<u>Graphs</u>

What is a graph? Abstractly, a set of vertices and a set of edges connecting pairs of vertices:

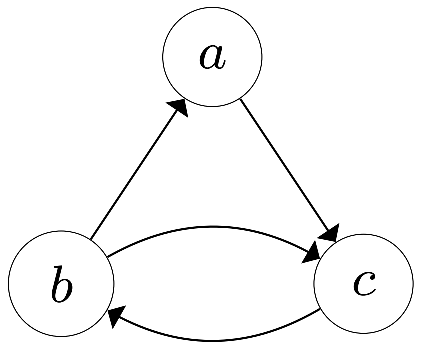
Graph $G = (V, E)$

- V = (finite) set of vertices $(V \neq \emptyset)$

- E = set of edges (vertex pairs) $\subseteq V \times V$
  - ordered pairs $(u, v) \Rightarrow$ <u>directed</u> graph
  - unordered pairs $\{u, v\} \Rightarrow$ <u>undirected</u> graph

Examples

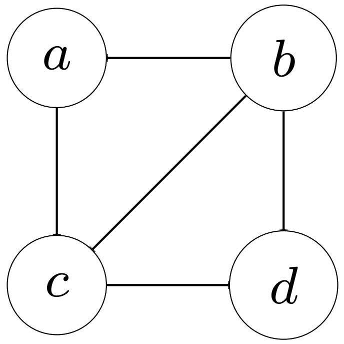
Directed graph



$V = \{a, b, c\}$
$E = \{(a, c), (b, a), (b, c), (c, b)\}$

Note: edge $(u, v) \neq (v, u)$
e.g., $(b, c) \neq (c, b)$

Undirected graph



$V = \{a, b, c, d\}$
$E = \{\{a, b\}, \{a, c\}, \{b, c\}, \{b, d\}, \{c, d\}\}$

Note: edge $(u, v) = (v, u)$
e.g., $(a, b) = (b, a)$

Note: often write (u, v) for edge even if G is undirected (understand that it means unordered pair, though)

Graph has <u>two</u> parameters describing its size:

$|V| =$ # of vertices

$|E| =$ # of edges $(0 \le |E| \le |V|^2)$

Running times may be described as function of both

Graph representation

Adjacency matrix representation

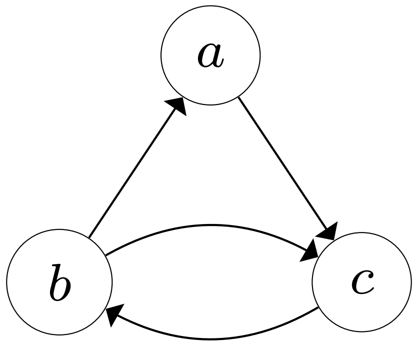
Assume $V = \{1, 2, \ldots, |V|\}$

Let $A = (a_{ij}) = |V| \times |V|$ matrix where

$$a_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

(Store as, e.g., array of arrays)

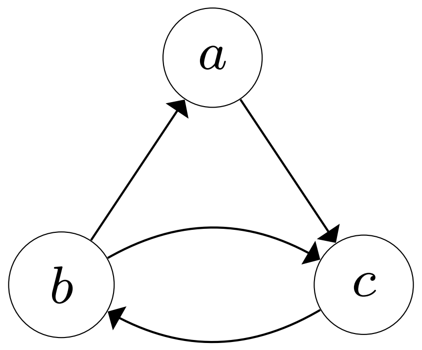
Adjacency matrix example



$$A = \begin{array}{c|ccc} & a & b & c \\ \hline a & 0 & 0 & 1 \\ b & 1 & 0 & 1 \\ c & 0 & 1 & 0 \end{array}$$

- Space: $\Theta(V^2)$
- Good for dense graphs (where $|E| \approx |V|^2$)
- $O(V^2)$ time to find all the edges

Adjacency lists: For each vertex u, keep a list $Adj[u]$ of vertices adjacent to u



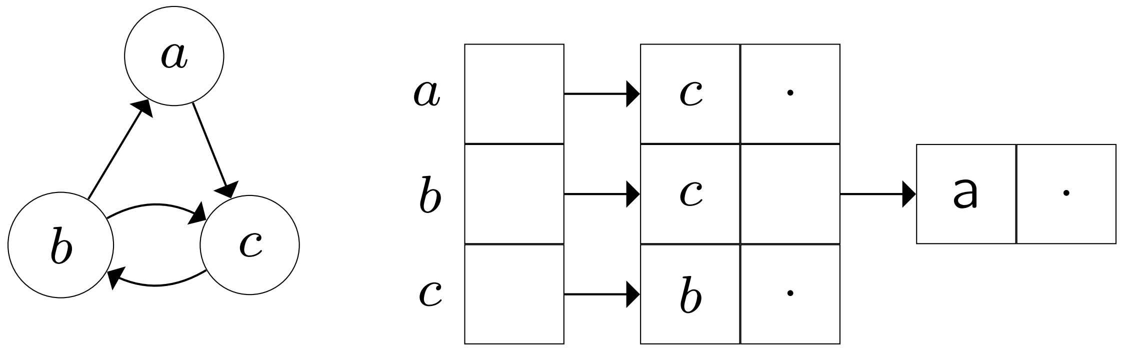
$$\begin{aligned} Adj[a] &= \{c\} \\ Adj[b] &= \{c, a\} \\ Adj[c] &= \{b\} \end{aligned}$$

Variation: Could also keep second list of edges coming *into* vertex

Adjacency list representation

Array A of size $|V|$, of $|V|$ linked lists

This array is indexed by a vertex; each element in array is a pointer to a linked list



- Space: $\Theta(V + E)$
- Good for sparse graphs (where $|E| \ll |V|^2$)
- $\Theta(V + E)$ time to find all the edges

Note:

- $Adj[u]$ lists u's neighbors in no particular order
- Have random access to vertices in the array but do not have random access to members of an adjacency list
- To find out if $(u, v) \in E$ need to scan all of $Adj[u]$ in worst case
- Can take $O(V)$ time

Graph Search:

Given: Graph $G = (V, E)$ (directed or undirected),
specific vertex $s \in V$

"Explore" a graph: ideas

1. Find a path from vertex s to another vertex t
2. Visit every vertex reachable from s
3. Visit every vertex and every edge of G

<u>Explore</u>: Visit every vertex reachable from s:

1. s is reachable from s
2. if u is reachable from s and $v \in Adj[u]$, then v is reachable from s
   (go to u, then follow edge (u, v))
3. only vertices reachable from s are those provably so by (1) & (2)

If u is reachable from s, then there is a path

$$s \to v_1 \to v_2 \to \cdots \to v_k \to u$$

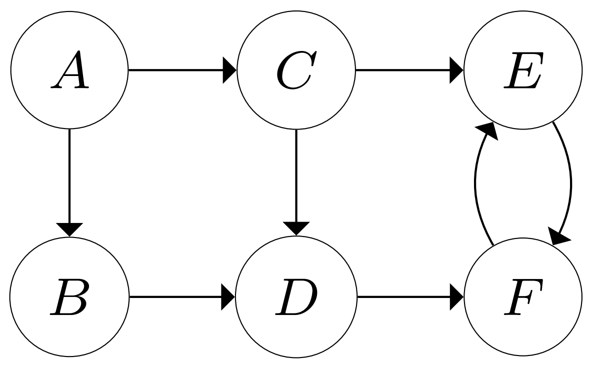
(of length $k + 1$) leading from s to u

We will find such paths ...

Breadth-first search (BFS):

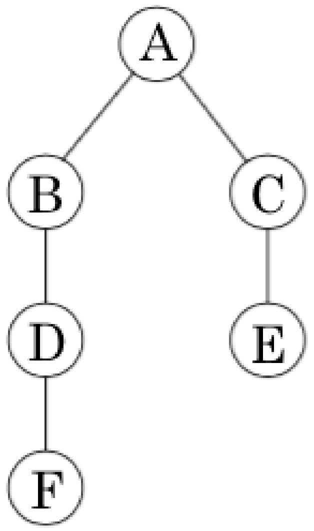
- visit all vertices reachable from given vertex $s \in V$
- $O(V + E)$ time
- look at vertices reachable from s:
  - in 0 edge length, i.e., $\{s\}$
  - then 1 edge length, i.e., $Adj[s]$
  - then 2 edge lengths, and so on
- avoid revisiting vertices: why?

To see how this works, consider this simple graph:



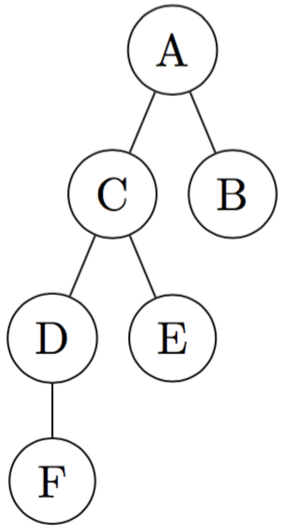
Given: source vertex A; graph G, in Adj list format

BFS tree:



Level of a vertex in the tree corresponds to the shortest distance from that vertex to source s

Would we get the same tree if processed $Adj[A]$ in $C, B, ...$ order?



BFS tree not unique

What is invariant? Distances from source s

Information that has to be kept track of:

1. The “time” when a vertex has been discovered
2. The “time” when its adjacency list has been scanned
3. Parent of the vertex
4. Distance of vertex from source s

Maintain 3 variables associated with each vertex v:

color[v]: White, Gray, Black

$d[v]$: distance of v from source vertex s

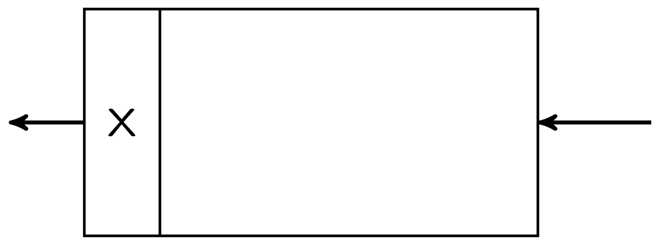
$\pi[v]$: parent of v: vertex w from which discover v

So if discovered A while scanning $Adj[S]$, then S is parent of A

Maintain a list of Gray vertices in the order of discovery:

- When discover a vertex, add it to end of the list
- When finish a vertex, remove it from beginning of the list

This is a FIFO list:



Best implemented as a linked list

Have link to the beginning and a link to the end of the list

To append an item to the list, go to the end and create a new link

To delete a item from the list, instead of removing item, just move link to next item

- To append an item to list is $O(1)$

- To delete an item from list is $O(1)$

The list is called a <u>Queue</u>

First-in First-out Queue

This is where we will put the vertices that are discovered but not finished

Only Gray vertices are in the Queue

The two operations are called Enqueue and Dequeue

- Enqueue(Q, key): append *key*

- Dequeue(Q): remove head of list

Breadth-first search (BFS):

Input: $G = (V, E)$, $s \in V$
G given in adjacency lists format

Output:

$d[v]$ = min # of edges in path from s to v

$\pi[v]$ = parent of v on path from s to v

Time: $O(V + E)$ linear time

```
BFS($G, s$)
    **for** each $u \in V - \{s\}$
        $color[u]$ = WHITE
        $d[u] = \infty$
        $\pi[u]$ = NIL
    $color[s]$ = GRAY
    $d[s] = 0$
    $Q = \emptyset$
    ENQUEUE($Q, s$)
    **while** $Q \neq \emptyset$
        $u$ = DEQUEUE($Q$) // remove $u$ from $Q$
        **for** each $v \in Adj[u]$
            **if** $color[v]$ == WHITE
                $color[v]$ = GRAY
                $d[v] = d[u] + 1$
                $\pi[v] = u$
                Enqueue($Q, v$) // put $v$ onto $Q$
        $color[u]$ = BLACK
```

<u>Running time</u>

- each vertex added to Q at most once
- each adjacency list examined at most once
- $\sum_u |Adj[u]| = |E|$ by definition
- running time is $O(V + E)$ <u>linear time</u>

Why does BFS work?

1st: What will be the color of the vertices at the end of the algorithm?

<u>Not</u> all Black

<u>Claim</u>: After BFS, vertices <u>reachable</u> from s will be Black; all others will be White

Gray vertices will disappear: as long as there is a Gray vertex, Q is not empty, and while Q is not empty, algorithm makes another iteration

Correctness of BFS

Claim: $d[v]$ is correctly computed for every $v \in V$, i.e., $d[v] =$ distance$(s, v) =$ min # of edges in path from s to v

Proof: Assume that some vertex v receives incorrect $d[v]$ value, i.e., $d[v] \neq$ distance(s, v)

Assume v is the first such vertex

Basis: Clearly $v \neq s$: $d[v] = 0$ only for $v = s$, and this is correctly set by the algorithm; all other vertices get $d[v] = \infty$ initially, and if v is reachable from s, $d[v] \leq 1$

Inductive step: Let u be the vertex immediately preceding v on a shortest path from s to v

Then distance$(s, v) =$ distance$(s, u) + 1$ (IH)

Since $d[u]$ is correctly set, $d[u] =$ distance(s, u), and distance$(s, v) = d[u] + 1$

So

$$d[v] > \text{distance}(s, v) = d[u] + 1 \qquad (*)$$

Now consider when algorithm dequeues vertex u from Q (line 10)

At this point vertex v is either White, Gray, or Black:

If White, then line 14 sets $d[v]$ to $d[u]+1$, contradicting $(*)$

If Black, then v was already removed from Q, and $d[v] < d[u]$, contradicting $(*)$

If Gray, then v was colored Gray on dequeuing some vertex w, which was removed from Q earlier than u and $d[v] = d[w] + 1$

But then $d[w] \leq d[u]$, and $d[v] = d[w] + 1 \leq d[u] + 1$, again contradicting $(*)$

$\therefore d[v] = \text{distance}(s, v)$ for all $v \in V$

Shortest paths:

BFS finds a shortest path in the graph from s to v (not necessarily the only one)

- for every vertex v, fewest edges to get from s to v is:

  $d[v]$ if v assigned a d-value

  ∞ else (no path)

- parent pointers form shortest-path tree = union of such a shortest path for each v

Print vertices on shortest path from s to v:

```
Print-Path($G, s, v$)
1  if $v == s$
2        print $s$
3  elseif $\pi[v]$ == NIL
4        print "no path from s to v"
5  else Print-Path($G, s, \pi[v]$)
6        print $v$
```

This procedure runs in $O(V)$ time