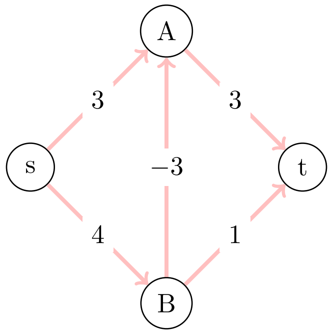
CSC 373 - Algorithm Design, Analysis, and Complexity **Summer 2016**

Lalla Mouatadid

DP: Single Source Shortest Paths, The Bellman-Ford Algorithm

Consider the following graph G:



Suppose we want to compute the shortest st path in G. Running Dijkstra's algorithm on G computes the following values: $d[A] = 3, d[B] = 4, d[t] = 5$, where in fact the shortest distance to A from s has length 1 and the shortest st path has length 4 and not 5.

As we discussed in class, the problem here is that Dijkstra's algorithm makes the assumption that the weights are always positive! And so when a vertex is processed (i.e. removed from the priority queue), we assume that its weight will never change. Which is the case if a graph has no negative edge weights. So how do we solve the shortest path problem if we have a graph with negative weights on the edges?

Notice first that if G has a cycle whose total weight is negative, then the shortest path problem does not make sense anymore, since every walk around the cycle will reduce the total length of a path and repeated walks will keep decreasing the length of the path over and over again. So we rephrase the previous question to how to compute the shortest path problem on a graph with negative edge weights but no negative cycles. Let's formally define our problem:

The Single Source Shortest Path:
Input: A directed edge-weighted graph $G(V, E, w)$ where $w : E \to \mathbb{R}$ and G has no negative cycles. A start vertex s.
Output: All su shortest paths for $u \in V$.

Notice that we need a directed graph, since otherwise an edge with negative weight is a "cycle" on its own in an undirected graph. How do we tackle this problem? Let's look at how Dijkstra's algorithm updates the distances. Suppose v is a vertex with sv distance $d[v]$. Suppose the neighbourhood of v is $N[v] = \{u_1, u_2, \ldots, u_k\}$, and let $d[u_1], d[u_2], \ldots, d[u_k]$ denote the respective shortest distances from s to $u_1, u_2, \ldots, u_k$. Then the shortest sv path would be the path that minimizes $d[u_i] + w(u_i, v)$. We can rewrite this as:

$$d[u] = \min_{1\leq i\leq k} \{d[u_i] + w(u_i, v)\}$$

When we find the right u_i that minimizes the distance, we **relax** $d[v]$. **Relaxation** across an edge (u, v) is the process of assigning a cheaper distance to a vertex v by using a shortest su path and the edge (u, v):

```
Relax(u,v)
If $d[v] > d[u] + w(u,v)$:
$d[v] = d[u] + w(u,v)$
```

The problem with he example above is that A was processed before B, so when we relax across the edge (B, A), A was already removed from the priority queue and thus $d[A]$ does not get updated. The question is whether we can find a way to *not* remove A from the priority queue until we are sure no more updates will decrease its shortest distance from s.

Notice that if we compute the shortest $d[v]$ for a given vertex v, then no matter how many relaxations we perform, $d[v]$ will never decrease (since we have the shortest path)! So here's an ugly algorithm:

Algorithm 1 Meh..

```
1: Set $d[s] = 0$ and $d[v] = \infty$ for all $v \neq s$.
2: for $i = 1 \ldots$ to some $k$ do
3:     Relax every edge in the graph.
4: end for
```

The idea is to relax every edge over and over again and hopefully, we'll *eventually* optimally compute $d[u]$ for some vertex u and then use $d[u]$ to compute the rest of the values. We just don't know how many of these relaxation rounds we need to perform, but it must be some constant k if G has no negative cycles! So what's k? Let's build some intuition.

Suppose the path $P_{sv} = \{s = w_0, w_1, \ldots, w_{x-1}, w_x = v\}$ is an optimal shortest sv path in G. To get to this conclusion (that P_{sv} is optimal), we would have had to relax over the edge (s, w_1), then (w_1, w_2),..., then $(w_{x-1}, w_x = v)$. In fact, since the algorithm above relaxes **all** the edges in G, it is then possible that we have relaxed other sv paths $Q_{sv} = \{s, u_1, \ldots, u_t, v\}$ that are **not** shortest sv paths. But the for **optimal** path P_{sv}, we had to relax across all its edges. So if this idea makes sense, we are formally claiming the following:

Claim: Consider two vertices $s, v \in V$, $s \neq v$, and let P_{sv} be a shortest path connecting s and v with the smallest number of edges $P_{sv} = \{s = w_0, w_1, \ldots, w_{x-1}, w_x = v\}$. Then after x relaxations (i.e. the length of P_{sv}), $d[v]$ is optimal.

Proof. We prove the claim by induction on x. If $x = 1$, then the shortest sv path is the edge (s, v). Therefore after one round of relaxation, $d[v]$ is minimized since we relax across the edge (s, v). The base case holds. Suppose the claim holds for all $j < x$.

Consider P_{sv} with x edges. Let $P_{sv} = \{s = w_0, w_1, w_2, \ldots, w_{x-1}, w_x = v\}$ and consider the distances $d[w_i]$ after $x - 1$ rounds of relaxation. By induction hypothesis, $d[s]$ and $d[w_i]$ for $i \in [x-1]$ are optimal. Now we just perform one more relaxation round, and the edge $(w_{x-1}, w_x = v)$ will be relaxed. Since $d[w_{x-1}]$ is minimal (i.e. optimal), relaxing over (w_{x-1}, v) will also be minimal since P_{sv} is a shortest sv path. $\square$

Tada! But we're not done yet. We said earlier that for some fixed k, we will eventually have optimal distances. And in the claim above, we say that for x relaxation rounds we will also have optimal distances. So what's k and what's x? Well, if we set k to be x, then for this specific P_{sv} path, we are guaranteed an optimal solution. What about other paths? We know for sure that any path from s to any vertex in G will have at most $n - 1$ vertices (i.e. this path contains all of V). Suppose P_{sv} has $n - 1$ edges, then we would visit all of $V \backslash v$ during relaxation rounds, before we get to v. So we would optimally relax all the paths P_{sy} for $y = v$ before we get to v, and thus optimally compute all $d[y]$'s. And by the claim above, we would also optimally compute $d[v]$. Therefore we need at most $n - 1$ relaxation rounds in order to compute $d[\cdot]$ for all the vertices in G. So the algorithm now looks like:

Algorithm 2 The Bellman-Ford Algorithm

```
1: Set $d[s] = 0$ and $d[v] = \infty$ for all $v \neq s$.
2: for $i = 1 \ldots n-1$ do
3:     for every edge $(u, v)$ in $G$ do
4:         Relax$(u, v)$
5:     end for
6: end for
```

This algorithm is known as the Bellman-Ford Algorithm. It is easy to see that it takes $\mathcal{O}(mn)$ time. Everything we have done so far is with respect to graphs with no negative cycles, can you modify or exploit this algorithm to test if G has a negative cycle? And if there is such a cycle, how would you find it?

Notice how this algorithm has a different flavour than all algorithms we've seen so far. Recall greedy algorithms are myopic: At every iteration we make a decision that is best for the moment. In Bellman Ford, we actually relax *all* the edges at every iteration, hoping to build short sub-paths as we go along, and eventually get all shortest paths. This algorithm paradigm is known as **Dynamic Programming**, instead of being myopic at every step, we try to solve the same problem on a smaller instance (in this case, computing optimal sub-paths), and eventually combine these sub-solutions to get a final optimal solution. We will formalize this idea in the next lecture.