Graph Representations

Let $G = (V, E)$ with $|V| = n$, $|E| = m$.

**Adjacency Matrix**

One way to represent $G$ is with an $n \times n$ matrix $A$ where $A[i][j] = 1$ if there is an edge from vertex $i$ to vertex $j$ and 0 otherwise. The primary advantage of this approach is that you can check whether or not there is an edge connecting two vertices in $O(1)$ time. The disadvantage, however, is that this representation takes up $O(n^2)$ space. When $n$ is large, this might become untenable.

Two things worth noting:
- If $G$ is undirected, then its adjacency matrix is symmetric.
- Entries along the diagonal of an adjacency matrix (technically representing the presence of edges from vertices to themselves) are 0 by convention, as our graphs are simple.

**Adjacency List**

Another way to represent $G$ is to use an adjacency list. Each vertex $v$ is associated to a list $\text{neighbors}(v)$ which contains the nodes $u$ such that $(v, u) \in E$. The advantage of this representation is that we use less space, $O(n + m)$, which is better than $O(n^2)$ of adjacency matrices as long as $m \ll n^2$. The disadvantage, though, is that checking whether $(u, v) \in E$ takes (potentially) linear time.

**Graph Traversals**

We now look at two ways to traverse a graph.

**BFS (Breadth First Search)**

In BFS, we begin at a node $v$ (level 0) and explore the graph in “layers.” First we would explore all children of $v$ (level 1), then the children of the nodes in level 1 (these would make up level 2), etc. The key point here is that we explore all nodes at level $i$ before exploring any nodes at level $i + 1$. The output of BFS is called a BFS tree. We typically use a queue to implement this algorithm. For implementation details, see https://en.wikipedia.org/wiki/Breadth-first_search

The running time of BFS is $O(n + m)$, because each vertex is added and removed from the queue once and, in the worst case, we need to traverse every edge to visit each node.

**DFS (Depth First Search)**

In DFS, we begin at a node $v$ and examine its neighbors. As soon as we encounter a neighbor that hasn’t been visited, visit it. Once we arrive at a node for which all of its neighbors have been visited, we “backtrack” until we reach a node that has still unvisited neighbors (in the form of returning from recursive visit calls). For more information, see https://en.wikipedia.org/wiki/Depth-first_search

The running time analysis for DFS is similar to that of BFS, giving a running time of $O(n + m)$.
Problem 1: Cycle Detection
Design an algorithm to determine whether or not a graph has a cycle.

Problem 2: Bipartite Graph Detection
Design an algorithm to determine whether or not a graph is bipartite (that there is a way to partition $V$ into sets $V_1$ and $V_2$ such that if $(u, v) \in E$, then $u, v$ cannot both be in $V_1$ or $V_2$).

Problem 3: Bipartite Graph Detection
Design an algorithm to find the shortest path between nodes $u$ and $v$ in a connected, unweighted graph.