Chapter 24
Single-Source Shortest Paths

A driver wishes to find the shortest possible route from Boston to Chicago, before the GPS age. She could call AAA to get a *TripTik*.

A shortest path is drawn, *by hand*, between the two cities, based on a map where distances between all the intermediate cities are listed, as well as the locations of the gas pump, and rest areas.

These days, you can also use *TripTiks* apps.

**Question:** How could *it* be done?
I am guessing...

A systematic way would be to enumerate all the possible routes from Boston to Chicago, add up the distance on each, and select the shortest, perhaps the one via I-90W, about 980 miles in between.

It simply takes too long to enumerate them, lots of them, an exponential amount, to be exact. Thus, such an algorithm is not efficient, not even effective. 😞

Many of the routes, e.g., the one via Houston, are poor choices, thus should not be considered.

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

This is the idea behind such popular apps as GPS, and Waze, dug out back in the 1950’s.
What do you mean?

Let \( p(v_0, \cdots, v_k) \) be a path in a weighted di-graph \( G(V, E, \omega) \).

By \( \omega(p) \), the weight of \( p(v_0, \cdots, v_k) \), we mean the sum of the weights of all the edges on the path, \( \omega(v_i, v_{i+1}) \), \( i \in [0, k-1] \), e.g., \( \omega(A, B, F, E) = 30 \), and \( \omega(A, C, E) = 25 \),

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

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

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

Here, \( (A, B, D, E) \) is the shortest path from \( A \) to \( E \), since its weight of 24 is the minimum.
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, profit, penalty, loss, reliability, and any other quantities that accumulates linearly along a path that one wishes to minimize.

*Appropriate applications can then be found....*

4. ...
Various versions of the problem

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 $V$, i.e., $V \setminus \{s\}$.

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, by repeating the above for all the other vertices in $V$.

Is it pretty general? 😊
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. We talked about this issue on Day 1 of this course. 😊

**Lemma 24.1.** Let \( p(v_1, \ldots, v_k) \) be a shortest path in a graph \( G \) 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 \( v_1 \) to \( v_k \), 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 as we went through in Chapter 16.

Notice that, as we saw earlier, a simple minded greedy solution won’t work. 😞
It does not always work!

Given the following digraph:

we want to find out the shortest path from $v_1$ to $v_5$. An intuitive way is to find it in stages. At a certain stage, if the path built so far ends at vertex $q$, we can select the nearest vertex that is adjacent to $q$, but not on the path yet.

For our example, this strategy leads to $(v_1, v_3, v_4, v_2, v_5)$ of length 10, which is certainly not the shortest one. 😞

We will find out several that do... 😊
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 negative cycle, the problem is undefined. For example, if $\omega(v_4, v_3) = -10$, then $\omega(v_1, v_4, v_3, v_1) = -5$. Starting with any vertex on such a cycle, e.g., $v_3$, no shortest algorithm would terminate.

*The more we drive, the less gas it burns.... 😊*
4. A shortest path should contain no *positive cycle*, either. Since if it does, we can obtain a strictly shorter path by removing the cycle from the path.

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

5. Besides calculating the distance of a shortest path between two vertices (*How much will that be?*), we also want to find out all the intermediate vertices (*How to get there?*). We thus, for each vertex, \( v \), maintain a predecessor, \( p[v] \) (*How to get to \( v \)?*).

For example, a shortest path from \( v_1 \) to \( v_6 \) can be obtained by flipping all the direction of the edges in: \( v_1 \leftarrow v_4 \leftarrow v_7 \leftarrow v_6 \).

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 $y$.

Sometimes, when a ship goes from $x$ to $y$, it carries no goods, thus making a negative profit, cost of labor, oil, ..., 😞. Such an edge should be labeled negative.

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

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

*Go for it!* 😊
I want to see...

Given the following graph

It is clear that the shortest path from \( s \) to \( t \) is \((s, w, t)\) with its length being 2.

To find the longest one, we just flip all the weights, then find out the “shortest path”, \((s, u, v, t)\) with its total weight being -5, the corresponding length of the longest path in the original graph is 5.

Another application is to find the most reliable path between two nodes in a network. Have a look of the next example if you are interested...
The most reliable path

Let $N$ denote a communication network, $V(N)$ the collection of all the processing nodes, and, for all $(x, y) \in E(N)$, $\omega(x, y) = \text{prob}(x, y)$, the reliability of $(x, y)$, or the probability that $(x, y)$ is reliable, $\omega(x, y) \in [0, 1]$.

Let $p(x_0, x_1, \ldots, x_n)$ be a path connecting $x_0$ and $x_n$, the reliability of $p(x_0, x_1, \ldots, x_n)$, is just $\omega(x_0, x_1) \times \cdots \times \omega(x_{n-1}, x_n)$, which should be maximized by maximizing $\log \omega(x_0, x_1) + \cdots + \log \omega(x_{n-1}, x_n)$, since $\log x$ is monotonically increasing.

Since $\omega(x_i, x_{i+1}) \leq 1$, $\log(\omega(x_i, x_{i+1})) \leq 0$. We now just reset $\log(\omega(x, y))$ to $-\log(\omega(x, y))$, and find a shortest path from $x_0$ to $x_n$, which minimizes $- [\log \omega(x_0, x_1) + \cdots + \log \omega(x_{n-1}, x_n)]$.

Since $-x < -y$ iff $x > y$, we will actually maximize $\log \omega(x_0, x_1) + \cdots + \log \omega(x_{n-1}, x_n)$, i.e., $\omega(x_0, x_1) \times \cdots \times \omega(x_{n-1}, x_n)$, thus finding a most reliable connection between $x_0$ and $x_n$. 
I want to see...

**Question:** Given the associated probabilities of the edges, which path should we take?

Since \( \text{prob}(s, u) = \text{prob}(v, t) = \frac{1}{4} \), and \( \text{prob}(u, v) = \text{prob}(s, w) = \text{prob}(w, t) = \frac{1}{2} \); and \( \log_2 \frac{1}{4} = -2 \), and \( \log_2 \frac{1}{2} = -1 \), we set \( \omega(s, u) = \omega(v, t) = - \left[ \log_2 \frac{1}{4} \right] = 2 \), \( \omega(u, v) = \omega(s, w) = \omega(w, t) = 1 \).

Now, \( \omega(s, w, t) = 2 \), less than \( 5 \) (\( = \omega(p(s, u, v, t)) \)).

This gives the shortest path, \( (s, w, t) \), in the flipped graph, i.e., the most reliable one in the original one, with a reliability of \( 2^{-2} = \frac{1}{4} \) (\( = \frac{1}{2} \times \frac{1}{2} \)), greater than \( 2^{-5} = \frac{1}{32} \) (\( = \frac{1}{2} \times \frac{1}{4} \times \frac{1}{4} \)), that of \( p(s, u, v, t) \).

We would take \( (s, w, t) \).
How could we find such a path?

All the algorithms we will study use an intuitive technique of relaxation, which 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 upper bound of the length of a shortest path from $s$ to $v$. It is the worst that we should expect. 😞

**Question**: Where should we start?

At the very beginning, we initialize $d[v]$, the distance, known so far, from a starting place to $v$, with the following $\Theta(|V|)$ process.

```
Initialize-Single-Source(G, s)
1. for each v in V
2.   do d[v]<-maxInt //Know nothing yet
3.       p[v]<-NIL //Have no idea how to get to v
4.   //Take nothing to come back to itself
5. d[s]<-0 // d(s, s)=0
```

Notice the source vertex $s$ does not have a predecessor, as set in Line 3, as well.
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 \(u\).

**Question:** Could we make it better?

Relax\((u, v, w)\)
1. if \(d[v] > d[u] + w(u, v)\) //Got a better deal
2. then \(d[v] <- d[u] + w(u, v)\) //grab it!
3. \(p[v] <- u\) //Where did we get it?

We found a better deal on the left, 😊 but not on the right, since \(5 + 2 = 7 > 6\). 😞
The Bellman-Ford algorithm

This algorithm, suggested in 1958 by..., 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 //a 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 $O(|V||E|)$.

Notice that the length of the longest path of $|V|$ vertices is $|V|−1$, thus we only need to run the loop this many times, when looking for a path in between s and another vertex.
An issue is that some of the $d[v]$ values, e.g., $t$ and $z$, have to be changed, through “feedback”, which explains the need of such a loop.

After working out relaxation step along the longest path, if the condition in Line 7 still holds, there is a negative cycle. For the 10 edges for this graph, nothing happens, so we are all set. For example,

$$2 = d(t) \leq 6 = d(s) + \omega(s, t).$$

**Assignment:** Exercise 24.1-1.
Correctness

**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 $\text{True}$, when $d[v] = \delta(s, v)$ for all $v \in V$.

If $G$ does contain a negative-weight cycle reachable from $s$, then the algorithm would fail. 😞

For example, in the following graph, $\omega(s, u) = 3$, $\omega(u, v) = 2$, and $\omega(v, s) = -6$. Thus, $(s, u, v, s)$ constitutes a negative cycle.

![Graph](image)

After applying the algorithm, using $(v, s)$,

\[ d(s) = 0 > -1 = d(v) + \omega(v, s) = 5 + (-6). \]

Bellman-Form returns $\text{False}$, thus failed. 😞
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 Pages 10 through 13 for details, and applications.)

On the other and, the Longest path problem, as commonly understood, 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 NP-Complete, and dynamic programming technique won’t help. 😞 But, for some special graphs, e.g., DAG, efficient algorithms do exist.

**Question:** What is the longest hiking trail in the United States? 😊

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

This algorithm, suggested in 1958 by Dijkstra, solves the single-source shortest-path problem on a weighted digraph for the case when all edge weights are nonnegative, i.e., for every edge \( e \in E, w(u, v) \geq 0 \). It is the basis of GPS and its company, as no distance in real life is negative.

Dijkstra(G, w, s)
0. //Check out the one on Page 14, where \( d[s] = 0 \)
1. Initialize-Single-Source(G, s)
2. \( S \leftarrow \text{NIL} \) //Nothing is known yet.
3. \( Q \leftarrow V \) //Build \( Q \), a minHeap
4. while !isEmpty(Q) //Not completed yet
5. do \( u \leftarrow \text{Extract-Min}(Q) \)
6. \( S \leftarrow S \cup \{u\} \) //\( u \) is done and labeled black.
7. for each \( v \) in \( \text{Adj}[u] \) //\( v \) still in \( Q \)
8. do Relax\((u, v, w)\) //A better deal?

This algorithm is based on the set structure, and \textit{minHeap}, which we have worked a few times, including two projects. We will see a lot more in \textit{CS 4310 Operating Systems}. 

20
I want to see...

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

![Diagrams](image)

**Assignment:** Apply this algorithm to the graph on Page 8 to get the correct shortest paths from $v_1$ to all the other vertices.

Read more about this influential algorithm through the link on the course page.
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 via $\text{Relax}(u, v, w)$ as shown on Page 15.

Both Line 1 and 3 take $\Theta(|V|)$. Since vertices are never inserted back into $Q$ after Line 3, Line 4 runs precisely $|V|$ times, and Line 7 runs $|E|$. If we use $\text{minHeap}$ to implement the priority queue, Line 8 runs in $O(|E| \log |V|)$, which is a $\text{decreaseKey}$ operation, maybe changing the value.

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

We once talked about a simple minded greedy approach won’t work. (Cf. Page 7 of this set of notes.)

Dijkstra algorithm works it out. 😊
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 $u \in V$.

**Assignment:** Exercise 23.3-1.

Notice that if a graph contains a negative edge, Dijkstra might fail. 😞

After applying the algorithm, $d(s) = 0$, $d(u) = 5$, and $d(v) = 1$. But, $\delta(v) = -5 < 1 = d(v)$. 😞

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

“If It Doesn’t Fit, You Must Acquit.” (Johnnie Cochran 1995)

We have to quit here, for another reason... .