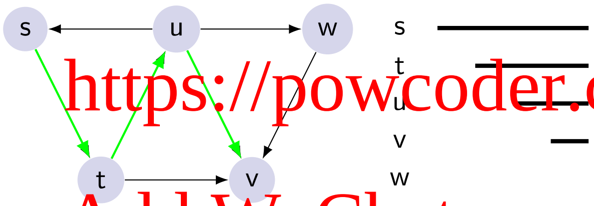
- If $DFS(v)$ has not yet been called when $DFS(u)$ begins then $DFS(v)$ must be called during the execution of $DFS(u)$. Thus u is a proper ancestor of v in the depth-first forest, and $u.pre < v.pre < v.post < u.post$.
  - If $DFS(u)$ calls $DFS(v)$ directly then $u \to v$ is a tree edge
  - Otherwise, $u \to v$ is called a forward edge.
- If $DFS(v)$ has already been called and is still active when $DFS(u)$ begins then v is already on the recursion stack which implies $v.pre < u.pre < u.post < v.post$. G must contain a directed path from v to u. $u \to v$ is then a back edge
- If $DFS(v)$ is finished when $DFS(u)$ begins, we immediately have $v.post < u.pre$. $u \to v$ is then a cross edge

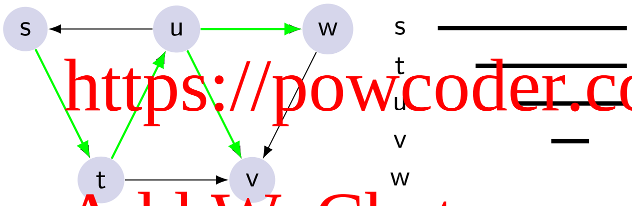


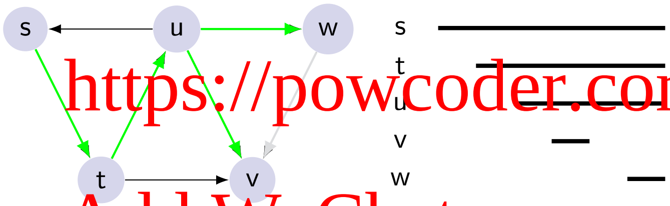
s —
t
u
v
w

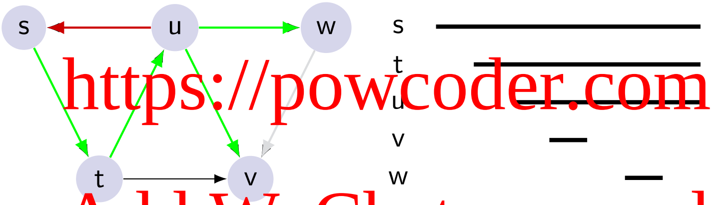

s
u
w
t
v
s
t
u
v
w

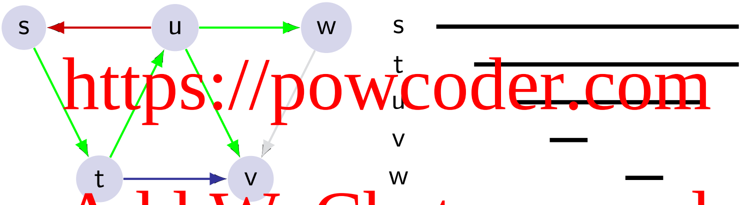

s
u
w
t
v
s
t
u
v
w

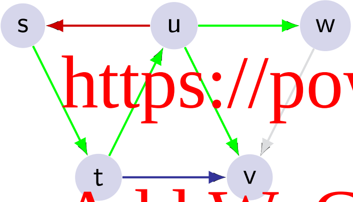

s
u
w
t
v
s
t
u
v
w

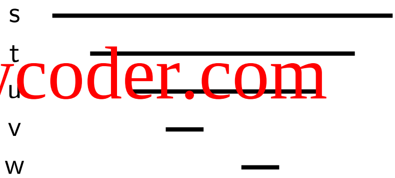

s
u
w
t
v
s
t
u
v
w


s
u
w
t
v
s
t
u
v
w


s
u
w
t
v
s
t
u
v
w


s
u
w
t
v
s
t
u
v
w


s
u
w
t
v


s
t
u
v
w

If $u.post < v.post$ for any edge $u \to v$, the graph contains a directed path from v to u, and therefore contains a directed cycle through the edge $u \to v$.

If $u.post < v.post$ for any edge $u \to v$, the graph contains a directed path from v to u, and therefore contains a directed cycle through the edge $u \to v$.

Thus, we can determine whether a given directed graph G is a dag in $O(V + E)$ time by computing $v.post$ at every vertex v.

A topological ordering of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$.

A topological ordering of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \rightarrow v$.

Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right.

A topological ordering of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \rightarrow v$.

Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right.

A topological ordering is impossible if the graph G has a directed cycle

A topological ordering of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$.

Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right.

A topological ordering is impossible if the graph G has a directed cycle

If $u.post < v.post$ for any edge $u \to v$ then G contains a directed cycle (through $u \to v$)

Topological Sort

A topological ordering of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \to v$.

Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right.

A topological ordering is impossible if the graph G has a directed cycle

If $u.post < v.post$ for any edge $u \to v$ then G contains a directed cycle (through $u \to v$)

Equivalently, if G is acyclic then $u.post > v.post$ for every edge $u \to v$.

A topological ordering of a directed graph G is a total order $\prec$ on the vertices such that $u \prec v$ for every edge $u \rightarrow v$.

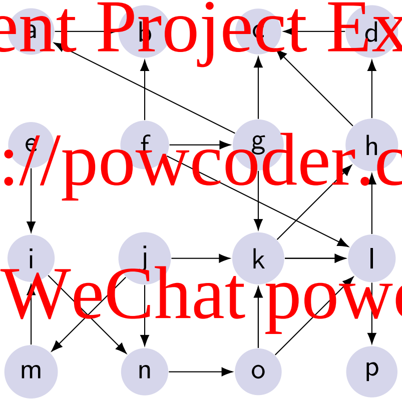
Less formally, a topological ordering arranges the vertices along a horizontal line so that all edges point from left to right.

A topological ordering is impossible if the graph G has a directed cycle

If $u.post < v.post$ for any edge $u \rightarrow v$ then G contains a directed cycle (through $u \rightarrow v$)

Equivalently, if G is acyclic then $u.post > v.post$ for every edge $u \rightarrow v$.

It follows that the vertex order given by reversing the $v.post$ values is a topological ordering of G


a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p