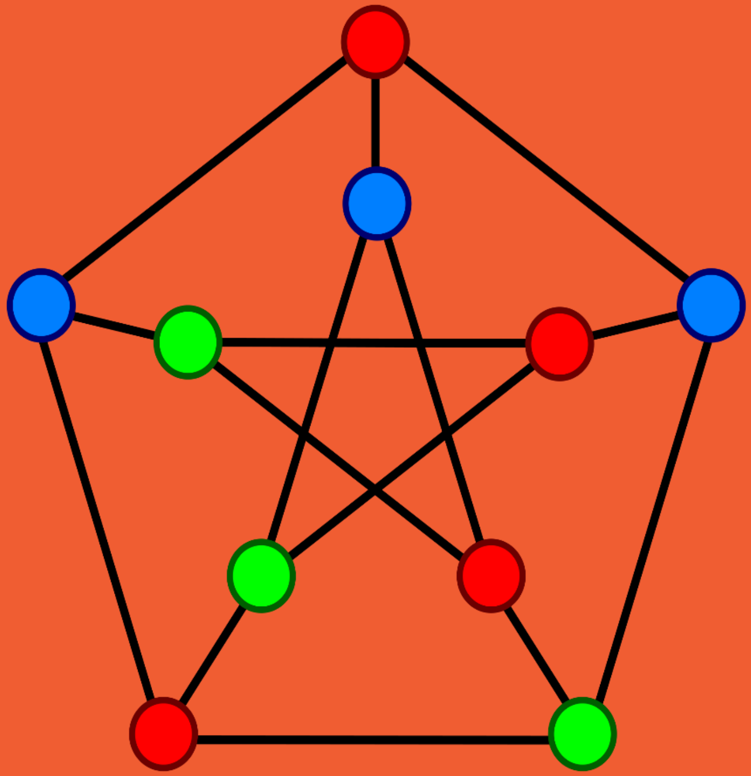
Lecture 2: Graphs

Joachim Gudmundsson



THE UNIVERSITY OF SYDNEY

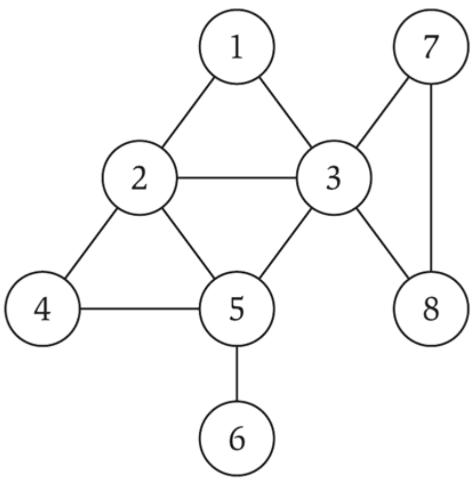
Lecture 2: Graphs

- Definitions
- Representations
- Breadth First Search
- Depth First Search
- Applications

3.1 Basic Definitions and Applications

Undirected Graphs G=(V,E)

- V = nodes (or vertices)
- E = edges between pairs of nodes
- Captures pairwise (symmetric) relationship between objects
- Graph size parameters: $n = |V|$, $m = |E|$



V = { 1, 2, 3, 4, 5, 6, 7, 8 }

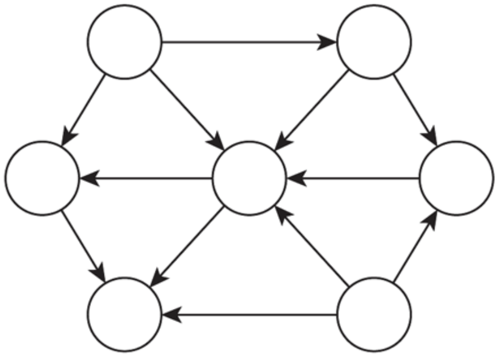
E = { 1-2, 1-3, 2-3, 2-4, 2-5, 3-5, 3-7, 3-8, 4-5, 5-6, 7-8 }

n = 8

m = 11

Directed Graphs G=(V,E)

Edge (u, v) goes from node u to node v.

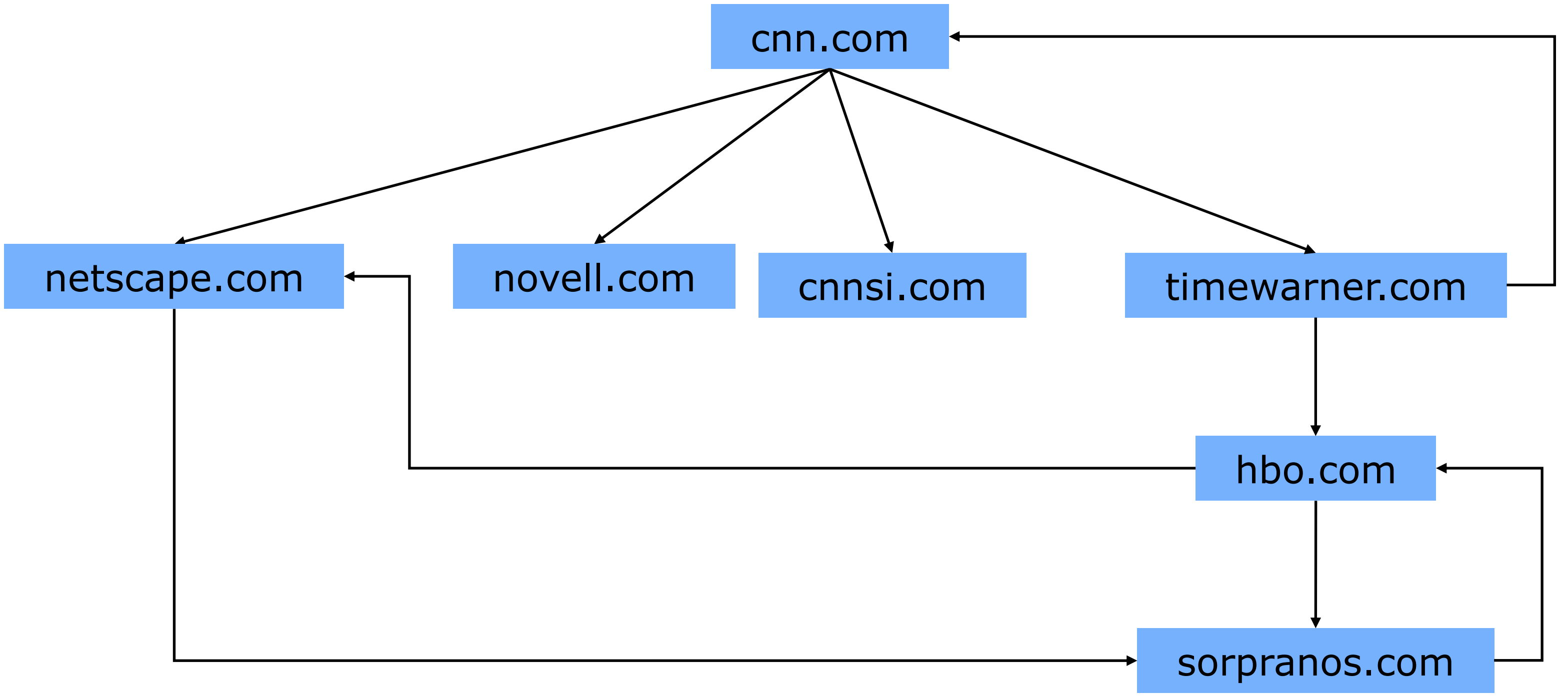


Some Graph Applications

Graph	*Nodes*	*Edges*
transportation	street intersections	highways
communication	computers	fiber optic cables
World Wide Web	web pages	hyperlinks
social	people	relationships
food web	species	predator-prey
software systems	functions	function calls
scheduling	tasks	precedence constraints
circuits	gates	wires

World Wide Web

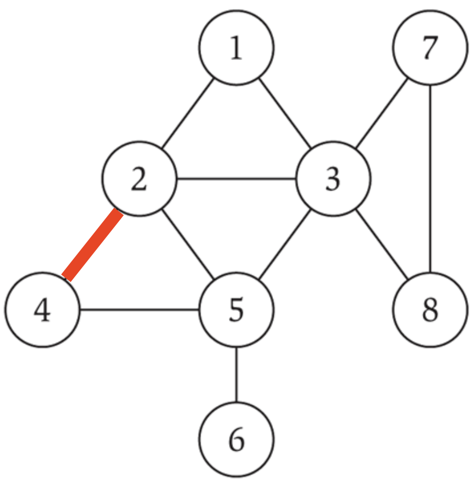
– Node: web page.
– Edge: hyperlink from one page to another.



Graph Representation: Adjacency Matrix

Adjacency matrix. n-by-n matrix with $A_{uv} = 1$ if (u, v) is an edge.

- Two representations of each edge (undirected graph).
- Space proportional to n^2.
- Checking if (u, v) is an edge takes $\Theta(1)$ time.
- Identifying all edges takes $\Theta(n^2)$ time.

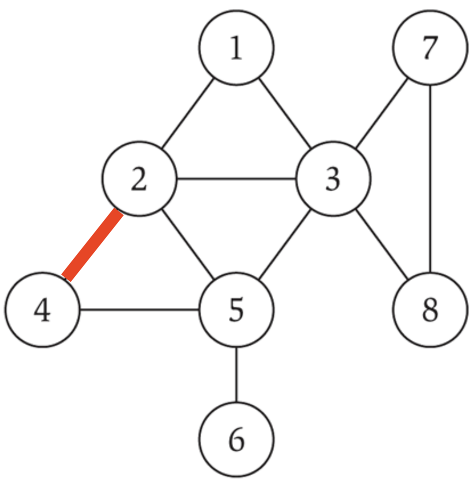


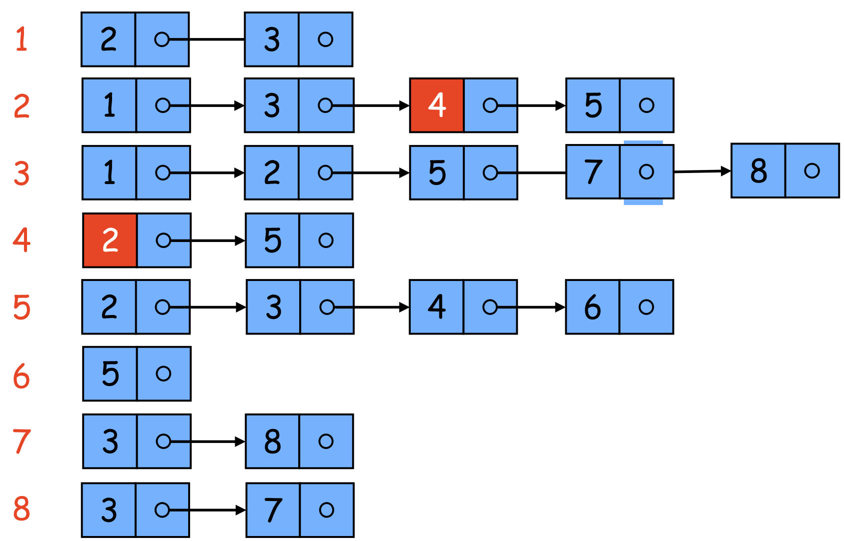
	1	2	3	4	5	6	7	8
1	0	1	1	0	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	0	1	0	1	1
4	0	1	0	0	1	0	0	0
5	0	1	1	1	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	1	0	0	0	0	1
8	0	0	1	0	0	0	1	0

Graph Representation: Adjacency List

Adjacency list. Node indexed array of lists.

- Two representations of each edge (undirected graph).
- Space proportional to m + n.
  degree = number of neighbors of u
- Checking if (u, v) is an edge takes O(deg(u)) time.
- Identifying all edges takes Θ(m + n) time.



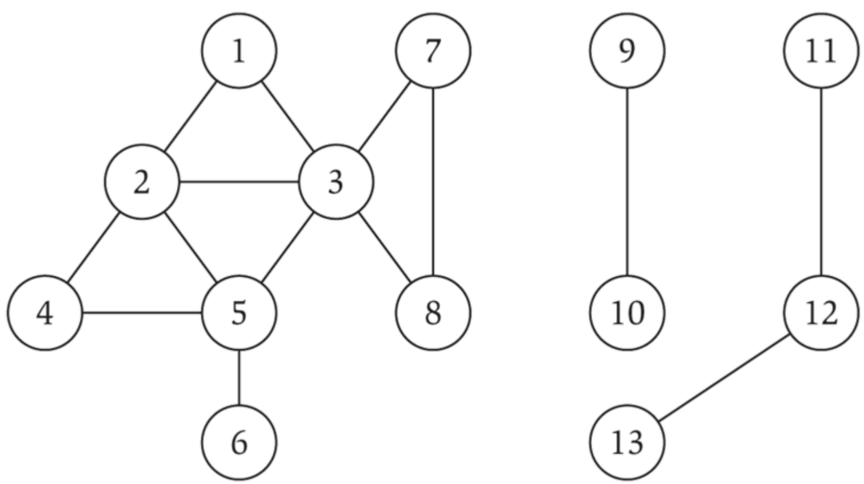


Paths and Connectivity

Definition: A path in an undirected graph $G = (V, E)$ is a sequence P of nodes $v_1, v_2, \ldots, v_{k-1}, v_k$ with the property that each consecutive pair v_i, v_{i+1} is joined by an edge in E.

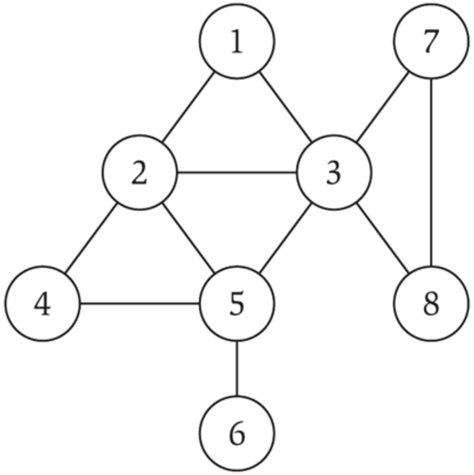
Definition: A path is simple if all nodes are distinct.

Definition: An undirected graph is connected if for every pair of nodes u and v, there is a path between u and v.



Cycles

Definition: A cycle is a path $v_1, v_2, \ldots, v_{k-1}, v_k$ in which $v_1 = v_k$, $k > 2$, and the first k-1 nodes are all distinct.

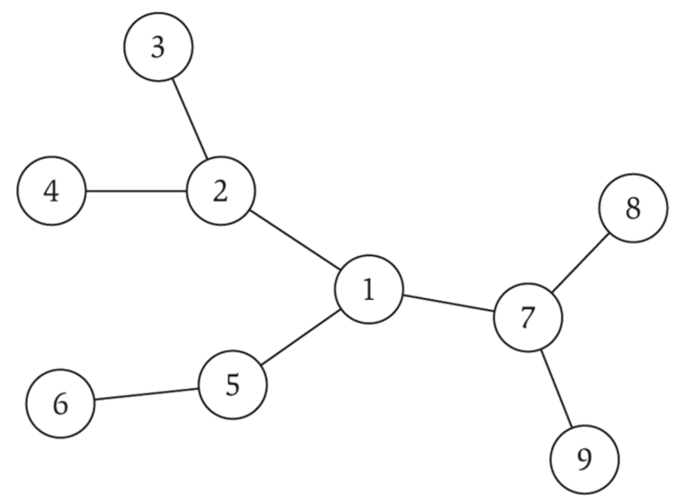


cycle C = 1-2-4-5-3-1

Trees

Definition: An undirected graph is a tree if it is connected and does not contain a cycle.

Number of edges in a tree?



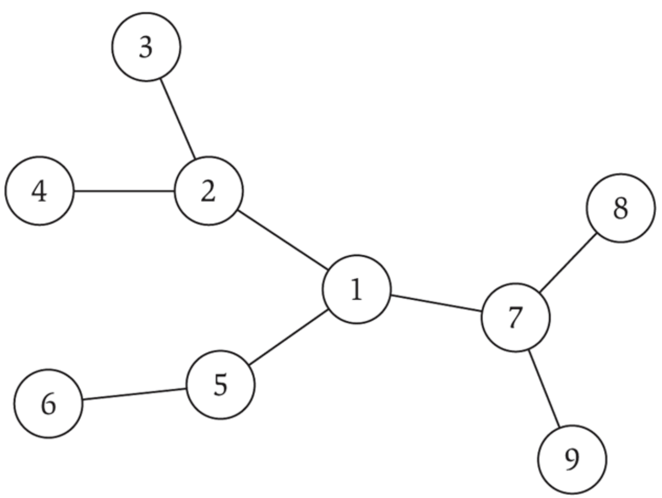
Trees

Definition: An undirected graph is a tree if it is connected and does not contain a cycle.

Number of edges in a tree?

Theorem: Let G be an undirected graph on n nodes. Any two of the following statements imply the third.

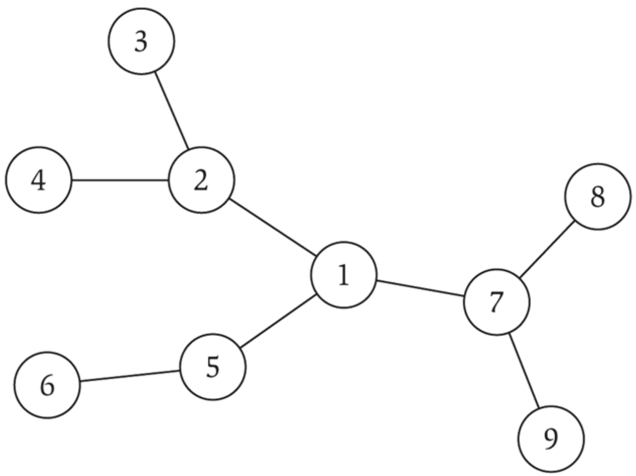
- G is connected.
- G does not contain a cycle.
- G has n-1 edges.



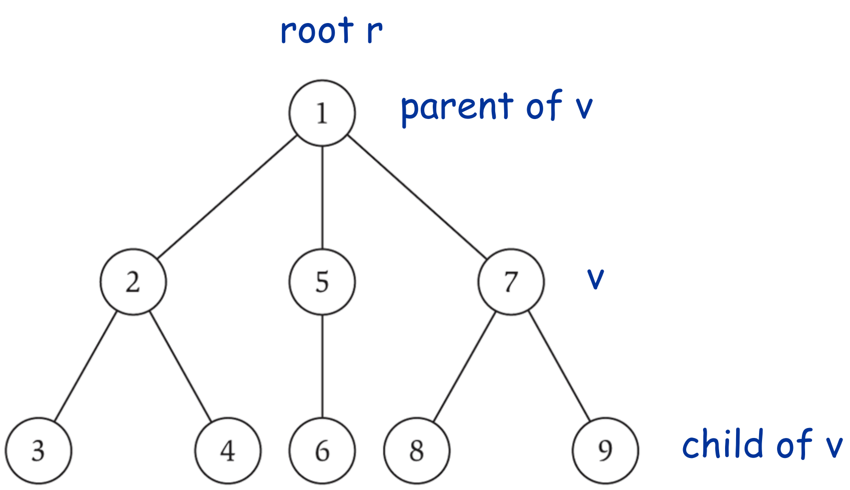
Rooted Trees

Rooted tree. Given a tree T, choose a root node r and orient each edge away from r.

Importance. Models hierarchical structure.



a tree



the same tree, rooted at 1

3.2 Graph Traversal

Connectivity

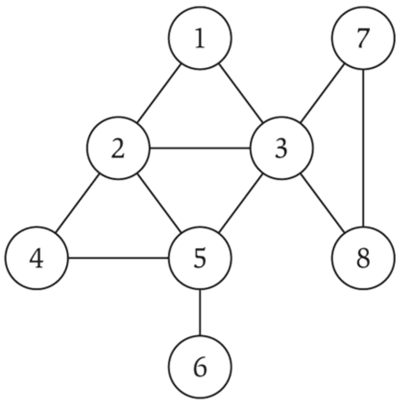
s-t connectivity problem. Given two nodes s and t, is there a path between s and t?

Length of path = number of links along path

s-t shortest path problem. Given two nodes s and t, what is the length of the shortest path between s and t?

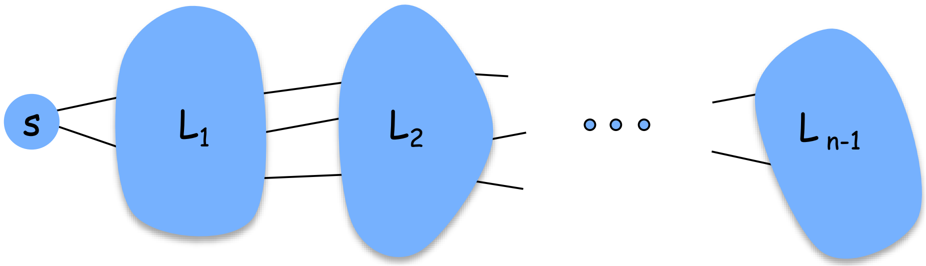
Applications.

Many. For example: Fewest number of hops in a communication network.



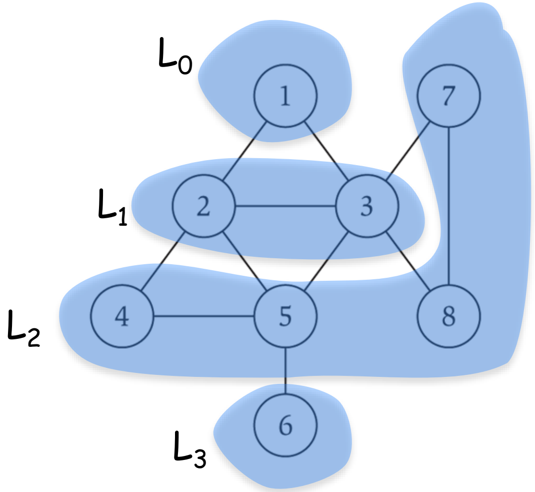
Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



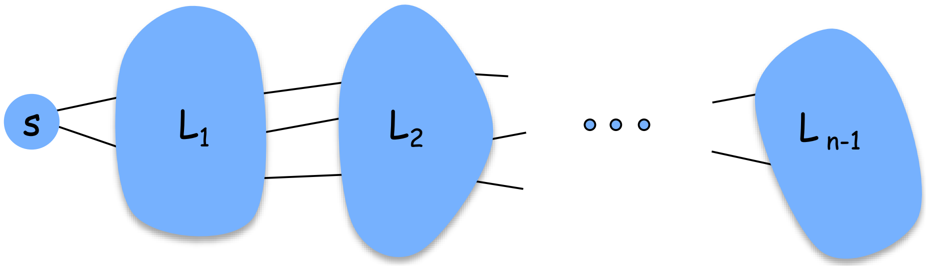
BFS algorithm.

- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.



Breadth First Search

BFS intuition. Explore outward from s in all possible directions, adding nodes one "layer" at a time.



BFS algorithm.

- $L_0 = \{ s \}$.
- L_1 = all neighbors of L_0.
- L_2 = all nodes that do not belong to L_0 or L_1, and that have an edge to a node in L_1.
- L_{i+1} = all nodes that do not belong to an earlier layer, and that have an edge to a node in L_i.

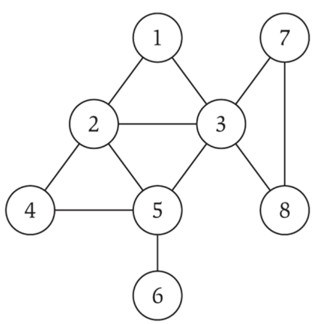
Theorem: For each i, L_i consists of all nodes at distance exactly i from s. There is a path from s to t if and only if t appears in some layer.

```
def BFS(G,s)

    layers = []
    next_layer = [s]
    “mark every vertex except s as not seen”
    while “current layer not empty” do
        layers.append(current_layer)
        for every u in current_layer do
            for every v in neighbourhood of u do
                if “haven’t seen v yet” then
                    next_layer.append(v)
                    “mark v as seen”
        current_layer = next_layer
        next_layer = []

    return layers
```

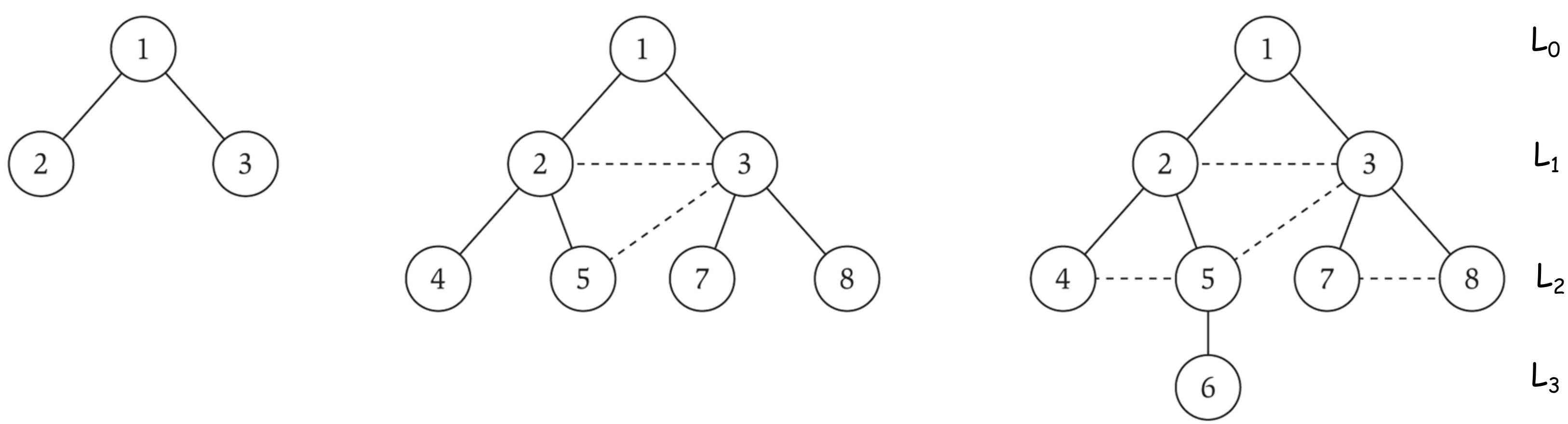
What if G is not connected?

Breadth First Search

BFS produces a tree T rooted at the start vertex on the set of nodes in G reachable from s.



Property. Let T be a BFS tree of G = (V, E), and let (x, y) be an edge of G. Then the level of x and y differ by at most 1.



Breadth First Search: Analysis

Theorem: The above implementation of BFS runs in $O(m + n)$ time if the graph is as an adjacency list.

Proof: Easy to prove $O(n^2)$ running time:

- at most n lists L[i]
- each node occurs on at most one list; for loop runs $\leq n$ times
- when we consider node u, there are $\leq n$ incident edges (u, v), and we spend $O(1)$ processing each edge

– Actually runs in $O(m + n)$ time:

- when we consider node u, there are deg(u) incident edges (u, v)
- total time processing edges is $\Sigma_{u \in V} \deg(u) = 2m$

each edge (u, v) is counted exactly twice
in sum: once in deg(u) and once in deg(v)

```
def BFS(G,s)

  layers = []
  next_layer = [s]
  "mark every vertex except s as not seen"
  while "current layer not empty" do
    layers.append(current_layer)
    for every u in current_layer do
      for every v in neighbourhood of u do
        if "haven't seen v yet" then
          next_layer.append(v)
          "mark v as seen"
    current_layer = next_layer
    next_layer = []

  return layers
```

This takes $O(|V|)$ time

This loop takes $O(|N(u)|)$ time

Adding up over all u, we get $O(\Sigma_u |N(u)|)=O(|E|)$

BFS implementation

Complexity? Depends on graph representation

Traverse all neighbours of a node u:

- Adjacency list: O(number of neighbours) = $O(|N(u)|)$
- Adjacency matrix: $O(n)$

Check if u and v are connected by an edge:

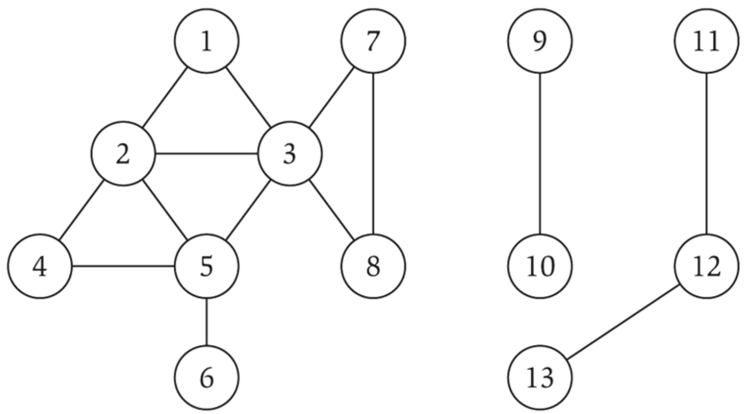
- Adjacency list: O(number of neighbours) = $O(|N(u)|)$ or $O(|N(v)|)$
- Adjacency matrix: $O(1)$

Space:

- Adjacency list: $O(|V|+|E|)$
- Adjacency matrix: $O(|V|^2)$

Connected Component

Find all nodes reachable from s.

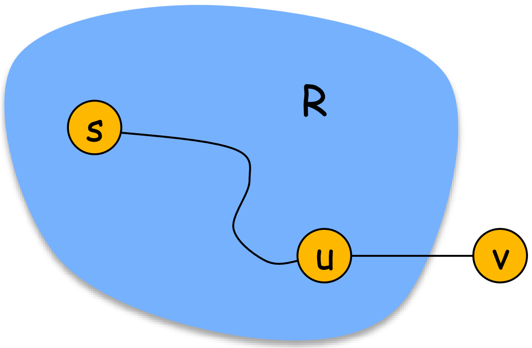


Connected component containing node 1
$= \{ 1, 2, 3, 4, 5, 6, 7, 8\}$

Connected Component

Find all nodes reachable from s.

R will consist of nodes to which s has a path
Initially $R = \{s\}$
While there is an edge (u, v) where $u \in R$ and $v \notin R$
  Add v to R
Endwhile



it's safe to add v

Theorem: Upon termination, R is the connected component containing s.

Shortest paths

The shortest path between two nodes u, v in a graph G, is the path with the minimum number of edges that connects u and v (if it exists).

The shortest path problem:

Input: a graph G=(V,E), and a node s in V

Output: the length of the shortest path between s and all other nodes in V.

Shortest paths by BFS

Recall the BFS algorithm

```
def BFS(G,s)

   layers = []
   next_layer = [s]
   “mark every vertex except s as not seen”
   while “current layer not empty” do
        layers.append(current_layer)
        for every u in current_layer do
           for every v in neighbourhood of u do
               if “haven’t seen v yet” then
                  next_layer.append(v)
                  “mark v as seen”
        current_layer = next_layer
        next_layer = []

   return layers
```

Shortest paths by BFS

Compute the shortest paths from a given node s to all other nodes

Let dist[u] = shortest path distance (hop distance)
from s to u

```
def BFS(G,s)

    layers = []
    next_layer = [s]
    “mark every vertex except s as not seen”
    while “current layer not empty” do
        layers.append(current_layer)
        for every u in current_layer do
            for every v in neighbourhood of u do
                if “haven’t seen v yet” then
                    next_layer.append(v)
                    “mark v as seen”
        current_layer = next_layer
        next_layer = []

    return layers
```

```
def BFS(G,s)

    layers = []
    next_layer = [s]
    for all u set dist[u] = infinity
    dist[s] = 0
    "mark every vertex except s as not seen"
    while "current layer not empty" do
        layers.append(current_layer)
        for every u in current_layer do
           for every v in neighbourhood of u do
               if "haven't seen v yet" then
                  next_layer.append(v)
                  "mark v as seen"
        current_layer = next_layer
        next_layer = []

    return layers
```

Initialize dist[ ]

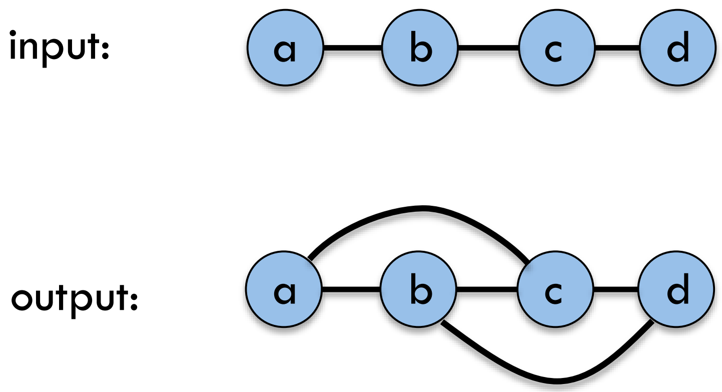
```
def BFS(G,s)

    layers = []
    next_layer = [s]
    for all u set dist[u] = infinity
    dist[s] = 0
    “mark every vertex except s as not seen”
    while “current layer not empty” do
        layers.append(current_layer)
        for every u in current_layer do
            for every v in neighbourhood of u do
                if “haven’t seen v yet” then
                    next_layer.append(v)
                    “mark v as seen”
                    dist[v] = dist[u] + 1
        current_layer = next_layer
        next_layer = []

    return layers
```

```
def ShortestPath(G,s)

   layers = []
   next_layer = [s]
   for all u set dist[u] = infinity
   dist[s] = 0
   “mark every vertex except s as not seen”
   while “current layer not empty” do
       layers.append(current_layer)
       for every u in current_layer do
           for every v in neighbourhood of u do
               if “haven’t seen v yet” then
                   next_layer.append(v)
                   “mark v as seen”
                   dist[v] = dist[u] + 1
       current_layer = next_layer
       next_layer = []

   return dist
```

Transitive closure of a graph

The transitive closure graph of G is a graph G':

- with the same vertices as G, and
- with an edge between all pairs of nodes that are connected by a path in G



Closure graph by BFS

How do we change BFS to compute the closure?

```
def BFS(G,s)

    layers = []
    next_layer = [s]
    “mark every vertex except s as not seen”
    while “current layer not empty” do
        layers.append(current_layer)
        for every u in current_layer do
            for every v in neighbourhood of u do
                if “haven’t seen v yet” then
                    next_layer.append(v)
                    “mark v as seen”
        current_layer = next_layer
        next_layer = []

    return layers
```

```
def BFS_closure(G,s)

    layers = []
    next_layer = [s]
    “mark every vertex except s as not seen”
    while “current layer not empty” do
        layers.append(current_layer)
        for every u in current_layer do
           for every v in neighbourhood of u do
               if “haven’t seen v yet” then
                  next_layer.append(v)
                  “mark v as seen”
                  add edge (s,v) to the graph G
        current_layer = next_layer
        next_layer = []

    return the new graph
```

For s in V:
BFS_closure(G,s)

```
def BFS_closure(G,s)

    layers = []
    next_layer = [s]
    “mark every vertex except s as not seen”
    while “current layer not empty” do
        layers.append(current_layer)
        for every u in current_layer do
            for every v in neighbourhood of u do
                if “haven’t seen v yet” then
                    next_layer.append(v)
                    “mark v as seen”
                    add edge (s,v) to the graph G
        current_layer = next_layer
        next_layer = []

    return the new graph
```

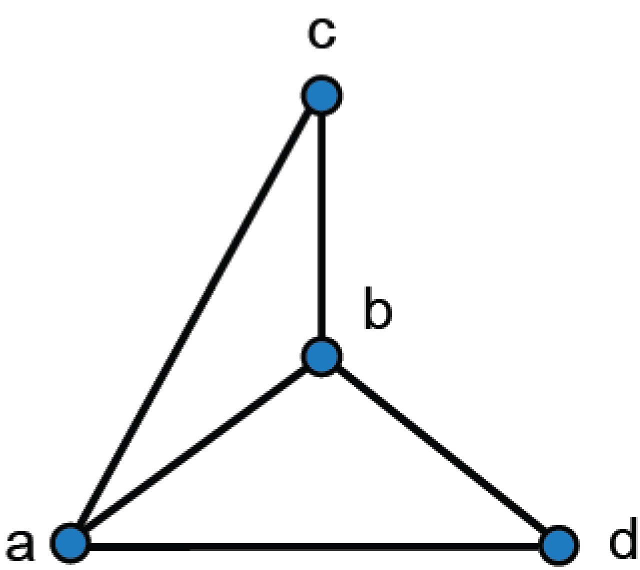
Running time? $O(|V|\cdot(|V|+|E|))$

For s in V:
    BFS_closure(G,s)

DFS – Depth first search

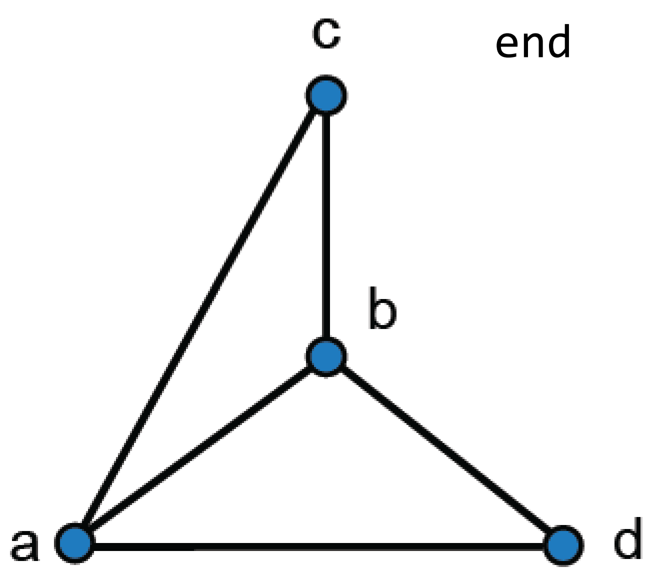
Algorithm: Pick a starting vertex, follow outgoing edges that lead to new vertices, and backtrack whenever “stuck”.

```
Algorithm DFS(G,u)
  Input: graph G(V,E) and a vertex u in V

  begin
     mark u as visited
     for each edge (u,v) in E do
        if v has not been visited then
           DFS(G,v)
  end
```



What if G is not connected?

```
Algorithm DFS(G,s)
  Input: graph G(V,E) and a vertex s in V

  begin
     initialise a stack S with node s
     while S is not empty do
        u=pop(S)
        if u not visited then
           set u as visited
           for each edge (u,v) do
              add v to S
end
```



Properties of DFS

Running time: $O(n+m)$

Subset of edges in DFS that "discover a new node" form a forest (a collection of trees).

A graph is **connected** if and only if DFS results in a **single tree.**

Each tree in the DFS result corresponds to a connected component

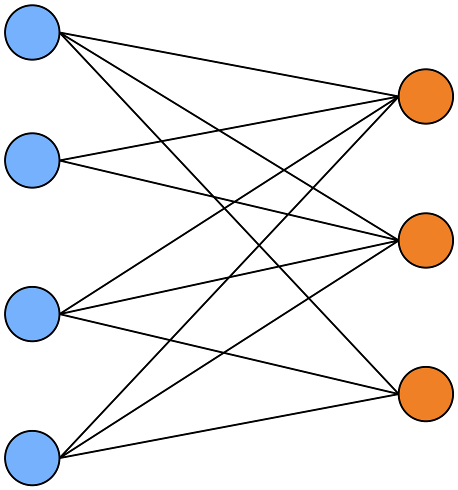
3.4 Testing Bipartiteness

Bipartite Graphs

Definition: An undirected graph G = (V, E) is bipartite if the nodes can be colored red or blue such that every edge has one red and one blue end.

Applications

- Stable marriage: men = red, women = blue.
- Scheduling: machines = red, jobs = blue.

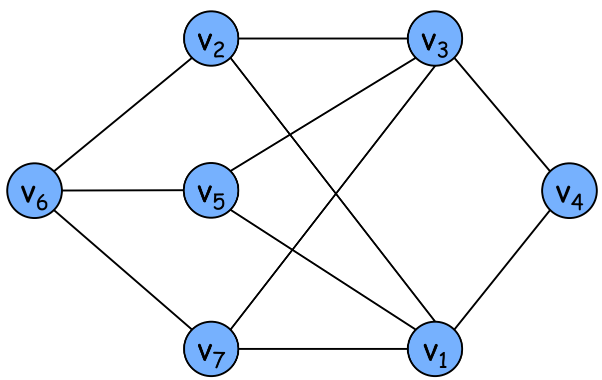


a bipartite graph

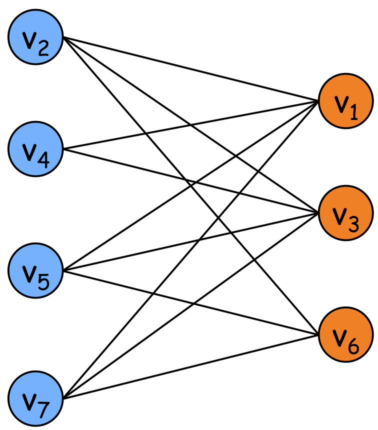
Testing Bipartiteness

Testing bipartiteness. Given a graph G, is it bipartite?

– Many graph problems become:
  - easier if the underlying graph is bipartite (matching)
  - tractable if the underlying graph is bipartite (independent set)
– Before attempting to design an algorithm, we need to understand structure of bipartite graphs.



a bipartite graph G

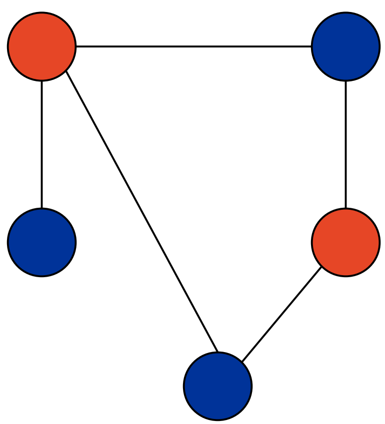


another drawing of G

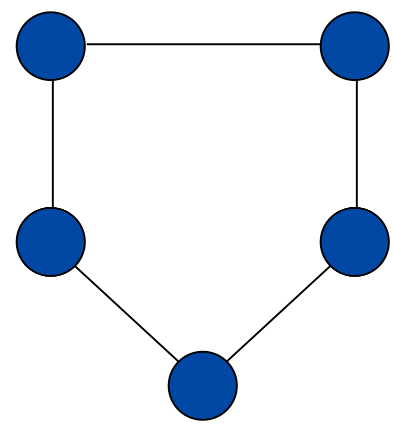
An Obstruction to Bipartiteness

Lemma: If a graph G is bipartite, it cannot contain an odd length cycle.

Proof: Not possible to 2-color the odd cycle, let alone G.



bipartite
(2-colorable)



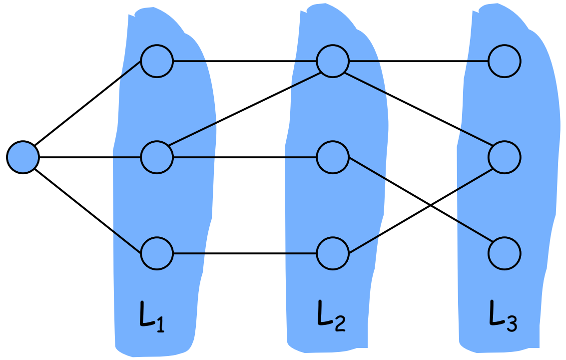
not bipartite
(not 2-colorable)

Bipartite Graphs

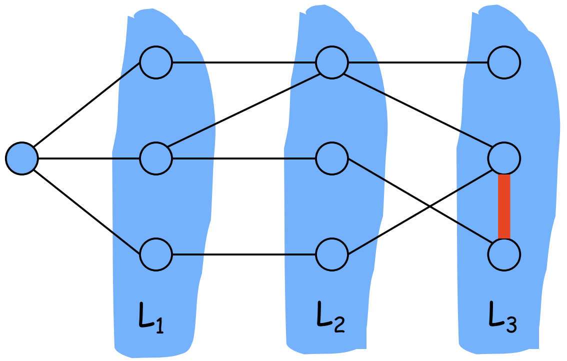
Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).



Case (i)



Case (ii)

Bipartite Graphs

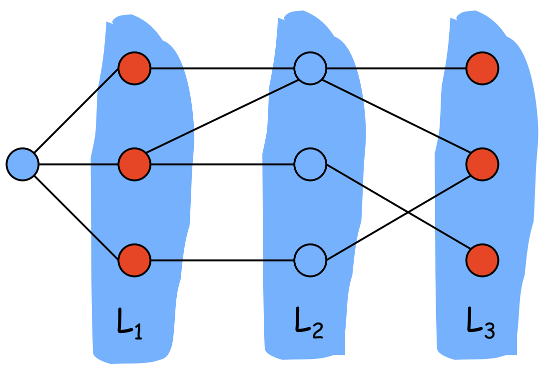
Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

Proof: Case (i)

- Suppose no edge joins two nodes in the same layer.
- By previous lemma, this implies all edges join nodes on adjacent level.
- Bipartition: red = nodes on odd levels, blue = nodes on even levels.



Case (i)

Bipartite Graphs

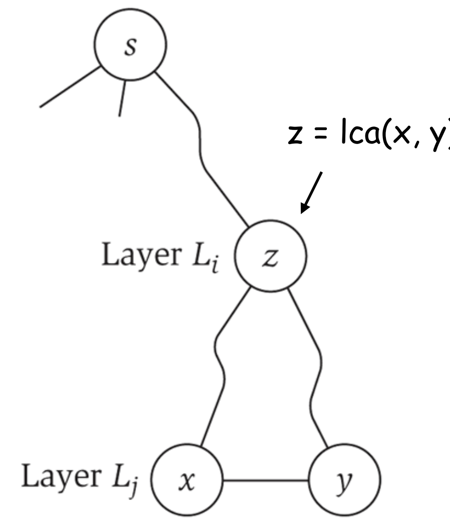
Lemma: Let G be a connected graph, and let $L_0, \ldots, L_k$ be the layers produced by BFS starting at node s. Exactly one of the following holds.

(i) No edge of G joins two nodes of the same layer, and G is bipartite.

(ii) An edge of G joins two nodes of the same layer, and G contains an odd-length cycle (and hence is not bipartite).

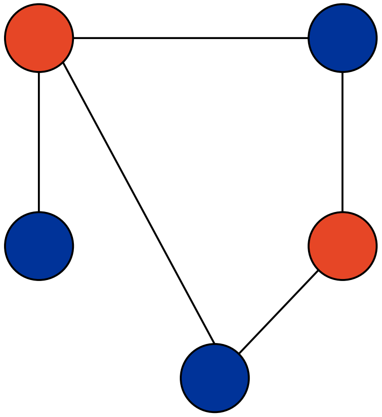
Proof: Case (ii)

- Suppose (x, y) is an edge with x, y in same level L_j.
- Let z = lca(x, y) = lowest common ancestor.
- Let L_i be level containing z.
- Consider cycle that takes edge from x to y, then path from y to z, then path from z to x.
- Its length is $\underbrace{1}_{(x,\,y)} + \underbrace{(j-i)}_{\text{path from } y \text{ to } z} + \underbrace{(j-i)}_{\text{path from } z \text{ to } x}$, which is odd. ▪

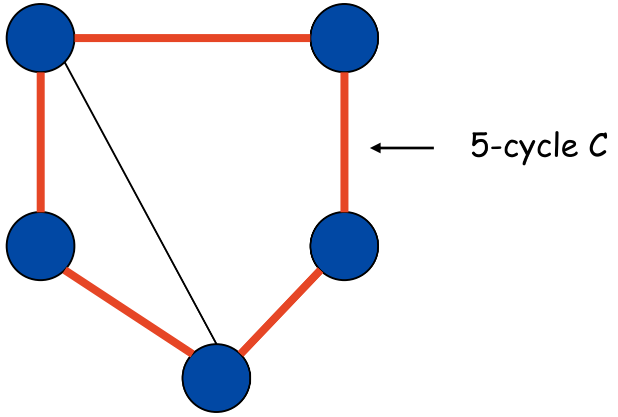


Obstruction to Bipartiteness

Corollary: A graph G is bipartite if and only if it contain no odd length cycle.



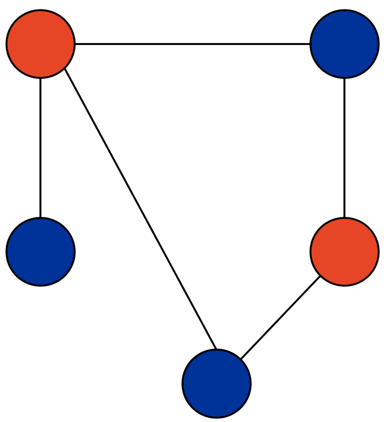
bipartite
(2-colorable)



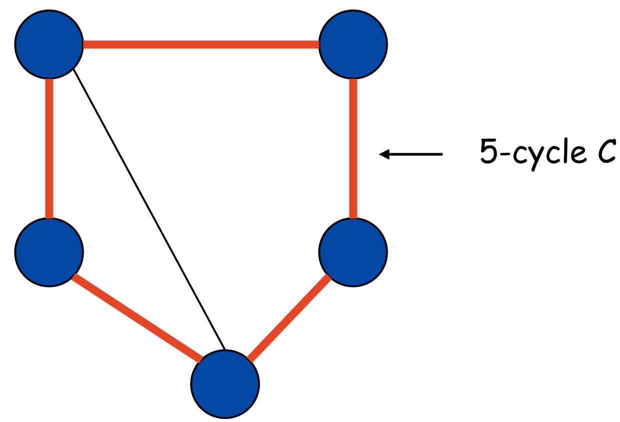
not bipartite
(not 2-colorable)

Testing bipartiteness

Theorem: Given a graph G=(V,E) one can test if G is bipartitie in O(n+m) time.



bipartite
(2-colorable)



not bipartite
(not 2-colorable)

Cut edges

Definition: In a connected graph, an edge e is called a "cut edge" if its removal would disconnect the graph

- $G=(V,E)$ is connected
- $G' = (V, E\setminus\{e\})$ is not connected

How do we find the cut edges of a graph?

Finding cut edges

Algorithm 1: (the straightforward one)

```
For every edge e in G
    remove e from G
    check if G is connected (running DFS for example)
```

Running time? $O(m^2+mn)$

More efficient algorithm?

Finding cut edges

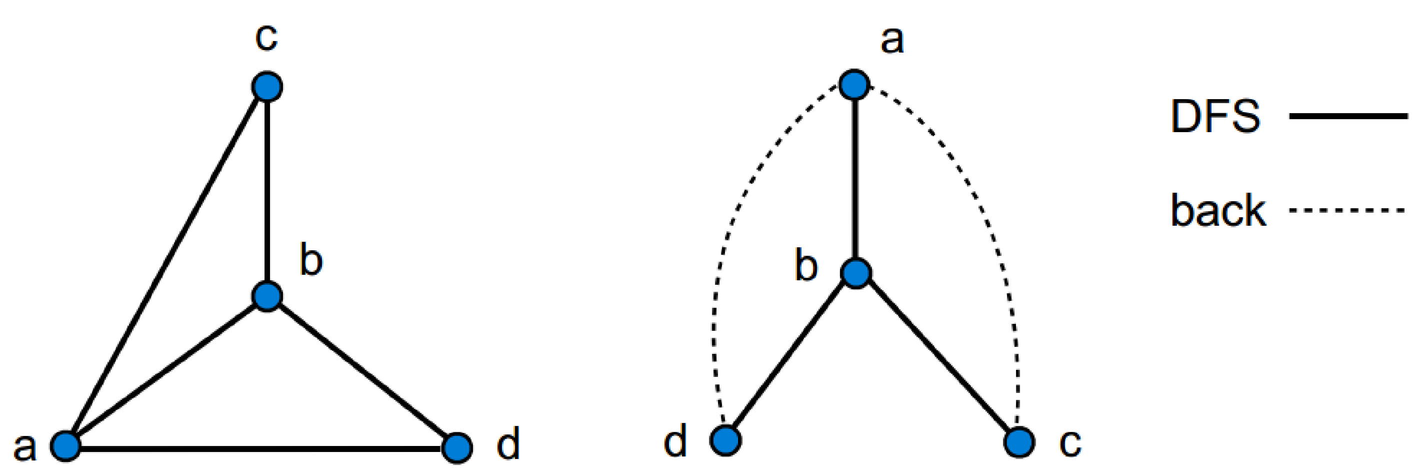
Algorithm 2:

```
Run DFS on the graph G
For each edge in the DFS tree
   remove that edge from the graph G
   check if G is now disconnected (using DFS)
```

Running time? O(nm)

```
Algorithm DFS(G,u)
  Input: graph G(V,E) and a vertex u in V

  begin
      mark u as visited
      for each edge (u,v) in E do
      if v has not been visited then
         DFS(G,v)
  end
```

In the DFS forest every non-tree edge is a back edge



Improved algorithm for finding cut edges

– Let (u,v) be an edge we would like to test.
  Assume u is the parent of v in the DFS tree.

– If (u,v) is not a cut edge then there must be a back edge from v or a descendant to a node above u in the DFS tree.

Running time: O(n+m)

Summary: Graphs

Graph representation:

- adjacency matrix or adjacency list

Basic notations and definitions:

- cycle, simple, connected, path, tree, directed,…

Traversing a graph (BFS or DFS): O(n+m)

- Applications of BFS/DFS: min link path, transitive closure, testing bipartitness, cut edges…