Algorithms

Graphs, breadth- and depth-first search

Emanuele Rodolà
rodola@di.uniroma1.it

SAPIENZA
UNIVERSITÀ DI ROMA

2nd semester a.y. 2019/2020 · April 02, 2020

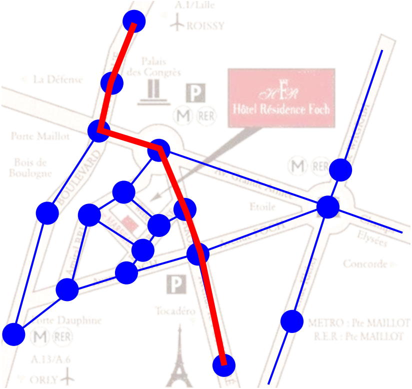
Python

For the coding exercises given in the previous lecture, you should make use of classes in Python.

See the description and examples here:

`https://docs.python.org/3/tutorial/classes.html`

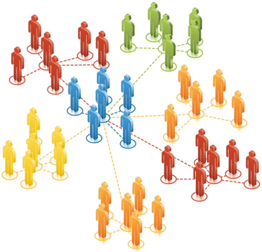
Graphs



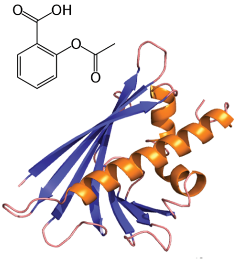
A graph $G = (V, E)$ is made of nodes V and edges E.

Graphs are used pervasively in data sciences.

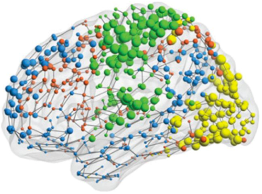
Graphs



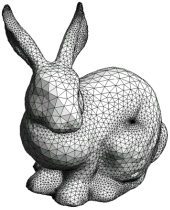
Social networks



Molecules

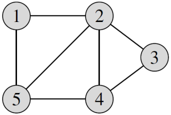


Functional networks



3D shapes

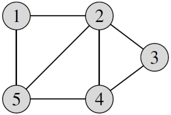
Representation



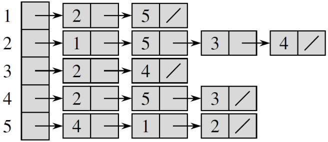
(a)

undirected graph

Representation



(a)

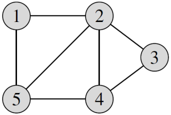


(b)

undirected graph

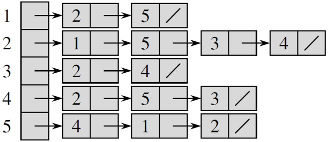
adjacency list

Representation



(a)

undirected graph



(b)

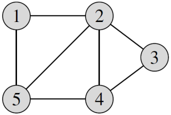
adjacency list

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

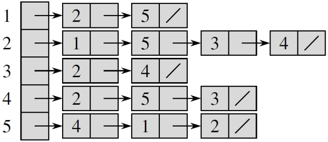
adjacency matrix

Representation



(a)

undirected graph



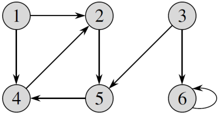
(b)

adjacency list

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

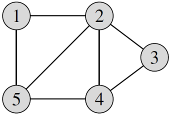
adjacency matrix



(a)

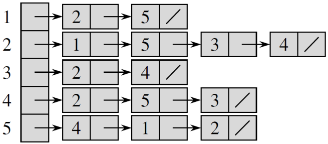
directed graph

Representation



(a)

undirected graph



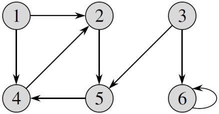
(b)

adjacency list

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

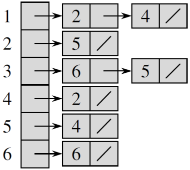
(c)

adjacency matrix



(a)

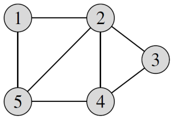
directed graph



(b)

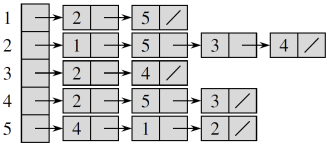
adjacency list

Representation



(a)

undirected graph



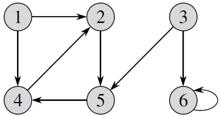
(b)

adjacency list

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

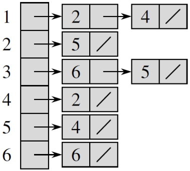
(c)

adjacency matrix



(a)

directed graph



(b)

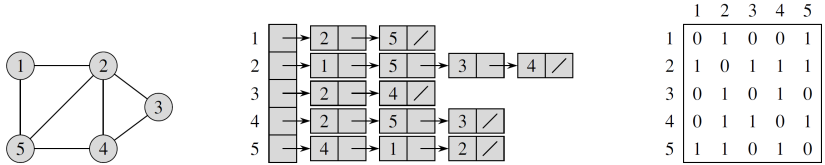
adjacency list

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

(c)

adjacency matrix

Representation efficiency



(a) (b) (c)

undirected graph

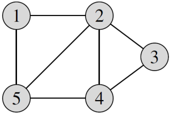
adjacency list

memory: $\Theta(|V| + |E|)$

adjacency matrix

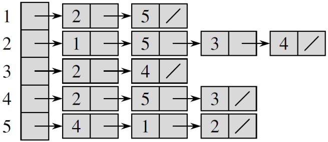
memory: $\Theta(|V|^2)$

Representation efficiency



(a)

undirected graph



(b)

adjacency list

memory: $\Theta(|V| + |E|)$

$$\begin{array}{c|ccccc|} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \\ \hline \end{array}$$

(c)

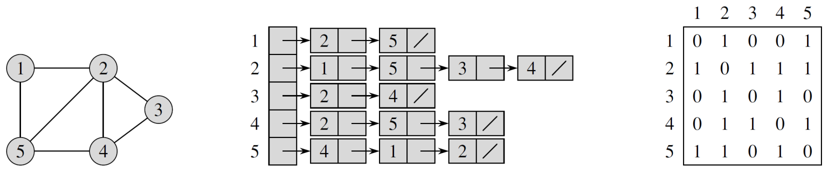
adjacency matrix

memory: $\Theta(|V|^2)$

For undirected graphs, the adjacency matrix is symmetric.

$\Rightarrow$ requires half of the memory.

Representation efficiency



(a) (b) (c)

undirected graph

adjacency list

memory: $\Theta(|V| + |E|)$

adjacency matrix

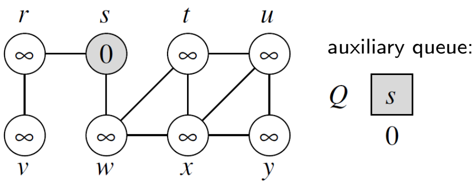
memory: $\Theta(|V|^2)$

For undirected graphs, the adjacency matrix is symmetric.

$\Rightarrow$ requires half of the memory.

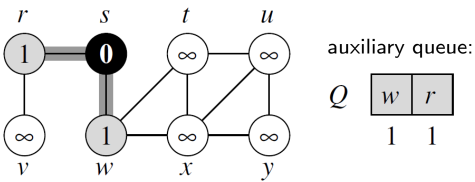
With adjacency matrices, most algorithms are lower bounded as $\Omega(|V|^2)$.

Breadth-first search (BFS)



We start from a source vertex s, and discover all the reachable vertices.

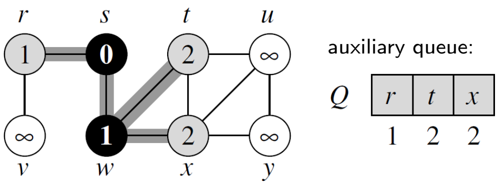
Breadth-first search (BFS)



We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

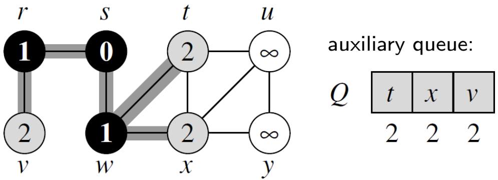
Breadth-first search (BFS)



We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

Breadth-first search (BFS)

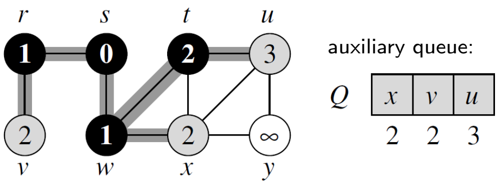


We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k+1$.

Breadth-first search (BFS)

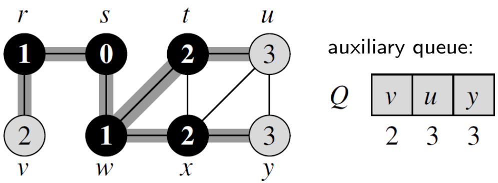


We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k+1$.

Breadth-first search (BFS)

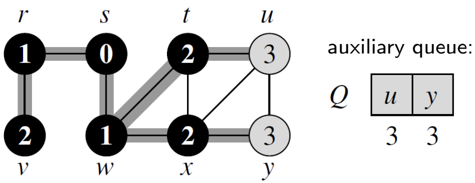


We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k + 1$.

Breadth-first search (BFS)

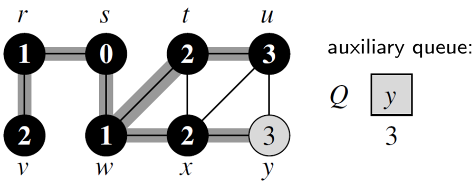


We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k + 1$.

Breadth-first search (BFS)

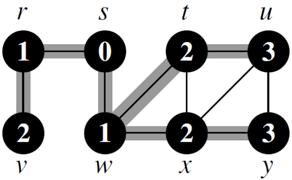


We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k+1$.

Breadth-first search (BFS)



auxiliary queue:

$Q \quad \emptyset$

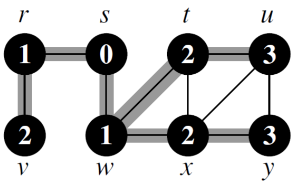
We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s (# edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k + 1$.

A breadth-first tree is obtained as a side-product (shaded edges).

Breadth-first search (BFS)



auxiliary queue:

$Q \quad \emptyset$

We start from a source vertex s, and discover all the reachable vertices.

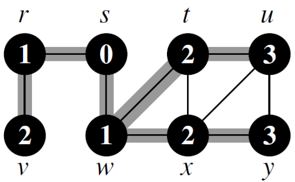
Each discovered vertex has its distance to s ($\#$ edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k+1$.

A breadth-first tree is obtained as a side-product (shaded edges).

- The tree changes if we change the source s (which is the root).

Breadth-first search (BFS)



auxiliary queue:

$Q \quad \emptyset$

We start from a source vertex s, and discover all the reachable vertices.

Each discovered vertex has its distance to s ($\#$ edges) computed.

Breadth-first: nodes at distance k explored before those at distance $k + 1$.

A breadth-first tree is obtained as a side-product (shaded edges).

- The tree changes if we change the source s (which is the root).
- The tree changes if we explore vertex x before vertex t.

Breadth-first search (BFS)

```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
```

* initialize all distances $d = \infty$
* the parent π of every vertex is NIL; parents constitute the tree
* WHITE vertices are undiscovered

Breadth-first search (BFS)

BFS(G, s)

for each vertex $u \in V[G] - \{s\}$
    do $color[u] \leftarrow$ WHITE
        $d[u] \leftarrow \infty$
        $\pi[u] \leftarrow$ NIL
$color[s] \leftarrow$ GRAY
$d[s] \leftarrow 0$
$\pi[s] \leftarrow$ NIL
$Q \leftarrow \emptyset$
ENQUEUE(Q, s)

$*$ GRAY vertices are discovered, but not yet explored

Breadth-first search (BFS)

```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
       do color[u] ← WHITE
          d[u] ← ∞
          π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
       do u ← DEQUEUE(Q)
          for each v ∈ Adj[u]
              do if color[v] = WHITE
                    then color[v] ← GRAY
                         d[v] ← d[u] + 1
                         π[v] ← u
                         ENQUEUE(Q, v)
          color[u] ← BLACK
```

Breadth-first search (BFS)

```
BFS(G, s)
Θ(|V|)   for each vertex u ∈ V[G] − {s}
             do color[u] ← WHITE
                d[u] ← ∞
                π[u] ← NIL
Θ(1)     color[s] ← GRAY
         d[s] ← 0
         π[s] ← NIL
         Q ← ∅
         ENQUEUE(Q, s)
         while Q ≠ ∅
             do u ← DEQUEUE(Q)
                for each v ∈ Adj[u]
                    do if color[v] = WHITE
                          then color[v] ← GRAY
                               d[v] ← d[u] + 1
                               π[v] ← u
                               ENQUEUE(Q, v)
                color[u] ← BLACK
```

Breadth-first search (BFS)

```
        BFS(G, s)
        for each vertex u ∈ V[G] − {s}
            do color[u] ← WHITE
Θ(|V|)         d[u] ← ∞
               π[u] ← NIL
        color[s] ← GRAY
        d[s] ← 0
Θ(1)    π[s] ← NIL
        Q ← ∅
        ENQUEUE(Q, s)
O(|V|) → while Q ≠ ∅
            do u ← DEQUEUE(Q)
               for each v ∈ Adj[u]
                   do if color[v] = WHITE
                         then color[v] ← GRAY
                              d[v] ← d[u] + 1
                              π[v] ← u
                              ENQUEUE(Q, v)
               color[u] ← BLACK
```

Breadth-first search (BFS)

```
              BFS(G, s)
                  for each vertex u ∈ V[G] − {s}
                        do color[u] ← WHITE
Θ(|V|)                     d[u] ← ∞
                           π[u] ← NIL
                  color[s] ← GRAY
                  d[s] ← 0
Θ(1)              π[s] ← NIL
                  Q ← ∅
                  ENQUEUE(Q, s)
O(|V|) →          while Q ≠ ∅
                        do u ← DEQUEUE(Q)
O(|E|) →                   for each v ∈ Adj[u]
                                do if color[v] = WHITE
                                      then color[v] ← GRAY
                                           d[v] ← d[u] + 1
                                           π[v] ← u
                                           ENQUEUE(Q, v)
                           color[u] ← BLACK
```

Breadth-first search (BFS)

$O(|V| + |E|)$

```
BFS(G, s)
for each vertex u ∈ V[G] − {s}
    do color[u] ← WHITE
       d[u] ← ∞
       π[u] ← NIL
color[s] ← GRAY
d[s] ← 0
π[s] ← NIL
Q ← ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    do u ← DEQUEUE(Q)
       for each v ∈ Adj[u]
           do if color[v] = WHITE
                 then color[v] ← GRAY
                      d[v] ← d[u] + 1
                      π[v] ← u
                      ENQUEUE(Q, v)
       color[u] ← BLACK
```

Exploring a path recursively

Assume a breadth-first tree has already been constructed.

```
PRINT-PATH(G, s, v)
if v = s
    then print s
    else  if π[v] = NIL
              then print "no path from" s "to" v "exists"
              else  PRINT-PATH(G, s, π[v])
                    print v
```

Exploring a path recursively

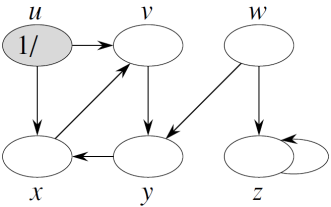
Assume a breadth-first tree has already been constructed.

```
PRINT-PATH(G, s, v)
1  if v = s
2      then print s
3      else  if π[v] = NIL
4               then print "no path from" s "to" v "exists"
5               else  PRINT-PATH(G, s, π[v])
6                     print v
```

Despite this being recursive, the total cost is $O(|V|)$.

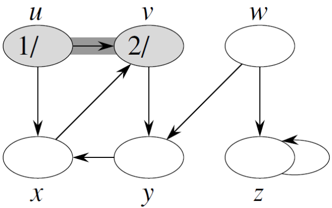
Line (5) is called on a path that is one edge shorter each time.

Depth-first search (DFS)



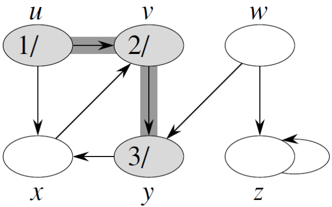
Inside each vertex, we write discovery time / finishing time.

Depth-first search (DFS)



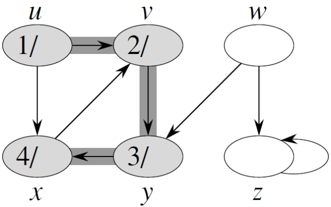
Inside each vertex, we write discovery time / finishing time.

Depth-first search (DFS)



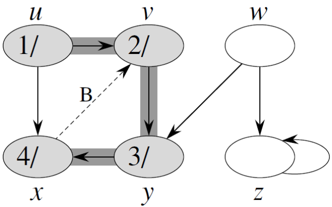
Inside each vertex, we write discovery time / finishing time.

Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

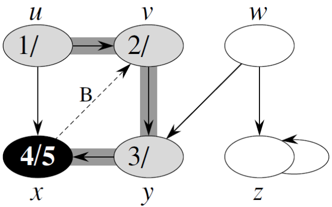
Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

Depth-first search (DFS)

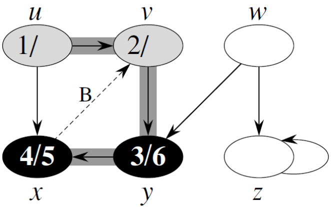


Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

Backtrack to the next available move.

Depth-first search (DFS)

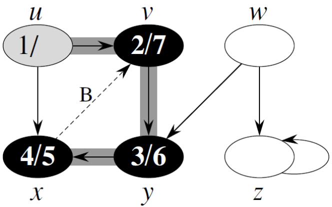


Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

Backtrack to the next available move.

Depth-first search (DFS)

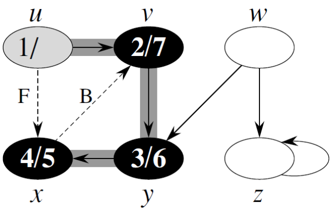


Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

Backtrack to the next available move.

Depth-first search (DFS)



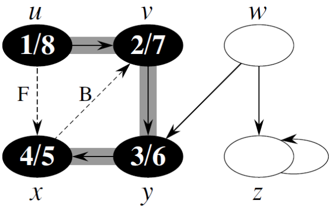
Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

Backtrack to the next available move.

Forward edge to an already discovered vertex.

Depth-first search (DFS)



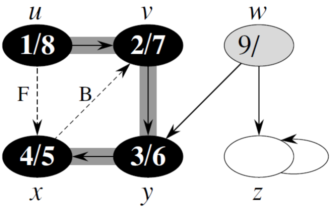
Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

Backtrack to the next available move.

Forward edge to an already discovered vertex.

Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

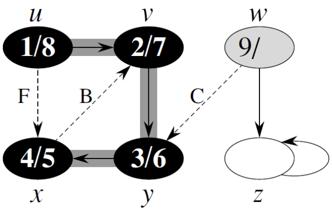
Back edge to an already discovered vertex.

Backtrack to the next available move.

Forward edge to an already discovered vertex.

Select a new source.

Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

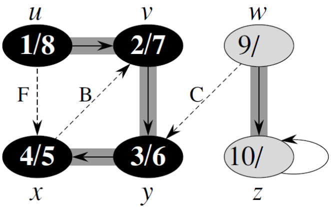
Backtrack to the next available move.

Forward edge to an already discovered vertex.

Select a new source.

Cross edge across two separate trees.

Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

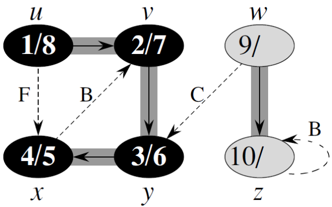
Backtrack to the next available move.

Forward edge to an already discovered vertex.

Select a new source.

Cross edge across two separate trees.

Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

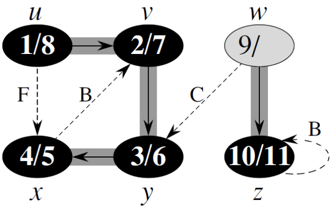
Backtrack to the next available move.

Forward edge to an already discovered vertex.

Select a new source.

Cross edge across two separate trees.

Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

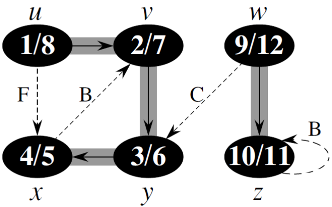
Backtrack to the next available move.

Forward edge to an already discovered vertex.

Select a new source.

Cross edge across two separate trees.

Depth-first search (DFS)



Inside each vertex, we write discovery time / finishing time.

Back edge to an already discovered vertex.

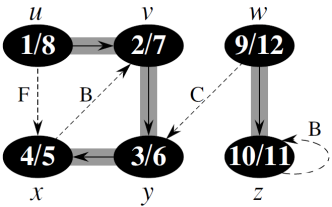
Backtrack to the next available move.

Forward edge to an already discovered vertex.

Select a new source.

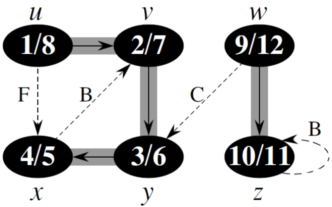
Cross edge across two separate trees.

Depth-first search (DFS)



Unlike BFS, we might have many disjoint trees (shaded edges), thus obtaining a depth-first forest.

Depth-first search (DFS)



Unlike BFS, we might have many disjoint trees (shaded edges), thus obtaining a depth-first forest.

Like BFS, the specific forest changes if the order of exploration changes.

Depth-first search (DFS)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

Depth-first search (DFS)

DFS(G)

for each vertex $u \in V[G]$
    do $color[u] \leftarrow$ WHITE
        $\pi[u] \leftarrow$ NIL
$time \leftarrow 0$
for each vertex $u \in V[G]$
    do if $color[u] =$ WHITE
        then DFS-VISIT(u)   (u will be the root of a new tree in the forest)

Depth-first search (DFS)

```
DFS(G)
for each vertex u ∈ V[G]
    do color[u] ← WHITE
       π[u] ← NIL
time ← 0
for each vertex u ∈ V[G]
    do if color[u] = WHITE
          then DFS-VISIT(u)
```

(u will be the root of a new tree in the forest)

```
DFS-VISIT(u)
color[u] ← GRAY                ▷ White vertex u has just been discovered.
time ← time + 1
d[u] ← time
for each v ∈ Adj[u]            ▷ Explore edge (u, v).
    do if color[v] = WHITE
          then π[v] ← u
               DFS-VISIT(v)
color[u] ← BLACK               ▷ Blacken u; it is finished.
f[u] ← time ← time + 1
```

Depth-first search (DFS)

```
         DFS(G)
         for each vertex u ∈ V[G]
Θ(|V|)       do color[u] ← WHITE
                π[u] ← NIL
         time ← 0
         for each vertex u ∈ V[G]
Θ(|V|)       do if color[u] = WHITE
                  then DFS-VISIT(u)      (u will be the root of a new tree in the forest)

         DFS-VISIT(u)
         color[u] ← GRAY          ▷ White vertex u has just been discovered.
         time ← time + 1
         d[u] ← time
         for each v ∈ Adj[u]      ▷ Explore edge (u, v).
             do if color[v] = WHITE
                  then π[v] ← u
                       DFS-VISIT(v)
         color[u] ← BLACK         ▷ Blacken u; it is finished.
         f[u] ← time ← time + 1
```

Depth-first search (DFS)

```
DFS(G)
            for each vertex u ∈ V[G]
Θ(|V|)          do color[u] ← WHITE
                   π[u] ← NIL
            time ← 0
            for each vertex u ∈ V[G]
Θ(|V|)          do if color[u] = WHITE
                      then DFS-VISIT(u)
```

(u will be the root of a new tree in the forest)

```
DFS-VISIT(u)
            color[u] ← GRAY          ▷ White vertex u has just been discovered.
            time ← time + 1
            d[u] ← time
            for each v ∈ Adj[u]      ▷ Explore edge (u, v).
Θ(|E|)          do if color[v] = WHITE
                      then π[v] ← u
                           DFS-VISIT(v)
            color[u] ← BLACK         ▷ Blacken u; it is finished.
            f[u] ← time ← time + 1
```

Suggested reading

Chapters 22.1, 22.2 (skip the "Shortest Paths" paragraph), and 22.3 (skip the "Properties of depth-first search" paragraph) of:

"Introduction to Algorithms – 2nd Ed.", Cormen et al.