Recitation 11

- Graphs
- Breadth First Search

Graphs

A graph is a collection of vertices \( V \) and edges \( E \) between the vertices, represented as pairs of vertices. We typically deal with simple graphs, consisting of no multiple edges and no loops (i.e. an edge between the same vertex twice). Graphs can be directed or undirected; a directed graph has an order relation on edges, typically represented as an arrow (a from vertex and a to vertex). Finally, graphs can be weighted or unweighted - in a weighted graph, every edge is further assigned a weight.

There are multiple representations of a graph. Typically we could use an adjacency-list representation or an adjacency-matrix representation. The adjacency-list is much more useful for sparse graphs (\(|V|^2 \gg |E|\)) because the entire matrix is not necessary - however, the adjacency-matrix has many added benefits for dense graphs.

For example, we can compute the number of paths of length \( k \) from vertex \( i \) to vertex \( j \) by looking at the \( ij^{th} \) entry of raising the adjacency matrix to the \( k^{th} \) power. (To see this, use strong induction.) Adjacency-matrices easily generalize to weighted graphs by replacing entries with the corresponding weight, and generalize to directed graphs by removing the symmetry aspect (only put a non-zero entry in \( ij \) and not \( ji \)).

There are many families of graphs that are useful to discuss. Two examples are paths and cycles - a path is a graph on \( n \) vertices, with edges \((1, 2), (2, 3), \ldots (n-2, n-1), (n-1, n)\). A cycle is similar to a path, with one additional edge \((n, 1)\).

A subgraph of a graph is some subset of vertices, and some subset of the edges between those vertices, in \( V \) and \( E \). For example, a path is a subgraph of the corresponding cycle. We refer to a graph as connected if there exists a path between every pair of vertices in the graph.

Breadth-First Search

A common problem in graph theory is the single-source shortest-paths problem. The idea is, for a single source vertex, to determine what the length of the shortest path (if it exists) between that
source vertex and every other vertex. Breadth-first search (BFS) is a convenient way for us to do so, in unweighted graphs.

**Motivation Example: Rubik’s Cube**

Suppose I had the following query: how many moves does it take to solve a Rubik’s cube in the worst case? In other words, perhaps it’s possible that in 20 moves or less, I can solve any given configuration (it is). But maybe this number can be lowered from 20 to something like 15. How would we determine computationally what this number is?

We can think of the set of Rubik’s Cube positions as an undirected graph. The set $V$ of vertices is the set of all possible unique positions, and two vertices have an edge in between them if one half-turn of any face can get from one to another.

One can imagine building this graph would be extremely difficult, but certainly not impossible (it has long been known how many vertices there are). Now, we can ask the simpler question - how many positions can we reach with 1 step from the starting position? Now we have a “frontier” set of all the vertices that are distance at most 1 away from the start.

Next, we want to find how many positions we can reach with 2 steps or less. So, from each of the positions in our “frontier”, we wish to expand 1 further to add to the set. However, if a visited position is already in the set, it means we already knew we could get to it in 2 steps or less, so we won’t need to add it again.

Iterating through this procedure, at some point we will have visited all the nodes (all positions can be visited in $k$ steps or less, for some $k$, so the procedure will terminate at step $k$). Thus, we can determine the answer to this problem in finite time, but each step will take progressively longer up to some point. Let’s formalize this idea in pseudocode, and analyze its runtime.
BFS Pseudocode, Runtime Analysis

We have already encountered trees, but formally, they are graphs in which there exist no undirected cycles (that is, if we replace every directed edge with an undirected edge, no subgraph is a cycle). In the process of breadth-first search, we form a BFS tree in which each layer $i$ consists of the graph vertices reachable with paths of length $i$ or less, the current “frontier”. We provide pseudocode and an example picture here.

```
BFS(G, s)
1   for each vertex $u \in G.V - \{s\}$
2       $u.color = \text{WHITE}$
3       $u.d = \infty$
4       $u.\pi = \text{NIL}$
5       $s.color = \text{GRAY}$
6       $s.d = 0$
7       $s.\pi = \text{NIL}$
8       $Q = \emptyset$
9       $\text{ENQUEUE}(Q, s)$
10      while $Q \neq \emptyset$
11         $u = \text{DEQUEUE}(Q)$
12         for each $v \in G.\text{Adj}[u]$
13            if $v.color == \text{WHITE}$
14               $v.color = \text{GRAY}$
15               $v.d = u.d + 1$
16               $v.\pi = u$
17               $\text{ENQUEUE}(Q, v)$
18         $u.color = \text{BLACK}$
```

Initializing the procedure takes $O(V)$, but the procedure runtime is dominated by lines 10 through 18, in which we traverse every edge a constant number of times (once for directed graphs and twice for undirected). The idea is that each edge will only be visited when the relevant “from” node tries to look for all of its neighbors, the “to” nodes (to see if they are unvisited and need to be
added to the queue). In general it’s clear that the runtime is linear in the size of the adjacency list of $G$ - that is, it runs in $O(V + E)$.

It also will mark every node (as black, in the algorithm above) iff the graph is connected. To see this, we need to demonstrate correctness for BFS, because then every vertex will be reachable with a path of finite length (i.e. marked/discovered in the process) iff the graph is connected, by definition.

**BFS Correctness**

We claim that BFS discovers the minimum distance from the source node to all nodes it is connected to in the graph. A skeleton of the proof is provided here, and a more detailed version can be found in section 22.2 of CLRS.

First, note that a vertex will be added to the BFS tree in finite time iff it is connected to the source. We induct on the size of the frontier; that is, suppose the frontier has reached size $k$, and the property holds for every vertex that the minimum length path from the source to that vertex has been discovered (and that the vertex has been enqueued). Now consider some vertex $v$ whose minimum length path to the source $u$ is $k + 1$. In the path of length $k + 1$ from $u$ to $v$, let the vertex immediately before $v$ be $w$. We claim $w$ is in the queue at this point (that is, $w$ belongs to the frontier of length $k$).

If $w$ had not been enqueued yet, then the inductive hypothesis is broken. However, it cannot have also been dequeued, because that would mean that $v$ would have been discovered already (added to the queue), which is not possible in $k$ steps or less. Thus, $w$ is still in the queue, and when we dequeue it, we will add its undiscovered neighbors, including $v$, to the queue in step $k + 1$. This completes the induction.