Testing for Connectivity

→ we can use DFS to determine if a graph is connected

→ In the case of an undirected graph, simply start a DFS at arbitrary vertex and test whether len(discovered) = n at end

→ for directed graph $\vec{G}$ we may want to test if it is strongly connected

- if for every pair of vertices u and v, both u reaches v and v reaches u
- → begin by performing DFS of $\vec{G}$ starting at arbitrary vertex s
  - if any vertex of $\vec{G}$ not visited by traversal it isn't strongly connected
  - if the DFS reaches each vertex, need to check that s is reachable from all vertices

Computing all Connected Components

→ when a graph is connected, next goal is identify all connected components of undirected graph, or strongly connected components of a directed graph.

→ if initial DFS call fails to reach all vertices of a graph, restart a new call to DFS at one unvisited vertex

```
// perform DFS on whole graph -- result maps each vertex v to edge used to discover it
def DFS_complete(g):
  forest = { }
  for u in g.vertices():
    if u not in forest:
      forest[u] = None        // u is root of tree
      DFS(g, u, forest)
  return forest
```

Detecting Cycles using DFS

→ a cycle exists if a back edge exists relative to DFS traversal of that graph

→ detecting back edge in undirected graph is easy -- all edges are either tree or back edges

→ In directed graph:

- when directed edge is explored leading to prev. visited vertex, must recognize whether that vertex is ancestor of curr vertex
- example: could tag vertices upon which a recursive call to DFS is still active