GRAPH REPRESENTATION

Given graph $G = (V, E)$. In pseudocode, represent vertex set by $G.V$ and edge set by $G.E$.

- G may be either directed or undirected.
- Two common ways to represent graphs for algorithms:
  1. Adjacency lists.
  2. Adjacency matrix.

When expressing the running time of an algorithm, it's often in terms of both $|V|$ and $|E|$. In asymptotic notation—and *only* in asymptotic notation—we'll drop the cardinality. Example: $O(V + E)$ really means $O(|V| + |E|)$.

ADJACENCY LISTS

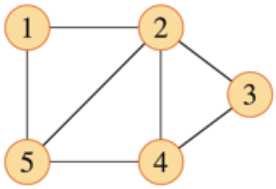
Array Adj of $|V|$ lists, one per vertex.

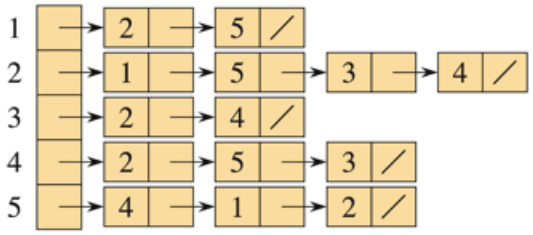
Vertex u's list has all vertices v such that $(u, v) \in E$. (Works for both directed and undirected graphs.)

In pseudocode, denote the array as attribute $G.Adj$, so will see notation such as $G.Adj[u]$.

EXAMPLE

For an undirected graph:





If edges have *weights*, can put the weights in the lists.

Weight: $w : E \to \mathbb{R}$

We'll use weights later on for spanning trees and shortest paths.

Space: $\Theta(V + E)$.

Time: to list all vertices adjacent to u: $\Theta(\text{degree}(u))$.

Time: to determine whether $(u, v) \in E$: $O(\text{degree}(u))$.

ADJACENCY MATRIX

$|V| \times |V|$ matrix $A = (a_{ij})$

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E \text{ ,} \\ 0 & \text{otherwise .} \end{cases}$$

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

	1	2	3	4
1	0	1	0	0
2	0	0	0	1
3	1	1	0	0
4	0	0	1	1

Space: $\Theta(V^2)$.

Time: to list all vertices adjacent to u: $\Theta(V)$.

Time: to determine whether $(u, v) \in E$: $\Theta(1)$.

Can store weights instead of bits for weighted graph.

Note the difference between the two examples adjacency matrices, one is from an ***undirected*** graph, and the other is from a ***directed*** graph.

Adjacency lists are more efficient to represent sparse graphs, and adjacency matrices are helpful to identify missing edges in nearly complete graphs.

This is foundational for the graph algorithm we will explore:

BREADTH-FIRST SEARCH (continued)

Intuition

Breadth-first search expands the frontier between discovered and undiscovered ver tices uniformly across the breath of the frontier.

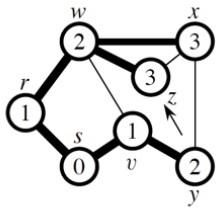
Discovers vertices in waves, starting from s.

- First visits all vertices 1 edge from s.
- From there, visits all vertices 2 edges from s.
- Etc.

Use FIFO queue Q to maintain wavefront.

- $v \in Q$ if and only if wave has visited v but has not come out of v yet.
- Q contains vertices at a distance k, and possibly some vertices at a distance $k + 1$. Therefore, at any time Q contains portions of two consecutive waves.

```
BFS(G, s)
 1  for each vertex u ∈ G.V − {s}
 2      u.color = WHITE
 3      u.d = ∞
 4      u.π = NIL
 5  s.color = GRAY
 6  s.d = 0
 7  s.π = NIL
 8  Q = ∅
 9  ENQUEUE(Q, s)
10  while Q ≠ ∅
11      u = DEQUEUE(Q)
12      for each vertex v in G.Adj[u]    // search the neighbors of u
13          if v.color == WHITE          // is v being discovered now?
14              v.color = GRAY
15              v.d = u.d + 1
16              v.π = u
17              ENQUEUE(Q, v)            // v is now on the frontier
18      u.color = BLACK                  // u is now behind the frontier
```



Q $\varnothing$

- Edges drawn with heavy lines are in the predecessor subgraph.
- Dashed lines go to newly discovered vertices. They are drawn with heavy lines because they are also now in the predecessor subgraph.
- Double-outline vertices have been discovered and are in Q, waiting to be visited.
- Heavy-outline vertices have been discovered, dequeued from Q, and visited.

BFS may not reach all vertices.

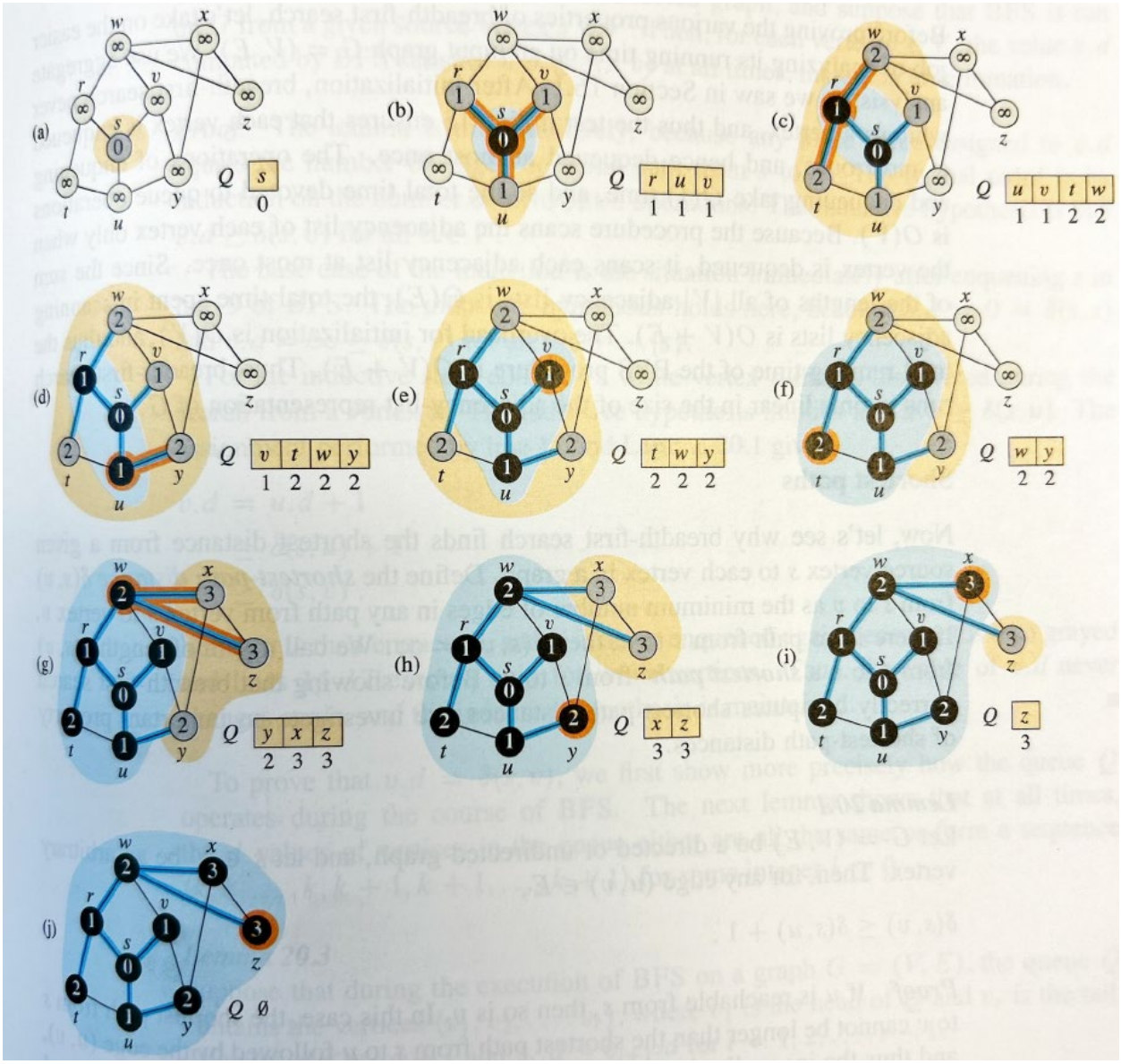
Time = $O(V + E)$.

- $O(V)$ because every vertex enqueued at most once.
- $O(E)$ because every vertex dequeued at most once and edge (u, v) is examined only when u is dequeued. Therefore, every edge examined at most once if directed, at most twice if undirected.

To print the vertices on a shortest path from s to v:

```
PRINT-PATH(G, s, v)
1  if v == s
2      print s
3  elseif v.π == NIL
4      print "no path from" s "to" v "exists"
5  else PRINT-PATH(G, s, v.π)
6      print v
```

Together, let's work through the following example from the text:



Run `BFS.ipynb`, which includes the pseudocode above, and confirm that the output agrees with the in-class example.

<u>In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):</u>

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Create a non-trivial undirected graph with your team.
4. Work through the BFS algorithm by hand.
5. In a new execution group, run your graph and confirm that the output agrees with your group work.