Shortest Paths

Let's define a few things:

- **Weighted graph**: a graph where each edge has a label (the weight), generally a real number. The weight of the edge from vertex u to vertex v is often denoted using the weight function $w(u, v)$. Weights are sometimes referred to as costs.
- **Unweighted graph**: a graph without edge labels. We generally assume that the edges all have weight 1.
- **Path**: a sequence of vertices $v_1 \to v_2 \to v_3 \to \ldots \to v_k$ formed by traveling along edges. The weight of a path p is the sum of the weights of the edges on the path and is often denoted as $w(p)$.
- **Shortest path** from u to v: the path p of minimum possible weight $w(p)$ that starts at u and ends at v. The weight of the shortest path from u to v is often denoted as $\delta(u, v)$.

For unweighted graphs, we are able to compute the shortest paths from a source vertex to all other vertices using a breadth-first search. A BFS will find the path between two vertices that uses the fewest edges. Because each edge is weighted equally in an unweighted graph, the path with the fewest edges also happens to be the shortest path. BFS will not work to compute shortest paths on weighted graphs since the path with the fewest edges may not be the shortest if the edges it contains are expensive. Figure **??** shows an example of a weighted graph and a shortest path.

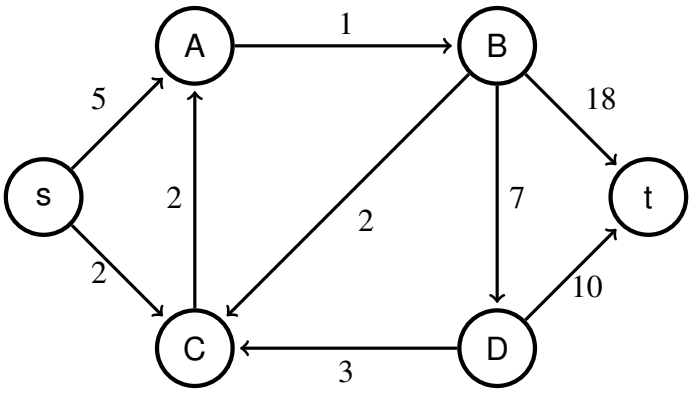


Figure 1: A weighted graph. The shortest path from s to t is $s \to C \to A \to B \to D \to t$. The weight of this path is 22.

There are several variants on the shortest paths problem and the algorithms that we will go over that correspond to solving each problem are in parentheses:

- **Single-source shortest-paths problem:** Find a shortest path from a source vertex to each other vertex in the graph (Bellman-Ford, Dijkstra).

- **Single-destination shortest-paths problem:** Find a shortest path to a destination vertex from each other vertex in the graph (Bellman-Ford/Dijkstra on reversed graph).
- **Single-pair shortest-path problem:** Find a shortest path between a vertex u and a vertex v in a graph (Bellman-Ford, Dijkstra).
- **All-pairs shortest-paths problem:** Find a shortest path between every two vertices in a graph (Bellman-Ford/Dijkstra $|V|$ times, Floyd-Warshall).

Special Cases

Sometimes, there is no path from u to v. For example, in Figure **??**, there is no path from t to s. In this case, we write $\delta(t, s) = \infty$.

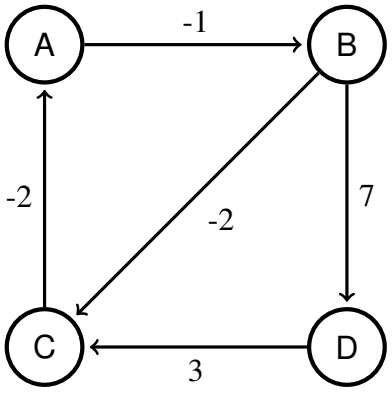


Figure 2: A weighted graph containing a negative cycle $A \to B \to C \to A$ with weight -5.

In addition, edge weights in graphs can be negative. If these edges form a cycle of negative weight, then some shortest paths are undefined. As an example, see the graph in Figure **??**. Because $A \to B \to C \to A$ forms a negative cycle, the shortest path from A to D is undefined. We can create a path with an arbitrarily low weight by traversing the cycle many times before heading over to D.

Notation: Shortest Paths

We will designate the source vertex as s. Every vertex v in the graph is augmented with the following parameters:

- $v.d$ - The weight of the current shortest path from s to v. This is initialized to be ∞ for all vertices besides the source vertex but decreases as paths are found and shorter paths are discovered. At the end of the algorithm, this will be the weight of the shortest path. $s.d$ is initialized to be 0.

- $v.\pi$ - The parent vertex of v in the current shortest path. This is initialized to be NIL but gets set to a vertex once a path is discovered from s to v. As shorter paths to v are discovered, the parent updates to reflect the change. At the end of the algorithms, this will be the parent of v in the shortest path to v. $s.\pi$ will always be NIL.

Also,

- $w(u, v)$ is the weight of the edge from vertex u to vertex v.
- $\delta(u, v)$ is the weight of the shortest path from vertex u to vertex v.

General Structure of Shortest Path Algorithms

Here is the general structure of shortest path algorithms for weighted graphs with no negative cycles. Usually what changes between different algorithms is the order in which edges are relaxed.

Initialize:	for $v \in V$: $v.d \leftarrow \infty$, $v.\pi \leftarrow$ NIL
	$s.d \leftarrow 0$
Main:	repeat
	select edge (u, v) [somehow]
"Relax" edge (u, v)	if $v.d > u.d + w(u, v)$:
	$v.d \leftarrow u.d + w(u, v)$
	$v.\pi \leftarrow u$
	until all edges have $v.d \leq u.d + w(u, v)$

Relaxation

Initializing the algorithms involves setting $v.d$ to ∞ and $v.\pi$ to NIL. Throughout the course of the algorithm, we will need to update these values to store the shortest paths that we find.

At any point, $v.d$ will store the shortest path found *so far* to v. We will repeatly look for better paths and update these $v.d$ values. To do this, we *relax* edges. That is, we will pick an edge and see if we can use it to improve upon currently known shortest paths.

Say we pick edge (u, v). To relax it, we see if we can use it to improve the shortest path found so far to v. We compare two paths and store the better one:

- Whatever best path we have found so far. This path has weight $v.d$.
- The path that travels from s to u, and then uses (u, v) to get to v. This path has weight $u.d + w(u, v)$.

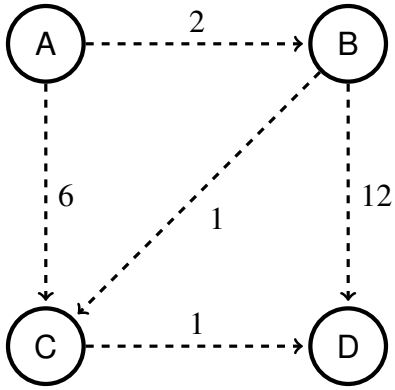
In pseudocode, relaxing the edge (u, v) is:

```
RELAX_EDGE (u, v):
    if v.d > u.d + w(u, v)          (if we find a shorter path to v through u)
        v.d ← u.d + w(u, v)         (update current shortest path weight to v)
        v.π ← u                     (update parent of v in current shortest path to v)
```

Relaxation Algorithm Example

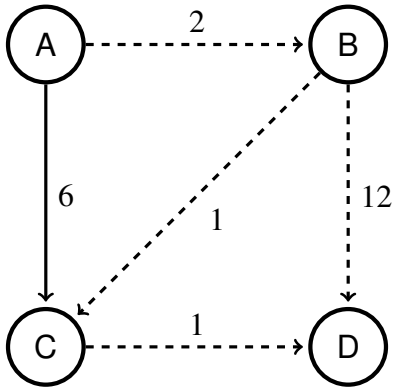
In the following graphs, a solid edge (u, v) means that $v.\pi = u$ (that (u, v) is used in the shortest path so far to v). A dashed edge means otherwise. Initially, all edges are dashed because we have not yet computed any shortest paths.

1. We start with the following graph. Let the source s be node A. We will compute all the shortest paths starting from A.



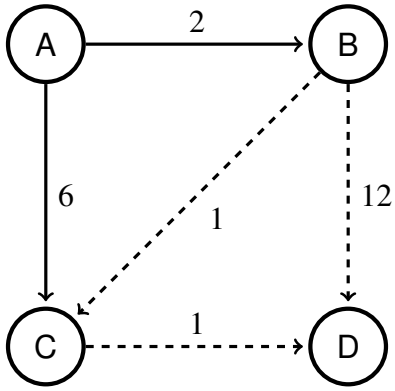
$A.d = 0$
$B.d = \infty$
$C.d = \infty$
$D.d = \infty$

2. We can first relax edge (A, C), setting $C.d = 6$.



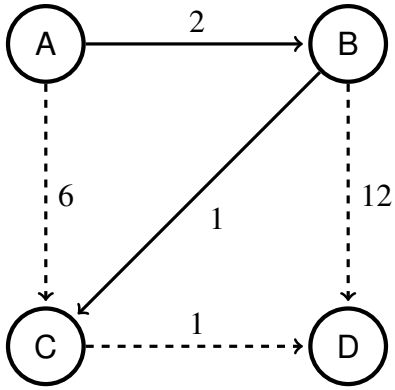
$A.d = 0$
$B.d = \infty$
$C.d = 6$
$D.d = \infty$

3. We can then relax edge (A, B), setting $B.d = 2$.



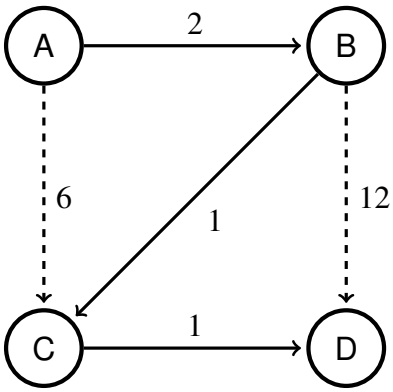
$A.d = 0$
$B.d = 2$
$C.d = 6$
$D.d = \infty$

4. Next, we can relax edge (B, C). We compare the previously computed shortest path to C (weight 6) to the one that goes through B and then uses the edge (B, C). Because the latter has weight 3, we update $C.d$ to 3, improving from the previous value we had stored.



$A.d = 0$
$B.d = 2$
$C.d = 3$
$D.d = \infty$

5. Finally, we can relax edge (C, D), setting $D.d = 4$.



$A.d = 0$
$B.d = 2$
$C.d = 3$
$D.d = 4$

6. At this point, no more edges can be relaxed. Thus, the shortest path lengths we have computed so far are optimal.

Worst Case Running Time

The running time of a shortest path algorithm heavily depends on the order in which edges are relaxed. In some cases the running time could become exponential.

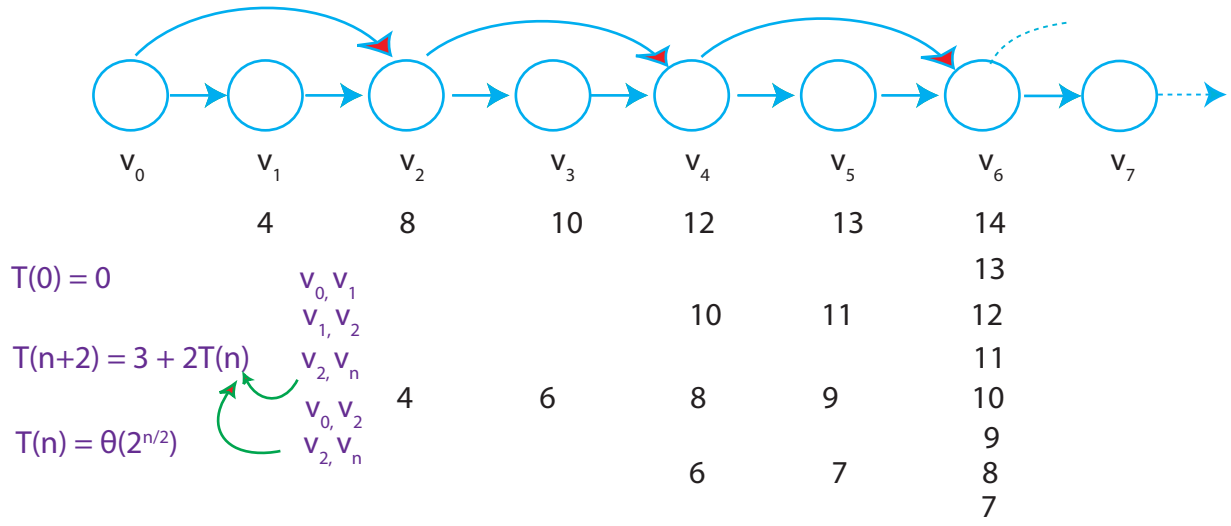


Figure 3: Running generic algorithm with worst-case relaxation order. The outgoing edges from v_0 and v_1 have weight 4, the outgoing edges from v_2 and v_3 have weight 2, the outgoing edges from v_4 and v_5 have weight 1. The order in which edges are relaxed: (v_0, v_1),(v_1, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6), (v_4, v_6), (v_2, v_4),(v_4, v_5), (v_5, v_6), (v_4, v_6), (v_0, v_2), (v_2, v_3), (v_3, v_4), (v_4, v_5), (v_5, v_6),... In this way everytime we relax an edge going into v_6 it's min path length drecreases only by 1.

In a generalized example based on Figure **??**, we have n nodes, and the weights of edges in the first 3-tuple of nodes are $2^{\frac{n}{2}-1}$. The weights on the second set are $2^{\frac{n}{2}-2}$, and so on. A pathological selection (using only bottom edges) of edges will result in the initial value of $(v_{n-1}).d$ to be $2 \times (2^{\frac{n}{2}-1}+2^{\frac{n}{2}-2}+\cdots+4+2+1)$. In this ordering, we may then relax the edge of weight 1 that connects v_{n-3} to v_{n-1}. This will reduce $(v_{n-1}).d$ by 1. After we relax the edge between v_{n-5} and v_{n-3} of weight 2, $(v_{n-2}).d$ reduces by 2. We then might relax the edges (v_{n-3}, v_{n-2}) and (v_{n-2}, v_{n-1}) to reduce $(v_{n-1}).d$ by 1. Then, we relax the edge from v_{n-3} to v_{n-1} *again*. In this manner, we might reduce $(v_{n-1}).d$ by 1 at each relaxation all the way down to $2^{\frac{n}{2}-1} + 2^{\frac{n}{2}-2} + \cdots + 4 + 2 + 1$. This will take $O(2^{\frac{n}{2}})$ time.

In future classes, we will learn how to intelligently pick edges to relax to avoid exponential running times.

Properties of Shortest Paths

Using our definitions of shortest paths and relaxations, we can come up with several properties. These, along with several more properties, can all be found in CLRS in chapter 24.

Triangle inequality: For any edge (u, v), we have $\delta(s, v) \leq \delta(s, u) + w(u, v)$. In english, the weight of the shortest path from s to v is no greater than the weight of the shortest path from s to u plus the weight of the edge from u to v. Note that the triangle inequality only applies to the shortests paths between nodes. It does not apply to the edges between nodes in the graph. A counter example is a triangle graph with two edges with weight 2 and one edge with weight 5. The triangle inequality does not hold for these edge weights but does hold for the minimum paths between nodes.

Optimal substructure: Let $\{v_1, v_2, v_3, ..., v_k\}$ be a shortest path that goes from v_1 to v_k through the vertices v_2 through v_{k-1}. Any subpath $\{v_i, v_{i+1}, ..., v_{j-1}, v_j\}$ must be a shortest path from v_i to v_j. That is, a shortest path is constructed of shortest paths between any two vertices in the path. Proof by contradiction: if the subpath is not a shortest path between v_i and v_j then we can find a shorter path between v_1 and v_k by using the shortest path between v_i and v_j.

Graph Practice Problems

Since we will delve further into specific shortest-path algorithms and their advantages in general over the next few lectures, a related algorithmic skill which we can develop right away is the ability to step back and think about how we can modify existing graphs or accurately model a problem and use Djikstra's, Bellman-Ford, or DAG-SP as block boxes.

Shortest Path with Even or Odd Length

Given an unweighted graph $G = (V, E)$, find the shortest path with an odd number of edges from s to t.

Solution: It's unclear how to solve the problem using G, but it turns out that if we transform G into a new graph G', the problem reduces to a shortest path problem. For every vertex u in G, there are two vertices u_e and u_o in G': these represent reaching the vertex u through even and odd number of edges respectively. For every edge (u, v) in G, there are two edges in G': (u_e, v_o) and (u_o, v_e). Both of these edges have the same weight as the original (weight 1). Constructing this graph takes linear time $O(|V| + |E|)$. Then we can run a BFS from s_e to t_o to find the shortest path.

The weighted graph version of this problem is also solvable using the same technique. The solution will use a weighted graph shortest path algorithm in place of a BFS.

Shortest Cycle

Given a directed weighted graph $G = (V, E)$, find the shortest cycle that starts at s. The cycle must be non-trivial (it must visit some vertex other than s).

Solution: We modify G to include a new vertex, s'. For every edge (u, s) in G, we also add the edge (u, s') to the graph. Then, we find the shortest path from s to s'.