Lecture 9: Breadth-First Search

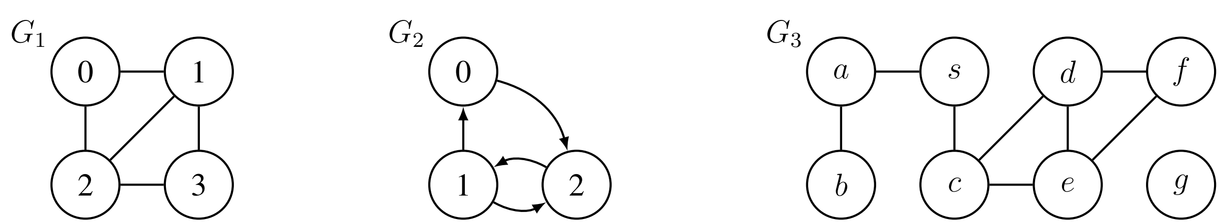
New Unit: Graphs!

- Quiz 1 next week covers lectures L01 - L08 on Data Structures and Sorting
- Today, start new unit, lectures L09 - L14 on Graph Algorithms

Graph Applications

- Why? Graphs are everywhere!
- any network system has direct connection to graphs
- e.g., road networks, computer networks, social networks
- the state space of any discrete system can be represented by a transition graph
- e.g., puzzle & games like Chess, Tetris, Rubik's cube
- e.g., application workflows, specifications

Graph Definitions



- Graph $G = (V, E)$ is a set of vertices V and a set of pairs of vertices $E \subseteq V \times V$.
- **Directed** edges are ordered pairs, e.g., (u, v) for $u, v \in V$
- **Undirected** edges are unordered pairs, e.g., $\{u, v\}$ for $u, v \in V$ i.e., (u, v) and (v, u)
- In this class, we assume all graphs are **simple**:
  - **edges are distinct**, e.g., (u, v) only occurs once in E (though (v, u) may appear), and
  - edges are **pairs of distinct vertices**, e.g., $u \neq v$ for all $(u, v) \in E$
  - Simple implies $|E| = O(|V|^2)$, since $|E| \leq \binom{|V|}{2}$ for undirected, $\leq 2\binom{|V|}{2}$ for directed

Neighbor Sets/Adjacencies

- The **outgoing neighbor set** of $u \in V$ is $\text{Adj}^+(u) = \{v \in V \mid (u, v) \in E\}$
- The **incoming neighbor set** of $u \in V$ is $\text{Adj}^-(u) = \{v \in V \mid (v, u) \in E\}$
- The **out-degree** of a vertex $u \in V$ is $\deg^+(u) = |\text{Adj}^+(u)|$
- The **in-degree** of a vertex $u \in V$ is $\deg^-(u) = |\text{Adj}^-(u)|$
- For undirected graphs, $\text{Adj}^-(u) = \text{Adj}^+(u)$ and $\deg^-(u) = \deg^+(u)$
- Dropping superscript defaults to outgoing, i.e., $\text{Adj}(u) = \text{Adj}^+(u)$ and $\deg(u) = \deg^+(u)$

Graph Representations

- To store a graph $G = (V, E)$, we need to store the outgoing edges $\text{Adj}(u)$ for all $u \in V$
- First, need a Set data structure Adj to map u to $\text{Adj}(u)$
- Then for each u, need to store $\text{Adj}(u)$ in another data structure called an **adjacency list**
- Common to use **direct access array** or **hash table** for Adj, since want lookup fast by vertex
- Common to use **array** or **linked list** for each $\text{Adj}(u)$ since usually only iteration is needed[1]
- For the common representations, Adj has size $\Theta(|V|)$, while each $\text{Adj}(u)$ has size $\Theta(\deg(u))$
- Since $\sum_{u \in V} \deg(u) \leq 2|E|$ by handshaking lemma, graph storable in $\Theta(|V| + |E|)$ space
- Thus, for algorithms on graphs, **linear time** will mean $\Theta(|V| + |E|)$ (linear in size of graph)

Examples

- Examples 1 and 2 assume vertices are labeled $\{0, 1, \ldots, |V| - 1\}$, so can use a direct access array for Adj, and store $\text{Adj}(u)$ in an array. Example 3 uses a hash table for Adj.

```
Ex 1 (Undirected)   | Ex 2 (Directed)   | Ex 3 (Undirected)
G1 = [              | G2 = [            | G3 = {
    [2, 1],    # 0  |     [2],     # 0  |     a: [s, b],    b: [a],
    [2, 0, 3], # 1  |     [2, 0],  # 1  |     s: [a, c],    c: [s, d, e],
    [1, 3, 0], # 2  |     [1],     # 2  |     d: [c, e, f], e: [c, d, f],
    [1, 2],    # 3  | ]                 |     f: [d, e],    g: [],
]                   |                   | }
```

- Note that in an undirected graph, connections are symmetric as every edge is outgoing twice

[1] A hash table for each $\text{Adj}(u)$ can allow checking for an edge $(u, v) \in E$ in $O(1)_{(e)}$ time

Paths

- A **path** is a sequence of vertices $p = (v_1, v_2, \ldots, v_k)$ where $(v_i, v_{i+1}) \in E$ for all $1 \le i < k$.
- A path is **simple** if it does not repeat vertices[2]
- The **length** $\ell(p)$ of a path p is the number of edges in the path
- The **distance** $\delta(u, v)$ from $u \in V$ to $v \in V$ is the minimum length of any path from u to v, i.e., the length of a **shortest path** from u to v
  (by convention, $\delta(u, v) = \infty$ if u is not connected to v)

Graph Path Problems

- There are many problems you might want to solve concerning paths in a graph:

- SINGLE_PAIR_REACHABILITY(G, s, t): is there a path in G from $s \in V$ to $t \in V$?
- SINGLE_PAIR_SHORTEST_PATH(G, s, t): return distance $\delta(s, t)$, and
  a shortest path in $G = (V, E)$ from $s \in V$ to $t \in V$
- SINGLE_SOURCE_SHORTEST_PATHS(G, s): return $\delta(s, v)$ for all $v \in V$, and
  a **shortest-path tree** containing a shortest path from s to every $v \in V$ (defined below)

- Each problem above is **at least as hard** as every problem above it
  (i.e., you can use a black-box that solves a lower problem to solve any higher problem)
- We won't show algorithms to solve all of these problems
- Instead, show one algorithm that solves the **hardest** in $O(|V| + |E|)$ time!

Shortest Paths Tree

- How to return a shortest path from source vertex s for every vertex in graph?
- Many paths could have length $\Omega(|V|)$, so returning every path could require $\Omega(|V|^2)$ time
- Instead, for all $v \in V$, store its **parent** $P(v)$: second to last vertex on a shortest path from s
- Let $P(s)$ be null (no second to last vertex on shortest path from s to s)
- Set of parents comprise a **shortest paths tree** with $O(|V|)$ size!
  (i.e., reversed shortest paths back to s from every vertex reachable from s)

[2] A path in 6.006 is a "walk" in 6.042. A "path" in 6.042 is a simple path in 6.006.

Breadth-First Search (BFS)

- How to compute $\delta(s,v)$ and $P(v)$ for all $v \in V$?
- Store $\delta(s,v)$ and $P(v)$ in Set data structures mapping vertices v to distance and parent
- (If no path from s to v, do not store v in P and set $\delta(s,v)$ to ∞)
- **Idea!** Explore graph nodes in increasing order of distance
- **Goal:** Compute **level sets** $L_i = \{v \mid v \in V \text{ and } d(s,v) = i\}$ (i.e., all vertices at distance i)
- Claim: Every vertex $v \in L_i$ must be adjacent to a vertex $u \in L_{i-1}$ (i.e., $v \in \text{Adj}(u)$)
- Claim: No vertex that is in L_j for some $j < i$, appears in L_i
- **Invariant:** $\delta(s,v)$ and $P(v)$ have been computed correctly for all v in any L_j for $j < i$

- Base case $(i = 1)$: $L_0 = \{s\}$, $\delta(s,s) = 0$, $P(s) = \texttt{None}$
- Inductive Step: To compute L_i:
  - for every vertex u in L_{i-1}:
    - for every vertex $v \in \text{Adj}(u)$ that does not appear in any L_j for $j < i$:
      - add v to L_i, set $\delta(s,v) = i$, and set $P(v) = u$
- Repeatedly compute L_i from L_j for $j < i$ for increasing i until L_i is the empty set
- Set $\delta(s,v) = \infty$ for any $v \in V$ for which $\delta(s,v)$ was not set

---

- Breadth-first search correctly computes all $\delta(s,v)$ and $P(v)$ by induction
- Running time analysis:
  - Store each L_i in data structure with $\Theta(|L_i|)$-time iteration and $O(1)$-time insertion (i.e., in a dynamic array or linked list)
  - Checking for a vertex v in any L_j for $j < i$ can be done by checking for v in P
  - Maintain δ and P in Set data structures supporting dictionary ops in $O(1)$ time (i.e., direct access array or hash table)
  - Algorithm adds each vertex u to ≤ 1 level and spends $O(1)$ time for each $v \in \text{Adj}(u)$
  - Work upper bounded by $O(1) \times \sum_{u \in V} \deg(u) = O(|E|)$ by handshake lemma
  - Spend $\Theta(|V|)$ at end to assign $\delta(s,v)$ for vertices $v \in V$ not reachable from s
  - So breadth-first search runs in linear time! $O(|V| + |E|)$
- Run breadth-first search from s in the graph in Example 3