Chapter 24
Single-Source Shortest Paths

A driver wishes to find the shortest possible route from Boston to Chicago, before the Google age. He called up AAA to get a TripTik, which would label the distance between each pair of adjacent towns. How can he determine the shortest route in between, about 1,000 miles?

One way could be to enumerate all the possible routes from Boston to Chicago, add up the distance on each, and select the shortest. It simply takes too long, and many of the routes, e.g., the one via Houston, are poor choices, and should not be considered.

We now discuss a few good, and classic, algorithms that will help us to find out the shortest paths from one source to all the other places.
A few notions

In a shortest-path problem, we are given a weighted digraph $G(V, E, \omega)$, with a weight function $\omega : E \rightarrow R$, and let $p(v_0, v_1, \cdots, v_k)$ be a path in $G$. By $\omega(p)$, the weight of $p$, we mean the sum of the weights of all the edges on the path, $\omega(v_i, v_{i+1}), i\in [0, k−1]$.

For a given pair of vertices, $u, v \in V$, we define the weight of the shortest-path from $u$ to $v$ by

$$\delta(u, v) = \begin{cases} \min_p \{\omega(p) | u \xrightarrow{p} v\} & p \text{ is a path from } u \text{ to } v; \\ \infty & \text{there is no such } p. \end{cases}$$

Finally, a shortest path from $u$ to $v$ is any path $p$ from $u$ to $v$ such that $\omega(p) = \delta(u, v)$.
Applications

1. If using a graph to represent a communication network, with weight representing the cost, then the shortest path leads to the most economic channel. Get there as quickly as possible.

2. If using a graph to represent a transportation network, with weight representing distance, then the shortest path leads to the geographically shortest path. Is this what GPS and Waze all about?

3. Edge weight can also represent time, cost, penalty, loss, and any other quantities that accumulates linearly along a path that one wishes to minimize.

4. ...
Various versions

In this chapter, we study the single source version of the shortest path problem: given a graph $G(V, E, \omega)$, and a given source, $s$, we want to find out the shortest paths from $s$ to all the other vertices in $V$.

If we want to solve the single destination version of the problem, we can reverse the direction of the graph, and apply the solution to the above problem.

Once the solution to the single source version is found out, we immediately have one to solve the single pair version of the problem.

We can also use it to solve the all pairs version of the problem (?).
The greedy nature

Shortest-path problems typically rely on the property that a shortest path between two vertices necessarily contains shortest paths for the intermediate vertices.

Lemma 24.1. Given $G(V, E, \omega)$. Let $p(v_1, \ldots, v_k)$ be a shortest path from vertex $v_1$ to $v_k$ and for any $i, j, 1 \leq i \leq j \leq k$, let $p_1(v_i, \ldots, v_j)$ be the subpath of $p$ from $v_i$ to $v_j$, then $p_1$ is a shortest path from $v_i$ to $v_j$.

Hence, to construct a shortest path, we should rely on those shortest partial paths, which immediately reminds us of the greedy algorithm technique.

But, as we saw earlier (Page 10 of the greedy chapter), a simple minded greedy solution won't work. 😞
A couple of points

1. Given the following graph, the shortest path from $v_1$ to $v_6$ is $v_1, v_4, v_7, v_6$ with weight 6.

2. The shortest path from any node to itself is empty, with length 0.

3. If there is a cycle containing a negative edge, the problem is undefined. For example, if $\omega(v_4, v_3) = -10$ in the above graph, then no algorithm will terminate.

The more we drive, the less gas we use.... 😊
4. A shortest path contains no “positive cycle”, either. Since if it does, we can obtain a strictly shorter path by removing the cycle from the path.

5. Even if the cycle has a weight 0, we can still remove it from the path and obtain a path with the same weight.

Hence, we assume that a shortest path contains no cycles.

6. Besides calculating the distance of a shortest path between two vertices (what), we also want to find out all the intermediate vertices (how). We thus, for each vertex, $v$, maintain a predecessor, $p[v]$.

The sequence of all such vertices going from $v$ back to $s$ provides the actual path, as GPS or Waze would.
Why negative edge?

Let a graph $P$ be a graph, where $V(P)$ collects all the ports that ships might go, and for each $(x, y) \in E(P)$, $\omega(x, y)$ represents the profit a ship makes while going from $x$ to $y$. Then, sometimes, when a ship goes from $x$ to $y$, it carries no goods, thus making a negative profit, cost of labor, oil, ..., which should be so labeled.

Notice that, given a graph, if we switch all the weights $\omega(x, y)$ to $-\omega(x, y)$, and finds a shortest path between two vertices, realizing the shortest total weight, we equivalently find a longest path between these two points, assuming no negative cycle exists.

For the shipping example, we would have found the itinerary that leads to a maximum profit.
The most reliable path

Let $N$ represent a network, $V(N)$ the collection of all the processing nodes, and, for all $(x, y) \in E(N)$, $\omega(x, y)$ represent $p(x, y)$, the reliability of the connection $(x, y)$.

Let $p(x_0, x_1, \ldots, x_n)$ be a path connecting $x_0$ and $x_n$, the reliability of $p(x, y)$, or the probability that $p$ is reliable, is just $p(x_0, x_1) \times \cdots \times p(x_{n-1}, x_n)$, which should be maximized. Equivalently, we want to maximize $\log p(x_0, x_1) + \cdots + \log p(x_{n-1}, x_n)$.

Since, for all $i \in [0, n-1], 0 < \omega(p_i, p_{i+1}) < 1$, $\log(\omega(p_i, p_{i+1})) \leq 0$. Thus, we set $\omega(x, y)$ to $-\omega(x, y)$ (\(= -\log p(x, y)\)), and find a shortest $\omega$-path from $x_0$ to $x_n$, which minimizes $- [\log p(x_0, x_1) + \cdots + \log p(x_{n-1}, x_n)]$.

Since $-x < -y$ iff $x > y$, we will find a most reliable connection between $x_0$ and $x_n$, since it maximizes $\log p(x_0, x_1) + \cdots + \log p(x_{n-1}, x_n)$. 
How could we find such a path?

All the algorithms we will study use an intuitive technique of relaxation, which always tries to find a better deal during the greedy process.

For each vertex \( v \in V \), we maintain an attribute \( d[v] \), which is an upbound of the length of a shortest path from \( s \) to \( v \).

At the very beginning, we initialize \( d[v] \) with the following \( \Theta(|V|) \) process.

**Initialize-Single-Source(G, s)**
1. for each \( v \) in \( V \)
2. do \( d[v]<\text{-maxInt} \) //Know nothing yet
3. \( p[v]<\text{-NIL} \) //What will be its predecessor?
4. \( d[s]<\text{-0} \) //Takes nothing to stay
The relaxing step

This relaxing process, when applied to an edge \((u, v)\), tests if we can improve the shortest path found so far for \(v\) with another one going through \(u\); and if we could, decrease \(d[v]\) and reset \(p[v]\) to \(u\).

Relax\((u, v, w)\)
1. if \(d[v] > d[u] + w(u, v)\) //A better deal
2. then \(d[v] \leftarrow d[u] + w(u, v)\)
3. \(p[v] \leftarrow u\)
The Bellman-Ford algorithm

This algorithm, suggested in 1958, solves the problem in the general case when edge weights may be negative.

Bellman-Ford\((G, w, s)\)
1. Initialize-Single-Source\((G, s)\)
2. for \(i<-1\) to \(|V|-1\) //longest path
3. do for each \((u, v)\) in \(E\)
4. do Relax\((u, v, w)\)
5. //Is there a negative cycle?
6. for each \((u, v)\) in \(E\)
7. do if \(d[v] > d[u] + w(u, v)\)
8. then return False
9. return True

It is clear that this algorithm runs in \(O(|V||E|)\).
An issue is that some of the $d[v]$ values, e.g., $t$ and $z$, have to be changed, which explains the need of such a loop. 😐

**Homework:** Exercise 24.1-1(*).
Theorem 24.4. Let Bellman-Ford be run on a weighted digraph $G(V, E, \omega)$ with source $s$. If $G$ contains no negative-weight cycles reachable from $s$, then the algorithm returns True, when $d[v] = \delta(s, v)$ for all $v \in V$, and the predecessor subgraph is a shortest-paths tree rooted at $s$.

If $G$ does contain a negative-weight cycle reachable from $s$, then the algorithm returns False.

For example, for $(s, u, v, s), \omega(s, u) = 8, \omega(u, v) = -16,$ and $\omega(v, s) = 4$, i.e., $(s, u, v, s)$ constitutes a negative cycle.

After applying the algorithm, $d(s) = 0, d(u) = 8$, and $d(v) = -8$. But, $d(s) = 0 > -4 = d(v) + \omega(v, s)$. Bellman-Form returns False.
How about a DAG?

Recall that, by DAG, we mean *directed acyclic graph*. (Cf. Page 36 of the Graph Basics notes) Since a DAG does not have a cycle, certainly not a negative-weight cycle, thus the single-source shortest-path problem is well defined for this type of graph.

We will show a $O(|V| + |E|)$ algorithm to find out all the shortest paths in a DAG from a source node $s$. What we will do is to run a topological sorting on its vertices to put the vertices into a linear ordering with the nice property that, if there is a path from $u$ to $v$, then $u$ proceeds $v$ in the topological order.

We thus only need to run the relaxing step in one pass through such an order, since no revision is needed once it is done. In other words, no path will go backwards, thus no revision is needed.

This consideration cuts down on its time. 😊
The code

When $G$ is a DAG, we can apply the following algorithm to solve the single source shortest path problem.

\[
\text{DAG-Shortest-Path}(G, w, s)
\]
1. topologically sort $V$
2. Initialize-Single-Source($G$, s)
3. for each $u$, in the topological order
   4. do for each $v$ in Adj[$u$]
   5. do Relax($u$, $v$, $w$)

It takes $O(|V| + |E|)$ to finish line 1, $\Theta(|V|)$ for line 2. Line 3 runs exactly $|V|$ times, and since the sum of all the length of the adjacency lists in a directed graph is $|E|$, (Eq. 1, Page 17 of the Basics notes), and line 5 runs $\Theta(|E|)$.

Hence, this algorithm runs in $O(|V| + |E|)$. 
An example

Below shows an example of applying the algorithm on a DAG, as shown in Part a.

![Diagram of a DAG with multiple states and transitions.](image)

**Homework:** Exercise 24.2-1(*).
Correctness

**Theorem 24.5.** When applied to a DAG with source node being $s$, the algorithm, when terminating, for all $v \in V$, $d[v] = \delta(s, v)$, and the predecessor subgraph is a shortest-path tree.

**Proof:** If $v$ is not reachable from $s$, then $d[v] = \infty = \delta(s, v)$, e.g., $d[r] = \infty$, by the no-path property (Cf. Corollary 24.12).

Otherwise, assume there is a shortest path $p = (v_0, v_1, \ldots, v_k)$ such that $s = v_0$ and $v = v_k$. Since we process the vertices in the topological order, the edges are relaxed in the order of $(v_0, v_1), \ldots, (v_{k-1}, v_k)$, by the path-relaxation property (Cf. Corollary 24.15), $d[v_i] = \delta(s, v_i)$, for all $i \in [0, k]$. In particular, $d[v] = \delta(s, v)$. 
The longest path problem

If we want to find a longest path between two vertices, we simply flip the weights, then apply the Bellman-Ford algorithm, and it will send back such a path, if no positive cycle exists in the original graph. Check out Page 8 and 9 for details, and applications.

On the other hand, the Longest path problem, as we know it, is to find out the longest simple path in a general graph. This problem is much more challenging as compared with its shortest cousin. In fact, it is actually NP-Complete, and dynamic programming technical won’t help. But, for some special graphs, e.g., DAG, efficient algorithms do exist.

For more details, check out the link in the course page.
Dijkstra’s algorithm

This algorithm, also suggested in 1958, solves the single-source shortest-paths problem on a weighted digraph for the case when all edge weights are nonnegative, i.e., for all edge \( e \in E, w(u,v) \geq 0 \). It is the basis of GPS and its company.

Dijkstra(G, w, s)
1. Initialize-Single-Source(G, s) //s gets 0
2. S<-Nil //Nothing is known yet.
3. Q <-V //Build minHeap
4. while !isEmpty(Q) //Not completed yet
5. do u<-Extract-Min(Q)
6. S<-S Union {u} //u is done.
7. for each v in Adj[u] //v still in Q!
8. do Relax(u, v, w) //Make it!

This one is also based on the set structure.
An example

Below shows an example of applying Dijkstra’s algorithm on a graph.

Let’s apply this algorithm to the graph on Page 6.
Discussion

After the usual initialization, the algorithm adds all the vertices into a minimal priority queue, and sets a set $S$ empty.

Every time we go through the while loop, a “minimum” vertex is taken out of the priority queue, added into $S$, and all its incident edges are relaxed.

Both Line 1 and 3 take $\Theta(|V|)$. Since vertices are never inserted back into $Q$ after Line 3, the while loop in Line 4 is executed precisely $|V|$ times. If using minHeap to implement the priority queue, it runs in $O(|E| \log |V|)$ (Lines 7 and 8, which might change the value.)

Since this algorithm always selects a smallest vertex, it also falls into the greedy algorithm category.

We will see more applications of priority queue in *CS4310 Operating Systems*. 
Correctness

Theorem 24.6. When applied to a weighted digraph $G = (V, E)$ with no negative weight function $w$ and a source $s$, Dijkstra’s algorithm terminates with $d[u] = \delta(s, u)$, for all $u \in V$.

Homework: Exercise 24.3-1(*) and 24.3-2(*).

Notice that if a graph contains a negative edge, Dijkstra might fail. For example, Let $V = \{s, u, v\}$, $\omega(s, u) = 5, \omega(u, v) = -10$, and $\omega(s, v) = 2$.

After applying the algorithm, $d(s) = 0, d(u) = 5$, and $d(v) = 2$. But, $\delta(s, v) = -5$.

“It does not fit, we have to quit.” ☹️