Recitation 11

Weighted Graphs

For many applications, it is useful to associate a numerical **weight** to edges in a graph. For example, a graph modeling a road network might weight each edge with the length of a road corresponding to the edge, or a graph modeling an online dating network might contain edges from one user to another weighted by directed attraction. A **weighted graph** is then a graph $G = (V, E)$ together with a **weight function** $w : E \to \mathbb{R}$, mapping edges to real-valued weights. In practice, edge weights will often not be represented by a separate function at all, preferring instead to store each weight as a value in an adjacency matrix, or inside an edge object stored in an adjacency list or set. For example, below are randomly weighted adjacency set representations of the graphs from Recitation 11. A function to extract such weights might be: `def w(u,v): return W[u][v].`

```
W1 = [0: {1: -2},          W2 = {0: {1: 1, 3: 2, 4: -1},    # 0
      1: {2:  0},                1: {0: 1},                  # 1
      2: {0:  1},                2: {3: 0},                  # 2
      3: {4:  3}}                3: {0: 2, 2: 0},            # 3
                                 4: {0: -1}}                 # 4
```

Now that you have an idea of how weights could be stored, for the remainder of this class you may simply assume that a weight function w can be stored using $O(|E|)$ space, and can return the weight of an edge in constant time[1]. When referencing the weight of an edge $e = (u, v)$, we will often use the notation $w(u, v)$ interchangeably with $w(e)$ to refer to the weight of an edge.

Exercise: Represent graphs W_1 and W_2 as adjacency matrices. How could you store weights in an adjacency list representation?

Weighted Shortest Paths

A **weighted path** is simply a path in a weighted graph as defined in Recitation 11, where the **weight of the path** is the sum of the weights from edges in the path. Again, we will often abuse our notation: if $\pi = (v_1, \ldots, v_k)$ is a weighted path, we let $w(\pi)$ denote the path's weight $\sum_{i=1}^{k-1} w(v_i, v_{i+1})$. The (single source) **weighted shortest paths** problem asks for a lowest weight path to every vertex v in a graph from an input source vertex s, or an indication that no lowest weight path exists from s to v. We already know how to solve the weighted shortest paths problem on graphs for which all edge weights are positive and are equal to each other: simply run breadth-first search from s to minimize the number of edges traversed, thus minimizing path weight. But when edges have different and/or non-positive weights, breadth-first search cannot be applied directly.

[1] We will typically only be picky with the distinction between worst-case and expected bounds when we want to test your understanding of data structures. Hash tables perform well in practice, so use them!

In fact, when a graph contains a **cycle** (a path starting and ending at the same vertex) that has negative weight, then some shortest paths might not even exist, because for any path containing a vertex from the negative weight cycle, a shorter path can be found by adding a tour around the cycle. If any path from s to some vertex v contains a vertex from a negative weight cycle, we will say the shortest path from s to v is undefined, with weight $-\infty$. If no path exists from s to v, then we will say the shortest path from s to v is undefined, with weight $+\infty$. In addition to breadth-first search, we will present three additional algorithms to compute single source shortest paths that cater to different types of weighted graphs.

Weighted Single Source Shortest Path Algorithms

Restrictions		SSSP Algorithm	
Graph	Weights	Name	Running Time $O(\cdot)$
General	Unweighted	BFS	$\lvert V\rvert + \lvert E\rvert$
DAG	Any	DAG Relaxation	$\lvert V\rvert + \lvert E\rvert$
General	Any	Bellman-Ford	$\lvert V\rvert \cdot \lvert E\rvert$
General	Non-negative	Dijkstra	$\lvert V\rvert \log \lvert V\rvert + \lvert E\rvert$

Relaxation

We've shown you one view of relaxation in lecture. Below is another framework by which you can view DAG relaxation. As a general algorithmic paradigm, a **relaxation** algorithm searches for a solution to an optimization problem by starting with a solution that is not optimal, then iteratively improves the solution until it becomes an optimal solution to the original problem. In the single source shortest paths problem, we would like to find the weight $\delta(s, v)$ of a shortest path from source s to each vertex v in a graph. As a starting point, for each vertex v we will initialize an upper bound estimate $d(v)$ on the shortest path weight from s to v, $+\infty$ for all $d(s, v)$ except $d(s, s) = 0$. During the relaxation algorithm, we will repeatedly **relax** some path estimate $d(s, v)$, decreasing it toward the true shortest path weight $\delta(s, v)$. If ever $d(s, v) = \delta(s, v)$, we say that estimate $d(s, v)$ is fully relaxed. When all shortest path estimates are fully relaxed, we will have solved the original problem. Then an algorithm to find shortest paths could take the following form:

```
def general_relax(Adj, w, s):        # Adj: adjacency list, w: weights, s: start
    d = [float('inf') for _ in Adj] # shortest path estimates d(s, v)
    parent = [None for _ in Adj]    # initialize parent pointers
    d[s], parent[s] = 0, s          # initialize source
    while True:                     # repeat forever!
        relax some d[v] ??          # relax a shortest path estimate d(s, v)
    return d, parent                # return weights, paths via parents
```

There are a number of problems with this algorithm, not least of which is that it never terminates! But if we can repeatedly decrease each shortest path estimates to fully relax each $d(s, v)$, we will have found shortest paths. How do we 'relax' vertices and when do we stop relaxing?

To relax a shortest path estimate $d(s, v)$, we will relax **an incoming edge** to v, from another vertex u. If we maintain that $d(s, u)$ always upper bounds the shortest path from s to u for all $u \in V$, then the true shortest path weight $\delta(s, v)$ can't be larger than $d(s, u) + w(u, v)$ or else going to u along a shortest path and traversing the edge (u, v) would be a shorter path[2]. Thus, if at any time $d(s, u) + w(u, v) < d(s, v)$, we can relax the edge by setting $d(s, v) = d(s, u) + w(u, v)$, strictly improving our shortest path estimate.

```
1 def try_to_relax(Adj, w, d, parent, u, v):
2     if d[v] > d[u] + w(u, v):    # better path through vertex u
3         d[v] = d[u] + w(u, v)    # relax edge with shorter path found
4         parent[v] = u
```

If we only change shortest path estimates via relaxation, than we can prove that the shortest path estimates will never become smaller than true shortest paths.

> **Safety Lemma:** Relaxing an edge maintains $d(s, v) \geq \delta(s, v)$ for all $v \in V$.
>
> *Proof.* We prove a stronger statement, that for all $v \in V$, $d(s, v)$ is either infinite or the weight of some path from s to v (so cannot be larger than a shortest path). This is true at initialization: each $d(s, v)$ is $+\infty$, except for $d(s) = 0$ corresponding to the zero-length path. Now suppose at some other time the claim is true, and we relax edge (u, v). Relaxing the edge decreases $d(s, v)$ to a finite value $d(s, u) + w(u, v)$, which by induction is a length of a path from s to v: a path from s to u and the edge (u, v). □

If ever we arrive at an assignment of all shortest path estimates such that no edge in the graph can be relaxed, then we can prove that shortest path estimates are in fact shortest path distances.

> **Termination Lemma:** If no edge can be relaxed, then $d(s, v) \leq \delta(s, v)$ for all $v \in V$.
>
> *Proof.* Suppose for contradiction $\delta(s, v) < d(s, v)$ so that there is a shorter path π from s to v. Let (a, b) be ther first edge of π such that $d(b) > \delta(s, b)$. Then edge (a, b) can be relaxed, a contradiction. □

So, we can change lines 5-6 of the general relaxation algorithm to repeatedly relax edges from the graph until no edge can be further relaxed.

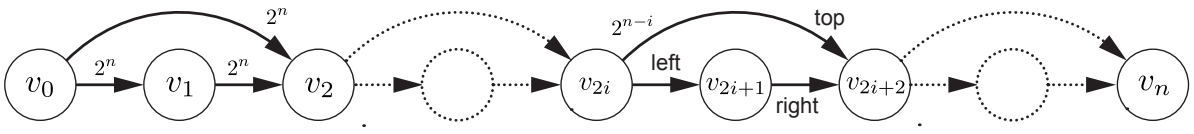
```
1     while some_edge_relaxable(Adj, w, d):
2         (u, v) = get_relaxable_edge(Adj, w, d)
3         try_to_relax(Adj, w, d, parent, u, v)
```

It remains to analyze the running time of this algorithm, which cannot be determined unless we provide detail for how this algorithm chooses edges to relax. If there exists a negative weight cycle in the graph reachable from s, this algorithm will never terminate as edges along the cycle could be relaxed forever. But even for acyclic graphs, this algorithm could take exponential time.

[2]This is a special case of the **triangle inequality**: $\delta(a, c) \leq \delta(a, b) + \delta(b, c)$ for all $a, b, c \in V$.

Exponential Relaxation

How many modifying edge relaxations could occur in an acyclic graph before all edges are fully relaxed? Below is a weighted directed graph on $2n + 1$ vertices and $3n$ edges for which the relaxation framework could perform an **exponential** number of modifying relaxations, if edges are relaxed in a bad order.



This graph contains n sections, with section i containing three edges, (v_{2i}, v_{2i+1}), (v_{2i}, v_{2i+2}), and (v_{2i+1}, v_{2i+2}), each with weight 2^{n-i}; we will call these edges within a section, **left**, **top**, and **right** respectively. In this construction, the lowest weight path from v_0 to v_i is achieved by traversing top edges until v_i's section is reached. Shortest paths from v_0 can easily by found by performing only a linear number of modifying edge relaxations: relax the top and left edges of each successive section. However, a bad relaxation order might result in many more modifying edge relaxations.

To demonstrate a bad relaxation order, initialize all minimum path weight estimates to ∞, except $d(s, s) = 0$ for source $s = v_0$. First relax the left edge, then the right edge of section 0, updating the shortest path estimate at v_2 to $d(s, v_2) = 2^n + 2^n = 2^{n+1}$. In actuality, the shortest path from v_0 to v_2 is via the top edge, i.e., $\delta(s, v_2) = 2^n$. But before relaxing the top edge of section 0, recursively apply this procedure to fully relax the remainder of the graph, from section 1 to $n - 1$, computing shortest path estimates based on the incorrect value of $d(s, v_2) = 2^{n+1}$. Only then relax the top edge of section 0, after which $d(s, v_2)$ is modified to its correct value 2^n. Lastly, fully relax sections 1 through $n - 1$ one more time recursively, to their correct and final values.

How many modifying edge relaxations are performed by this edge relaxation ordering? Let $T(n)$ represent the number of modifying edge relaxations performed by the procedure on a graph containing n sections, with recurrence relation given by $T(n) = 3 + 2T(n - 2)$. The solution to this recurrence is $T(n) = O(2^{n/2})$, exponential in the size of the graph. Perhaps there exists some edge relaxation order requiring only a **polynomial** number of modifying edge relaxations?

DAG Relaxation

In a directed acyclic graph (DAG), there can be no negative weight cycles, so eventually relaxation must terminate. It turns out that relaxing each outgoing edge from every vertex exactly once in a topological sort order of the vertices, correctly computes shortest paths. This shortest paths algorithm is sometimes called **DAG Relaxation**.

```
def DAG_Relaxation(Adj, w, s):      # Adj: adjacency list, w: weights, s: start
    _, order = dfs(Adj, s)          # run depth-first search on graph
    order.reverse()                 # reverse returned order
    d = [float('inf') for _ in Adj] # shortest path estimates d(s, v)
    parent = [None for _ in Adj]    # initialize parent pointers
    d[s], parent[s] = 0, s          # initialize source
    for u in order:                 # loop through vertices in topo sort
        for v in Adj[u]:            # loop through out-going edges of u
            try_to_relax(Adj, w, d, parent, u, v)  # try to relax edge from u to v
    return d, parent                # return weights, paths via parents
```

Claim: The DAG Relaxation algorithm computes shortest paths in a directed acyclic graph.

Proof. We prove that at termination, $d(s, v) = \delta(s, v)$ for all $v \in V$. First observe that Safety ensures that a vertex not reachable from s will retain $d(s, v) = +\infty$ at termination. Alternatively, consider any shortest path $\pi = (v_1, \ldots, v_m)$ from $v_1 = s$ to any vertex $v_m = v$ reachable from s. The topological sort order ensures that edges of the path are relaxed in the order in which they appear in the path. Assume for induction that before edge $(v_i, v_{i+1}) \in \pi$ is relaxed, $d(s, v_i) = \delta(s, v_i)$. Setting $d(s, s) = 0$ at the start provides a base case. Then relaxing edge (v_i, v_{i+1}) sets $d(s, v_{i+1}) = \delta(s, v_i) + w(v_i, v_{i+1}) = \delta(s, v_{i+1})$, as sub-paths of shortest paths are also shortest paths. Thus the procedure constructs shortest path weights as desired. Since depth-first search runs in linear time and the loops relax each edge exactly once, this algorithm takes $O(|V| + |E|)$ time. □

Exercise: You have been recruited by MIT to take part in a new part time student initiative where you will take only one class per term. You don't care about graduating; all you really want to do is to take 19.854, Advanced Quantum Machine Learning on the Blockchain: Neural Interfaces, but are concerned because of its formidable set of prerequisites. MIT professors will allow you take any class as long as you have taken **at least one** of the class's prerequisites prior to taking the class. But passing a class without all the prerequsites is difficult. From a survey of your peers, you know for each class and prerequisite pair, how many hours of stress the class will demand. Given a list of classes, prerequisites, and surveyed stress values, describe a linear time algorithm to find a sequence of classes that minimizes the amount of stress required to take 19.854, never taking more than one prerequisite for any class. You may assume that every class is offered every semester.

Solution: Build a graph with a vertex for every class and a directed edge from class a to class b if b is a prerequisite of a, weighted by the stress of taking class a after having taken class b as a prerequisite. Use topological sort relaxation to find the shortest path from class 19.854 to every other class. From the classes containing no prerequisites (sinks of the DAG), find one with minimum total stress to 19.854, and return its reversed shortest path.