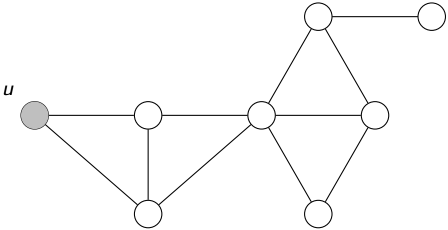
Depth-First Search

Specify or arbitrarily pick a start vertex.

0. visit the start vertex
1. choose one adjacent, unvisited vertex of the previous; visit it
2. choose one adjacent, unvisited vertex of the previous; visit it
3. ...
4. whenever you have no choice, backtrack to the last time you had a choice, choose another one

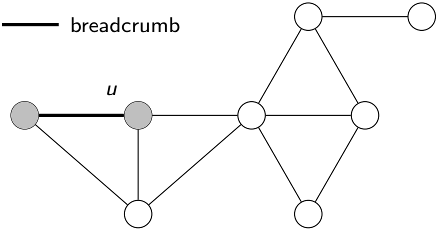
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

choose an adjacent, unvisited vertex to visit

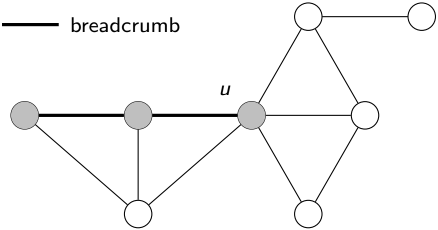
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

choose an adjacent, unvisited vertex to visit

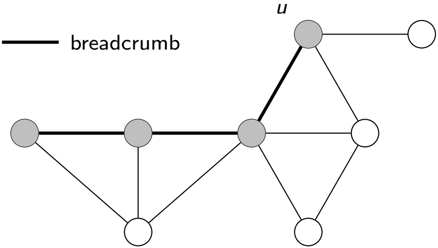
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

choose an adjacent, unvisited vertex to visit

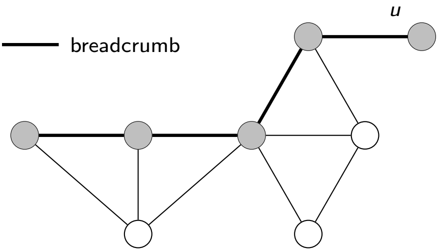
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

choose an adjacent, unvisited vertex to visit

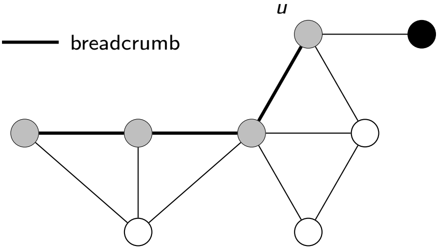
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

no adjacent, unvisited vertex; backtrack

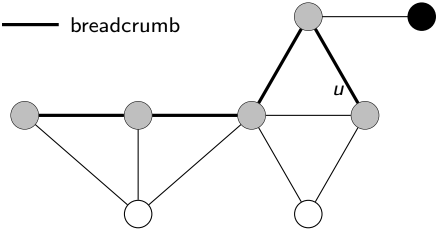
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

choose an adjacent, unvisited vertex to visit

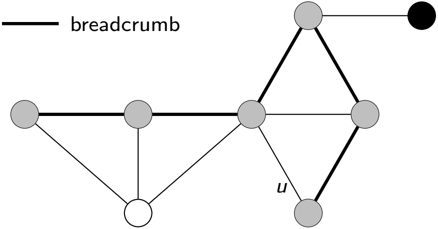
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

choose an adjacent, unvisited vertex to visit

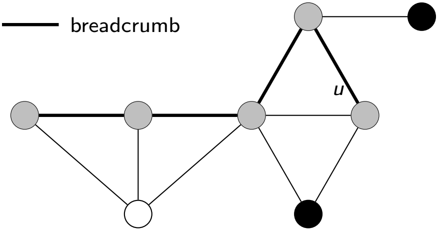
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

no adjacent, unvisited vertex; backtrack

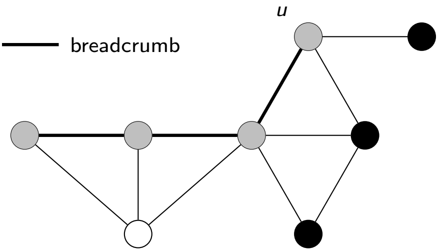
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

no adjacent, unvisited vertex; backtrack

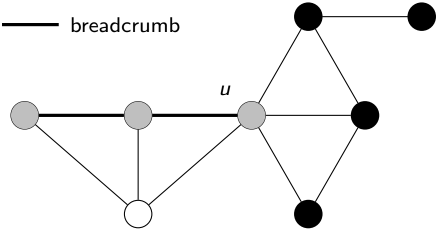
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

no adjacent, unvisited vertex; backtrack

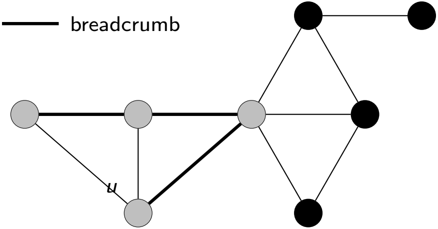
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

choose an adjacent, unvisited vertex to visit

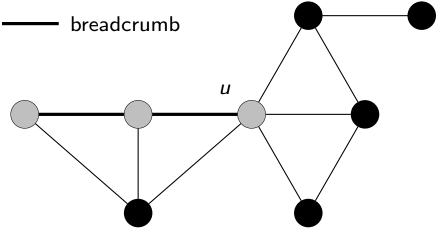
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

no adjacent, unvisited vertex; backtrack

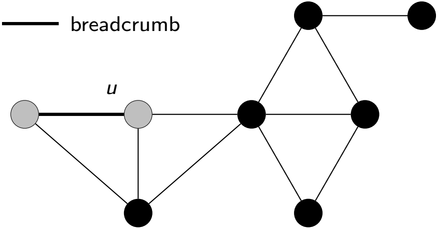
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

no adjacent, unvisited vertex; backtrack

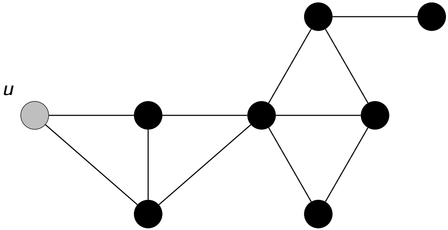
Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

no adjacent, unvisited vertex; backtrack

Depth-First Search



(White: unvisited. Gray: in progress. Black: done.)

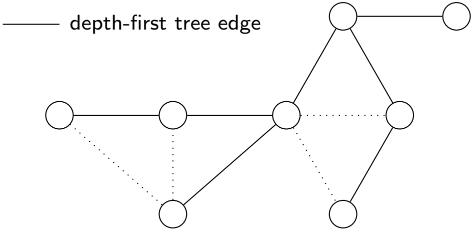
no adjacent, unvisited vertex; nothing to backtrack, the end.

Depth-First Search

```
0. mark all vertices white
1. time := 0
2. start := pick a vertex
3. DFS-visit(start)

4. DFS-visit(u):
5.   discovery-time(u) := ++time
6.   mark u gray
7.   for each v in u's adjacency list:
8.     if v is white:
         // edge {u,v} is a depth-first tree edge
         // predecessor(v) = u
9.       DFS-visit(v)
10.  mark u black
11.  finish-time(u) := ++time
```

Depth-First Search



DFS finds:

- whether a vertex is reachable from *start*
- a tree consisting of the reachable vertices from *start*
- the component containing *start*
- (with a small modification) whether a cycle exists

Depth-First Search

DFS running time:

1. we visit each vertex once:
   - only visit white vertices; mark gray when visit
2. we consider each edge twice:
   - each edge incident on 2 vertices
3. we find each vertex's adjacency list once:
   - right after mark gray (line 7)
4. check v's colour $deg(v)$ times:
   - once from every node adjacent to it (line 8)

Assume $\Theta(1)$ time for

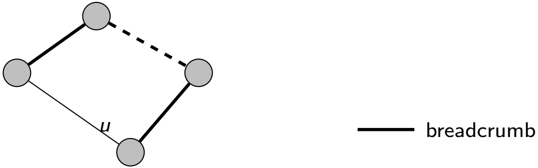
- marking/checking a vertex's colour
- finding a vertex's adjacency list

Then DFS total time: $\Theta(|V| + |E|)$.

Exercise: What if the assumption doesn't hold?

cycle detection

During DFS, if something like this happens:



When u has an edge to a gray vertex that is not its predecessor.

Then it must be because. . . you have found a cycle.

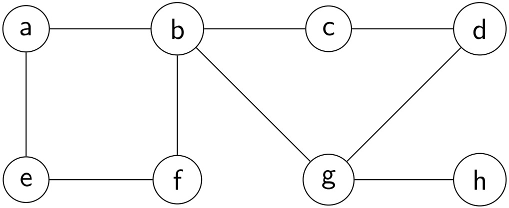
Conversely, if this never happens, there is no cycle. (Harder to prove.)

cycle detection

```
mark all vertices white
for each vertex s:
  if s is white:
    if has-cycle(s): return True
return False

has-cycle(u):
  mark u gray
  for each v in u's adjacency list:
    if v is white:
      predecessor(v) = u
      if has-cycle(v): return True
    elif v is gray and v is not predecessor(u):
      return True
  mark u black
  return False
```

cycle detection: example



directed graph

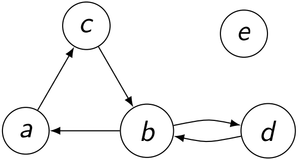
A directed graph G is a pair (V, E) of:

- V — a set of vertices
- E — a set of edges, where an edge is a pair of vertices (usually, we disallow edges from a vertex to itself)

Each edge specifies one direction.
(a, b) lets you go from a to b, if present.
(b, a) lets you go from b to a, if present.

Many definitions need small modifications.

storing a directed graph: adjacency lists



	adjacency list
a	c
b	a, d
c	b
d	b
e	

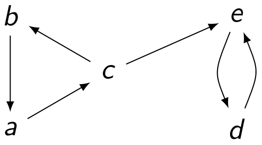
"c is adjacent to a", but not "a is adjacent to c".

directed graph: modified definitions

- out-degree: how many edges go out of a vertex
  in-degree: how many edges go into a vertex
  degree: out-degree + in-degree
- path, reachable: must comply with edge directions
  path $\langle v_0, \ldots, v_k \rangle$ requires $(v_0, v_1) \in E$, ..., $(v_{k-1}, v_k) \in E$
- cycle: must comply with edge directions
  cycle $\langle v_0, \ldots, v_{k-1}, v_0 \rangle$ requires $(v_0, v_1) \in E$, ..., $(v_{k-1}, v_0) \in E$
  Note: $\langle b, d, b \rangle$ is a simple cycle this time: (b, d) and (d, b) are two different edges.
- BFS, DFS: no change needed because:
  - "for each v in u's adjacency list" already complies with edge direction (u, v)

directed graph: BFS/DFS

BFS/DFS depend on the choice of the start vertex:

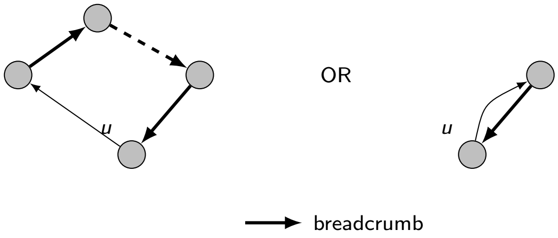


- will visit every vertex if start at: a, b, c
- will not visit every vertex if start at: d, e

(Unlike in undirected graphs.)

directed graph: cycle detection

If something like this happens:



- if we encounter an edge to a gray vertex, then
- we found a cycle, even if the vertex is u's predecessor

Different from undirected graphs.

directed graph: cycle detection

```
mark all vertices white
for each vertex s:
  if s is white:
    if has-cycle(s): return True
return False

has-cycle(u):
  mark u gray
  for each v in u's adjacency list:
    if v is white:
     if has-cycle(v): return True
    elif v is gray:
      return True
  mark u black
  return False
```