Depth-First Search

Announcements/ Reminders

- Homework 0 feedback will be returned soon
- Solutions added to GLOW->Files
- Discussion:
    - Geometric series question
    - Induction question
- Homework 1 is due Wednesday March 3 by 11 pm
- Help hours today:
    - Me: 2-3.30 pm, TAs: 3.30-5.30 pm, 9-11 pm
- Help hours tomorrow:
    - Me: 3-5 pm, TAs: 7-10 pm

Wrapping Up

Theorem. The following statements are **equivalent** for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Note: Conditions (a) and (b) seem hard to check directly; but conditions (c) and (d) allow an easy check!

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (a) ⇒ (b)

Vertices must alternate between V_1 and V_2.

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (b) $\Rightarrow$ (c)

Contradiction: Such an edge implies an odd cycle

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (c) $\Rightarrow$ (d)

If all BFS trees have a property then some do as well

Bipartite Testing: Using BFS

Theorem. The following statements are equivalent for a connected graph G :

(a) G is bipartite

(b) G has no odd-length cycle

(c) No BFS tree has edges between vertices at same level

(d) Some BFS tree has no edges between 2 vertices at same level

Proof. (d) $\Rightarrow$ (a)

Edges must span consecutive levels: levels provide bipartition of G

Implications of the Theorem

How to check if a graph is bipartite?

- When we visit an edge during BFS, we know the level of both of its endpoints
- So if both ends have the same level, then we can stop ! (G is not bipartite)
- If no such edge is found during traversal, G is bipartite
- Alternate levels give the bipartition

Running time?

- Still $O(n + m)$
- **Certificate.** If G is not bipartite this algorithm gives us a proof of it (the odd cycle that is found)!

Depth-First Search

Stack Instead of Queue

If we change how we store the visited vertices (the data structure we use), it changes how we traverse the graph

```
BFS (G, s):
 Set status of all nodes to unmarked
 Mark s and put in the queue Q
 While Q is not empty
    Extract v from Q
          For each edge (v, w):
              If w is unmarked
                  Mark w
                  Put w into the queue Q
```

Stack Instead of Queue

Depth-first search: Store visited nodes in a **stack**

```
DFS (G, s):
 Set status of all nodes to unmarked
 Mark s and put in the stack S
 While S is not empty
   Extract v from S
       For each edge (v, w):
          If w is unmarked
              Mark w
              Put w into the stack S
```

Depth-First Search: Recursive

- Perhaps the most natural traversal algorithm
- Can be written **recursively** as well
- Both versions are the same; can actually see the “recursion stack” in the iterative version

```
Recursive-DFS(u):
   Set status of u to marked # discovered u
   for each edges (u, v):
      if v's status is unmarked:
         DFS(v)
   # done exploring neighbors of u
```

DFS Running Time

- Same analysis as BFS
- Inserts and extracts to a stack: $O(1)$ time
- Setting status of each node to unmarked: $O(n)$
- Each node is set marked at most once; equivalently DFS(u) is called at most once for each node
- For every node v, explore degree(v) edges
    - $\sum_{v} \text{degree(v)} = 2m$
- Overall, running time $O(n + m)$

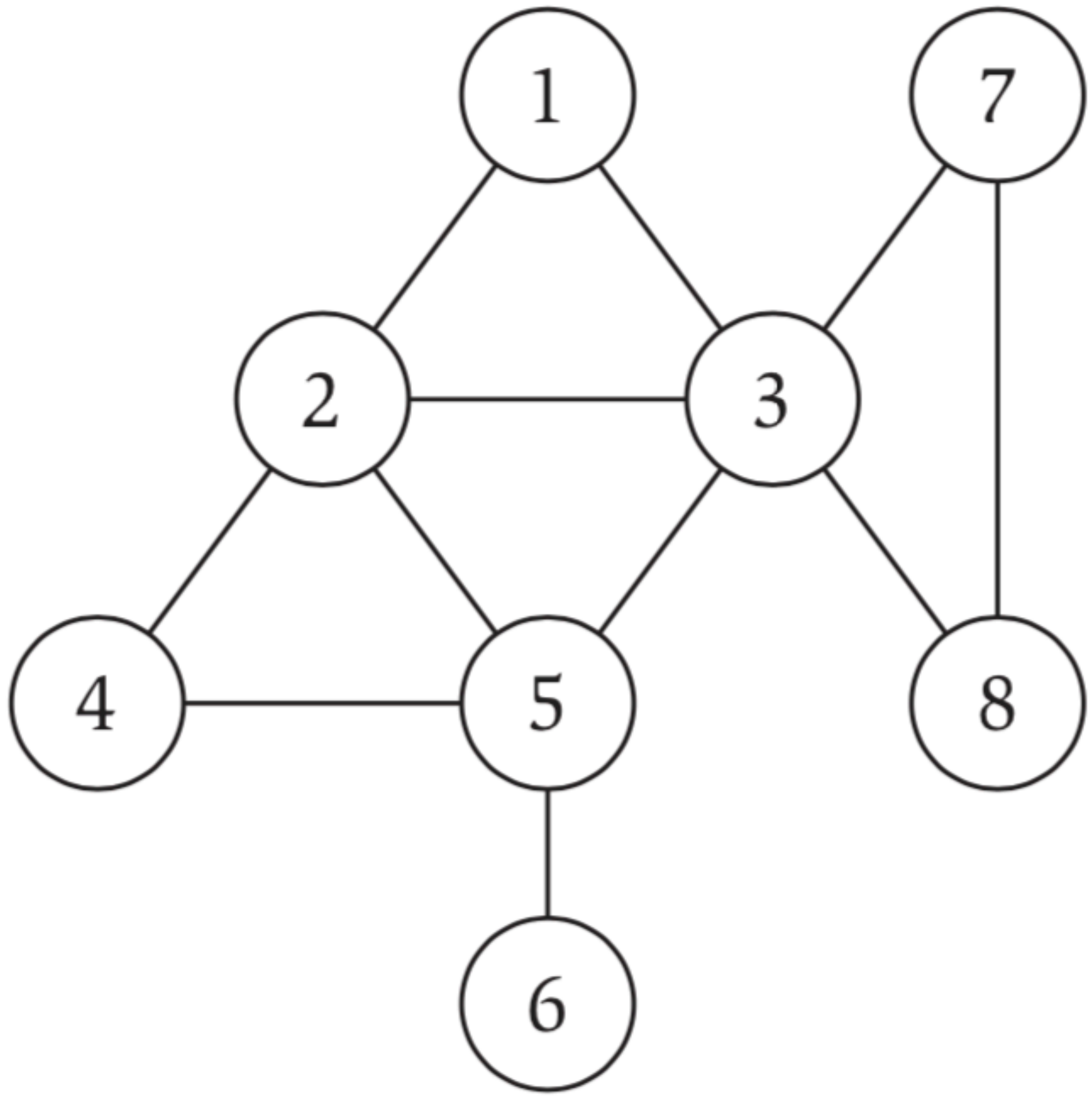
Depth-First Search Tree

- DFS returns a spanning tree, similar to BFS

```
DFS-Tree(G, s):
 Put (∅, s) in the stack S
 While S is not empty
   Extract (p, v) from S
     If v is unmarked
       Mark v
       parent(v) = p
       For each edge (v, w):
          Put (v, w) into the stack S
```

- The spanning tree formed by parent edges in a DFS are usually long and skinny

Example Graph



DFS Correctness

- DFS finds precisely the set of nodes reachable from start node s
- That is, DFS(s) marks node x iff node x is reachable from s
- **Proof**. $(\Rightarrow)$
    - Since x is marked, $(x,$ parent$(x))$) is an edge in the graph
    - Claim. $x \rightarrow$ parent(x) $\rightarrow$ parent(parent(x)) $\rightarrow \cdots$ leads to s
    - Induction on the order in which vertices are marked
    - Suppose claim holds for all vertices before some vertex u
    - Consider u: parent(u) must be discovered before u, and thus the claim holds for it, since ($u,$ parent(u)) is an edge, we have a path from u to s

DFS Correctness

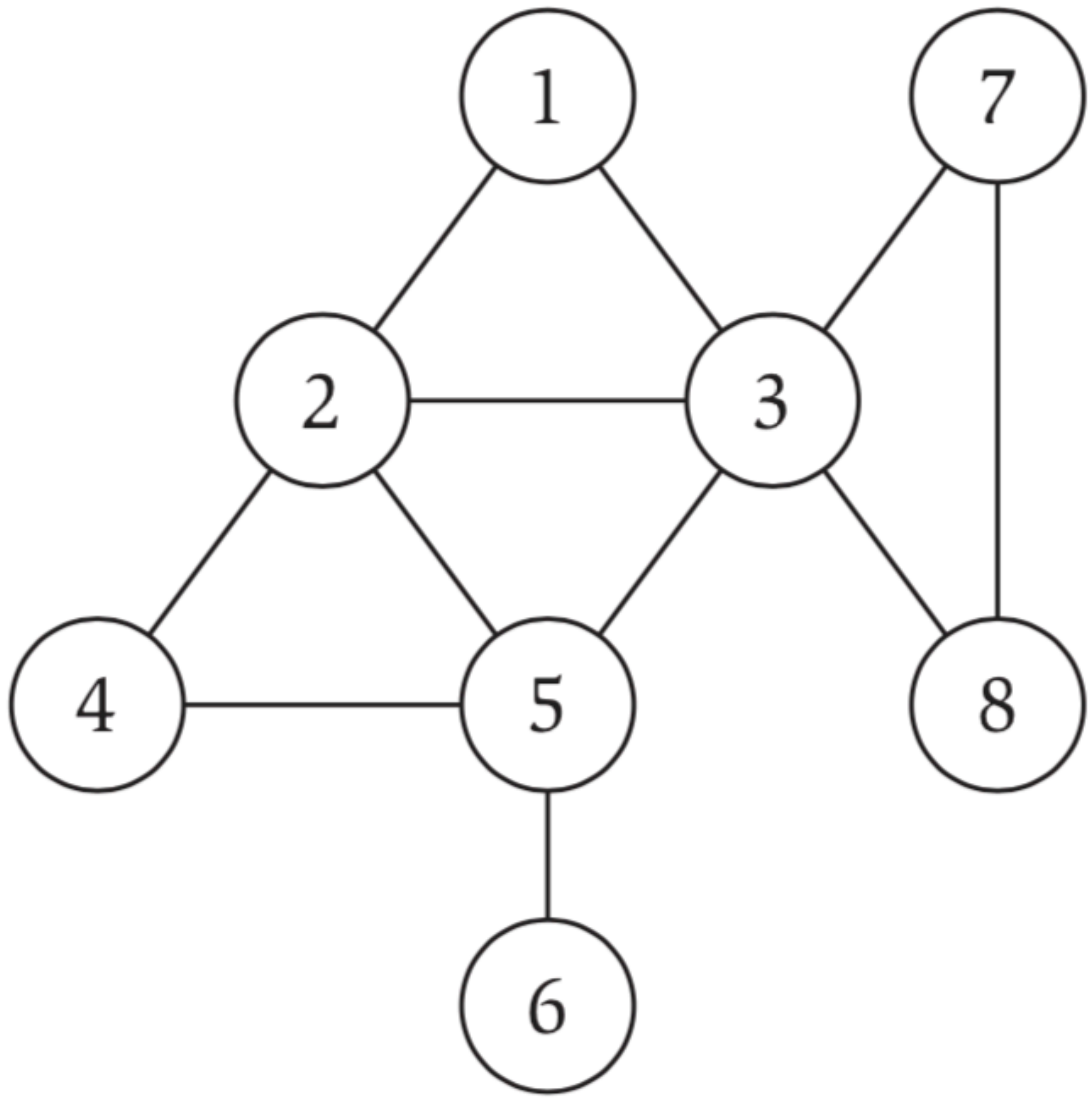
- DFS finds precisely the set of nodes reachable from start node s
- That is, DFS(s) marks node x iff node x is reachable from s
- **Proof**. $(\Leftarrow)$
    - Suppose node x is reachable from s via path P, but x is not marked by DFS
    - Since s is marked by DFS and x is not, there must be a first node v on P that is not marked by DFS
    - Thus, there is an edge $(u, v) \in P$ such that u is marked and v is not marked
    - But this cannot happen, since when u is marked, all its neighbors are also marked $\Rightarrow \Leftarrow$ ∎

BFS vs DFS

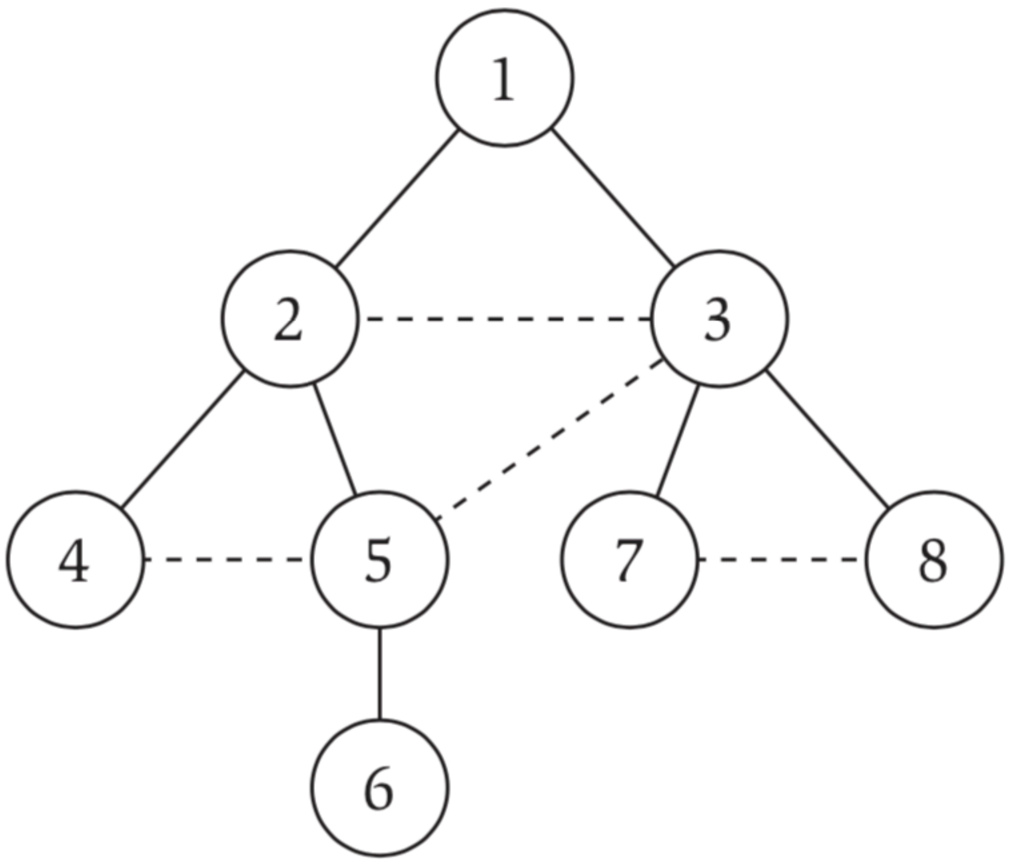
Similarities: Reachability

- Similar to BFS, you can use DFS to verify is a graph is connected
- To answer reachability questions: is s reachable from t
- Or to find all the connected components of a graph

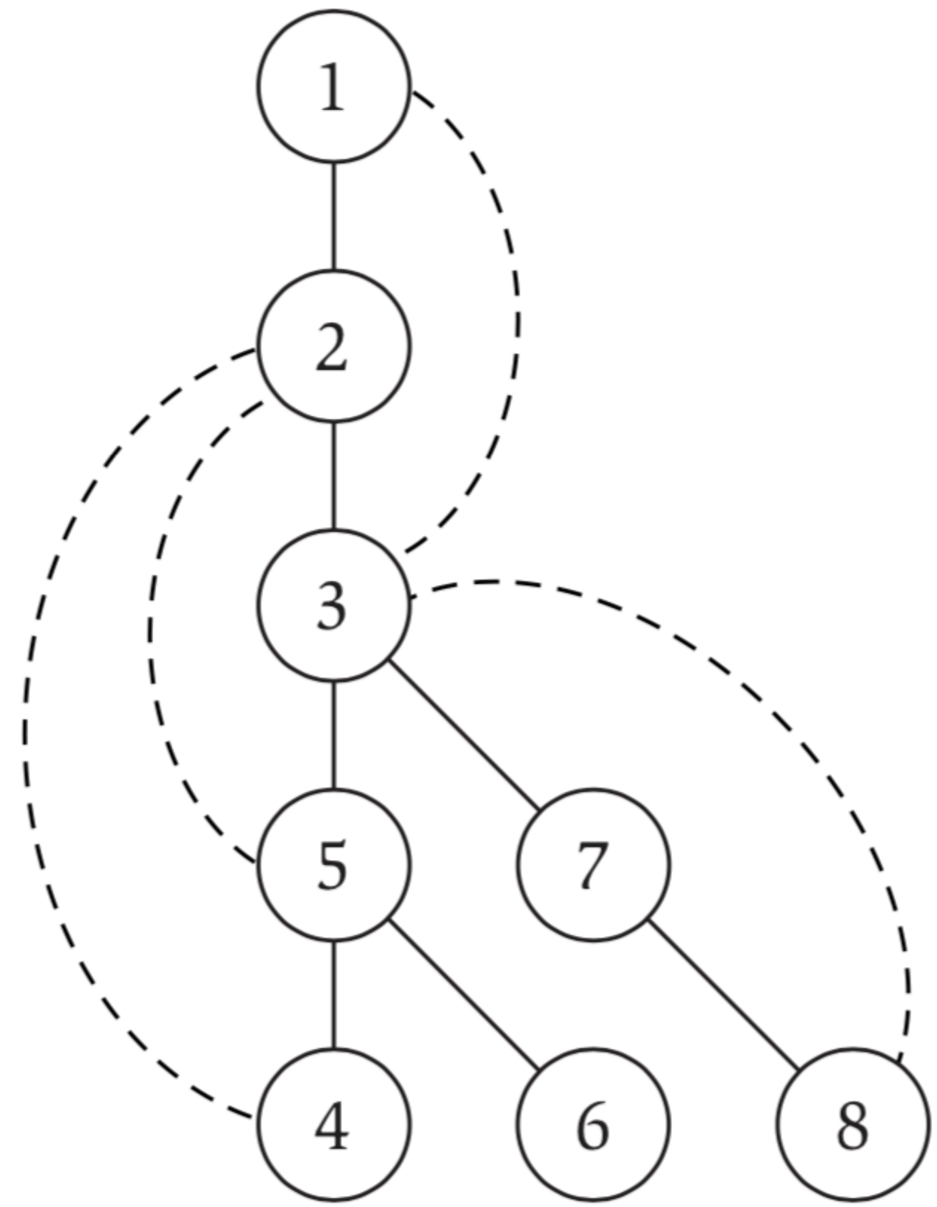
Example Graph



BFS vs DFS Tree



BFS tree

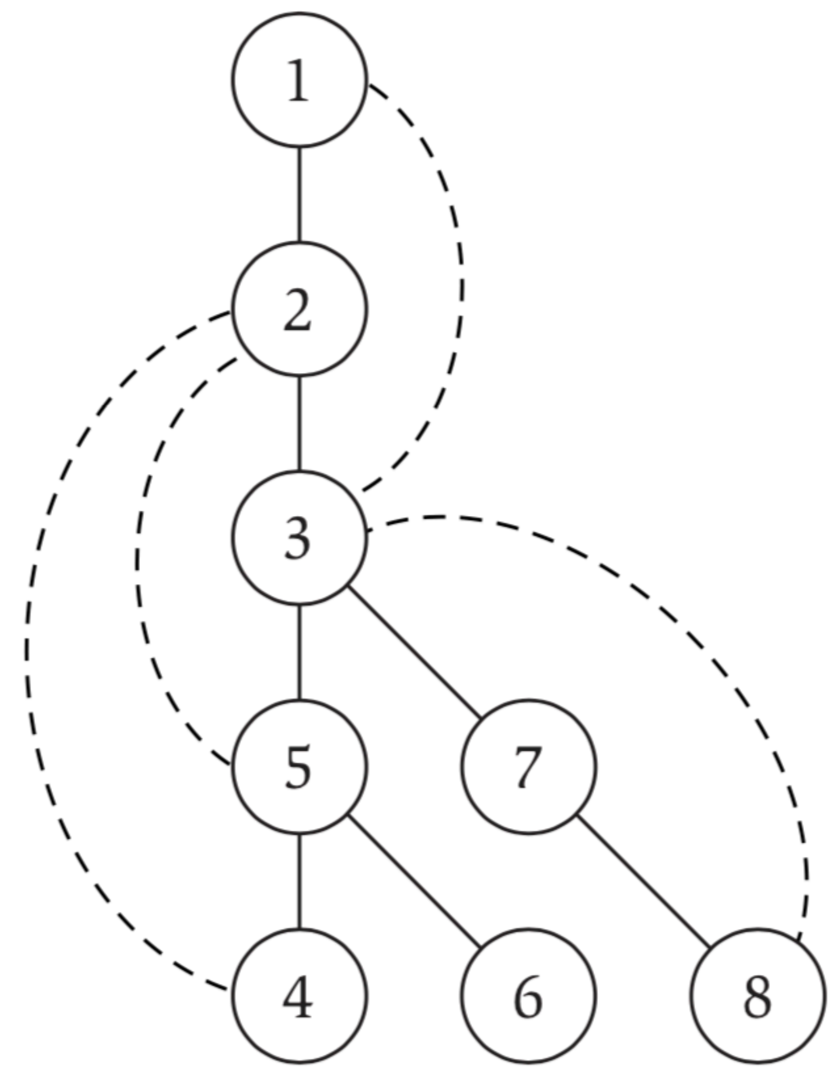


DFS tree

DFS Property

Property. For a given recursive call DFS(u), all nodes that are marked between the invocation and end of this recursive call are descendants of u in T

- A node y is a descendant of x and x is an ancestor of y if you can reach x from y by following the edges $y \rightarrow$ parent(y) $\rightarrow$ parent(parent(y)) $\rightarrow \cdots$
- Equivalently if x is on the path from the root to y



DFS Tree Property

Lemma. For every edge $e = (u, v)$ in G, one of u or v is an ancestor of the other in T.

Proof. Obvious if edge e is in T.

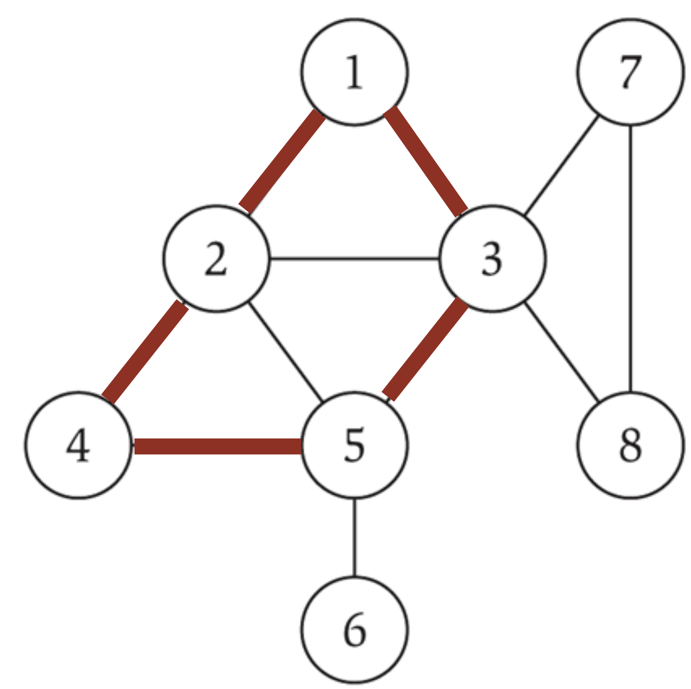
Suppose edge e is not in T. Without loss of generality, suppose DFS is called on u before v.

- When the edge u, v is inspected v must have been already marked visited (why?)
  - Or else $(u, v) \in T$ and we assumed otherwise
- Since $(u, v) \notin T$, v is not marked visited during the DFS call on u
- Must have been marked during a recursive call within DFS(u)
  - Thus v is a descendant of u ∎

In-Class Exercise

Question. Given an undirected connected graph G, how can you detect (in linear time) that it contains a cycle?

[Hint. Use DFS]



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

In-Class Exercise

Question. Given an undirected connected graph G, how can you detect (in linear time) that contains a cycle?

Idea. When we encounter a back edge (u, v) during DFS, that edge is necessarily part of a cycle (cycle formed by following tree edges from u to v and then the back edge from v to u).

```
Cycle-Detection-DFS(u):
   Set status of u to marked # discovered u
   for each edges (u, v):
      if v's status is unmarked:
         DFS(v)
      else    # found an edge to a marked node
         found a back edge, report a cycle!
   # done exploring neighbors of u
```

Cycle Detection Analysis

- Running time.
  - Same as DFS $O(n+m)$
- Correctness:
  - Follows from the observation that a graph G has a cycle if and only if there exists a back edge wrt to any DFS tree of G