Given a (weighted, directed) graph \( G \) and a pair of vertices \( s \) and \( t \), we would like to find a shortest path from \( s \) to \( t \).

A fundamental task with many applications:

- Internet routing (e.g. the OSPF routing algorithm)
- VLSI routing
- Traffic information systems
- Robot motion planning
- Routing telephone calls
- Avoiding nuclear contamination
- Destabilising currency markets
- . . .

Formally, a shortest path from \( s \) to \( t \) in a graph \( G \) is a sequence \( v_1, v_2, \ldots, v_m \) such that the total weight of the edges \( s \rightarrow v_1, v_1 \rightarrow v_2, \ldots, v_m \rightarrow t \) is minimal.
Single-source shortest paths

In fact, the algorithms we will discuss for this problem give us more: given a source $s$, they output a shortest path from $s$ to every other vertex.

This is known as the single-source shortest path problem (SSSP).

![Graph with weights]

Negative-weight edges

If some of the edges have negative weights, the idea of a shortest path might not make sense.

If there is a cycle in $G$ which is reachable on a path from $s$ to $t$, and the sum of the weights of the edges in the cycle is negative, then we can get from $s$ to $t$ with a path of arbitrarily low weight by repeatedly going round the cycle.

![Graph with negative weights]

Today’s lecture

Today we will discuss an algorithm for the single-source shortest paths problem called the Bellman-Ford algorithm.

The algorithm can be used for graphs with negative weights and can detect negative-weight cycles.

It also has applications to solving systems of difference constraints and detecting arbitrage.

Remark: One algorithmic idea to solve the SSSP that doesn’t work is to try every possible path from $s$ to $t$ in turn.

There can be exponentially many paths so such an algorithm cannot be efficient.

Notation

We will use the following notation (essentially the same as CLRS):

- We always let $G$ denote the graph in which we want to find a shortest path. We use $V$ for the number of vertices in $G$, and $E$ for the number of edges. $s$ always denotes the source.
- We write $u \rightarrow v$ for an edge from $u$ to $v$, and $w(u, v)$ for the weight of this edge.
- We write $\delta(u, v)$ for the distance from $u$ to $v$, i.e. the length (total weight) of a shortest path from $u$ to $v$.
- We write $\delta(u, v) = \infty$ when there is no path from $u$ to $v$.
  (Mathematical note: in practice, $\infty$ would be represented by a number so large it could never occur in distance calculations...)
- For each vertex $v$, we will maintain a guess for its distance from $s$; call this $v.d.$
Predecessors and shortest paths

- For each vertex $v$, we try to determine its predecessor $v.\pi$, which is the previous vertex in some shortest path from $s$ to $v$.

- Knowledge of $v$'s predecessor suffices to compute the whole path from $s$ to $v$, by following the predecessors back to $s$ and reversing the path.

A general framework

The basic idea behind both shortest-path algorithms we will discuss is:

1. Initialise a guess $v.d$ for the distance from the source $s$: $s.d = 0$, and $v.d = \infty$ for all other vertices $v$.

2. Update our guesses by relaxing edges:
   - If there is an edge $u \rightarrow v$ and our guess for the distance from $s$ to $v$ is greater than our guess for the distance from $s$ to $u$, plus $w(u, v)$, then we can improve our guess by using this edge.

\[
\text{Relax}(u, v) = \begin{cases} 
  1. & v.d > u.d + w(u, v) \\
  2. & v.d \leftarrow u.d + w(u, v) \\
  3. & v.\pi = u 
\end{cases}
\]

Note that $\infty + x = \infty$ for any real number $x$.

Example 1: no negative-weight cycles

Imagine we want to find shortest paths from vertex $A$ in the following graph:

The Bellman-Ford algorithm

This algorithm simply consists of repeatedly relaxing every edge in $G$.

\[
\text{BellmanFord}(G, s) = \begin{cases} 
  1. & \text{for each vertex } v \in G: v.d \leftarrow \infty, v.\pi \leftarrow \text{nil} \\
  2. & s.d \leftarrow 0 \\
  3. & \text{for } i = 1 \text{ to } V - 1 \\
  4. & \text{for each edge } u \rightarrow v \text{ in } G \\
  5. & \text{Relax}(u, v) \\
  6. & \text{for each edge } u \rightarrow v \text{ in } G \\
  7. & \text{if } v.d > u.d + w(u, v) \\
  8. & \text{error("Negative-weight cycle detected")}
\end{cases}
\]

- Time complexity: $\Theta(V) + \Theta(VE) + \Theta(E) = \Theta(VE)$. 
Example 1: no negative-weight cycles

At the start of the algorithm:

- In the above diagram, the red text is the distance from the source $A$, (i.e. $v.d$), and the green text is the predecessor vertex (i.e. $v.\pi$).

Note that the edges are picked in arbitrary order.

Example 1: no negative-weight cycles

The first iteration of the for loop:

So the shortest path from $A$ to $G$ (for example) has weight 1.

To output a shortest path itself, we can trace back the predecessor values from $G$. 

Example 1: no negative-weight cycles

The second iteration of the for loop:

Example 1: no negative-weight cycles

The 4 iterations of the for loop that follow do not update any distance or predecessor values, so the final state is:
Example 2: negative-weight cycle

We now consider an input graph that has a negative-weight cycle.

At the start of the algorithm:

The first iteration of the for loop:

The second iteration of the for loop:

As before, the order in which we consider the edges is arbitrary (here we use the order $A \to B$, $C \to A$, $B \to C$).

At the end of the algorithm, $B.d > A.d + w(A, B)$.

So the algorithm terminates with “Negative-weight cycle detected”.

▶ As before, the order in which we consider the edges is arbitrary (here we use the order $A \to B$, $C \to A$, $B \to C$).

▶ At the end of the algorithm, $B.d > A.d + w(A, B)$.

▶ So the algorithm terminates with “Negative-weight cycle detected”.
Proof of correctness: Preliminaries

Claim (cycles)
If $G$ does not contain any negative-weight cycles reachable from $s$, a shortest path from $s$ to $t$ cannot contain a cycle.

Proof
If a path $p$ contains a cycle $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_0$ such that the sum of the weights of the edges is non-negative, deleting this cycle from $p$ cannot increase $p$'s total weight.

Finally, an important property of relaxation, which can be proven by induction and using the triangle inequality, is called path-relaxation:

Claim (path-relaxation)
Assume that:
- $p = s \rightarrow v_1 \rightarrow \cdots \rightarrow v_k \rightarrow v$ is a shortest path from $s$ to $v$;
- $s.d$ is initially set to 0 and $u.d$ is initially set to $\infty$ for all $u \neq s$;
- the edges in $p$ are relaxed in the order they appear in $p$ (possibly with other edges relaxed in between).

Then, at the end of this process, $v.d = \delta(s, v)$.

Proof: exercise.

Proof of correctness

Claim
If $G$ does not contain a negative-weight cycle reachable from $s$, then at the completion of BellmanFord, $v.d = \delta(s, v)$ for all vertices $v$.

Proof
- Write $v_0 = s$, $v_m = v$. If $v$ is reachable from $s$, there must exist a shortest path $v_0 \rightarrow v_1 \rightarrow \cdots \rightarrow v_m$.
- A shortest path cannot contain a cycle, so $m \leq V - 1$.
- In the $i$th iteration of the for loop, the edge $v_{i-1} \rightarrow v_i$ is relaxed (among others).
- By the path-relaxation property, after $V - 1$ iterations, $v.d = \delta(s, v)$.
- So $V - 1$ iterations suffice to set $v.d$ correctly for all $v$. \[\square\]
**Proof of correctness**

**Claim**

If $G$ does not contain a negative-weight cycle reachable from $s$, then BellmanFord does not exit with an error.

**Proof**

- By the triangle inequality, for all edges $u \rightarrow v$,
  
  $\delta(s, v) \leq \delta(s, u) + w(u, v)$.

- By the claim on the previous slide, $v.d = \delta(s, v)$ for all vertices $v$.

- So, for all edges $u \rightarrow v$, $v.d \leq u.d + w(u, v)$.

- So the check in step (7) of the algorithm never fails.

**Proof of correctness**

**Claim**

If $G$ contains a negative-weight cycle reachable from $s$, then BellmanFord exits with an error.

**Proof**

- We will assume that $G$ contains a negative-weight cycle reachable from $s$, and that BellmanFord does not exit with an error, and prove that this implies a contradiction.

- Let $v_0, \ldots, v_k$ be a negative-weight cycle, where $v_k = v_0$.

- Then by definition $\sum_{i=1}^{k} w(v_{i-1}, v_i) < 0$.

- As BellmanFord does not exit with an error, for all $1 \leq i \leq k$, $v_i.d \leq v_{i-1}.d + w(v_{i-1}, v_i)$.

  $\ldots$

**Application 1: difference constraints**

- A system of difference constraints is a set of inequalities of the form $x_i - x_j \leq b_j$, where $x_i$ and $x_j$ are variables and $b_j$ is a real number.

- For example:

  
  $x_1 - x_2 \leq 5$, $x_2 - x_3 \leq -2$, $x_1 - x_4 \leq 0$.

- Given a system of $m$ difference constraints in $n$ variables, we would like to find an assignment of real numbers to the variables such that the constraints are all satisfied, if such an assignment exists.

- For example, the above system is satisfied by $x_1 = 0$, $x_2 = -1$, $x_3 = 1$, $x_4 = 7$ (among other solutions).

- We will show that this problem can be solved using Bellman-Ford in time $O(nm + n^2)$. 

### Claim
Let $G$ be the graph corresponding to a system of difference constraints. If $G$ does not contain a negative-weight cycle, the assignment $x_i = \delta(v_0, v_i)$, for all $1 \leq i \leq n$, is a valid solution to the system of constraints.

### Proof
- We need to prove that 
  \[ \delta(v_0, v_i) - \delta(v_0, v_j) \leq b_{ij} \]
  for all $i, j$ in the list of constraints.
- This follows from the triangle inequality 
  \[ \delta(v_0, v_i) \leq \delta(v_0, v_j) + \delta(v_j, v_i) \leq \delta(v_0, v_j) + w(v_j, v_i) = \delta(v_0, v_j) + b_{ij} \]
  and rearranging.

### Example
The set of inequalities 
\[
    x_1 - x_2 \leq 5, \quad x_2 - x_3 \leq -2, \quad x_1 - x_4 \leq 0
\]
corresponds to the graph

With shortest paths
\[
    \delta(v_0, v_1) = 0, \quad \delta(v_0, v_2) = -2, \quad \delta(v_0, v_3) = 0, \quad \delta(v_0, v_4) = 0.
\]
So
\[
    x_1 = 0, \quad x_2 = -2, \quad x_3 = 0, \quad x_4 = 0
\]
is a solution to the constraints.
Solving difference constraints

- We can run Bellman-Ford with $v_0$ as the source.
- If there is a negative-weight cycle, the algorithm detects it (and we output “no solution”); otherwise, we output $x_i = \delta(v_0, v_i)$ as the solution.
- For a solution to a system of $m$ difference constraints on $n$ variables, the graph produced has $n + 1$ vertices and $m + n$ edges.
- The running time of Bellman-Ford is thus $O(VE) = O(mn + n^2)$.
- This can be improved to $O(mn)$ time (CLRS exercise 24.4-5).

Application 2: Currency exchange

Imagine we have $n$ different currencies, and a table $T$ whose $(i, j)$'th entry $T_{ij}$ represents the exchange rate we get when converting currency $i$ to currency $j$. For example:

<table>
<thead>
<tr>
<th>£</th>
<th>$</th>
<th>€</th>
</tr>
</thead>
<tbody>
<tr>
<td>£ 1</td>
<td>1.61</td>
<td>1.18</td>
</tr>
<tr>
<td>$ 0.62</td>
<td>1</td>
<td>0.74</td>
</tr>
<tr>
<td>€ 0.85</td>
<td>1.35</td>
<td>1</td>
</tr>
</tbody>
</table>

- If we convert currency $i \rightarrow j \rightarrow k$, the rate we get is the product of the individual rates.
- If we convert $i \rightarrow j \rightarrow \cdots \rightarrow i$, and the product of the rates is greater than 1, we have made money by exploiting the exchange rates! This is called arbitrage.
- We can use Bellman-Ford to determine whether arbitrage is possible.

Application: Currency exchange

We produce a weighted graph $G$ from the currency table, where the weight of edge $i \rightarrow j$ is $-\log_2 T_{ij}$. For example:

Then the weight of a cycle $c_0 \rightarrow c_1 \rightarrow \cdots \rightarrow c_k$ (with $c_k = c_0$) is

$$- \sum_{j=1}^{k} \log_2 T_{c_j c_{j-1}} = - \log_2 \prod_{j=1}^{k} T_{c_j c_{j-1}}.$$  

- This will be negative if and only if $\prod_{j} T_{c_j c_{j-1}} > 1$, i.e. the sequence of transactions corresponds to an arbitrage opportunity.
- So $G$ has a negative-weight cycle if and only if arbitrage is possible.

Summary

- The Bellman-Ford algorithm solves the single-source shortest paths problem in time $O(VE)$.
- It works if the input graph has negative-weight edges, and can detect negative-weight cycles.
- Although the proof of correctness is a bit technical, the algorithm is easy to implement and doesn't use any complicated data structures.
- It can be used to solve a system of difference constraints and to determine whether arbitrage is possible.
Biographical notes

Richard E. Bellman (1920–1984)

- American mathematician who worked at Princeton, Stanford, the RAND Corporation and the University of Southern California.
- Author of at least 621 papers and 41 books, including 100 papers after the removal of a brain tumour left him severely disabled.
- Winner of the IEEE Medal of Honor in 1979 for his invention of dynamic programming.

Lester Ford, Jr. (1927–)

- Another American mathematician whose other contributions include the Ford-Fulkerson algorithm for maximum flow problems.
- His father was also a mathematician and, at one point, President of the Mathematical Association of America.