Graph Algorithms
Single-source shortest path in DAG

Shortest paths are always well defined in a DAG, since there are no negative-weight cycle in a graph.

- If the DAG contains a path from \( u \) to \( v \), \( u \) precedes \( v \) in the topological sort.
- If \( u \) comes before \( v \) in the topological order, there is no path from \( v \) to \( u \).
Single-source shortest path in DAG

DAG-Shortest-Paths(G, s)

1. Topologically sort the vertices of G
2. $d[s] \leftarrow 0$
3. For each $v \in V - \{s\}$ do $d[v] \leftarrow \infty$
4. For each vertex $u$, taken in topologically sorted order
   1. For each $v \in Adj[u]$
   2. If $d[v] > d[u] + w(u, v)$ then $d[v] \leftarrow d[u] + w(u, v)$
Single-source shortest path in DAG: Example

Why is it working in graphs with negative edges?
Single-source shortest path in DAG: Runtime: $\Theta(V+E)$

DAG-Shortest-Paths(G, s)

Topologically sort the vertices of G

$d[s] \leftarrow 0$

for each $v \in V - \{s\}$

\[ d[v] \leftarrow \infty \]

for each vertex $u$, taken in topologically sorted order

for each $v \in Adj[u]$

if $d[v] > d[u] + w(u, v)$

\[ d[v] \leftarrow d[u] + w(u, v) \]
Single-source shortest path in DAG: Correctness

**Theorem.** When the algorithm terminates, \( d[v] = \delta(s, v) \) for all vertices \( v \in V \)

**Proof.**

- If \( v \) is not reachable from \( s \), then \( d[v] = \delta(s, v) = \infty \)
- If \( v \) is reachable from \( s \), there is a shortest path \( p=\langle v_0, v_1, \ldots, v_k \rangle \) where \( v_0 = s \) and \( v_k = v \).
- The algorithm process the vertices in topologically sorted order.
- Therefore, the edges on \( p \) are relaxed in the order \( (v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k) \)
- We can prove by induction on the number of relaxation steps that \( d[v] = \delta(s, v) \)
**Theorem.** After the k-th edge of path p is relaxed, we have $d[v_k] = \delta(s, v_k)$

**Proof by induction:** induction on the number of relaxation steps.

**Induction hypothesis:** After the i-th edge of path p is relaxed, $d[v_i] = \delta(s, v_i)$

**Base Case: i=0**
- before any edge of p have been relaxed, we have $d[v_0] = d[s] = 0 = \delta(s, s)$

**Induction step.** Assuming $d[v_{i-1}] = \delta(s, v_{i-1})$ after the (i-1)-th edge was relaxed → we want to show that $d[v_i] = \delta(s, v_i)$ after the i-th edge is relaxed

- $d[v_i] \leq \delta(s, v_i)$
  - After relaxing edge $(v_{i-1}, v_i)$, we have $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$
    - before relaxing the edge, there are two cases
      - $d[v_i] > d[v_{i-1}] + w(v_{i-1}, v_i)$ if this is the case the algorithm does the following
        - $d[v_i] = d[v_{i-1}] + w(v_{i-1}, v_i)$
      - $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ if this is the case, no change happen and the property holds
  - $d[v_i] \leq \delta(s, v_{i-1}) + w(v_{i-1}, v_i) = \delta(s, v_i)$ (subpaths of shortest path are also shortest path)

- $d[v_i] \geq \delta(s, v_i)$

Therefore $d[v_i] = \delta(s, v_i)$
Theorem. \( d[v] \geq \delta(s, v) \) for all \( v \)

Proof by induction: induction on the number of relaxation steps.

Induction hypothesis: After \( j \) relaxation steps, \( d[v] \geq \delta(s, v) \) for all \( v \).

Base Case: after initialization, \( d[v] = \infty \rightarrow d[v] \geq \delta(s, v) \)
\( d[s] = 0 \geq \delta(s, s) = 0 \)

Induction step. Assuming that induction hypothesis is true for \( j \), we want to prove it is true for \( j+1 \):
Consider relaxation of edge \((u,v)\). There are two cases:

- \( d[v] \) does not change \( \rightarrow d[v] \geq \delta(s, v) \) (induction assumption)
- \( d[v] \) will change: \( d[v] = d[u] + w(u,v) \geq \delta(s, u) + w(u,v) \geq \delta(s, v) \)

Triangle inequality
Bellman-Ford Algorithm
Bellman-ford Algorithm

- If G has no negative cycles, then there exists a shortest path from s to any node u that uses at most n-1 edges.

**Proof.** Suppose there exists a shortest path from s to u consisting of n or more edges
  - A path of length at least n must visit at least n+1 nodes
  - There exists a node x that is repeated (pigeonhole principle) → There is a cycle C
  - Can remove C without increasing cost of path
Bellman-ford Algorithm

**Intuition.** Although Dijkstra’s algorithm may not compute all distances in one pass, it will compute the distance to some vertices correctly, e.g. first vertex on a shortest path.

How many iterations of dijkstra algorithm is required?

If there is no negative-weight cycle

- shortest path is a simple path
- Shortest path is of length at most n-1

→ At most n-1 iterations of Dijkstra is needed

→ Each iteration starts at the next node in the shortest path
Bellman-ford Algorithm: dynamic programming approach

- The problem has the optimal substructure property:
  - All subpaths of a shortest path are shortest paths.
- Can we solve the problem using dynamic programming?
- Can we solve the problem recursively?
- What is the subproblem?
Bellman-ford Algorithm: dynamic programming approach

- $P = \text{shortest path from } u \text{ to } v \text{ with at most } i \text{ edges}$
- $P = P' + (t, v)$
  - $P'$: (shortest path from $u$ to $t$ with at most $i-1$ edge)
Bellman-ford Algorithm: dynamic programming approach

$D(i,v) =$ weight of a shortest path from $s$ to $v$ that uses at most $i$ edges

- Goal $D(n-1, v)$ for each $v$
  - If there is no negative cycle, then there exists a shortest path that is simple

\[
D(i, v) = \min \left\{ D(i-1, v), \min_{(u,v) \in E} \{D(i-1, u) + w(u,v)\} \right\}
\]

- Shortest path uses at most $i-1$ edges
- Shortest path uses exactly $i$ edges

$D(0, s) = 0$

$D(0, v) = \infty$ where $v \neq s$
Bellman-ford Algorithm: dynamic programming approach

For each node $v \in V$

$M[0, v] = \infty$

$M[0, s] = 0$

for $i = 1$ to $n - 1$

for each node $v \in V$

$M[i, v] = M[i - 1, v]$

for each edge $(u, v) \in E$:

$M[i, v] \leftarrow \min \{ M[i, v], M[i - 1, u] + w(u, v) \}$.

- Runtime: $O(nm)$
- Space Complexity: $O(n^2)$
  - Could be improved to $O(n)$
  - To compute $M[i, v]$ only $M[i-1,v]$ values are needed
Bellman-ford Algorithm: DP: example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D(0, v)$</td>
<td>0</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$D(1, v)$</td>
<td>0</td>
<td>-1</td>
<td>4</td>
<td>$\infty$</td>
<td>$\infty$</td>
</tr>
<tr>
<td>$D(2, v)$</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$D(3, v)$</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>-2</td>
<td>1</td>
</tr>
<tr>
<td>$D(4, v)$</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>-2</td>
<td>1</td>
</tr>
</tbody>
</table>

![Graph with nodes A, B, C, D, E and edges with weights 3, 2, 2, 5, -3, 1, 4, -1]
Bellman-ford algorithm: simple version

For each node $v \in V$
\[ d[v] = \infty \]
\[ d[s] = 0 \]
for $i = 1$ to $n - 1$
  for each edge $(u, v) \in E$
    If $d[v] > d[u] + w(u, v)$
      \[ d[v] \leftarrow d[u] + w(u, v) \]
      \[ \text{parent}[v] = u \]

- Re-use same $d[v]$
- Runtime: $O(nm)$
- Space Complexity: $O(n)$
- The set of $\{v, \text{parent}[v]\}$ form a shortest path tree
- Allows to recover path $s$ to $v$ backward from $v$
- How to detect negative weight cycle reachable from $s$
  - Run 1 more iteration and see if any $d$ value changes
Bellman-ford algorithm: Proof of correctness

**Theorem.** At the end, $D(n-1, v)$ is the cost of the shortest path from $s$ to $v$ with at most $n-1$ edges for all $v \in V$

**Proof.** Proof by induction.

**Induction hypothesis.** $D(k, v)$ is the cost of the shortest path from $s$ to $v$ with at most $k$ edges for all $v \in V$

**Base Case:** $i=0$

**Induction step:** Assuming $D(i-1, v)$ is the cost of the shortest path from $s$ to $v$ with at most $i-1$ edges for all $v \in V$, prove that $D(i, v)$ is the cost of the shortest path from $s$ to $v$ with at most $i$ edges for all $v \in V$
Detecting negative cycles

- Given a directed graph $G=(V,E)$ with edge-weights $w_e$ (can be negative), determine if $G$ contains a negative cycle.
- We reduce this to a slightly different problem and will use Bellman-Ford algorithm to solve it.
- **Problem.** Given $G$ and source $s$, find if there is negative cycle on a path from $s$ to $v$ for any node $v$. 
Negative Cycles

**Claim 1.** If there is a negative cycle on a \( s \rightarrow t \) path, then \( D(k, v) \rightarrow -\infty \) as \( k \rightarrow \infty \) for some \( v \in V \)

Example: \( D(t, 3) = -1, D(t, 6) = -4, D(t, 9) = -7 \)
Negative Cycles

- **Claim 2.** If the graph does not have any negative cycle, then $D(n, v) = D(n-1, v)$ for all $v \in V$
  - **Proof.** Any cycle is non-negative, so we can assume that any shortest path from $s$ to $v$ has no cycle and thus it is of length at most $n-1$

- **Claim 3.** If $D(n, v) = D(n-1, v)$ for all $v \in V$, then the graph has no negative cycles
  - **Proof.** We can show that $D(k, v)$ is finite when $k$ goes to infinity for all $v \in V$
    - By claim 1, there are no negative cycles in graph

- A graph has no negative cycles iff $D(n, v) = D(n-1, v)$ for all $v \in V$
  - →There is an $O(mn)$ algorithm for checking
Algorithm for detecting Negative Cycles

- **Lemma.** If $D(n, v) < D(n-1, v)$ for some $v$, then any shortest path from $s$ to $v$ contains a negative cycle.

- **Proof.** by contradiction.
  - Suppose $G$ does not contain a negative cycle.
  - Since $D(n, v) < D(n-1, v)$, the shortest path from $s$ to $v$ has exactly $n$ edges.
    - Otherwise, $D(n, v) = D(n-1, v)$ (according to the algorithm).
    - By pigeonhole principle, a path of length $n$ must have a repeated vertex, and thus a cycle $c$.
      - We claim that $C$ must be a negative cycle.
      - If $C$ has non-negative weight, removing it would give us a shortest path with less than $n$ edges → contradiction: the path contained exactly $n$ edges.

- **there is a negative cycle. How do we find it?**
Algorithm for detecting Negative Cycles

- So, to detect a negative cycle reachable from s:
  - We run one more iteration, and check if any \( d \) value changes.
  - By tracing out the parents using the stored information, we can find \( P \) and thus the cycle \( C \).
  
This gives an \( O(mn) \) time algorithm to find a negative cycle, using \( \theta(n^2) \) space.
All pairs shortest path problem

- **Input.**
  - A directed graph $G = (V, E)$ with a weight on each edge
  - The edge weight could be negative, but there is not negative-weight cycle
- **Output:** The shortest path distance from $u$ to $v$ for all pairs of $u,v \in V$.

- **Brute-force solution.**
  - Apply Bellman-Ford on each node $u \in V$
  - **Runtime** = (n. mn) = $O(n^2m)$
  - **Floyd-Warshall** algorithm: $O(n^3)$
All pairs shortest path problem: First solution

- Subproblem is a path to the predecessor node. To find the optimal solution, we try all possible predecessor nodes $x$

$$D_i(u, v) = \min \begin{cases} D_{i-1}(u, v) \\ \min_{x \in V} \{D_{i-1}(u, x) + w(x, v)\} \end{cases}$$

- Shortest path uses at most $i-1$ edges
- Shortest path uses exactly $i$ edges

$$D_i(u, v) = \min_{x \in V} \{D_{i-1}(u, x) + w(x, v)\}$$

$$D_0(u, u) = 0$$

$$D_0(u, v) = \infty \text{ where } u \neq v$$

$$D_0(u, v) = w(u, v) \text{ where } (u, v) \in E$$

Runtime: $O(n^4)$
All pairs shortest path problem

- $V = \{1, 2, \ldots, n\}$
- **Subproblems are paths in which all interior nodes are in $\{1..k-1\}$**
  - We restrict paths to $u$
  - To find the optimal solution, try all ways to use node $k$ as an interior node
- $D_k[i, j] = \text{weight of shortest } ij \text{ path using only intermediate vertices in } \{1 \ldots k\}$
  - Goal: finding $D_n[i, j]$
- Let $P$ be a min-weight $i,j$-path in which all interior nodes are in $\{1, \ldots, k\}$
- There are two cases
  - Case 1: $k$ is not used in $P$
    - Interior nodes are all in $\{1, \ldots, k-1\}$
  - Case 2: $k$ is used in $P$
    - Interior nodes on paths $i$ to $k$ and $k$ to $j$ are all in $\{1, \ldots, k-1\}$
All pairs shortest path problem

- $D_{k}[i, k] =$ weight of shortest $ij$ path using only intermediate vertices in $\{1...k\}$
  - Goal. finding $D_{n}[i, j]$

- **Base cases:**
  - $D_{0}[i, j] :$ shortest path length from $i$ to $j$ without using intermediate vertices
  - $D_{0}[i, j] = 0$ if $u=v$
  - $D_{0}[i, j] = w(u,v)$ if $(u,v) \in E$
  - $D_{0}[i, j] = \infty$ otherwise

- $D_{k}[i, j] =$ min
  - $D_{k-1}[i, k] + D_{k-1}[k, j]$ use vertex $k$
  - $D_{k-1}[u, v]$ don’t use vertex $k$

- **Correctness:** this considers all possibilities for $k_i$. Then induction on $i$. 
All pairs shortest path problem

Initialize $D_0[i, j]$ as above
for $k$ from 0 to $n-1$ do
    for $i$ from 1 to $n$ do
        for $j$ from 1 to $n$ do
            $D_k[i, j] := \min\{D_{k-1}[i, j], D_{k-1}[i, k] + D_{k-1}[k, j]\}$

- Runtime: $O(n^3)$
- Space: $O(n^3)$
  - Need to store two $n$-by-$n$ arrays, and the original graph.
  - As with Bellman-Ford, we don’t really need to store all $n$ of the $D_k$
All pairs shortest path problem

Initialize $D_0[i, j]$ as above
for $k$ from 0 to $n-1$ do
  for $i$ from 1 to $n$ do
    for $j$ from 1 to $n$ do
      $D[i, j] := \min\{D[i, j], D[i, k] + D[k, j]\}$

- Runtime: $O(n^3)$
- Space: $O(n^2)$
All pairs shortest path problem

- What if we want the actual path?
  - Along with $D[u, v]$, compute $\text{Next}[u, v] = \text{the first vertex after } u \text{ on a shortest } u \text{ to } v \text{ path}$.
  - Exercise. Check how this works.