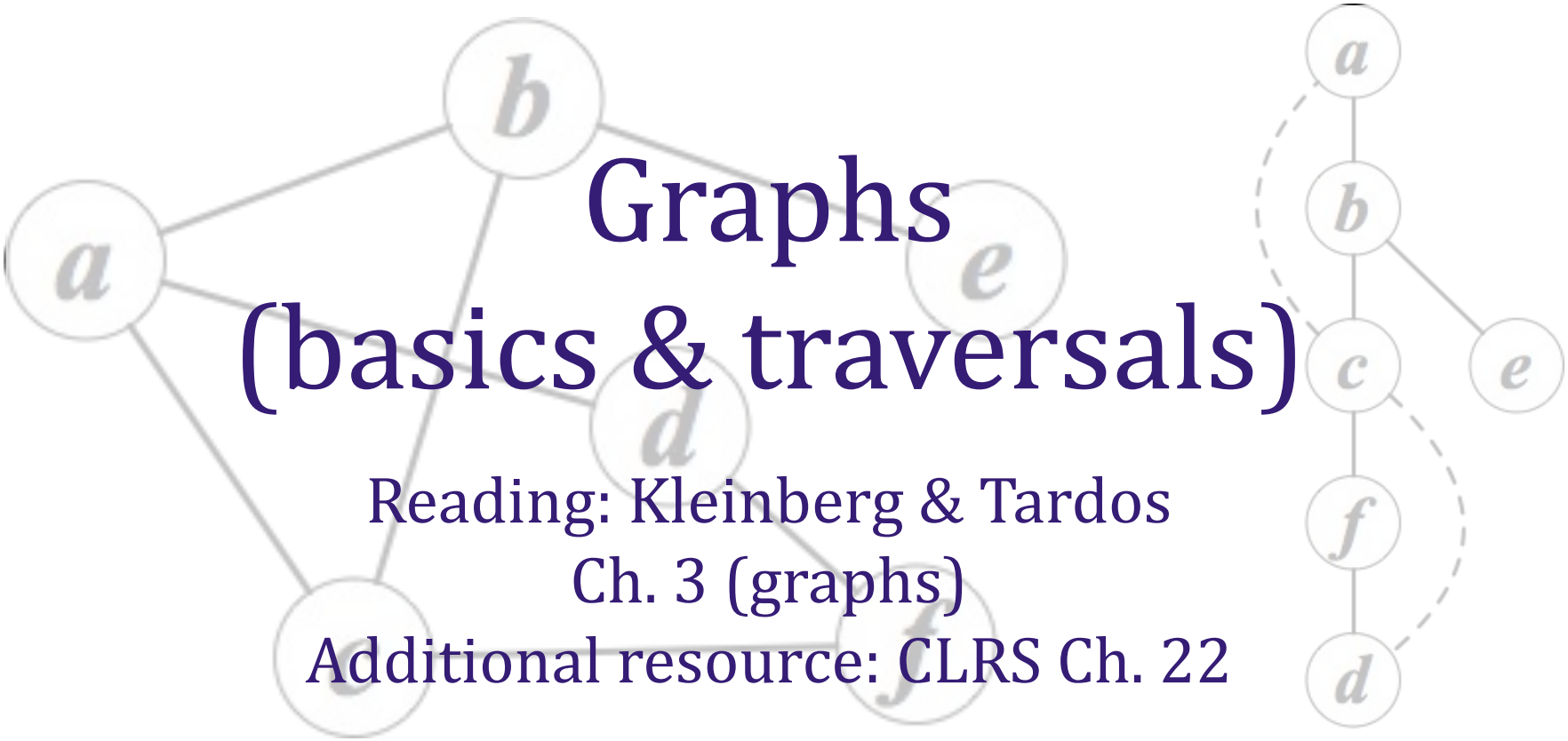
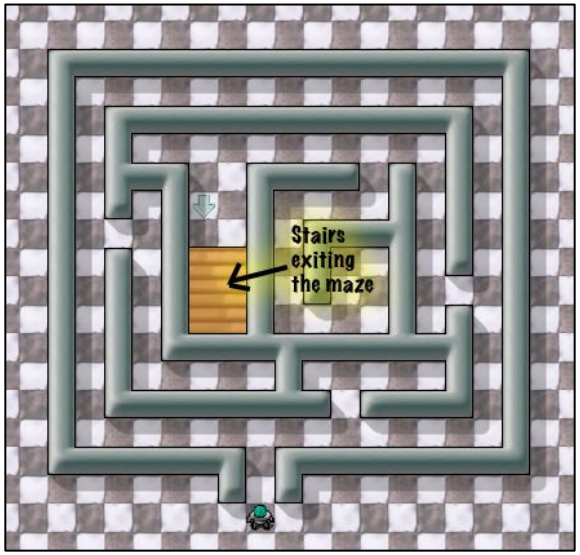
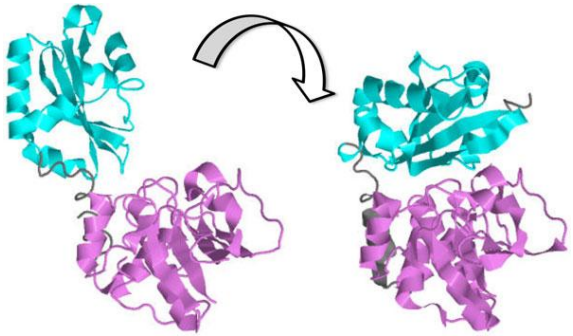


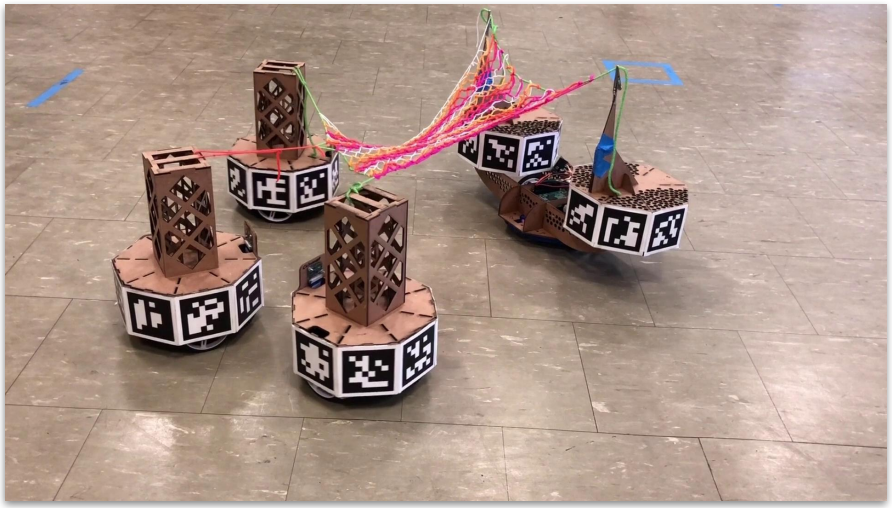
Graphs (basics & traversals)

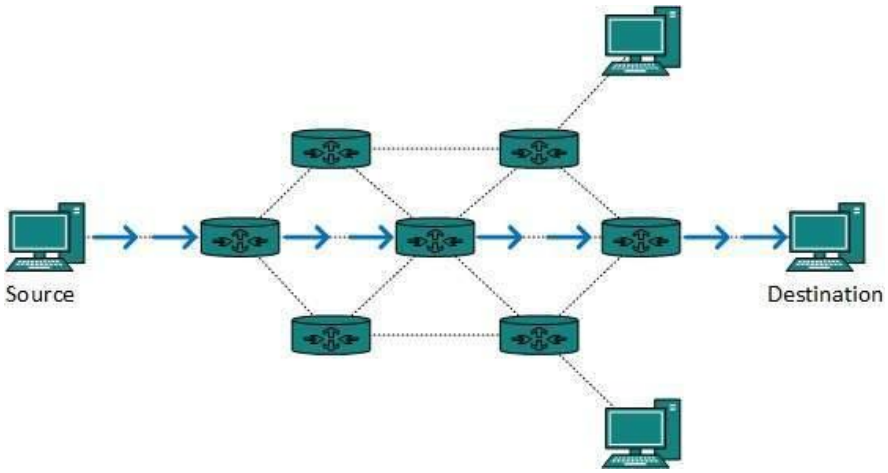
Reading: Kleinberg & Tardos
Ch. 3 (graphs)
Additional resource: CLRS Ch. 22

What do these have in common?







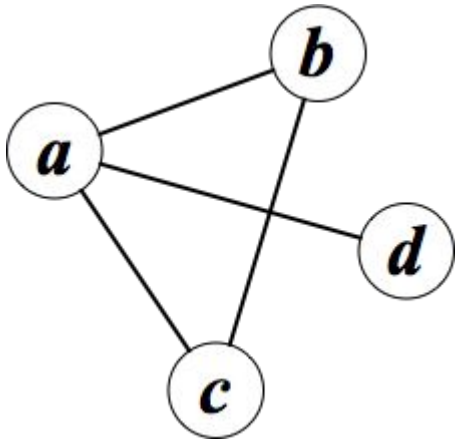


Undirected graph

- A graph $G = (V, E)$
  - V is a set of *vertices*
  - $E \subseteq V \times V$ is the set of *edges*
- Typically denote $|V| = n$ and $|E| = m$
- Exercise:

V = ?

E = ?

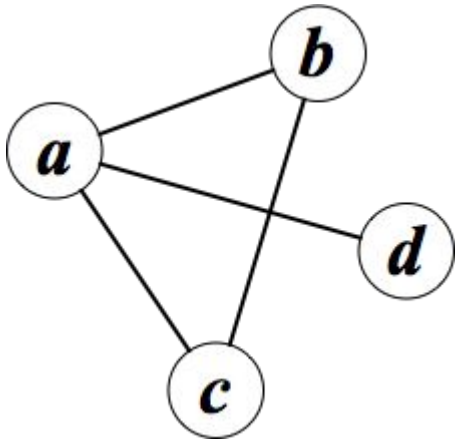


Undirected graph

- A graph $G = (V, E)$
  - V is a set of *vertices*
  - $E \subseteq V \times V$ is the set of *edges*
- Typically denote $|V| = n$ and $|E| = m$
- Exercise:

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

$E = \{ab,ac,ad,bc\}$



Graph representation

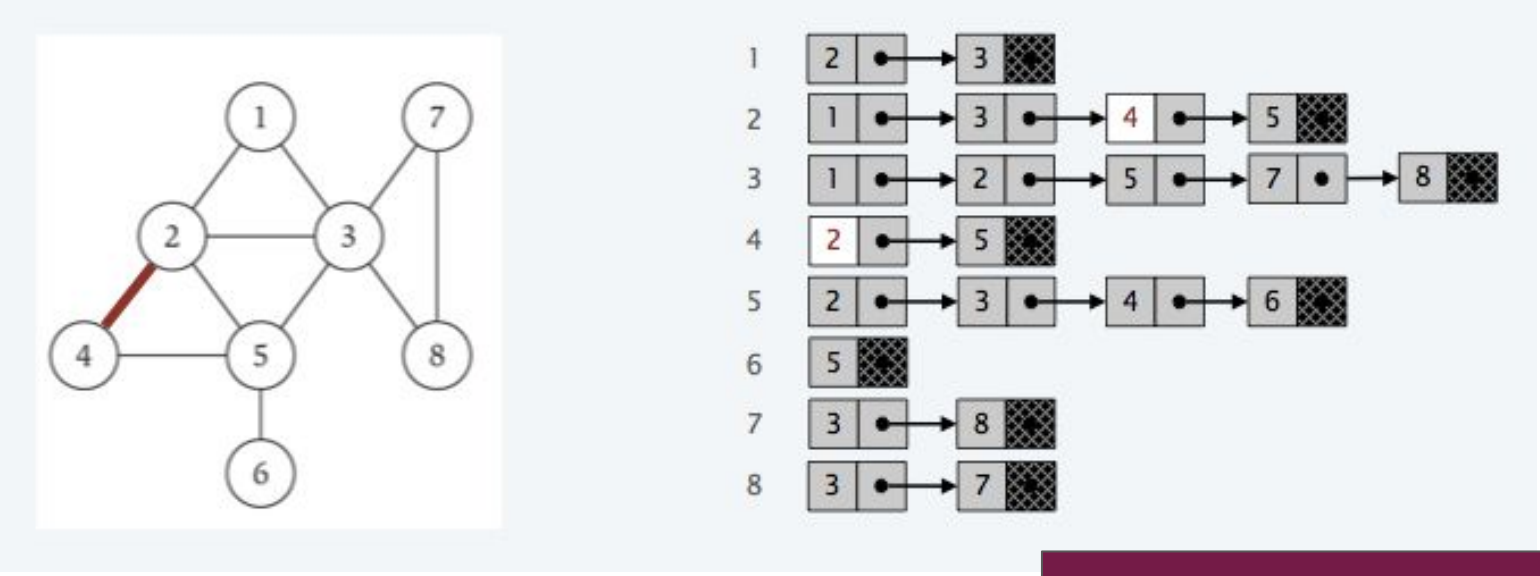
- Adjacency list
- Adjacency matrix
- Incidence matrix
- Edge sets
- …
- Data structures impact running time!

Graph representation

- **Adjacency list**
- Adjacency matrix
- Incidence matrix
- Edge sets
- …
- Data structures impact running time!

Graph representation

- **Adjacency list**



degree: # of neighbors

- Each edge stored twice (undirected)
- Space? $\Theta(m+n)$
- Checking if uv is an edge? $O(\text{degree}(u))$ time

Path

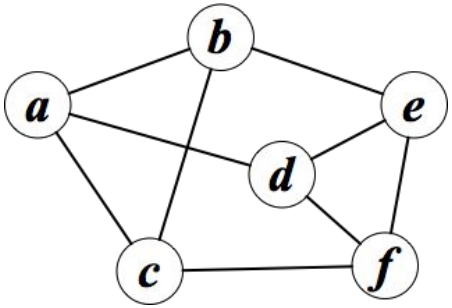
- ***path*** P: sequence of vertices

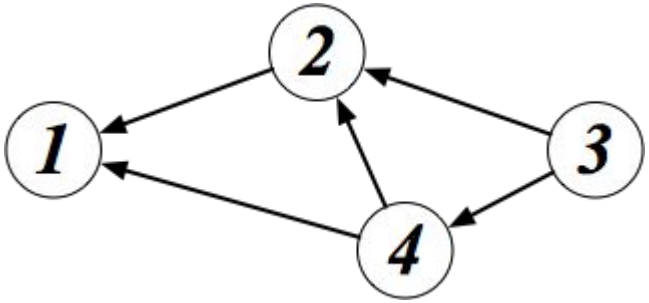
  $v_1, v_2, \ldots, v_{k-1}, v_k$

  such that, for each consecutive pair v_i, v_{i+1} (for all $i = 1,\ldots,k\text{-}1$),

  $v_i v_{i+1} \in E$

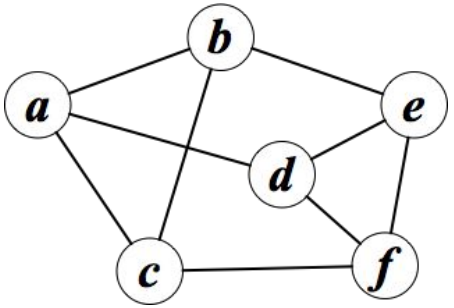
- "path from v_1 to v_k" or "v_1-v_k path"
- ***length:*** #edges traversed on path (k-1)
- ***simple path***: path where all vertices distinct (no repetitions)





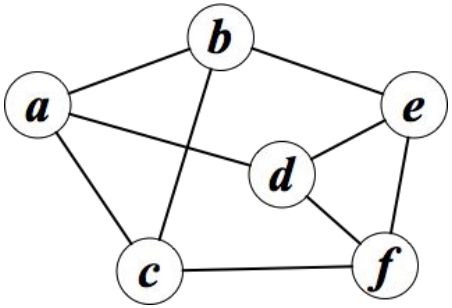
Distance and cycles

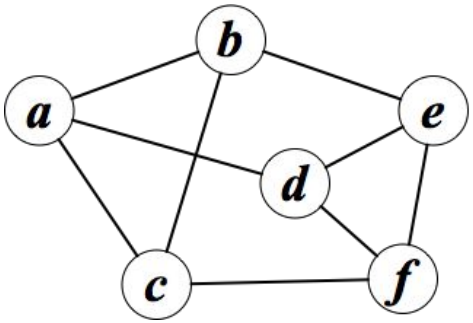
- ***distance*** from u to v: minimum length of a u-v path
- ***cycle*****:** path $v_1, v_2, \ldots, v_{k-1}, v_k$ where $v_1 = v_k$
  - Start and end at the same vertex.
- ***simple cycle***: cycle where all vertices $v_1, v_2, \ldots, v_{k-1}$ are distinct

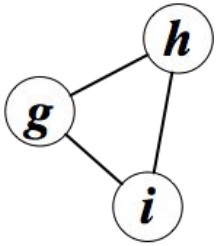


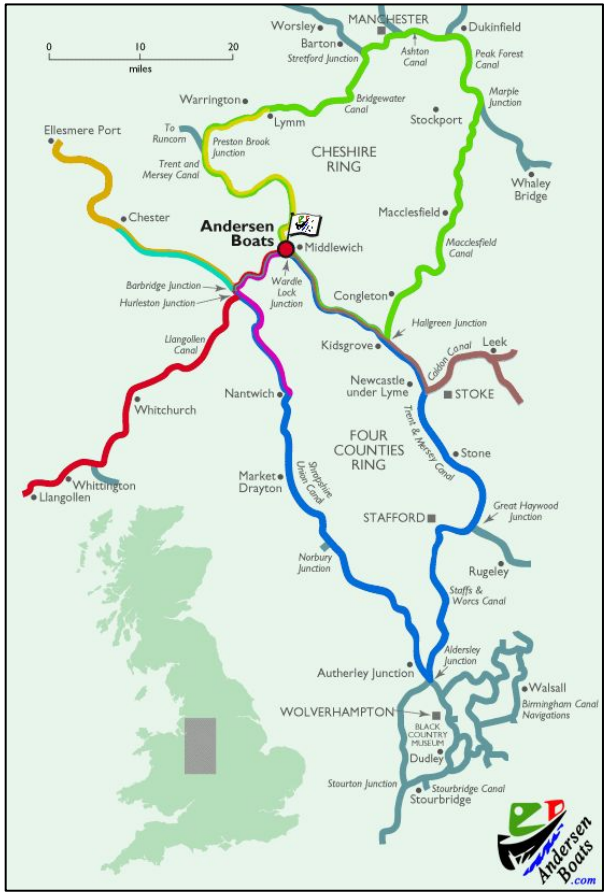
Connectivity

- graph is ***connected***: exists path for every pair of vertices









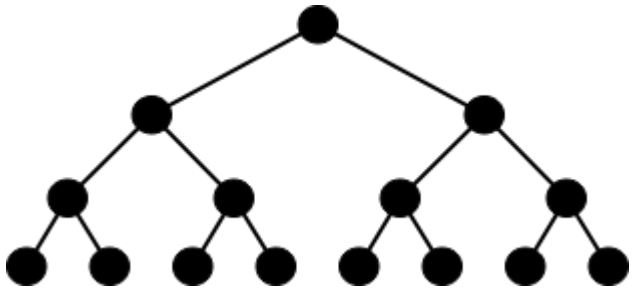
Trees

tree: connected (undirected) graph with no cycles

- **Tree properties**
  Let *G* be an (undirected) graph with *n* vertices. Then any two of the following statements implies the third:
  - *G* is connected
  - *G* has no cycles
  - *G* has *n-1* edges

You've probably seen

- **Rooted** tree: tree with (directed) parent-child relationship
- Pick root *r* & orient all edges away from root
- Parent of *v*: predecessor on path from *r* to *v*



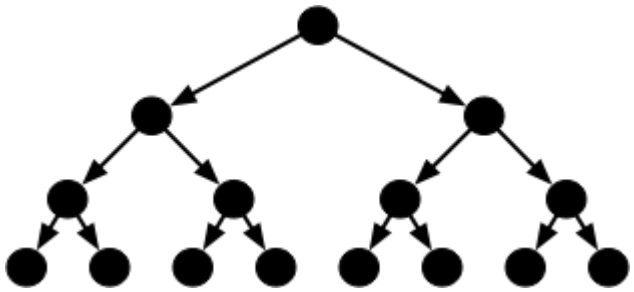
Trees

tree: connected (undirected) graph with no cycles

- **Tree properties**
  Let *G* be an (undirected) graph with *n* vertices. Then any two of the following statements implies the third:
  - *G* is connected
  - *G* has no cycles
  - *G* has *n-1* edges

You've probably seen

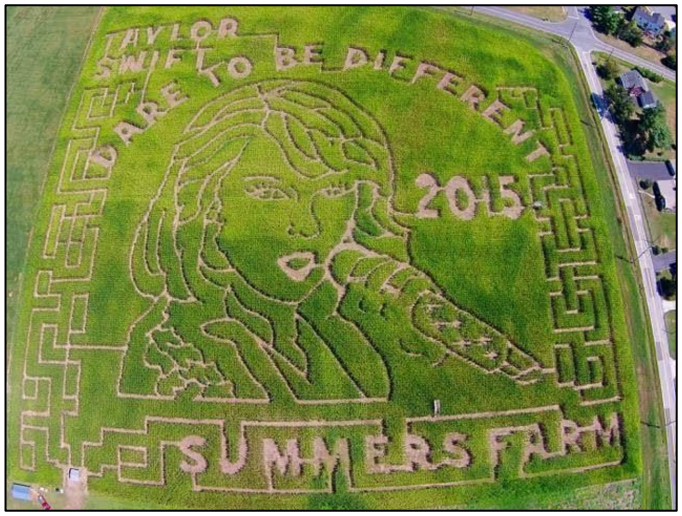
- **Rooted** tree: tree with (directed) parent-child relationship
- Pick root *r* & orient all edges away from root
- Parent of *v*: predecessor on path from *r* to *v*



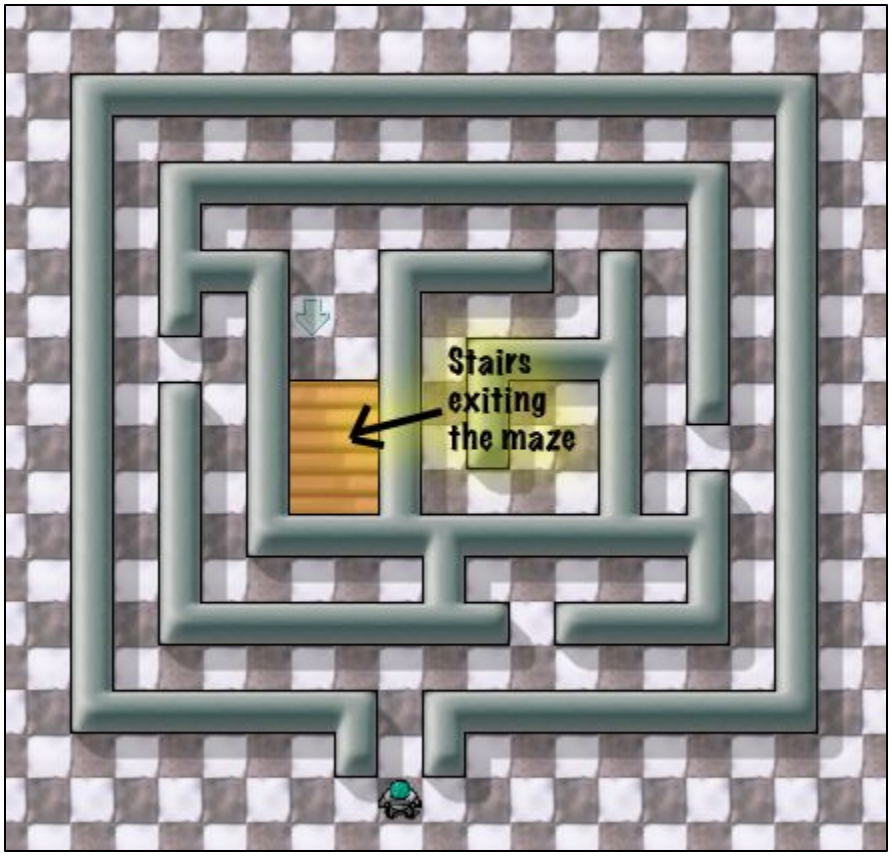
DFS Traversal

Don’t get lost!

“The “wall follower” rule, as it’s known among maze-solving experts, is simple: If you put your right hand on a corn maze wall and walk, it will, eventually, lead you to the exit... Sounds simple, right? But here’s the catch: The rule only works if the maze is **simply connected**...term to describe a maze that consists of walls that are all *connected to the outside wall...*”

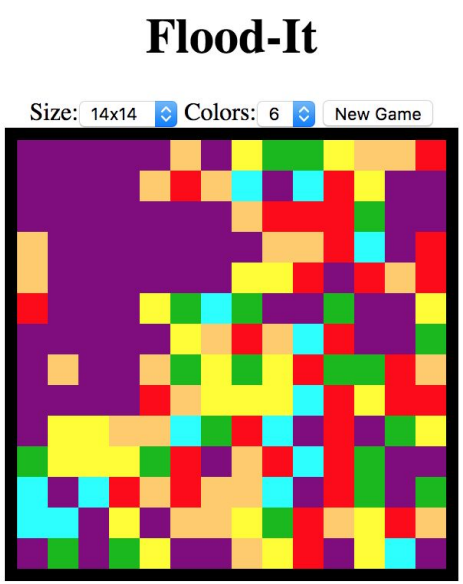


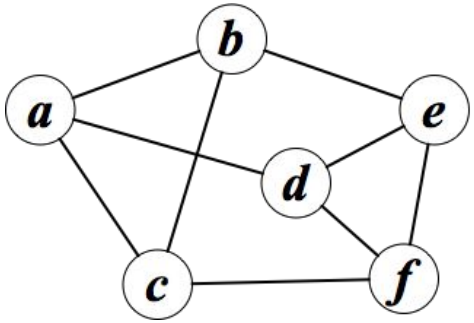
Connected?

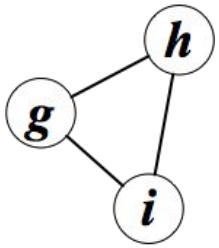


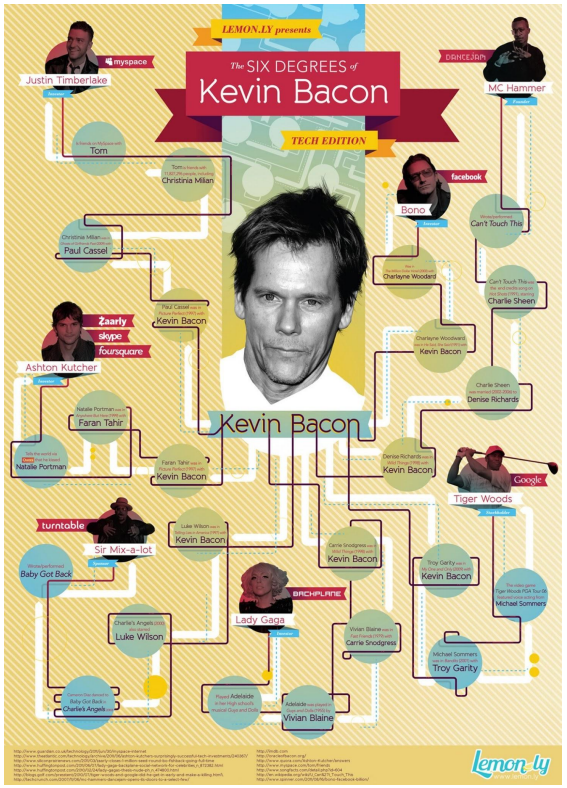
Connected components - applications

- ***connected component***: maximal set of vertices s.t. exists path for every pair of vertices



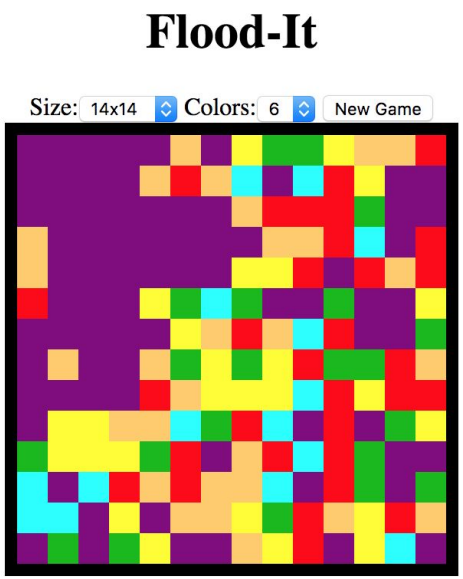


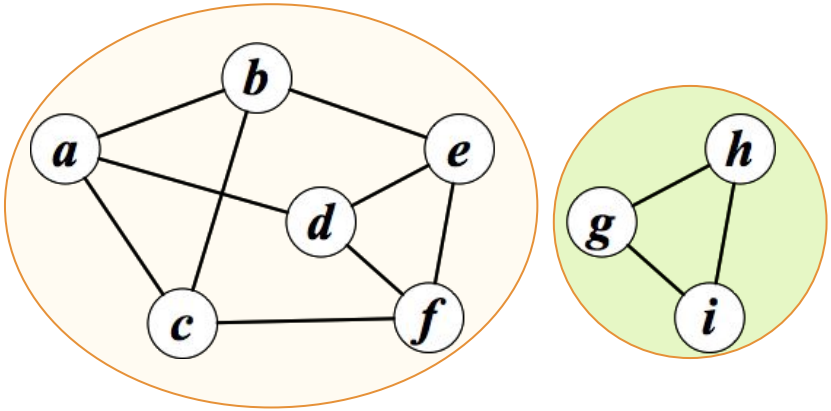


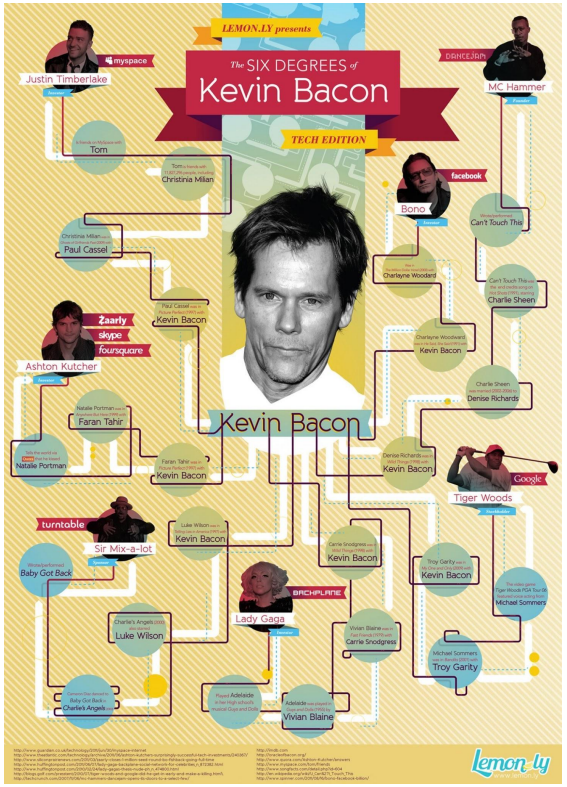


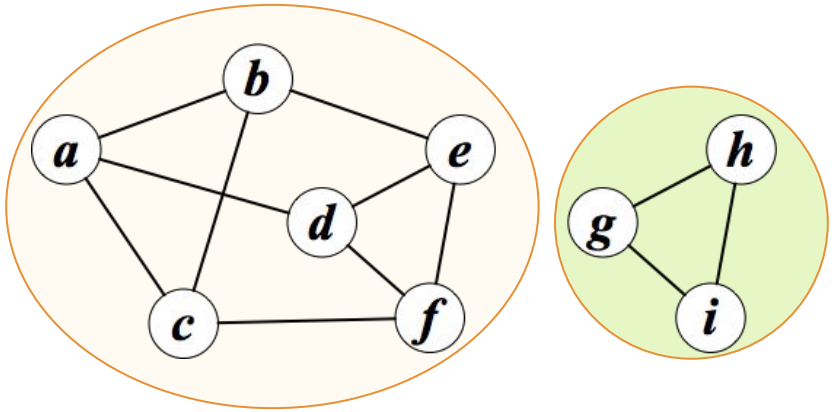
Connected components - applications

- ***connected component***: maximal set of vertices s.t. exists path for every pair of vertices

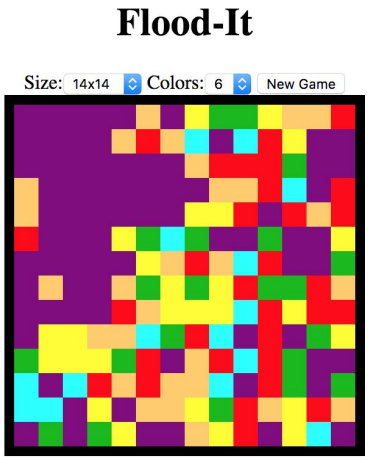








So…
how can we find a connected component?



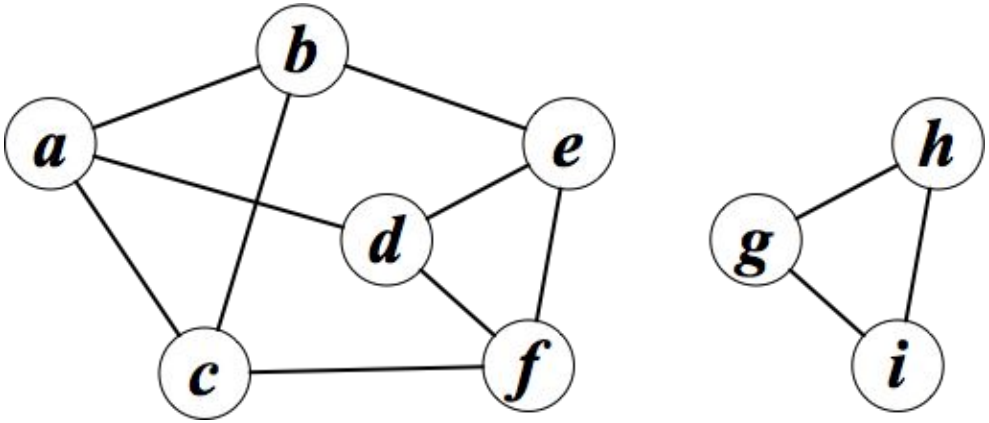
How can we find a connected component?

find-component(*G*,*v*):

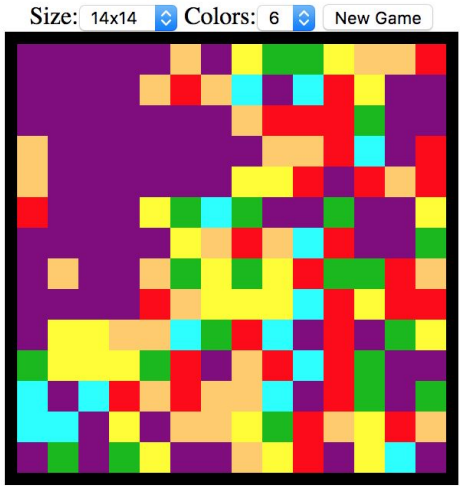
    for each vertex *u* in *G*.*V*
        u.visited = false
    C = {}
    explore(*G*,*v*,*C*)

explore(*G*,*x*,*C*):

    x.visited = true
    C = *C* ∪ {*x*}
    for each neighbor *y* of *x*
        if !(*y*.visited)
            explore(*G*,*y*,*C*)



Flood-It



Find a connected component -- running time

find-component(G,v):

    for each vertex u in $G.V$
        u.visited = false
    C = {}
    explore(G,v,C)

explore(G,x,C):

    x.visited = true
    $C = C \cup \{x\}$
    for each neighbor y of x
        if !(y.visited)
            explore(G,y,C)

n iterations

$\leq n$ invocations

2m iterations = $O(m)$

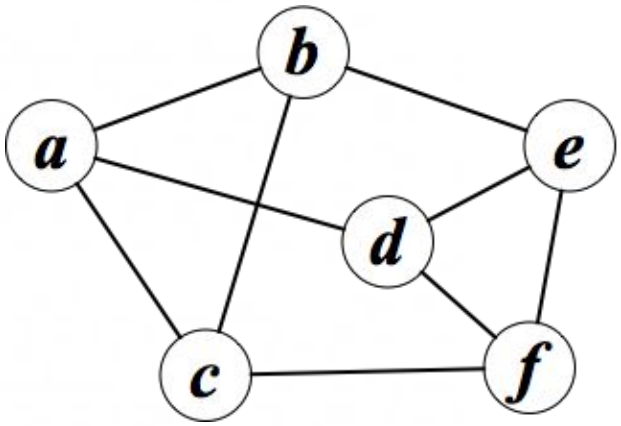
Total: $O(m + n)$

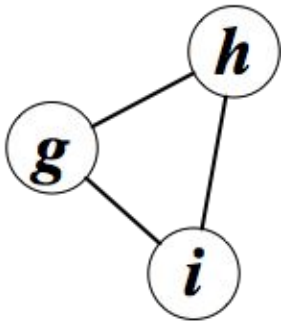
How can we experience all the storylines?

- Can you describe a process?
- In pseudocode?
- What is the running time?

Depth-first search (backtracking)

```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)
```






```
DFS-VISIT(G, u)
 1  time = time + 1          // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]    // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK          // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

Constructs a rooted tree for each connected component

From CLRS

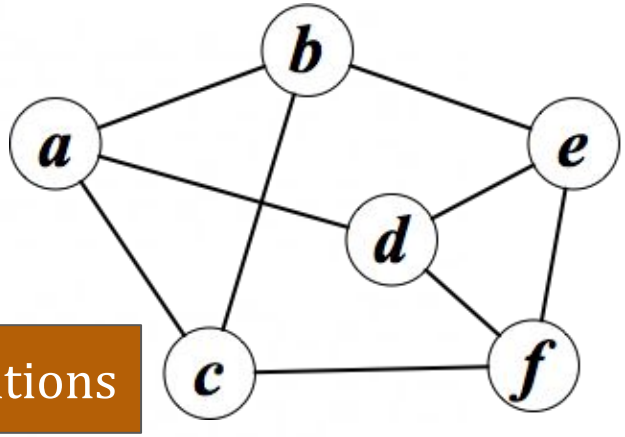
Depth-first search - running time?

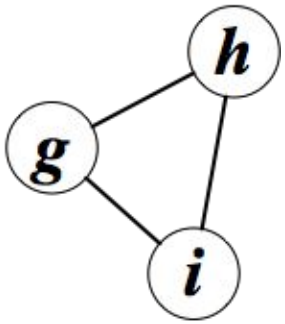
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1          // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]    // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK          // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```

O(n)

n invocations

2m =O(m)



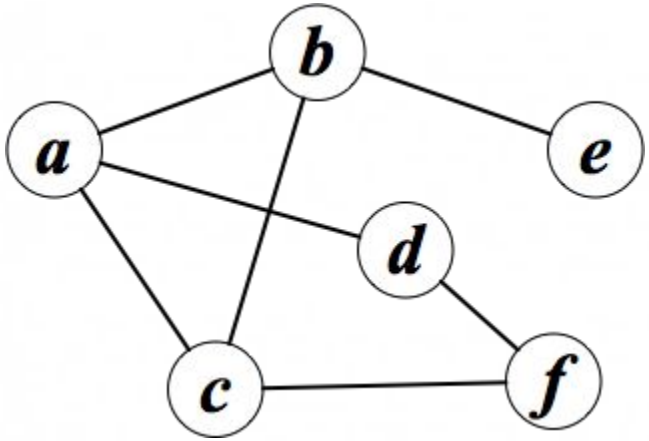


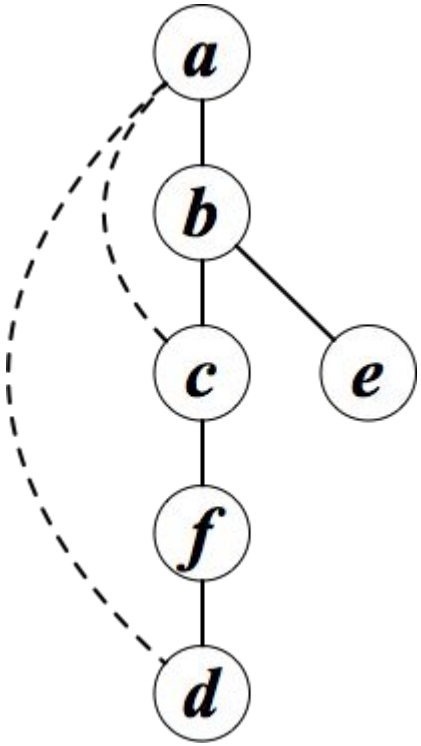
Total:
O(m + n)

From CLRS

For example…

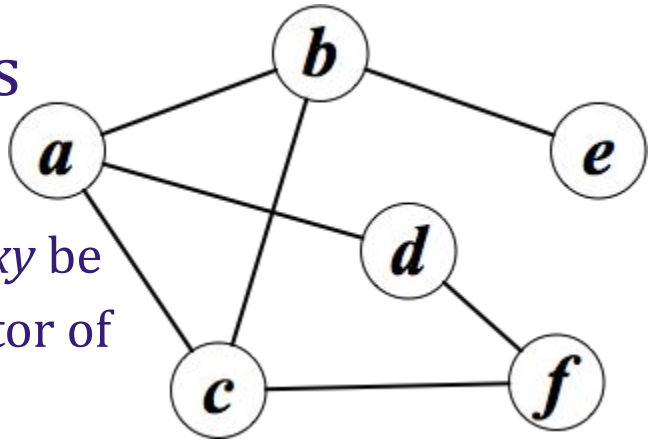
```
DFS(G)
1  for each vertex u ∈ G.V
2      u.color = WHITE
3      u.π = NIL
4  time = 0
5  for each vertex u ∈ G.V
6      if u.color == WHITE
7          DFS-VISIT(G, u)

DFS-VISIT(G, u)
 1  time = time + 1                  // white vertex u has just been discovered
 2  u.d = time
 3  u.color = GRAY
 4  for each v ∈ G.Adj[u]            // explore edge (u, v)
 5      if v.color == WHITE
 6          v.π = u
 7          DFS-VISIT(G, v)
 8  u.color = BLACK                  // blacken u; it is finished
 9  time = time + 1
10  u.f = time
```





From CLRS

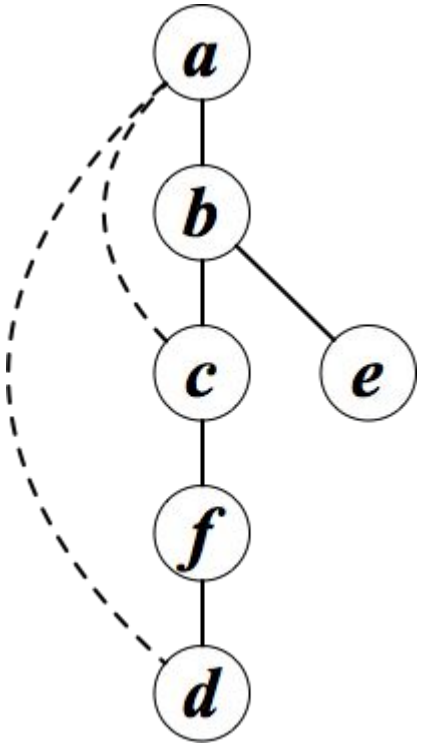
DFS: a small result on non-tree edges



Claim: Let T be the tree discovered by DFS, and let xy be an edge of G not in T. Then one of x or y is an ancestor of the other.

Proof: Let u be the first of the two nodes explored and v *the other*.

- Is v explored at beginning of DFS(u)? No.
- At some point during recursive calls from DFS(u), we examine the edge uv.
- Is v explored then? Yes; otherwise uv would be in T
- $\Rightarrow$ v was explored during DFS(u)
- $\Rightarrow$ v is a descendant of u

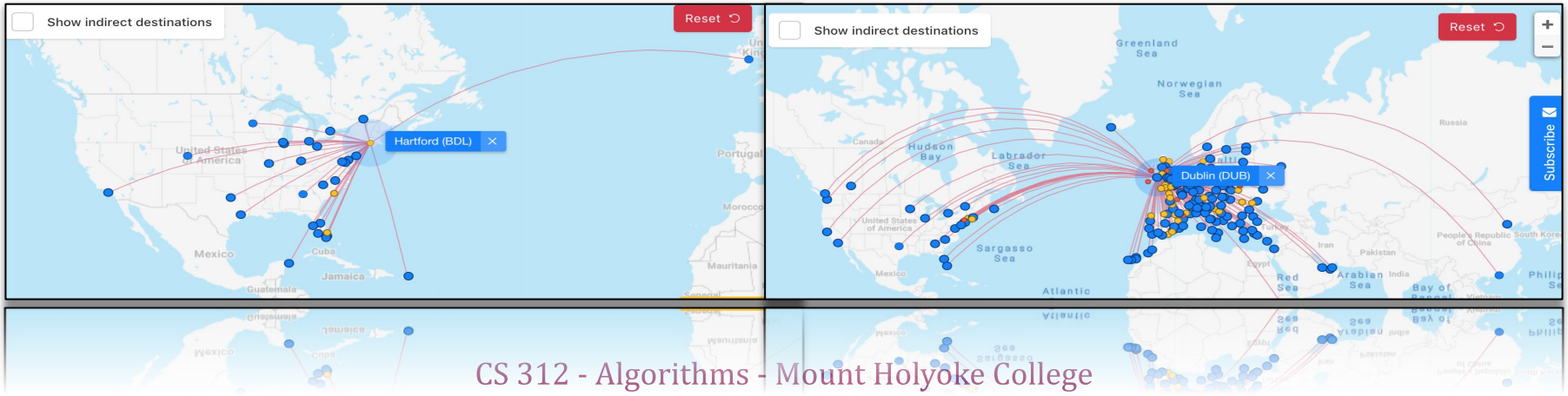


BFS Traversal

Do you want to travel the world?

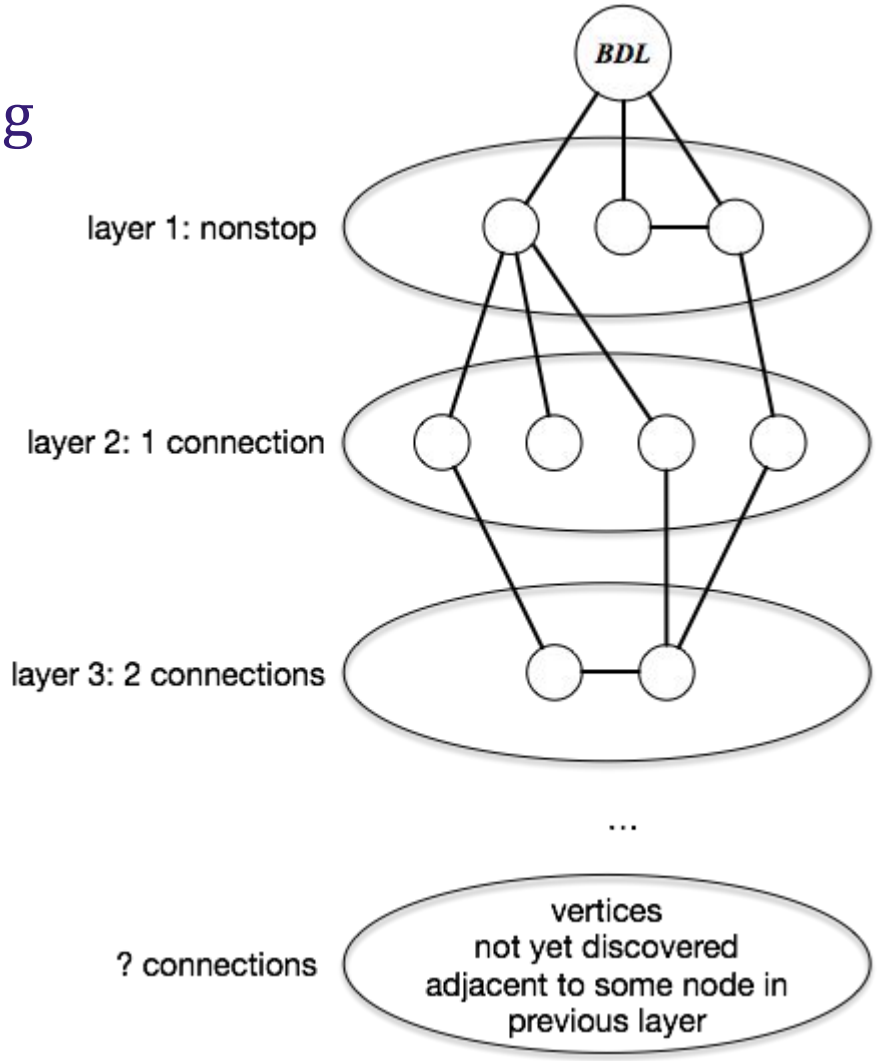
Flight hopping

- Construct graph $G = (V, E)$
  - V: airports
  - E: edge between two airports if nonstop flight
- Where could we go on a nonstop?
- With 1 connection? 2 connections?
- Can you fly to Boston?



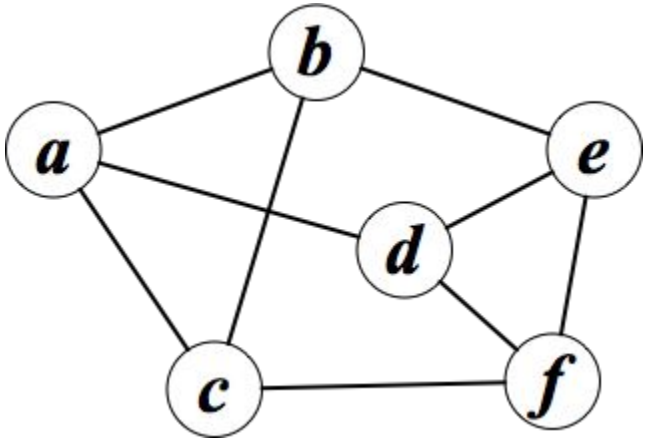
Breadth-first search

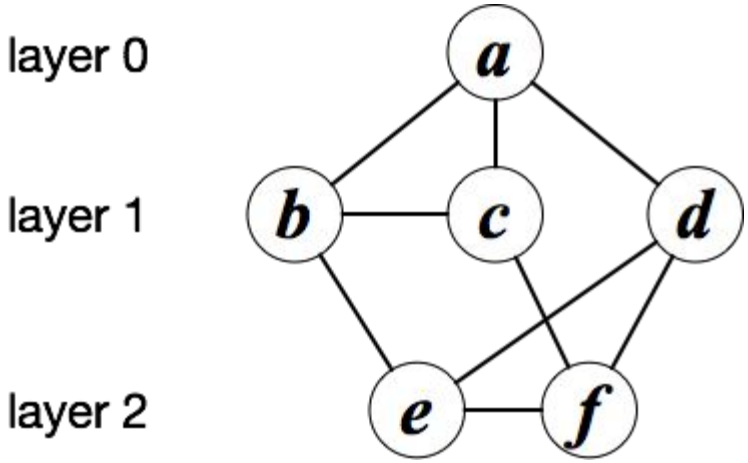
- Explore outward from starting vertex by distance
- "Expanding wave"



BFS layers

BFS from *a*





BFS implementation

```
BFS(G,v):
    for each vertex u in G.V
        u.visited = false
    v.layer = 0
    Q = empty queue
    Q.enqueue(v)
    v.visited = true
    while !Q.empty()
        x = Q.dequeue()
        for each neighbor y of x
            if !(y.visited)
                y.visited = true
                y.layer = x.layer + 1
                Q.enqueue(y)
```

n iterations

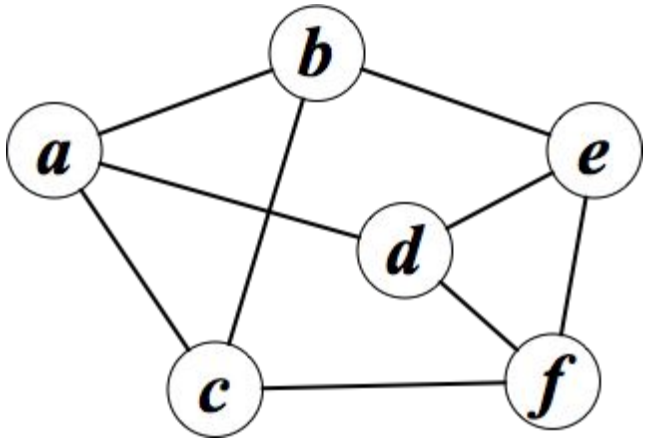
≤*n* iterations

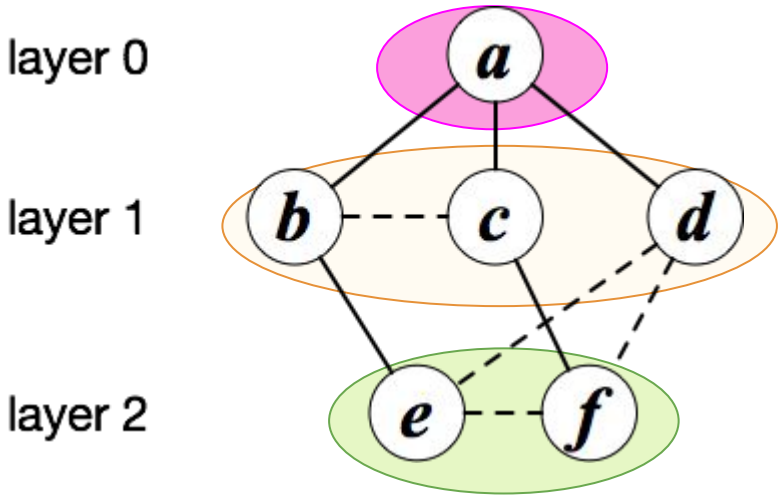
≤2*m* iterations

Total:
$O(m + n)$

BFS can produce a tree

```
BFS(G,v):
    for each vertex u in G.V
        u.visited = false
    v.layer = 0
    v.parent = null
    Q = empty queue
    Q.enqueue(v)
    v.visited = true
    while !Q.empty()
        x = Q.dequeue()
        for each neighbor y of x
            if !(y.visited)
                y.visited = true
                y.parent = x
                y.layer = x.layer + 1
                Q.enqueue(y)
```

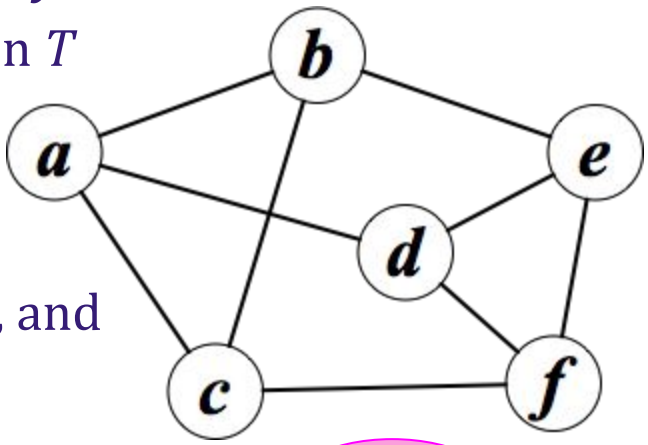




[CLRS]

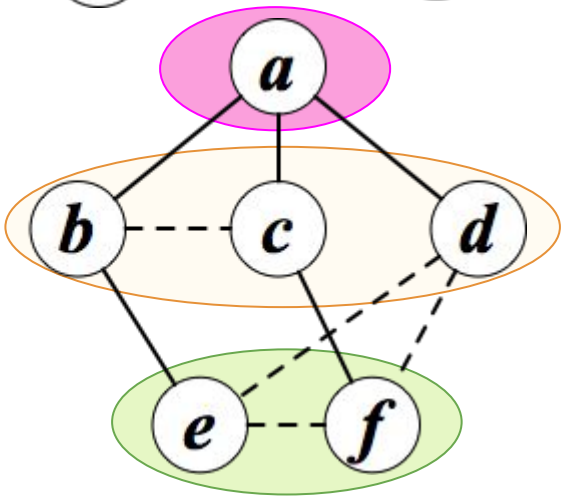
BFS: a small result on non-tree edges

Claim: Let T be the tree discovered by BFS, and let xy be an edge of G not in T. Then the layer of x and y in T differ by at most 1.



Proof: Let u be the first of the two nodes explored, and v the other. Let L_i be u's layer in T.

- Then $v \in L_j$ for $j \geq i$.
- When neighbors of u are explored, v is either already in L_i or L_{i+1}, or is discovered and added to L_{i+1}.



Generic graph traversal

Let A = data structure of discovered vertices

traverse(G,v):

```
    for each vertex u in G.V
        u.explored = false
    A = empty data structure
    A.add(v)
    while !A.empty()
        x = A.remove()
        if !x.explored:
            x.explored = true
            for each neighbor y of x
                A.add(y)
```

BFS: A is a queue (FIFO)

DFS: A is a stack (LIFO)

- Can a node be discovered (placed in A) multiple times? Yes.
- For DFS, node is explored from parent that added it last (LIFO).
- For BFS, can avoid by not adding discovered nodes.

What about the rest of the graph?

- How to explore entire graph even if it is disconnected?

  while (there is some unexplored node v)
  traverse(v) // Run BFS/DFS starting from v

- Running time -- does it change?
  - naive: $O(m+n)$ per component $\Rightarrow O(c(m+n))$ if c components
  - better: traversal for component C only works on vertices/edges in C
    - Time for component C: O(#edges in C + #vertices in C)
    - Total time still $O(m+n)$
- Usually OK to assume graph is connected.
  - State if you are doing so and why it does not trivialize the problem.

Traversals

- BFS/DFS: basic algorithmic primitive used in many other algorithms
  - Different versions of general exploration strategy
- Can answer:
  - Is there a path from u to v?
  - Find all connected components
  - Produce trees with different properties
- $\Theta(m+n)$ time