Analysis of Algorithms, I
CSOR W4231

Eleni Drinea
Computer Science Department

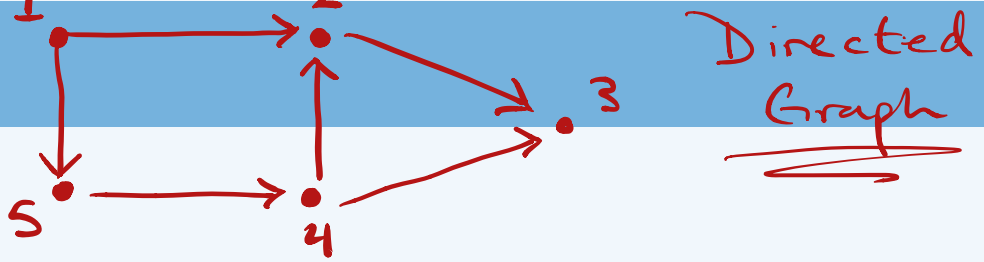
Columbia University

Graphs, Breadth-First Search (BFS)

Outline

Today

1. Graphs
2. Representing graphs
3. Breadth-first search (BFS)
4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness



Definition 1.

A **directed** graph consists of a finite set V of vertices (or nodes) and a set E of directed edges. A directed edge is an ordered pair of vertices (u, v).

- In mathematical terms, a directed graph $G = (V, E)$ is just a binary relation $E \subseteq V \times V$ on a finite set V.
- An **undirected** graph is the special case of a directed graph where $(u, v) \in E$ if and only if $(v, u) \in E$. In this case, an edge may be indicated as the unordered pair $\{u, v\}$.
- **Notational conventions:** $|V| = n$, $|E| = m$

If weights are not specified on the edges, the default weight is 1 for each edge.

Node degrees

- **Undirected** graphs

$$deg(v) \triangleq \text{ number of edges incident to } v$$

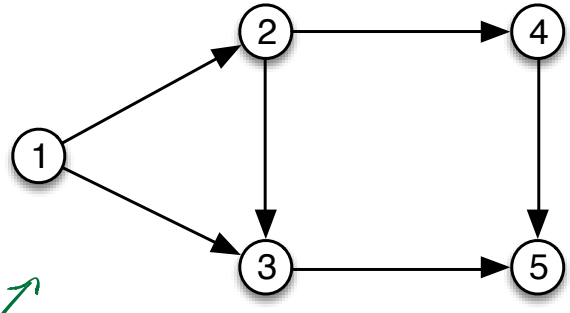
- **Directed** graphs

$$\begin{aligned} indeg(v) &\triangleq \text{ number of edges entering } v \\ outdeg(v) &\triangleq \text{ number of edges leaving } v \end{aligned}$$

Example graphs

Circles denote vertices (nodes).
Lines denote edges connecting vertices.
Arrows on lines indicate the direction along which the edge may be traversed.

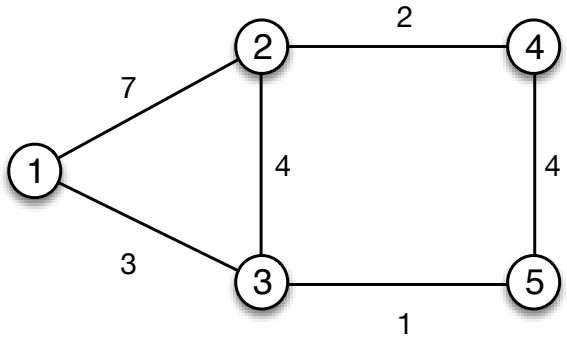
A directed, unweighted graph G
(default edge weight w(e) = 1)



indeg(1) = 0
indeg(3) = 2

outdeg(1) = 2
outdeg(3) = 1

An undirected, weighted graph G'



deg(1) = 2
deg(3) = 3

$E = \{(1,2), (1,3), (2,3), (2,4), (3,5), (4,5)\}$

Examples of graphs (networks)

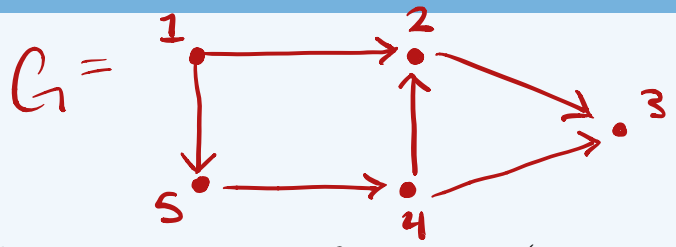
- **Transportation** networks: e.g., nodes are cities, edges (potentially weighted) are highways connecting the cities
  - *Can we reach a city j from a city i?*
  - *If* **yes**, *what is the shortest (or cheapest) path?*

  Reachability Questions / Problems

  For unweighted graphs, this problem can be reduced to finding the shortest path since all edges have a weight = 1
- **Information** networks: e.g., the World Wide Web can be modeled as a directed graph Nodes ≡ Web Pages ; Edges ≡ Hyperlinks
- **Wireless** networks: nodes are devices sitting at locations in physical space and there is an edge from u to v if v is close enough to u to hear from it.

  Directed since one website could link to another website, but not necessarily the other way around
- **Social** networks: e.g., nodes are people, edges represent friendship
- **Dependency** networks: e.g., given a list of functions in a large program, find an order to test the functions.

Useful definitions



1-3 paths in G:
(1, 2, 3)
(1, 5, 4, 3)

- A **path** is a sequence of vertices $(x_1, x_2, \ldots, x_n)$ such that consecutive vertices are adjacent, that is, there exists an edge $(x_i, x_i + 1) \in E$ for all $1 \leq i \leq n - 1$.

  Example: $(1, 2, 3, 2, 4)$ in G' is a path. ← Slide 6

- A path is **simple** when all vertices are distinct.

  Example: $(1, 2, 4)$ in G' is a simple path.

- A **simple cycle** is a simple path that ends where it starts $(x_n = x_1)$.

  Example: $(1, 2, 3, 1)$ in G' is a simple cycle.

The only vertex that may be repeated in a simple cycle is the first vertex of the path, but can only be repeated twice (start and end).

Useful definitions (cont'd)

Distance from v_1 to v_2 is the length of the shortest path from to v_1 to v_2.

The **distance** from u to v, denoted by $dist(u, v)$, is the *length* of the *shortest* path from u to v.

- **Unweighted** graphs:

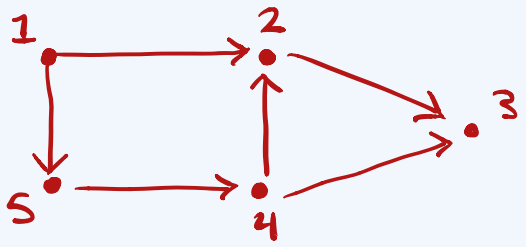
$$\textit{length of path } P \triangleq \#\text{ edges on } P$$

Since $\forall$ edges in an unweighted graph have weight = 1.

- **Weighted** graphs: *coming up in a few lectures.*

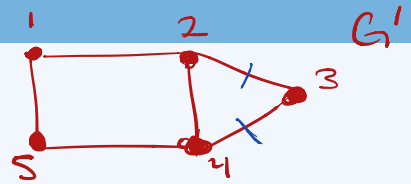
- *Shortest* path from u to v: a path of minimum length among all paths from u to v.

Example: in G, $dist(1, 4) = 2$.



Useful definitions (cont'd)

What is the minimum # of edges to remove to make G' disconnected? 2



- An undirected graph is **connected** when there is a path between every pair of vertices.

  Example: G' is connected.

If the graph is connected the connected components will be the entire set of vertices.

- The **connected component** of a node u is the set of all nodes in the graph reachable by a path from u.

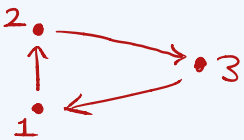
  Example: the connected component of 1 in G' is $\{1, 2, 3, 4, 5\}$.

- A directed graph is **strongly connected** if for every pair of vertices u, v, there is a path from u to v and from v to u.

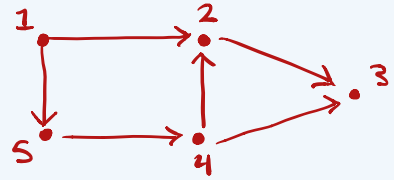
For a graph with $V \geq 2$ to have a SCC, it must have cycles.

- The **strongly connected component** of a node u in a directed graph is the set of nodes v in the graph such that there is a path from u to v and from v to u.

  Example: the strongly connected component of 1 in G is $\{1\}$.



Strongly Connected



Strongly connected component of 1 is just {1}

Trees and tree properties

Trees are always undirected.

A tree is the minimal connected graph. If you remove any edge it is no longer connected

Definition 2.

1. A tree is a **connected acyclic** graph (undirected graphs). Or;
2. A **rooted** graph such that there is a unique path from the root to any other vertex (all graphs).

↳ Rooted graphs are either directed or undirected.

A tree is the most widely used special type of graph: it is the minimal connected graph. — If you remove one edge, the graph becomes disconnected.

Lemma 3.

Let G be an undirected graph. Any two of the following properties imply the third property, and that G is a tree.

1. *G is connected;*
2. *G is acyclic;*
3. *$|E| = |V| - 1$.*

(Memorize this Lemma!!)

- If 1 and 2 are true, then G is a tree, by definition.
  1 and 2 $\Rightarrow$ 3
- 1 and 3 $\Rightarrow$ 2 and G a tree
- 2 and 3 $\Rightarrow$ 1 and G a tree.

Trees and tree properties

A connected graph H is a tree if and only if for each pair of vertices (x,y), there is exactly one path joining x and y.

Definition 2.

A tree is a **connected acyclic** graph (undirected graphs). Or;
A **rooted** graph such that there is a unique path from the root to any other vertex (all graphs).

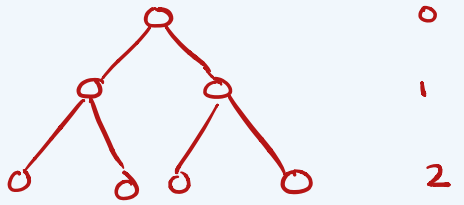
← exactly one path.

A tree is the most widely used special type of graph: it is the minimal connected graph. If you remove any edge from a tree, it is no longer connected.

Lemma 3.

Let G be an undirected graph. Any two of the following properties imply the third property, and that G is a tree.

1. *G is connected;*
2. *G is acyclic;*
3. $|E| = |V| - 1$.



TRUE FOR Directed graphs too.

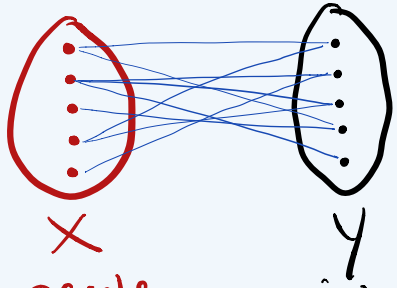
↑ true for directed graphs?? Yes

Matchings and bipartite graphs

Bipartite graphs: vertices can be split into two subsets such that there are no edges between vertices in the same subset.

- Applications: social networks, coding theory
- **Notation:** $G = (X \cup Y, E)$, where $X \cup Y$ is the set of vertices in G and every edge in E has one endpoint in X and one endpoint in Y.

Example: suppose there are 5 people and 5 jobs and certain people qualify for certain jobs.

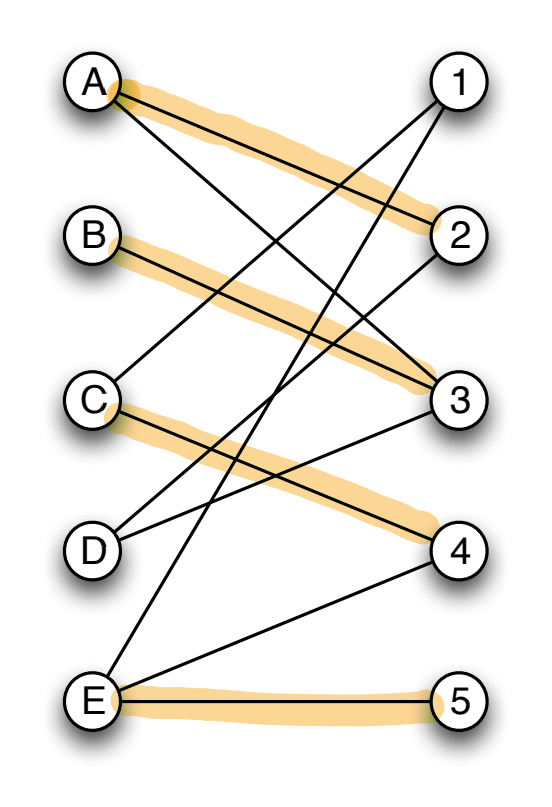


Matchings in bipartite graphs

people Jobs

Matching: a subset of the edges where every node appears at most once.

Perfect Matching: Looking for a set of edges in the bipartite graph such that every node appears exactly once (1-to-1 matching)

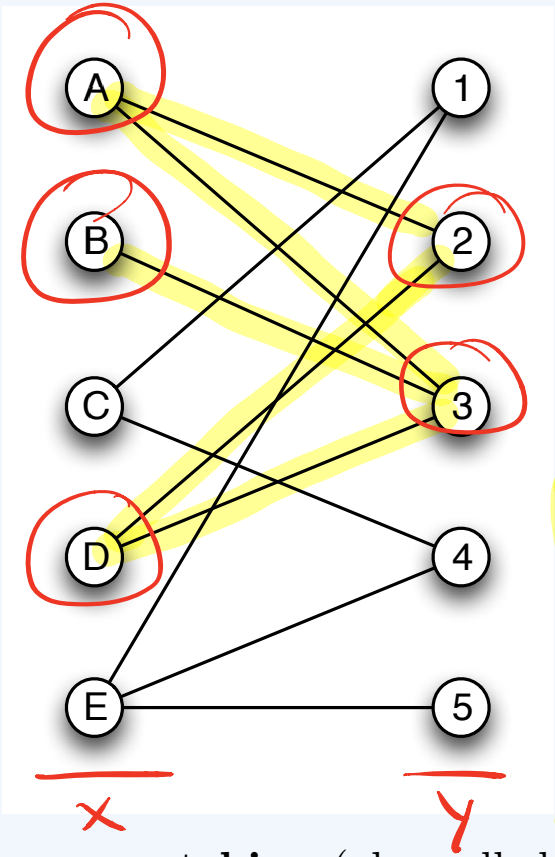


- $\{(A,2)\}$ is a matching of size 1.
- $\{(A,2),(B,3)\}$ is a matching of size 2.
- $\{(A,2),(B,3),(C,4)\}$ is a matching of size 3.
- $\{(A,2),(B,3),(C,4),(E,1)\}$ is a matching of size 4
- No matchings of size 5.

Goal: find a **one-to-one matching** (also called, a perfect matching) of people to jobs, if one exists.

Matchings in bipartite graphs

Matching: a subset of the edges where every node appears at most once.



No perfect matchings since there are 3 people who qualify for the same 2 jobs.

For a perfect matching to exist any subset of people, the # of jobs they collectively qualify for should be at least as many people in the subset.

If $\exists$ PM, then $\forall M \subseteq X$, $|N(M)| \geq |M|$

$N(M)$: neighbors of M in Y

Goal: find a **one-to-one matching** (also called, a perfect matching) of people to jobs, if one exists.

Matchings in bipartite graphs

Matching: a subset of the edges where every node appears at most once.

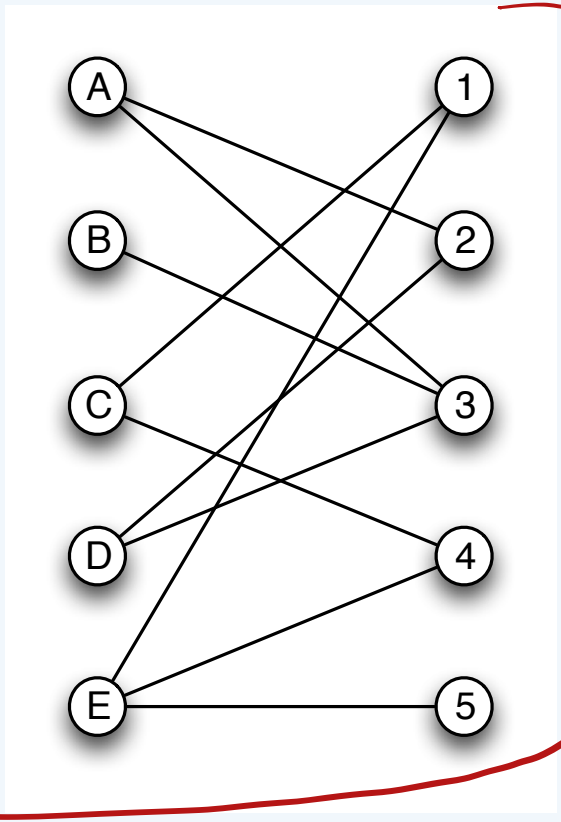
Is this a sufficient cond.?

Necessary Condition:

If $\exists$ PM, then $\forall M \subseteq X$, $|N(M)| \geq |M|$

$N(M)$: neighbors of M in Y

Sufficient Condition?



If for every subset of X, the # of neighbors they have in Y is at least the cardinality of that subset – do we necessarily have a perfect matching?

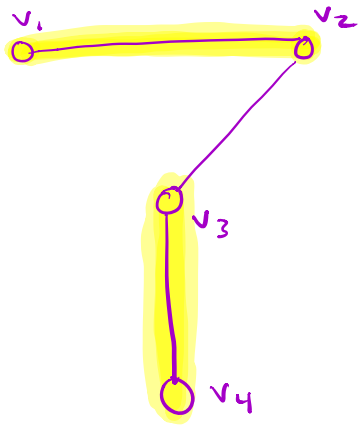
Yes

Not clear why – will discuss later.

Goal: find a **one-to-one matching** (also called, a perfect matching) of people to jobs, if one exists.

Tree on $n=4$ vertices $\{v_1, v_2, v_3, v_4\}$.

Perfect matching is highlighted

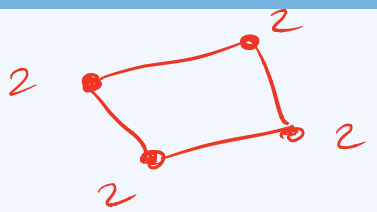


For a perfect matching to exist in bipartite graph $G = (X \cup Y, E)$, the following must hold:

$$\forall A \subseteq X, \quad |N(A)| \geq |A|$$

where $N(A) \equiv$ neighbors of A in the set Y

Degree theorem



undir : $\sum_{v \in V} \deg(v) = 2|E|$

dir : $\sum_{v \in V} \text{indeg}(v) + \sum_{v \in V} \text{outdeg} = 2|E|$

Theorem 4.

In any graph, the sum of the degrees of all vertices is equal to twice the number of the edges.

Going to use this to evaluate run-time of graph algorithms.

Proof.

Every edge is incident to two vertices, thus contributes twice to the total sum of the degrees. (Summing the degrees of all vertices simply counts all instances of some edge being incident to some vertex.) □

$\sum \text{outdeg}(v) = |E|$

Because every edge must originate at some node.

$\sum \text{indeg}(v) = |E|$

Because every edge has to enter some node.

Running time of graph algorithms

Notational convention that we will use for graph algorithms in this course:

Input: graph $G = (V, E)$, $|V| = n, |E| = m$

- Linear graph algorithms run in $O(n+m)$ time
  - *Lower bound on m (assume connected graphs)?* → $m \geq n-1$, $m = \Omega(n)$
  - *Upper bound on m (assume simple graphs)?*
    → $m \leq \binom{n}{2} = \frac{n(n-1)}{2} \Rightarrow m = \overline{O}(n^2)$ (for complete graphs)
- More general running times: the best performance is determined by the relationship between n and m
  - For example, $O(n^3)$ is better than $O(m^2)$ if the graph is dense (that is, $m = \Omega(n^2)$ edges)

Lower-bound on the # of edges in a connected graph?
$m \geq n-1$ since the minimal connected graph is a tree and the # of edges in a tree is $n-1$. So $m = \Omega(n)$

Running time of graph algorithms

Notational convention that we will use for graph algorithms in this course:

In general, the run-time depends on the density of the graph.

Input: graph $G = (V, E), |V| = n, |E| = m$

- Linear graph algorithms run in $O(n+m)$ time
  - *Lower bound on m (assume connected graphs)?* $m \geq n-1$
  - *Upper bound on m (assume simple graphs)?*

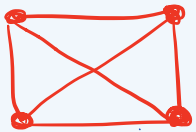
For dir. graphs $m < n \cdot (n-1)$

$$m \leq \binom{n}{2} = \frac{n(n-1)}{2} \Rightarrow m = \bar{O}(n^2)$$

- More general running times: the best performance is determined by the relationship between n and m
  - For example, $O(n^3)$ is better than $O(m^2)$ if the graph is dense (that is, $m = \Omega(n^2)$ edges)

$K_n \equiv$ complete graph on n vertices

has $\binom{n}{2} = \frac{n(n-1)}{2}$ edges.



K_4

Running time of graph algorithms

sparse graph: $O(n)$ edges

dense graph: $O(n^2)$ edges

Input: graph $G = (V, E), |V| = n, |E| = m$

- Linear graph algorithms run in $O(n + m)$ time
  - *Lower bound on m (assume connected graphs)?*
  - *Upper bound on m (assume simple graphs)?*

- More general running times: the best performance is determined by the relationship between n and m
  - For example, $O(n^3)$ is better than $O(m^2)$ if the graph is dense (that is, $m = \Omega(n^2)$ edges)

Ex. 1: If we have a sparse graph: $O(n)$ edges (e.g., a tree) would we prefer the $O(n^3)$ algorithm or the $O(m^2)$ alg.?

$O(m^2)$ algorithm.

Ex. 2: Which algorithm would we prefer on a dense graph (such as a complete graph)?

$O(n^3)$ algorithm

[Because the $O(n^2)$ alg. would become $O(n^4)$.]

- If you have 2 graph algorithms that solve the same problem and they have different complexities, you should look at your inputs in the particular application before you decide which one to use.

sparse graph: A graph in which the number of edges is much less than the possible number of edges. → $O(n)$ edges

dense graph: A graph in which the number of edges is close to the maximal number of edges. → $\Omega(n^2)$ edges

Today

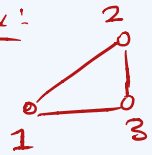
1. Graphs
2. Representing graphs
3. Breadth-first search (BFS)
4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Representing graphs: adjacency matrix

We want to represent a graph $G = (V, E)$, $|V| = n$, $|E| = m$.

Adjacency matrix for G: an $n \times n$ matrix A such that

Ex:



$$A[i,j] = \begin{cases} 1, & \text{if edge } (i,j) \in E \\ 0, & \text{otherwise} \end{cases}$$

$E = \{(1,2), (1,3), (2,3)\}$

$$A = \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{bmatrix}$$

Space required for adjacency matrix A: $\Theta(n^2)$.

Notice that A is symmetric.

Remark 1.

Space requirements can be improved if the graph is

- *undirected:* *A is symmetric $\Rightarrow$ only store its upper triangle*
- *unweighted:* *only need 1 bit per entry*

Since its a 0-1 matrix.

Since A is symmetric we only need to store its upper triangle.

Pros/cons of adjacency matrix representation

Representing $G = (V, E)$, $|V| = n$, $|E| = m$ by its adjacency matrix has the following pros/cons.

Advantages:

1. check whether edge $e \in E$ in constant time
2. easy to adapt if the graph is weighted
3. suitable for dense graphs where $m = \Theta(n^2)$

Just check entry (Random Access Model)

just replace 1's w/ the weights.

↑ since A has n^2 entries.

Drawbacks:

1. requires $\Omega(n^2)$ space even if G is sparse ($m = o(n^2)$).
2. does not allow for linear time algorithms in sparse graphs (at least when all matrix entries must be examined).

Should not use for sparse graphs (e.g., trees) since using $\Theta(n^2)$ space to store $O(n)$ edges – this is VERY inefficient in terms of space usage.

Representing graphs: adjacency list

Adjacency list representations allow for linear time graph algorithms (given that the algorithm has linear complexity).

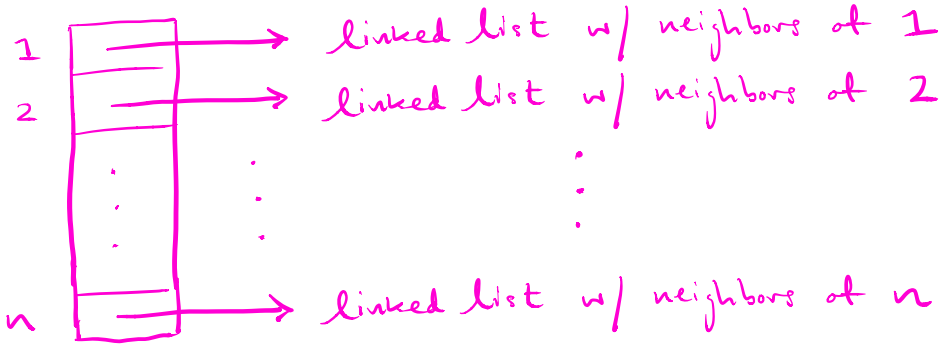
An alternative representation for graph $G = (V, E)$, $|V| = n$, $|E| = m$ is as follows.

Adjacency list: recall that vertex j is adjacent to vertex i if $(i, j) \in E$; then the adjacency list for vertex i is simply the list of vertices adjacent to vertex i.

The adjacency list representation of a graph consists of an array A with n entries such that $A[i]$ points to the adjacency list of vertex i.

For all of our linear time graph algorithms, we will be using adjacency list representations of graphs.

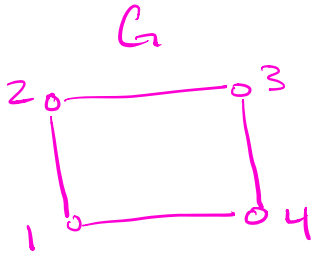
Adjacency list:



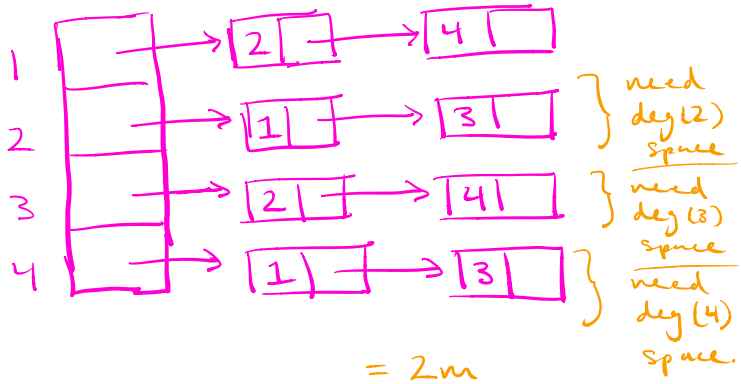
an array with n entries, each of which corresponds to a vertex in the graph.

Every entry contains a pointer to a linked list with the neighbors of that node.

Ex:



need deg(1) space to store linked list for node 1.



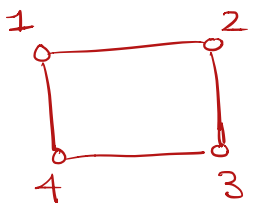
Space Required for this representation: $O(n) + \underbrace{\sum \deg(v)}_{= 2m}$

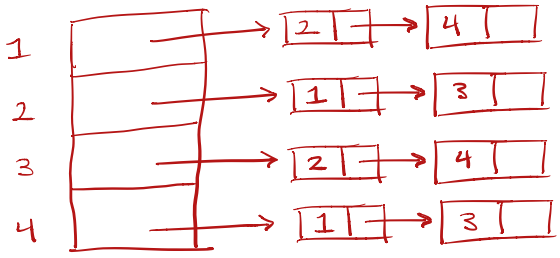
$O(n)$: for the array

$\sum \deg(v)$: for all the linked lists.

$= O(n+m)$

Adjacency list Example :





Array will be of size 4:
One entry for each vertex.

(# of nodes in the adjacency list of 1 is deg(1))

Space Requirement : $O(n) + \sum_{v \in V} \deg(v)$

$O(n)$ { In general, you have an array of size n, 1 entry for each vertex in the graph.

Space needed to store all the adjacency lists

$\sum \deg(v)$ { Space req'd to store all adjacency lists in the array: $\deg(v_k)$ for each node v_k, $k \in \{1, \dots, n\}$

$\Rightarrow$ Space Req'ment :

$$O(n) + \underbrace{\sum \deg(v)}_{= 2m} = O(n) + O(m) = O(n+m)$$

Linear → Good !!

- In terms of space, the adjacency list representation is excellent.

* Linear space requirement.

- Any disadvantages?

  - Finding an edge takes time.

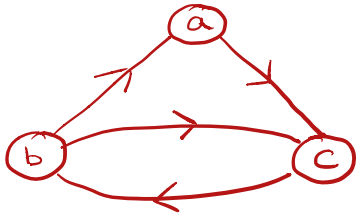
    * Linear search → will have to scan the adjacency list of the particular node and that list can have at most $n-1$ nodes — so that could take linear time.

Adjacency matrix representation, however, only takes $O(1)$ time.

Your choice of representation should depend on whether or not you're going to ask about an edge's existence many times.

Adjacency List Example

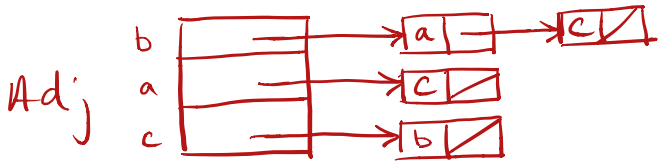
G:



$Adj[b] = \{a, c\}$

$Adj[a] = \{c\}$

$Adj[c] = \{b\}$



For $w \in Adj[s]$:

s.paths = s.paths + CountSimplePaths(z, t)

Space requirements for adjacency list

Need

- an array of n pointers: $O(n)$ space; plus
- the sum of the lengths of all adjacency lists:
  - directed G: maintain the list of vertices with incoming edges from v and the list of vertices with outgoing edges to v.
    - length of adjacency lists of $v = outdeg(v) + indeg(v)$
    - length of all adjacency lists $= \sum_v outdeg(v) + indeg(v) = 2m$
  - undirected G: maintain the list of vertices adjacent to v
    - length of adjacency list of $v = deg(v)$
    - length of all adjacency lists $= \sum_v deg(v) = 2m$.

$\Rightarrow$ Total space: $O(n + m)$

Pros/cons of representing G by its adjacency list

Representing $G = (V, E)$, $|V| = n$, $|E| = m$ using adjacency lists has the following pros/cons.

Advantages:

- allocates no unnecessary space: $O(n + m)$ space to represent a graph on n vertices and m edges
- suitable for linear or near-linear time algorithms

Drawbacks:

- searching for an edge can take $O(n)$ time

As oppose to O(1) time for adjacency matrix.

Adjacency list vs Adjacency matrix

We prefer **adjacency matrix** when

- we need determine quickly whether an edge is in the graph
- the graph is dense
- the graph is small (it is a simpler representation).

We use an **adjacency list** otherwise.

- if the graph is sparse.
- for linear time algorithms.

Today

1. Graphs
2. Representing graphs
3. Breadth-first search (BFS)
4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Searching a graph

Want to find all vertices reachable from s.

Given a transportation network and a city s, we want to find all cities reachable from s.

This problem is known as *s-t* connectivity.

Input: a graph $G = (V, E)$, a vertex $s \in V$

Output: all vertices $t \in V$ such that there is a path from s to t

Notice that if we solve this problem for an undirected graph, we get the connected component of s.
The same is NOT true for directed graphs.

An algorithm for s-t connectivity: breadth-first search

For undirected graphs, the output of BFS is the connected component of s.

Breadth-first search (BFS): explore G starting from s **outward in all possible directions**, adding reachable nodes one **layer** at a time.

- First add all nodes that are joined by an edge to s: these nodes form the first layer.
  If G is unweighted, these are the nodes at distance 1 from s.
- Then add all nodes that are joined by an edge to a node in the first layer: these nodes form the second layer.
  If G is unweighted, these are the nodes at distance 2 from s.
- And so on and so forth.

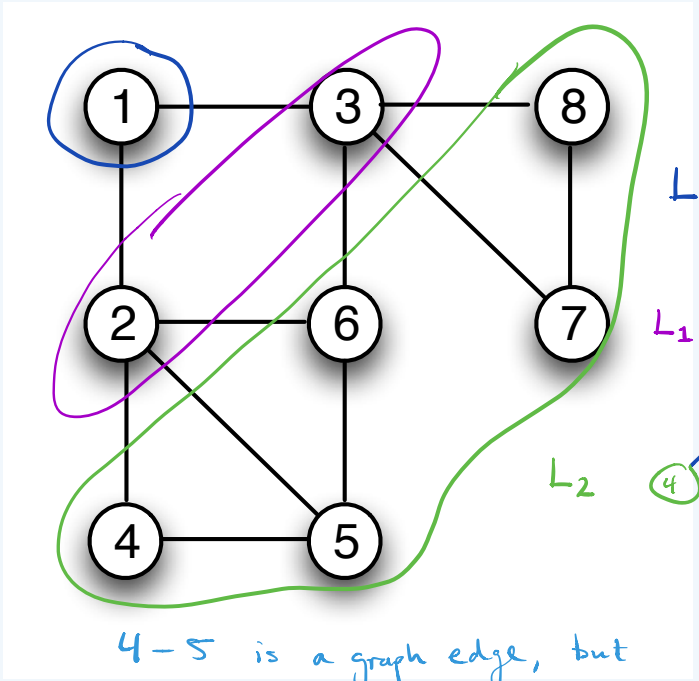
1st. Adds all immediate neighbors of s. These nodes form the first layer.

Example graph G_1

BFS$(G_1, 1)$

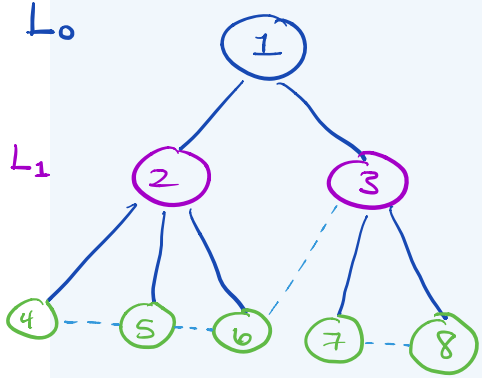
2nd: Explores the immediate neighbors of the nodes in the 1st layer.

Break ties toward the node w/ the smallest index.



Ex: Start @ node 1.

BFS$(G_1, 1)$



The nodes in layer 1 (L_1) are distance 1 from s.:

$\forall v \in L_i$,
$dist(s, v) = i$

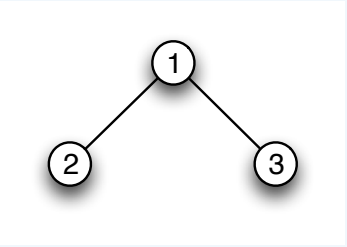
4 – 5 is a graph edge, but not an output that belongs to the output of BFS$(G, 1)$, so that edge gets a dotted line to indicate this.

The output of BFS is a tree. Indeed, the new graph from BFS is connected (every new node in the output is a child of some existing node), and acyclic because every

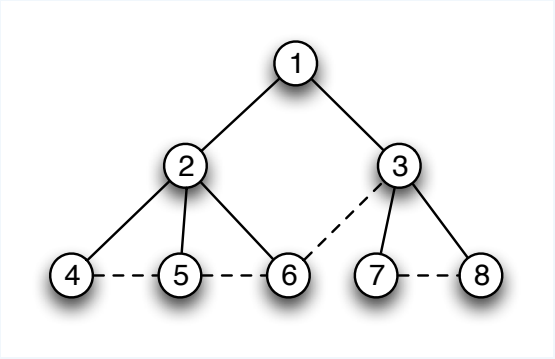
node has exactly one parent.

The BFS layers for the example graph G_1

1st layer



2nd layer



Solid edges appear in G_1 and in the output of BFS.
Dotted edges appear in G_1 but do not in the output of BFS.
Ties are broken by selecting the node with the smallest index.

Properties of the layers of the output of BFS

Formally,

- **Layer** L_0 contains s.
- **Layer** L_1 contains all nodes v such that $(s, v) \in E$.
- For $i \geq 1$, **layer** L_i contains all nodes that
  1. have an edge from a node in layer L_{i-1}; and
  2. do **not** belong to a previous layer.

Claim

Fact 5. CLAIM:

L_i is the set of nodes that are at distance i from s.

Equivalently, the length of the shortest s-v path for all $v \in L_i$ equals i.

PROVE BY Induction : Base Case is layer 0 (L_0):
s is distance 0 from itself. ✓

Induction Hypothesis: $L_i = \{v \in V : d(s,v) = i\}$.

Proof of Fact 5

Inductive Step: Consider the nodes in L_{i+1}

All nodes in L_{i+1} are joined to a node in L_i by exactly one edge. Since $d(s,v) = i$ for all $v \in L_i$ and $d(v,u) = 1$ for all $u \in L_{i+1}$ and $v \in L_i$, then $d(s,u) = i+1$ // □

By induction.

- **Basis:** true for layer L_0.
- **Hypothesis:** suppose L_i is the set of nodes at distance i from s, for some $i \geq 0$.
- **Step:** The only vertices added to L_{i+1} are those that
  1. have an edge from a node in L_i; and
  2. do not have an edge from a node in **any previous** layer L_k, for $k < i$

  Then L_{i+1} contains the nodes that are at distance $1 + i$ from s.

Output of BFS

For undirected graphs,
If a node does not belong to the connected component of the origin node, that node will not appear in the BFS output.

- *When is a node v added to the graph produced by BFS?*
  - Add a node when exploring a neighboring node in previous layer.
- *Why would a node fail to appear in the BFS output?*
  if it is not reachable from the origin node.
- *Which problems are solved by BFS?*
  - s-t connectivity
  - shortest s-v paths in unweighted graphs
- *Why is the graph produced by BFS a tree?*

[Suppose G is undirected.

- *Consider an edge $(u, v) \in E$ that does* **not** *appear in the BFS tree, while u and v both appear in the BFS tree, say at layers L_i and L_j respectively. How far apart can these layers be?* $\Rightarrow$ at most 1 layer apart (0 or 1)

BFS is not suitable for shortest path problems on weighted graphs, but is for unweighted graphs.

- Could $i-j=0$? Yes ✓
- Could $i-j=1$? Yes ✓
- Could $i-j=2$? No ✗

NO!

If you're running BFS, if $uv \in E$, then u and v will be at most one layer apart in the BFS tree.

- They can be either in the same layer or 1 layer apart. This is a property of <u>all</u> BFS trees.

Output of BFS

- *When is a node v reachable from s added to the BFS graph?*

  A node v is added to the tree when it is *discovered*, that is, when some node u is being *explored* and an edge (u, v) is found for the first time. Then u becomes the parent of v since u is responsible for *discovering* v.

- *Why would a node fail to appear in the BFS graph?*

  Because there is no path from s to that node.

BFS in G started at vertex s answers

1. s-t connectivity;
2. shortest s-v paths in *unweighted* graphs. (from the origin vertex)

BFS solves shortest s-v paths for all nodes in G.

Properties of the BFS tree (cont'd)

- *Why is the graph produced by BFS a tree?*

  Because it is connected and acyclic.

  acyclic because we only connect a node w/ its parent.

  * The order in which the vertices are visited matters for the final tree but **not** for the distances computed.

- *What are graph edges that do not appear in the BFS tree?*

  They are either
  - edges between nodes in the same layer; or
  - edges between nodes in adjacent layers.

This is a property of **all** BFS trees.

Holds for undirected graphs.

Non-tree edges in BFS trees

Claim 1.

Let T be a BFS tree, let x and y be nodes in T belonging to layers L_i and L_j respectively, and let (x, y) be an edge in G. Then i and j differ by at most 1.

Proof of Claim 1

Proof.

Without loss of generality, assume that x was *discovered* **first** during execution of BFS. Therefore, $i \leq j$ and x is *explored* before y. When x is *explored*, there are two possibilities.

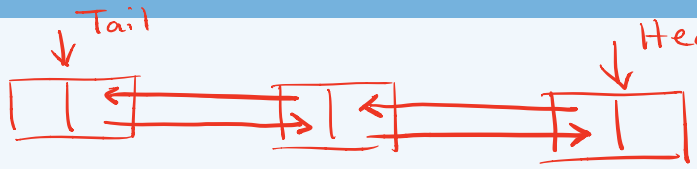
1. y is *discovered* then, hence y is added at layer L_{i+1} as a child of x and the claim holds. Or;
2. y was *discovered* **before** x is *explored*. Thus y appears in the tree at some layer L_j with $j \leq i + 1$. Since $j \geq i$, the claim holds.

□

Implementing BFS

Queue discussed in textbook (Ch. 6 or 10)

Queue:



Add to the tail, extract from the head

We need to store the nodes *discovered* at layer L_i in order to *explore* them later (once we have finished exploring layer L_i).

To this end we use a queue.

{ queue is a first-in-first-out data structure.

- FIFO data structure: add to the end of the queue, extract from the head of the queue.
- Implemented as a double-linked list: maintain explicit pointers to the head and tail elements. Then **enqueue** and **dequeue** operations take constant time.

dequeue: extracting an element from the queue.

enqueue: adding an element to the queue.

If you maintain explicit pointers to the head and tail of the queue, then the operation of adding an element to the tail takes O(1) time and extracting an element at the head takes constant O(1) time.

Pseudocode for BFS —running time?

initializing an array is a for-loop in the RAM model.

BFS$(G = (V, E), s \in V)$

O(n) time

array $discovered[V]$ initialized to 0 ← No node has been disc. yet.

array $dist[V]$ initialized to ∞ ← Because a priori, you don't know if any node is reachable from s.

array $parent[V]$ initialized to NIL ← Don't know if any node is reachable from s.

queue q

O(1)

$discovered[s] = 1$

$dist[s] = 0$

$parent[s] = NIL$ ← root of the tree

enqueue(q, s) ← adds s to the queue

while $size(q) > 0$ **do**

extract top element (from Head) → $u =$dequeue(q) O(1)

O(n)

for $(u, v) \in E$ **do** ← for a fixed $(u,v) \in E$

O(deg(u))

if $discovered[v] == 0$ **then**

$discovered[v] = 1$

$dist[v] = dist[u] + 1$

$parent[v] = u$

enqueue(q, v) ← adds v to the queue. (v in next layer)

O(1) for fixed u,v

end if

end for

end while

How many times do we enter the while-loop? That is, how many times can we enqueue every node?

At most n-times

enqueue each node at most once, and thus dequeue every node at most once.

Pseudocode for BFS —running time?

```
BFS(G = (V, E), s ∈ V)
    array discovered[V] initialized to 0
    array dist[V] initialized to ∞
    array parent[V] initialized to NIL
    queue q
    discovered[s] = 1
    dist[s] = 0
    parent[s] = NIL
    enqueue(q, s)
    while size(q) > 0 do
        u = dequeue(q)
        for (u, v) ∈ E do
            if discovered[v] == 0 then
                discovered[v] = 1
                dist[v] = dist[u] + 1
                parent[v] = u
                enqueue(q, v)
            end if
        end for
    end while
```

Node u has exactly $\deg(u)$ neighbors and for each node u, spend $O(1)$ time.

$\deg(u) \cdot O(1)$ = time spent in the for-loop for fixed node u.

Then,

$$\sum_{u \in V} \deg(u) \cdot O(1)$$

$$= C \cdot \sum_{u \in V} \deg(u)$$

$$= O(m)$$

Therefore,

Running Time $= O(n + m)$

Whenever you can use the degree of a node to determine an upper-bound the # of neighbors of a node, you should do that.

Adequate representation of the BFS tree since it tells you which nodes are parents and children and the layers they are in.

- REMEMBER: Initializing an array of size n requires $O(n)$ time, since each entry of the array must be initialized. Think of it like a for-loop that initializes each of the n positions over n iterations.

NOTE: For dense graphs $O(n+m) = O(n^2)$

This is not the case for sparse graphs.

Today

1. Graphs
2. Representing graphs
3. Breadth-first search (BFS)
4. Applications of BFS
   - Connected components in undirected graphs
   - Testing bipartiteness

Connected components in undirected graphs

Say,

Want an algorithm to decompose G into all its cc

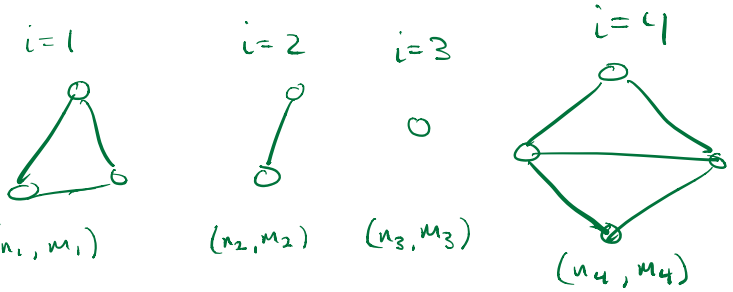
BFS?

4 connected components.

- BFS(s) naturally produces the connected component $R(s)$ of vertex s, that is, the set of nodes reachable from s.
  - Exploring the vertices in a different *order* can yield different algorithms for finding connected components (*coming up in the next lecture*).
- *How can we produce all the connected components of G?*
  - Consider two distinct vertices s and t in G: *how do their connected components compare?*

You should be able to decompose a graph into its connect components in linear time: $O(n+m)$

Say $G =$



(n_i, m_i) = # of vertices and edges in connected component i.

Then, the running time to find connected component $i = 1$ is $O(n_1 + m_1)$

and in general is $O(n_i + m_i)$

(once you've found a node in the respective component.)

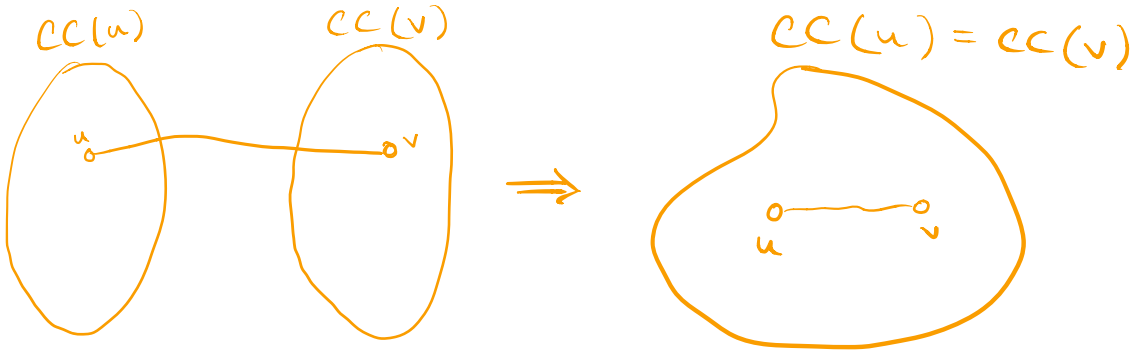
Need to find <u>any</u> node in the connected component <u>first</u>.

Assuming you can find a node in each CC, the running time of the entire algorithm is linear in the size of the graph:

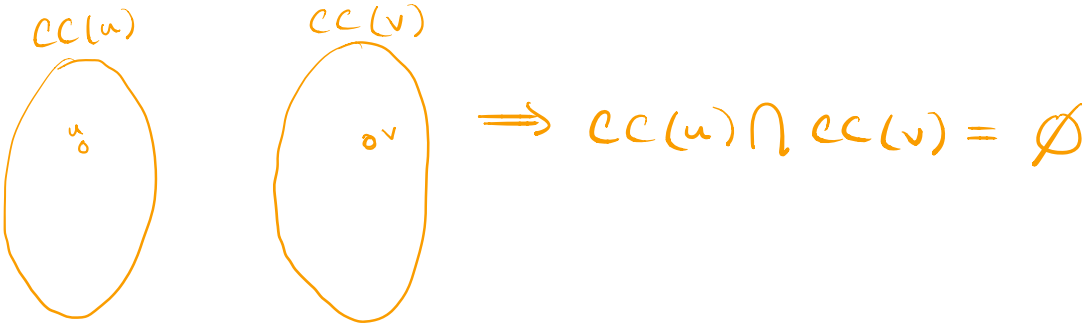
$$O\left(\sum_{i=1}^{c} n_i + \sum_{i=1}^{c} m_i\right) = O(n + m)$$

For a graph $G = (V, E)$, consider the connected components of any two vertices $u, v \in V$. There are two cases:

1. If $\exists$ (u,v)-path: then $CC(u) = CC(v)$. That is, u and v must be in the same connected component.



2. If $\nexists$ (u,v)-path: $CC(u)$ and $CC(v)$ are disjoint. That is, $\nexists\, x \in V$ such that $x \in CC(u)$ and $x \in CC(v)$.



Proof of 2': (Proof by contradiction)

Suppose to the contrary that $\exists x \in V$ such that $x \in CC(u)$ and $x \in CC(v)$, i.e., $x \in CC(u) \cap CC(v)$.

Since $x \in CC(u)$, there exists a (u,x)-path in G.

Since $x \in CC(v)$, there exists a (x,v)-path in G.

Therefore, there is a path from u to v by taking the (u,x)-path and the (x,v)-path. A contradiction. $\not\Rightarrow$

Hence, if $\nexists$ (u,v)-path in G,

$$CC(u) \cap CC(v) = \emptyset$$

($CC(u)$ and $CC(v)$ are disjoint). $\square$

Connected components of different vertices

Loop over all nodes and run BFS over all undiscovered nodes.

Fact 6.

For any two vertices u and v their connected components are either the same or disjoint.

Proof.

Consider any two nodes s, t such that there is a path between them: their connected components are the same (*why?*).

Now consider any two nodes s, t such that there is no path between them: their connected components are disjoint. If not, there is a node v that belongs to both components, hence a path between s and v and a path between t and v. Then there is a path between s and t, contradiction. □

Finding all the connected components in a graph

Worst case: Every CC is a singleton set. Then BFS runs n times

`AllConnectedComponents`$(G = (V, E))$

1. Start with an arbitrary node s; run `BFS`(G, s) and output the resulting BFS tree as one connected component.

2. Continue with any node u that has not been visited by `BFS`(G, s); run BFS from u and output the resulting BFS tree as one connected component.

3. Repeat until all nodes in V have been visited.

Running time of `AllConnectedComponents` ?

Answer: $O(n + m)$. Why? ?

You start BFS from node 1, and you find the connected component of node 1, CC_1, in time $O(|V_1| + |E_1|)$, where V_1 is the set of nodes in CC_1 and E_1 is the set of edges in CC_1.

Next, start BFS again from undiscovered node in V. Let CC_2 denote the connected component returned from second run of BFS. Then run-time of this execution of BFS is $O(|V_2| + |E_2|)$.

⋮

For the last run of BFS, connected component CC_x is returned w/ run time $O(|V_x| + |E_x|)$.

Total Run-time:

$$O(|V_1| + |E_1|) + O(|V_2| + |E_2|) + \cdots + O(|V_x| + |E_x|)$$
$$= O(\Sigma|V_i| + \Sigma|E_i|) = O(|V| + |E|) = O(n+m)$$

Therefore, this procedure (`AllConnectedComponents`) is linear, assuming you can find the next undiscovered node to continue quickly.

By "quickly" we mean, finding the next node in time $O(n)$ or $O(n+m)$.

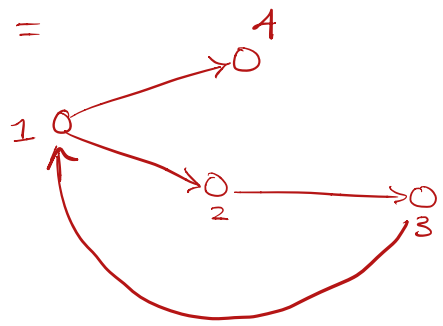
If it takes time $O(n)$ or $O(n+m)$ to find the next node, then the running time of the entire algorithm remains $O(n+m)$.

* Figure out on own how to find the next (undiscovered) node in time $O(n)$.

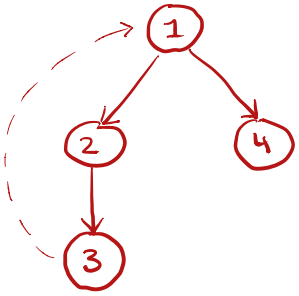
Can we find the connected components of a **directed** graph using BFS?

Yes, we can

$G =$



BFS tree from node 1:



$BFS(G,1) = \{1,2,3,4\} = T_1$

but,

Strongly connected component of node 1 is:

$SCC(G,1) = \{1,2,3\}$

and then

$BFS(G,1) = \{1,2,3,4\}$

which is NOT the SCC of node 1.

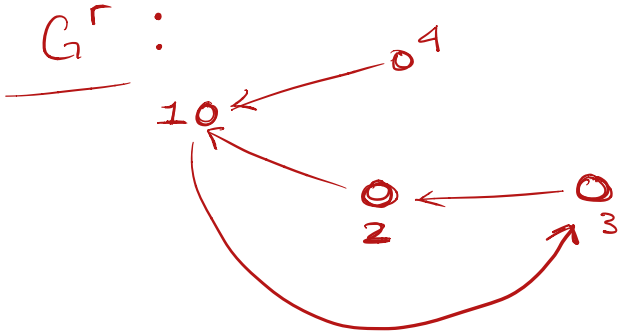
Then, BFS does not output SCC of node for directed graphs, in general. Instead, it returns the set of nodes that are reachable from that node.

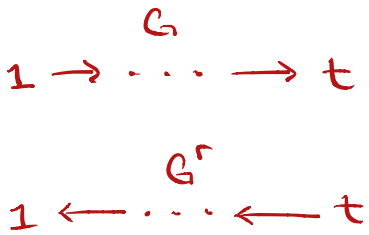
How can we use BFS to find the strongly connected components of a directed graph then??

Solution:

Define the reverse graph G^r, of G, as that w/ the same node set and edge set, but all edges point in the opposite direction:

G^r :

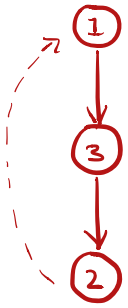




Path from $1 \to \cdots \to t$ in G will correspond to $t \to \cdots \to 1$ in G^r.

Run BFS again, but on G^r from node 1. :

$BFS(G^r, 1)$:



It will give us the set of nodes that are reachable from 1 in G^r and that is the set of nodes in G that can reach 1 (in G), which is exactly the set of nodes we are looking for. Take the intersection of the two outputs, and that will give you the strongly connected components of node 1 in G.

$BFS(G^r, 1) = \{1, 3, 2\} = T_2$ ← set of nodes that can reach back node 1 in G.

$SCC(1) = T_1 \cap T_2 = \{1, 2, 3\}$

Because T_1 is the set of nodes that 1 can reach and T_2 is the set of nodes that can reach 1 in G.

Complexity of this algorithm is

$O(n + m)$ (efficient!)

A more elegant way of finding the SCCs in directed graphs uses the Depth-First-Search (DFS) algorithm. The DFS algorithm will be covered in the next lecture.

Testing bipartiteness & graph 2-colorability

It might be the case that your graph is not actually bipartite either due to data corruption or whatever, so we need to check whether it is or is not.

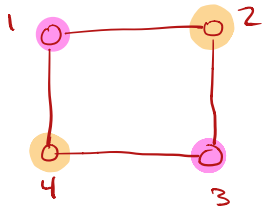
Testing bipartiteness

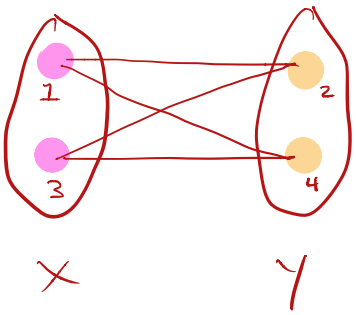
- **Input:** a graph $G = (V, E)$
- **Output:** **yes** if G is bipartite, **no** otherwise

Equivalent problem (*why?*)

- **Input:** a graph $G = (V, E)$
- **Output:** **yes** if and only if we can color all the vertices in G using at most 2 colors –say red and white– so that no edge has two endpoints with the same color.

(2-colorable)

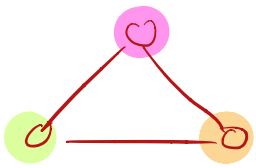




A graph is bipartite iff it is 2-colorable

When is a graph not 2-colorable?

A complete graph on 3 nodes (a triangle)?



K_3

Not 2-colorable.

- Complete graphs, in general, can not be 2-colorable.
- Graphs w/ odd cycles cannot be 2-colorable.

FACT: If G contains an odd-length cycle, then G is not 2-colorable.

Necessary condition for Bipartite graphs:

If G is 2-colorable, then G does not contain any odd-length cycles.

Is this a sufficient condition?

If G does not contain odd-length cycles, then G is bipartite — is this true?

Equivalently (contrapositive): If G is not 2-colorable, then G contains an odd-length cycle?

Going to prove that this is a necessary and sufficient condition. We will use this to determine if a graph is bipartite in our graph algorithms.

Why wouldn't we be able to 2-color a graph?

Fact: If a graph contains an odd-length cycle, then it is not 2-colorable.

So a **necessary** condition for a graph to be 2-colorable is that it does not contain odd-length cycles.

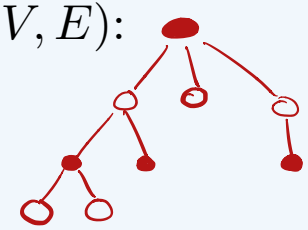
Is this condition also **sufficient**, *that is, if a graph does not contain odd-length cycles, then is it* 2*-colorable?*

In other words, are odd cycles the only obstacle to bipartiteness?

Algorithm for 2-colorability

(Input graph is undirected)

BFS provides a natural way to 2-color a graph $G = (V, E)$:

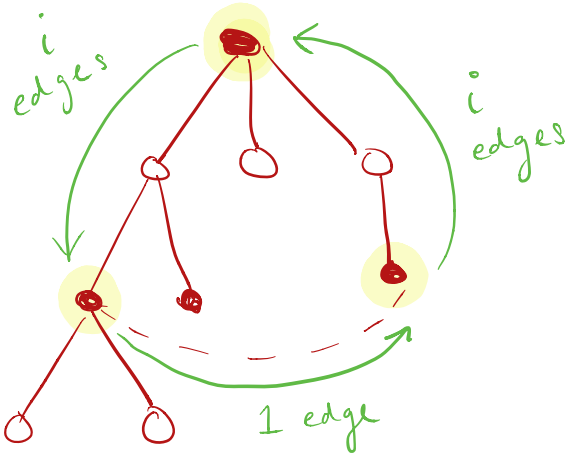


- Start BFS from any vertex; color it red.
- Color white all nodes in the first layer L_1 of the BFS tree. If there is an edge between two nodes in L_1, output **no** and stop. => If no, can we find an odd-length cycle??
- Otherwise, continue from layer L_1, coloring red the vertices in even layers and white in odd layers.
- If BFS terminates and all nodes in V have been explored (hence 2-colored), output **yes**.

efficiency is linear since BFS.

- Think about how to maintain linear time.
- Consider time to compute intersection of 2 sets

If no, then find odd-length cycle



Find the lowest common answer store of between points.
In this case, the lowest common answer store is the root

Follow the path from the root to one end point, that has i edges. Then follow the single edge and then the path from that node back to the root which should also have i edges. This is an odd-length cycle: $2i + 1$ edges

This proves that if G is NOT 2-colorable then G contains an odd-length cycle, and that this is a necessary and sufficient condition.

Analyzing the algorithm

Upon termination of the algorithm

- either we successfully 2-colored all vertices and output **yes**, that is, declared the graph bipartite;
- or we stopped at some level because there was an edge between two vertices of that level and output **no**; in this case, we declared the graph non-bipartite.

This algorithm is **efficient**. *Is it a* **correct** *algorithm for* 2*-colorability?*

Showing correctness

To prove correctness, we must show the following statement: if our algorithm outputs

1. **yes**, the 2-coloring it provides is a valid 2-coloring of G;
2. **no**, G is indeed not 2-colorable by **any** algorithm (e.g., because it contains an odd-length cycle).

The next claim proves that this is indeed the case by examining when the algorithm succeeds to 2-color G. Recall that the only reason why the algorithm fails to 2-color G is because it found an edge between two nodes of the same layer.

Correctness of algorithm for 2-colorability

Claim 2.

Let G be a connected graph, and let L_1, L_2, ... be the layers produced by BFS starting at node s. Then exactly one of the following two is true.

1. *There is no edge of G joining two nodes of the same layer. Then G is bipartite and has no odd length cycles.*

2. *There is an edge of G joining two nodes of the same layer. Then G contains an odd length cycle, hence is not bipartite.*

Corollary 7.

A graph is bipartite if and only if it contains no odd length cycle.

Proof of Claim 2, part 1

1. **Assume** that no edge in G joins two nodes of the same layer of the BFS tree.

By Claim 1, all edges in G not belonging to the BFS tree are

- either edges between nodes in the same layer;
- or edges between nodes in adjacent layers.

Our assumption implies that all edges of G not appearing in the BFS tree are between nodes in adjacent layers.

Since our coloring procedure gives such nodes different colors, the whole graph can be 2-colored, hence it is bipartite.

Proof of Claim 2, part 2

2. **Assume** that there is an edge $(u, v) \in E$ between two nodes u and v in the same layer.

Obviously G is not 2-colorable by our algorithm: both endpoints of edge (u, v) are assigned the same color.

Our algorithm returns **no**, hence declares G non-bipartite.

Can we show existence of an odd-length cycle and prove that G indeed is not 2-colorable by **any** *algorithm?*

Proof of correctness, part 2

- Let u, v appear at layer L_j and edge $(u, v) \in E$.
- Let z be the lowest common ancestor of u and v in the BFS tree (z might be s). Suppose z appears at layer L_i with $i < j$.
- Consider the following path in G: from z to u follow edges of the BFS tree, then edge (u, v) and back to z following edges of the BFS tree. This is a cycle starting and ending at z, consisting of $(j - i) + 1 + (j - i) = 2(j - i) + 1$ edges, hence of odd length.