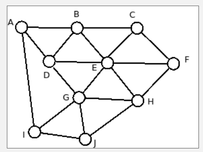
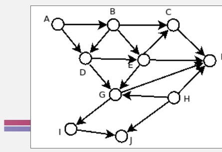
BFS-Based Algorithms

- reachability
  - run bfs(s) – the vertices *reachable* from *s* are those marked as "processed" by bfs(s)
  - works with both undirected and directed graphs

intuition – we follow every edge leaving each discovered vertex, and every vertex put in the queue is eventually removed and marked as processed

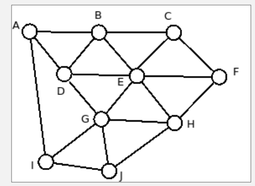





```
state[s] = "discovered"
Q.enqueue(s)
while Q is not empty do
  u = Q.dequeue()
  for each edge (u,v) in G.incidentEdges(u) do
    if state[v] = "undiscovered" then
      state[v] = "discovered"
      Q.enqueue(v)
  state[u] = "processed"
```


BFS-Based Algorithms

- unweighted shortest path
  - initialize dist[s] = 0, dist[v] = ∞ for all other vertices v
  - set dist[v] = dist[u]+1 when vertex v is discovered from vertex u
  - at the end, dist[v] has the length of the shortest path from s to v for all vertices v in the graph



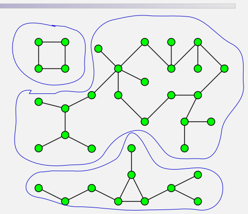
intuition – all dist 1 vertices are discovered before dist 2, etc

BFS finds the unweighted shortest path because of how it traverses the graph

```
unweighted-shortest-path(G,s)
  for each vertex u in V-{s} do
    state[u] = "undiscovered"
    prev[u] = null
    dist[v] = ∞
  state[s] = "discovered"
  prev[s] = null
  dist[s] = 0
  Q.enqueue(s)
  while Q is not empty do
    u = Q.dequeue()
    for each edge (u,v) in G.incidentEdges(u) do
      if state[v] = "undiscovered" then
        dist[v] = dist[u]+1
        state[v] = "discovered"
        prev[v] = u
        Q.enqueue(v)
    state[u] = "processed"
```

BFS-Based Algorithms

a graph with three connected components (circled)

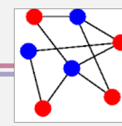


- connected components
  - a *connected component* is a subgraph where every pair of vertices are connected by a path and there are no connections to other vertices not in the subgraph

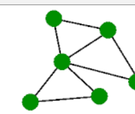
```
c = 0
for each vertex v of graph G
  if v has not been discovered
    run bfs(v), setting comp[u] = c when each vertex u is
     processed
    c++
```

intuition – BFS finds all vertices reachable from *v* along a path

BFS-Based Algorithms



bipartite



not bipartite

- bipartite graph detection / two-coloring
  - a *bipartite* graph is one whose vertices can be divided into two sets such that every edge connects a vertex in one set with a vertex in the other
  - *coloring* refers to assigning labels (colors) to vertices so that no two adjacent vertices have the same label (color)
    - a *two-coloring* uses two colors

```
color[s] = 0
run bfs(s), setting color[v] = the opposite color of
 color[u] for each discovery edge (u,v) and checking that
 color[v] is the opposite color of color[u] for each non-
 discovery edge (u,v)
```

  - if there is an edge (u,v) for which color[u] = color[v], the graph is not bipartite / two-colorable

intuition – following a path along discovery edges must alternate colors, since those edges are graph edges
– can't change the color of any vertex without changing them all
– non-discovery edges are also graph edges, and ends must be opposite colors

Takeaways

- BFS algorithm
- BFS-based algorithms
  - graph traversal
  - reachability
  - unweighted shortest path
  - connected components
  - 2-coloring / detecting bipartite graphs