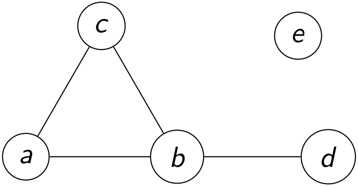
CSCB63 – Design and Analysis of Data Structures

Akshay Arun Bapat[1]

[1] based on notes by Anya Tafliovich, Anna Bretscher and Albert Lai

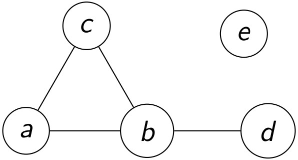
Introduction

- cities and highways between them
- computers and network cables between them
- people and relationships
- in a board game: a state and legal moves to other states



a graph

Undirected graph



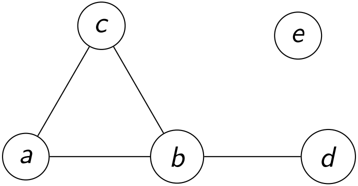
An <u>undirected graph</u> is a pair (V, E) of:

- V: a set of vertices (above: $\{a, b, c, d, e\}$)
- E: a set of edges, where an edge is a pair of vertices
  (above: $\{\{a, c\}, \{a, b\}, \{b, c\}, \{b, d\}\}$)
  (usually, no edge from a vertex to itself)
  undirected graph — no direction specified, bidirectional

Graph terminology: incident, endpoint, degree

Edge incident on vertex, vertex is an endpoint of edge: e.g.,
$\{a, c\}$ is incident on a; a is an endpoint of $\{a, c\}$
$\{a, c\}$ is incident on c; c is an endpoint of $\{a, c\}$
$\{a, c\}$ is not incident on b; b is not an endpoint of $\{a, c\}$

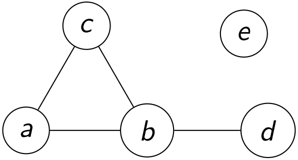
Degree of vertex: how many edges are incident on it.



vertex	a	b	c	d	e
degree	2	3	2	1	0

Graph terminology: adjacent

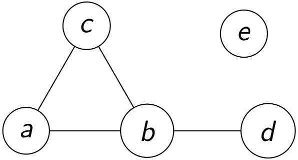
Two vertices are adjacent iff there is an edge between them.



	a	b	c	d	e
a		$\surd$	$\surd$		
b	$\surd$		$\surd$	$\surd$	
c	$\surd$	$\surd$			
d		$\surd$			
e					

	is adjacent to
a	b, c
b	a, c, d
c	a, b
d	b
e	

Storing a graph: adjacency matrix



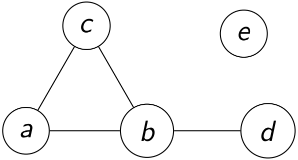
	a	b	c	d	e
a		$\surd$	$\surd$		
b	$\surd$		$\surd$	$\surd$	
c	$\surd$	$\surd$			
d		$\surd$			
e					

Adjacency matrix = store this in a 2D array

Let $n = |V|$ and $m = |E|$. Then in terms of n and m:

- space: $\Theta(n^2)$
- "who are adjacent to v?" time: $\Theta(n)$
- "are v and w adjacent?" time: $\Theta(1)$

Storing a graph: adjacency lists



	is adjacent to
a	b, c
b	a, c, d
c	a, b
d	b
e	

Adjacency lists = store this in a 1D array or dictionaryUse a list or a set for each entry on the right.

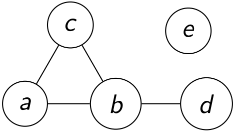
Let $n = |V|$ and $m = |E|$. Then in terms of n, m, and $degree(v)$:

- space: $\Theta(n + m)$
- "who are adjacent to v?" time: $\Theta(deg(v))$
- "are v and w adjacent?" time: $\Theta(deg(v))$
- optimal for graph searches

Graph terminology: (simple) path, reachable

A (simple) path is a non-empty sequence of vertices in which

- consecutive vertices are adjacent
- vertices are distinct



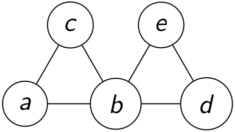
$\langle d \rangle$ is a path, length 0.
$\langle d, b, c \rangle$ is a path, length 2.
$\langle d, b, c, b \rangle$ is a not a (simple) path.
$\langle d, a, b \rangle$ is not a path.

v is reachable from u iff there is a path from u to v.

Graph terminology: (simple) cycle

A (simple) cycle is a non-empty sequence of vertices in which

- consecutive vertices are adjacent
- first vertex = last vertex
- vertices are distinct except first=last; edges used are distinct
- $\langle v \rangle$ is not a cycle



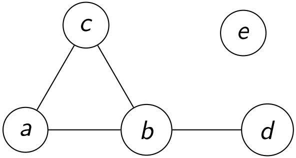
$\langle b, c, a, b \rangle$ is a simple cycle, length 3. ($\langle b, c, a \rangle$ in some books.)
$\langle b, c, a, b, d, e, b \rangle$ is not a (simple) cycle: uses b in the middle
$\langle b, d, b \rangle$ is not a cycle: it uses $\{b, d\}$ twice

Graph terminology: (dis)connected, component

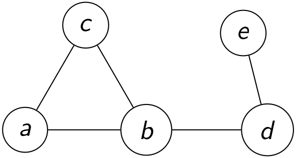
A graph is connected iff between every two distinct vertices there is a path.

A graph is disconnected iff it is not connected.

Disconnected:



Connected:



Component: maximal subset of vertices reachable from each other. (Sometimes also include their edges.)

E.g., the graph on the left has two components: $\{a, b, c, d\}$, $\{e\}$

Tree: definition and results

A tree is a graph that is connected and has no cycles.

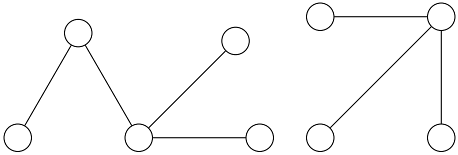
Equivalently:

- between every two vertices, a unique simple path
- connected, but disconnected if any edge removed
- connected, and $|E| = |V| - 1$
- no cycles, but has a cycle if any edge added
- no cycles, and $|E| = |V| - 1$

Exercise: convince yourself that these are equivalent!

Graph terminology: forest

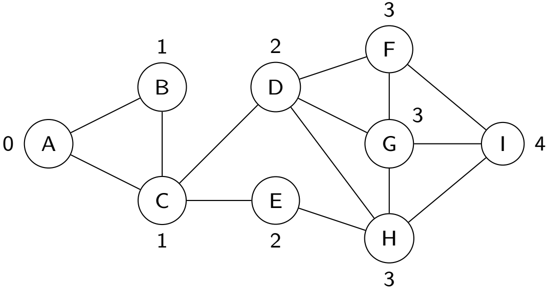
A forest is a collection of trees (may be disconnected). A forest has no cycles.



Breadth-First Search

Specify or arbitrarily pick a start vertex.

0. visit the start vertex
1. visit vertices 1 edge away from the above
2. visit unvisited vertices 1 edge away from the above
3. visit unvisited vertices 1 edge away from the above
4. . . .

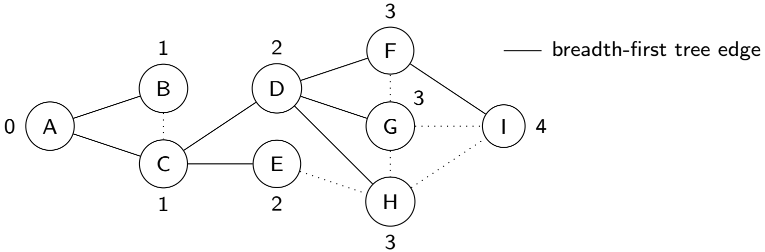


Breadth-First Search

```
0. start := pick a vertex
1. queue := new Queue()
2. queue.enqueue(start)
3. mark start as seen
   // distance(start) = 0

4. while not queue.is_empty():
5.   u := queue.dequeue()
6.   for each v in u's adjacency list:
7.     if v is not seen:
8.       queue.enqueue(v)
9.       mark v as seen
         // edge {u,v} is a "breadth-first tree edge"
         // u is v's "predecessor"
         // distance(v) = distance(u) + 1
```

Breadth-First Search



BFS finds:

- whether a vertex is reachable from *start*
- if yes, a shortest path and distance
- a tree consisting of the reachable vertices from *start*
- the component containing *start*

Shortest paths and the tree are non-unique: sensitive to orders of vertices in adjacency lists.

Unvisited Vertices

What if some vertex is not visited?
Depends on your purpose.

- To determine which vertices are reachable from the start: now you have the answer.
- To check the whole graph: another round of BFS from an unvisited vertex.
  E.g., to find all components.

Breadth-First Search

BFS running time:

1. we enqueue and dequeue each vertex once:
   - only enqueue unseen vertices; mark as seen right after enqueue
2. we consider each edge twice:
   - each edge incident on 2 vertices
3. we find each vertex's adjacency list once:
   - right after dequeue (line 6)
4. check v's "seen" status $deg(v)$ times:
   - once from every node adjacent to it (line 7)

Assume $\Theta(1)$ time for

- marking/checking a vertex's "seen" status
- finding a vertex's adjacency list

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

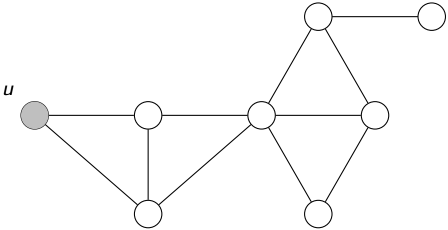
Exercise: What if the assumption doesn't hold?

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

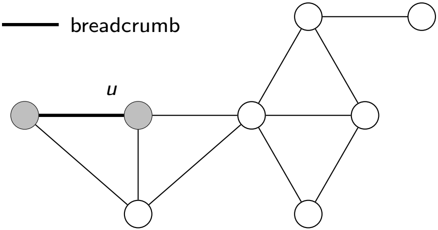
Depth-First Search



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

choose an adjacent, unvisited vertex to visit

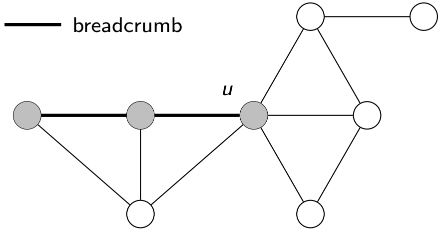
Depth-First Search



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

choose an adjacent, unvisited vertex to visit

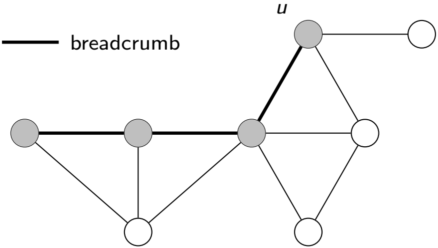
Depth-First Search



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

choose an adjacent, unvisited vertex to visit

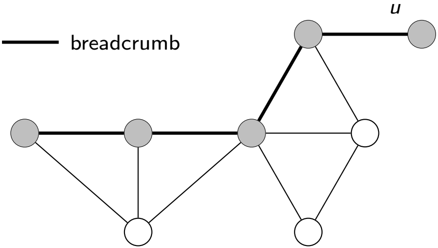
Depth-First Search



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

choose an adjacent, unvisited vertex to visit

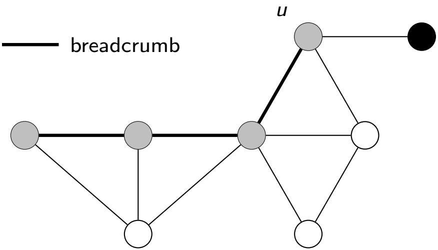
Depth-First Search



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

no adjacent, unvisited vertex; backtrack

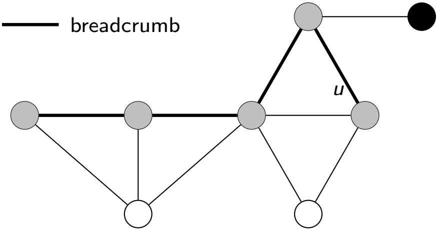
Depth-First Search



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

choose an adjacent, unvisited vertex to visit

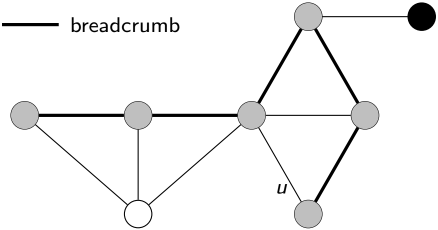
Depth-First Search



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

choose an adjacent, unvisited vertex to visit

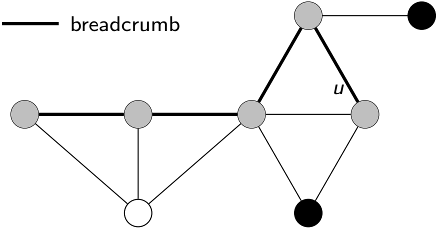
Depth-First Search



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

no adjacent, unvisited vertex; backtrack

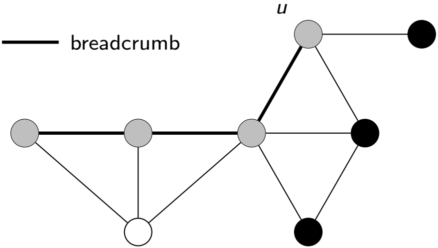
Depth-First Search



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

no adjacent, unvisited vertex; backtrack

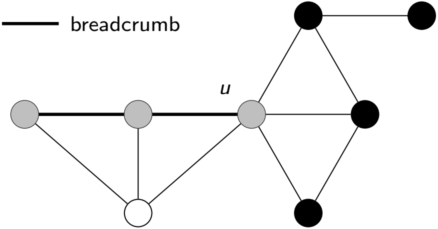
Depth-First Search



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

no adjacent, unvisited vertex; backtrack

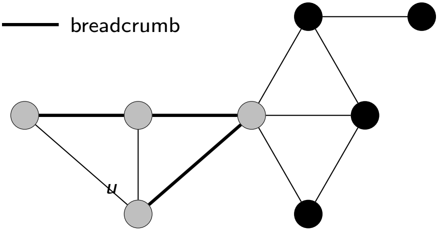
Depth-First Search



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

choose an adjacent, unvisited vertex to visit

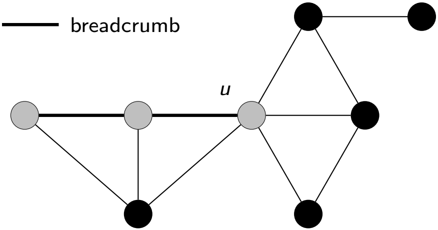
Depth-First Search



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

no adjacent, unvisited vertex; backtrack

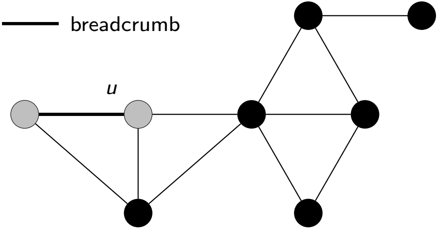
Depth-First Search



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

no adjacent, unvisited vertex; backtrack

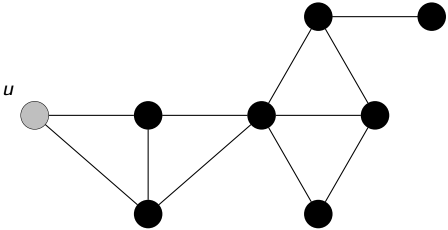
Depth-First Search



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

no adjacent, unvisited vertex; backtrack

Depth-First Search



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

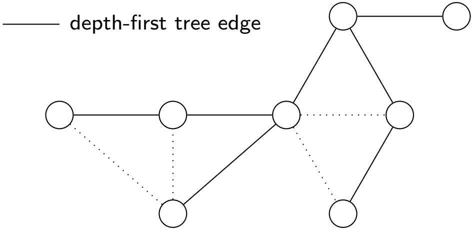
no adjacent, unvisited vertex; nothing to backtrack, the end.

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

Unvisited Vertices

What if some vertex is not visited?
Depends on your purpose.

- To determine which vertices are reachable from the start: now you have the answer.
- To check the whole graph: another round of DFS from an unvisited vertex.
  E.g., to find all components; to find a cycle.

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

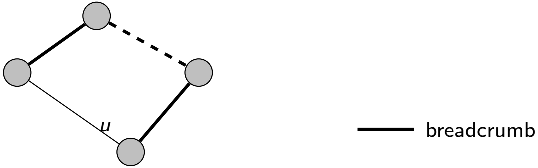
- marking/checking a vertex's colour
- finding a vertex's adjacency list

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.

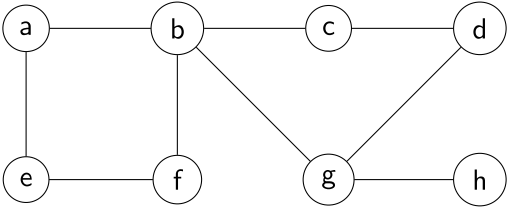
Conversely, if this never happens, there is no cycle. (Harder to prove.)

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

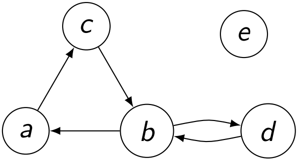
A directed graph G is a pair (V, E) of:

- 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	

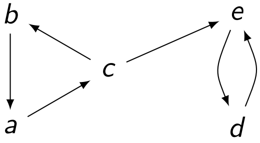
"c is adjacent to a", but not "a is adjacent to c".

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 $(v_0, v_1) \in E$, $\ldots$, $(v_{k-1}, v_k) \in E$
- cycle: must comply with edge directions
  cycle $\langle v_0, \ldots, v_{k-1}, v_0 \rangle$ requires $(v_0, v_1) \in E$, $\ldots$, $(v_{k-1}, v_0) \in E$
  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:

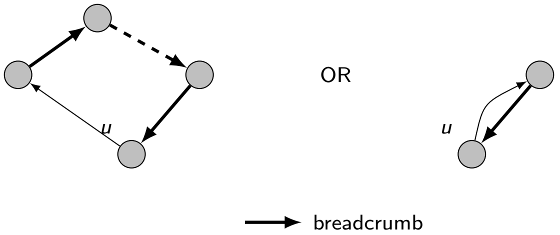


- will visit every vertex if start at: a, b, c
- will not visit every vertex if start at: d, e

(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
```