Graph Traversal

One of the basic things you can do with any collection is *traversal* – i.e. visit everything in the collection exactly once.

- for graphs, this is vertices and edges
- need a systematic method to ensure
  - correctness – don't leave anything out
  - efficiency – don't waste time visiting the same things over and over

Graph Traversal

Building blocks and observations –

- Graph ADT provides operations for getting edges incident on a vertex, and end vertices of an edge
  - from a vertex you can find edges, and from an edge you can find the vertex at the other end
- there may be more than one vertex adjacent to another –
  - can't just trace through the graph using a single finger to point at where you are (loop)
  - the possibility of cycles means it is necessary to know what has been visited already
    - → need a container to hold *discovered* vertices

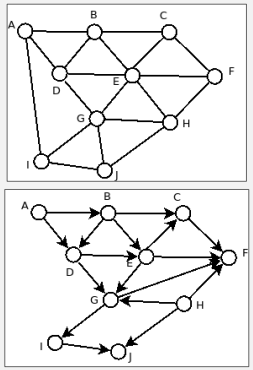
Using a queue for the container leads to breadth-first search.

Breadth-First Search

```
bfs(G,s)      G is the graph, s is the starting vertex
  for each vertex u in V-{s} do
    state[u] = "undiscovered"
    prev[u] = null
  state[s] = "discovered"
  prev[s] = null
  Q.enqueue(s)                    Q is the queue of
                                  discovered vertices
  while Q is not empty do
    u = Q.dequeue()
    process vertex u (early)      "process" is application-specific
    for each edge (u,v) in G.incidentEdges(u) do
      if state[v] = "undiscovered" then
        process edge (u,v)
        state[v] = "discovered"
        prev[v] = u
        Q.enqueue(v)
    state[u] = "processed"
    process vertex u (late)
```

this is a generalized form of the algorithm which allows for both early (before visiting incident edges) and late (after visiting incident edges) operations


```
bfs(G,s)
  for each vertex u in V-{s} do
    state[u] = "undiscovered"
    prev[u] = null
  state[s] = "discovered"
  prev[s] = null
  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"
        prev[v] = u
        Q.enqueue(v)
    state[u] = "processed"
```



H is unreachable starting from A – it is never marked as processed

	adjacency list	adjacency matrix
numVertices(), numEdges()	O(1)	O(1)
vertices(), edges()	O(1) per element	O(1) per element
aVertex()	O(1)	O(1)
degree(v)	O(1)	O(1)
adjacentVertices(v)	O(1) per element	O(n) – to scan row/column of array
incidentEdges(v)	O(1) per element	O(n) – to scan row/column of array
endVertices(e)	O(1)	O(1)
opposite(v,e)	O(1)	O(1)
areAdjacent(v,w)	O(min(deg(v,w))) – search list for vertex with smaller degree	O(1)
insertEdge(v,w,o)	O(1)	O(1)
insertVertex(o)	O(1)	O(n) – to initialize row/col of array $O(n^2)$ – if array needs to grow
removeVertex(v)	O(deg(v)) – to remove each incident edge	O(1) – with clever bookkeeping (and wasted space) $O(n^2)$ – shifting in array
removeEdge(e)	O(1)	O(1)
space	O(n+m)	$O(n^2)$

BFS – Implementation and Running Time

```
bfs(G,s)
  for each vertex u in V-{s} do
    state[u] = "undiscovered"
    prev[u] = null
  state[s] = "discovered"
  prev[s] = null
  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"
        prev[v] = u
        Q.enqueue(v)
    state[u] = "processed"
```

O(1) per element → O(n) total

O(1)

n repetitions

O(1)

O(1) per element = O(deg(u)) (adj list)
O(n) (adj matrix)

O(1)

O(1)

enqueue, dequeue, isEmpty are O(1)
set, access state – want O(1), can do that with Map (hashtable)

total over all iterations of `while` loop – O(n)

total over all iterations of `while` loop –
- O(m) for adj list because the sum of the degrees is 2m – each edge is counted twice
- $O(n^2)$ for adj matrix because `while` loop repeats n times

⇒ O(n+m) for adjacency list implementation
$O(n^2)$ for adjacency matrix implementation