Graph Representations

Let $G = (V, E)$ with $|V| = n$, $|E| = m$. In other words, for some graph $G$, it contains $n$ vertices and $m$ edges.

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. That is, flipping the matrix along its main diagonal will produce the same matrix.
- 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. Non-simple graphs have self-loops, where vertices contain edges to themselves (these will not be dealt with in this course).

Adjacency List

Another way to represent $G$ is to use an adjacency list. Each vertex $u$ is associated to a list $\text{neighbors}(v)$ which contains the nodes $v$ such that $(u, v) \in E$. 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). We typically use a stack. There is also a recursive method to implement this algorithm. Please see both implementation methods in the following link: [https://en.wikipedia.org/wiki/Depth-first_search](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: More Cycle Detection!**

Design an algorithm to determine whether or not a connected undirected graph has a cycle in \( O(n) \) time.

**Problem 3: Shortest Path in an Unweighted Graph**

Design an algorithm to find the shortest path between nodes \( u \) and \( v \) in a connected, unweighted graph.

**Problem 4: Tic-tac-toe**

Suppose we are given a graph of tic-tac-toe moves such that nodes are board states and an edge from \( u \) to \( v \) means that \( v \) is reachable from \( u \) in one move. Design an algorithm that takes in a board state and determines the best possible next move (i.e. in most cases the move that will guarantee a draw or a win, unless of course every move results in a loss).

**Problem 5: Recursive Permutation**

Recursively generate all the permutations of the character sequence ‘ABCD’.

**Dijkstra’s Algorithm**

Dijkstra’s algorithm finds the shortest path between two given vertices in a weighted graph, assuming that the graph’s edge weights are non-negative. The running time of the algorithm is \( O(E \log V + V \log V) \) when the graph is implemented using adjacency lists. With a special transformation (use of Fibonacci heaps) this can be reduced to \( O(E + V \log V) \), which is the fastest version of this algorithm. The pseudo-code for the algorithm is given below.
Pseudocode

Dijkstra(G, s)
1  for each vertex v ∈ VG
2    dist[v] = ∞
3    parent[v] = NIL
4  dist[s] = 0
5  Q = VG
6  while Q ≠ ∅
7    u = Extract-Min(Q)
8    for each vertex v ∈ G.Adj[u]
9      if dist[v] > dist[u] + w(u, v)
10         dist[v] = dist[u] + w(u, v)
11         parent[v] = u

Edge-Weighted DAGs (Directed Acyclic Graphs)

The algorithm for shortest path on edge weighted DAGs is simpler and faster than Dijkstra’s algorithm. However, instead of considering vertices by priority of their distance estimates, we consider the vertices of the DAG in a topological order. (Why must a DAG always have a topological order?) Then we just relax each vertex in the topological ordering. Running time: O(|V| + |E|).

Dijkstra Questions

Problem 1. Find the shortest path between vertices E and G in the graph provided
**Problem 2.** Explain why Dijkstra’s algorithm is a *greedy algorithm.*

**Problem 3.** Does Dijkstra’s Algorithm work with negative weights? Why or why not?

**Problem 4.** True or false: Dijkstra’s algorithm will not terminate if run on a graph with negative edge weights.

**Problem 5.** True or false: The shortest path algorithm in an edge weighted DAG works even with negative edge weights.

**Problem 6.** How could you modify Dijkstra’s algorithm to find all shortest paths?

**Problem 7.** How could you modify Dijkstra’s algorithm to stop once it’s found the shortest path to a particular node?

**Problem 8.** Explain the running time of Dijkstra’s algorithm.

**Problem 9.** True or false: If we double the weights of all the edges in a graph, then Dijkstra’s algorithm will produce the same shortest path.

**Problem 10.** Say we are given a graph $G$ where all edges are positively weighted. Construct graph $G'$ where for all edges $e$ with weight $w(e)$ and endpoints $u$ and $v$, we replace $e$ with $w(e)$ edges of weight 1 in $G'$, such that the path from $u$ to $v$ in $G'$ consists of $w(e) - 1$ middle nodes. How could you use this method to find the shortest path between two vertices in $G$? What problem do you see with this approach?