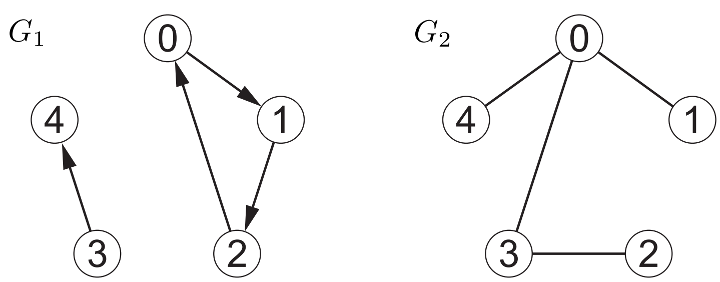
Recitation 9

Graphs

A graph $G = (V, E)$ is a mathematical object comprising a set of **vertices** V (also called nodes) and a set of **edges** E, where each edge in E is a two-element subset of vertices from V. A vertex and edge are **incident** or **adjacent** if the edge contains the vertex. Let u and v be vertices. An edge is **directed** if its subset pair is **ordered**, e.g., (u, v), and **undirected** if its subset pair is **unordered**, e.g., $\{u, v\}$ or alternatively both (u, v) and (v, u). A directed edge $e = (u, v)$ extends from vertex u (e's **tail**) to vertex v (e's **head**), with e an **incoming** edge of v and an **outgoing** edge of u. In an undirected graph, every edge is incoming and outgoing. The **in-degree** and **out-degree** of a vertex v denotes the number of incoming and outgoing edges connected to v respectively. Unless otherwise specified, when we talk about degree, we generally mean out-degree.

As their name suggest, graphs are often depicted **graph**ically, with vertices drawn as points, and edges drawn as lines connecting the points. If an edge is directed, its corresponding line typically includes an indication of the direction of the edge, for example via an arrowhead near the edge's head. Below are examples of a directed graph G_1 and an undirected graph G_2.

$$G_1 = (V_1, E_1) \qquad V_1 = \{0, 1, 2, 3, 4\} \qquad E_1 = \{(0, 1), (1, 2), (2, 0), (3, 4)\}$$
$$G_2 = (V_2, E_2) \qquad V_2 = \{0, 1, 2, 3, 4\} \qquad E_2 = \{\{0, 1\}, \{0, 3\}, \{0, 4\}, \{2, 3\}\}$$



A **path**[1] in a graph is a sequence of vertices $(v_0, \ldots, v_k)$ such that for every ordered pair of vertices (v_i, v_{i+1}), there exists an outgoing edge in the graph from v_i to v_{i+1}. The **length** of a path is the number of edges in the path, or one less than the number of vertices. A graph is called **strongly connected** if there is a path from every node to every other node in the graph. Note that every connected undirected graph is also strongly connected because every undirected edge incident to a vertex is also outgoing. Of the two connected components of directed graph G_1, only one of them is strongly connected.

[1] These are "walks" in 6.042. A "path" in 6.042 does not repeat vertices, which we would call a **simple path**.

Graph Representations

There are many ways to represent a graph in code. The most common way is to store a Set data structure Adj mapping each vertex u to another data structure $\text{Adj}(u)$ storing the **adjacencies** of v, i.e., the set of vertices that are accessible from v via a single outgoing edge. This inner data structure is called an **adjacency list**. Note that we don't store the edge pairs explicitly; we store only the out-going neighbor vertices for each vertex. When vertices are uniquely labeled from 0 to $|V| - 1$, it is common to store the top-level Set Adj within a direct access array of length $|V|$, where array slot i points to the adjacency list of the vertex labeled i. Otherwise, if the vertices are not labeled in this way, it is also common to use a hash table to map each $u \in V$ to $\text{Adj}(u)$. Then, it is common to store each adjacency list $\text{Adj}(u)$ as a simple unordered array of the outgoing adjacencies. For example, the following are adjacency list representations of G_1 and G_2, using a direct access array for the top-level Set and an array for each adjacency list.

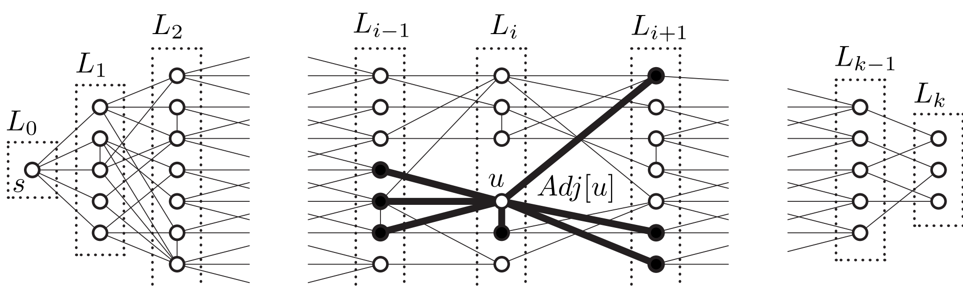
```
A1 = [[1],                A2 = [[1, 4, 3],            # 0
      [2],                      [0],                  # 1
      [0],                      [3],                  # 2
      [4],                      [0, 2],               # 3
      []]                       [0]]                  # 4
```

Using an array for an adjacency list is a perfectly good data structures if all you need to do is loop over the edges incident to a vertex (which will be the case for all algorithms we will discuss in this class, so will be our default implementation). Each edge appears in any adjacency list at most twice, so the size of an adjacency list representation implemented using arrays is $\Theta(|V| + |E|)$. A drawback of this representation is that determining whether your graph contains a given edge (u, v) might require $\Omega(|V|)$ time to step through the array representing the adjacency list of u or v. We can overcome this obstacle by storing adjacency lists using hash tables instead of regular unsorted arrays, which will support edge checking in expected $O(1)$ time, still using only $\Theta(|V|+|E|)$ space. However, we won't need this operation for our algorithms, so we will assume the simpler unsorted-array-based adjacency list representation. Below are representations of G_1 and G_2 that use a hash table for both the outer Adj Set and the inner adjacency lists $\text{Adj}(u)$, using Python dictionaries:

```
S1 = {0: {1},             S2 = {0: {1, 3, 4},         # 0
      1: {2},                   1: {0},               # 1
      2: {0},                   2: {3},               # 2
      3: {4}}                   3: {0, 2},            # 3
                                4: {0}}               # 4
```

Breadth-First Search

Given a graph, a common query is to find the vertices reachable by a path from a queried vertex s. A **breadth-first search** (BFS) from s discovers the **level sets** of s: level L_i is the set of vertices reachable from s via a **shortest** path of length i (not reachable via a path of shorter length). Breadth-first search discovers levels in increasing order starting with $i = 0$, where $L_0 = \{s\}$ since the only vertex reachable from s via a path of length $i = 0$ is s itself. Then any vertex reachable from s via a shortest path of length $i + 1$ must have an incoming edge from a vertex whose shortest path from s has length i, so it is contained in level L_i. So to compute level L_{i+1}, include every vertex with an incoming edge from a vertex in L_i, that has not already been assigned a level. By computing each level from the preceding level, a growing frontier of vertices will be explored according to their shortest path length from s.



Below is Python code implementing breadth-first search for a graph represented using index-labeled adjacency lists, returning a parent label for each vertex in the direction of a shortest path back to s. Parent labels (**pointers**) together determine a **BFS tree** from vertex s, containing some shortest path from s to every other vertex in the graph.

```
1  def bfs(Adj, s):                        # Adj: adjacency list, s: starting vertex
2      parent = [None for v in Adj]        # O(V) (use hash if unlabeled)
3      parent[s] = s                       # O(1) root
4      level = [[s]]                       # O(1) initialize levels
5      while 0 < len(level[-1]):           # O(?) last level contains vertices
6          level.append([])                # O(1) amortized, make new level
7          for u in level[-2]:             # O(?) loop over last full level
8              for v in Adj[u]:            # O(Adj[u]) loop over neighbors
9                  if parent[v] is None:   # O(1) parent not yet assigned
10                     parent[v] = u       # O(1) assign parent from level[-2]
11                     level[-1].append(v) # O(1) amortized, add to border
12     return parent
```

How fast is breadth-first search? In particular, how many times can the inner loop on lines 9–11 be executed? A vertex is added to any level at most once in line 11, so the loop in line 7 processes each vertex v at most once. The loop in line 8 cycles through all $\deg(v)$ outgoing edges from vertex v. Thus the inner loop is repeated at most $O(\sum_{v \in V} \deg(v)) = O(|E|)$ times. Because the parent array returned has length $|V|$, breadth-first search runs in $O(|V| + |E|)$ time.

Exercise: For graphs G_1 and G_2, conducting a breadth-first search from vertex v_0 yields the parent labels and level sets below.

```python
      P1 = [0,       L1 = [[0],      P2 = [0,      L2 = [[0],         # 0
            0,              [1],            0,             [1,3,4],   # 1
            1,              [2],            3,             [2],       # 2
            None,           []]             0,             []]        # 3
            None]                           0]                        # 4
```

We can use parent labels returned by a breadth-first search to construct a shortest path from a vertex s to vertex t, following parent pointers from t backward through the graph to s. Below is Python code to compute the shortest path from s to t which also runs in worst-case $O(|V| + |E|)$ time.

```python
def unweighted_shortest_path(Adj, s, t):
    parent = bfs(Adj, s)                    # O(V + E) BFS tree from s
    if parent[t] is None:                   # O(1) t reachable from s?
        return None                         # O(1) no path
    i = t                                   # O(1) label of current vertex
    path = [t]                              # O(1) initialize path
    while i != s:                           # O(V) walk back to s
        i = parent[i]                       # O(1) move to parent
        path.append(i)                      # O(1) amortized add to path
    return path[::-1]                       # O(V) return reversed path
```

Exercise: Given an unweighted graph $G = (V, E)$, find a shortest path from s to t having an **odd** number of edges.

Solution: Construct a new graph $G' = (V', E')$. For every vertex u in V, construct two vertices u_E and u_O in V': these represent reaching the vertex u through an even and odd number of edges, respectively. For every edge (u, v) in E, construct the edges (u_E, v_O) and (u_O, v_E) in E'. Run breadth-first search on G' from s_E to find the shortest path from s_E to t_O. Because G' is bipartite between even and odd vertices, even paths from s_E will always end at even vertices, and odd paths will end at odd vertices, so finding a shortest path from s_E to t_O will represent a path of odd length in the original graph. Because G' has $2|V|$ vertices and $2|E|$ edges, constructing G' and running breadth-first search from s_E each take $O(|V| + |E|)$ time.