Analysis of Algorithms, I
CSOR W4231

Eleni Drinea
Computer Science Department
Columbia University

Shortest paths in weighted graphs (Bellman-Ford, Floyd-Warshall)
Outline

1. Shortest paths in graphs with non-negative edge weights (Dijkstra’s algorithm)
   - Implementations
   - Graphs with negative edge weights: why Dijkstra fails

2. Single-source shortest paths (negative edges): Bellman-Ford
   - A DP solution
   - An alternative formulation of Bellman-Ford

3. All-pairs shortest paths (negative edges): Floyd-Warshall
Shortest paths in graphs with non-negative edge weights (Dijkstra’s algorithm)
  - Implementations
  - Graphs with negative edge weights: why Dijkstra fails

Single-source shortest paths (negative edges): Bellman-Ford
  - A DP solution
  - An alternative formulation of Bellman-Ford

All-pairs shortest paths (negative edges): Floyd-Warshall
Graphs with non-negative weights

Input

- a weighted, directed graph $G = (V, E, w)$; function $w : E \rightarrow R^+$ assigns non-negative real-valued weights to edges;
- a source (origin) vertex $s \in V$.

Output: for every vertex $v \in V$

1. the length of a shortest $s$-$v$ path;
2. a shortest $s$-$v$ path.
Dijkstra’s algorithm (Input: $G = (V, E, w), s \in V$)

**Output:** arrays $dist$, $prev$ with $n$ entries such that

1. $dist(v)$ stores the length of the shortest $s$-$v$ path
2. $prev(v)$ stores the node before $v$ in the shortest $s$-$v$ path

At all times, maintain a set $S$ of nodes for which the distance from $s$ has been determined.

- Initially, $dist(s) = 0$, $S = \{s\}$.
- Each time, add to $S$ the node $v \in V - S$ that
  1. has an edge from some node in $S$;
  2. minimizes the following quantity among all nodes $v \in V - S$

\[
d(v) = \min_{u \in S : (u,v) \in E} \{dist(u) + w(u,v)\}
\]

- Set $prev(v) = u$. 

Implementation

Dijkstra-v1(\(G = (V, E, w), s \in V\))

Initialize(\(G, s\))

\(S = \{s\}\)

while \(S \neq V\) do

Select a node \(v \in V - S\) with at least one edge from \(S\) so that

\[
    d(v) = \min_{u \in S, (u,v) \in E} \{\text{dist}[u] + w(u,v)\}
\]

\(S = S \cup \{v\}\)

\(\text{dist}[v] = d(v)\)

\(\text{prev}[v] = u\)

end while

Initialize(\(G, s\))

for \(v \in V\) do

\(\text{dist}[v] = \infty\)

\(\text{prev}[v] = \text{NIL}\)

end for

\(\text{dist}[s] = 0\)
Improved implementation (I)

Idea: Keep a conservative overestimate of the true length of the shortest $s$-$v$ path in $\text{dist}[v]$ as follows: when $u$ is added to $S$, update $\text{dist}[v]$ for all $v$ with $(u,v) \in E$.

Dijkstra-v2($G = (V, E, w), s \in V$)

Initialize($G, s$)

$S = \emptyset$

while $S \neq V$ do

Pick $u$ so that $\text{dist}[u]$ is minimum among all nodes in $V - S$

$S = S \cup \{u\}$

for $(u, v) \in E$ do

Update($u, v$)

end for

end while

Update($u, v$)

if $\text{dist}[v] > \text{dist}[u] + w(u, v)$ then

$\text{dist}[v] = \text{dist}[u] + w(u, v)$

prev[v] = u

end if
Improved implementation (II): binary min-heap

Idea: Use a priority queue implemented as a binary min-heap: store vertex \( u \) with key \( \text{dist}[u] \). Required operations: \text{Insert}, \text{ExtractMin}; \text{DecreaseKey} for \text{Update}; each takes \( O(\log n) \) time.

Dijkstra-v3(\( G = (V, E, w), s \in V \))

\begin{align*}
\text{Initialize}(G, s) \\
Q = \{V; \text{dist}\} \\
S = \emptyset \\
\text{while } Q \neq \emptyset \text{ do} \\
\quad u = \text{ExtractMin}(Q) \\
\quad S = S \cup \{u\} \\
\quad \text{for } (u, v) \in E \text{ do} \\
\quad\quad \text{Update}(u, v) \\
\quad \text{end for} \\
\text{end while}
\end{align*}

Running time: \( O(n \log n + m \log n) = O(m \log n) \)

When is Dijkstra-v3() better than Dijkstra-v2()?
Example graph with **negative** edge weights
Dijkstra’s output and correct output for example graph

Dijkstra’s output

Correct shortest paths

Dijkstra’s algorithm will first include $a$ to $S$ and then $c$, thus missing the shorter path from $s$ to $b$ to $c$. 
Intuitively, a path may start on long edges but then compensate along the way with short edges.

Formally, in the proof of correctness of the algorithm, the last statement about $P$ does not hold anymore: even if the length of path $P_v$ is smaller than the length of the subpath $s \to x \to y$, negative edges on the subpath $y \to v$ may now result in $P$ being shorter than $P_v$. 
Bigger problems in graphs with negative edges?

$\text{dist}(a) =$?
Bigger problems in graphs with negative edges?

1. $\text{dist}(v)$ goes to $-\infty$ for every $v$ on the cycle $(a, b, c, a)$
2. **no** solution to shortest paths when negative cycles

$\Rightarrow$ need to detect negative cycles
Today

1. Shortest paths in graphs with non-negative edge weights (Dijkstra’s algorithm)
   - Implementations
   - Graphs with negative edge weights: why Dijkstra fails

2. Single-source shortest paths (negative edges): Bellman-Ford
   - A DP solution
   - An alternative formulation of Bellman-Ford

3. All-pairs shortest paths (negative edges): Floyd-Warshall
Input: weighted directed graph $G = (V, E, w)$ with $w : E \rightarrow R$; a source (origin) vertex $s \in V$.

Output:

1. If $G$ has a negative cycle reachable from $s$, answer “negative cycle in $G$”.
2. Else, compute for every $v \in V$
   2.1 the length of a shortest $s$-$v$ path;
   2.2 a shortest $s$-$v$ path.
Properties of shortest paths

Suppose the problem has a solution for an input graph.

- *Can there be negative cycles in the graph?*
- *Can there be positive cycles in the graph?*
- *Can the shortest paths contain positive cycles?*
- *Consider a shortest s-t path; are its subpaths shortest? In other words, does the problem exhibit optimal substructure?*
**Key observation:** if there are no negative cycles, a path cannot become shorter by traversing a cycle.

**Fact 1.**

*If $G$ has no negative cycles, then there is a shortest $s$-$v$ path that is simple, thus has at most $n - 1$ edges.*

**Fact 2.**

*- The shortest paths problem exhibits optimal substructure.*

Facts 1 and 2 suggest a DP solution.
Subproblems

Let

\[ OPT(i, v) = \text{cost of a shortest } s-v \text{ path with at most } i \text{ edges} \]

Consider a shortest \( s-v \) path using at most \( i \) edges.

- If the path uses at most \( i - 1 \) edges, then
  \[
  OPT(i, v) = OPT(i - 1, v).
  \]

- If the path uses \( i \) edges, then
  \[
  OPT(i, v) = \min_{x : (x, v) \in E} \{ OPT(i - 1, x) + w(x, v) \}.
  \]
Recurrence

Let

\[ OPT(i, v) = \text{cost of a shortest } s-v \text{ path using at most } i \text{ edges} \]

Then

\[
OPT(i, v) = \begin{cases} 
0, & \text{if } i = 0, v = s \\
\infty, & \text{if } i = 0, v \neq s \\
\min \left\{ OPT(i - 1, v) \right. & \\
\left. \min_{x : (x, v) \in E} \{ OPT(i - 1, x) + w(x, v) \} \right\}, & \text{if } i > 0
\end{cases}
\]
$n \times n$ dynamic programming table $M$ such that $M[i, v] = OPT(i, v)$.

Bellman-Ford($G = (V, E, w), s \in V$)

for $v \in V$ do
  $M[0, v] = \infty$
end for

$M[0, s] = 0$

for $i = 1, \ldots, n - 1$ do
  for $v \in V$ (in any order) do
    $M[i, v] = \min \left\{ M[i - 1, v], \min_{x:(x,v)\in E} \left\{ M[i - 1, x] + w(x, v) \right\} \right\}$
  end for
end for
Running time & Space

- **Running time**: $O(nm)$

- **Space**: $\Theta(n^2)$ — can be improved (*coming up*)

To reconstruct actual shortest paths, also keep array $prev$ of size $n$ such that

$$prev[v] = \text{predecessor of } v \text{ in current shortest } s-v \text{ path.}$$
Compute shortest $s-v$ paths in the graph below, for all $v \in V$. 

![Graph Image]
Only need two rows of $M$ at all times.

Actually, only need one (see Remark 1)! Thus drop the index $i$ from $M[i, v]$ and only use it as a counter for $\#$repetitions.

$$M[v] = \min \left\{ M[v], \min_{x: (x, v) \in E} \{ M[x] + w(x, v) \} \right\}$$

**Remark 1.**

*Throughout the algorithm, $M[v]$ is the length of some $s-v$ path. After $i$ repetitions, $M[v]$ is no larger than the length of the current shortest $s-v$ path with at most $i$ edges.*

**Early termination condition:** if at some iteration $i$ no value in $M$ changed, then stop (*why?*)
An alternative way to view Bellman-Ford

Let $P = (s = v_0, v_1, v_2, \ldots, v_k = v)$ be a shortest $s$-$v$ path.

Then $P$ can contain at most $n - 1$ edges.

How can we correctly compute $\text{dist}(v)$ on this path?
Key observations about subroutine $\text{Update}(u, v)$

Recall subroutine $\text{Update}$ from Dijkstra’s algorithm:

$$\text{Update}(u, v) : \text{dist}(v) = \min\{\text{dist}(v), \text{dist}(u) + w(u, v)\}$$

**Fact 3.**

*Suppose $u$ is the last node before $v$ on the shortest $s$-$v$ path, and suppose $\text{dist}(u)$ has been correctly set. The call $\text{Update}(u, v)$ returns the correct value for $\text{dist}(v)$.***

**Fact 4.**

*No matter how many times $\text{Update}(u, v)$ is performed, it will never make $\text{dist}(v)$ too small. That is, $\text{Update}$ is a safe operation: performing few extra updates can’t hurt.*
Suppose we update the edges on the shortest path $P$ in the order they appear on the path (though not necessarily consecutively). Hence we update

$$(s, v_1), (v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v).$$

This sequence of updates correctly computes $dist(v_1), dist(v_2), \ldots, dist(v)$ (by induction and Fact 3).

How can we guarantee that this specific sequence of updates occurs?
Consider the shortest $s$-$b$ path, which uses edges $(s, a), (a, b)$.

*How can we guarantee that our algorithm will update these two edges in this order?* (More updates in between are allowed.)
Update all $m$ edges in the graph, $n - 1$ times in a row!

- By Fact 4, it is ok to update an edge several times in between.
- All we need is to update the edges on the path in this particular order. This is guaranteed if we update all edges $n - 1$ times in a row.
We will use Initialize and Update from Dijkstra’s algorithm.

Initialize($G, s$)
   \[\text{for } v \in V \text{ do}\]
   \[\text{dist}[v] = \infty\]
   \[\text{prev}[v] = NIL\]
   \[\text{end for}\]
   \[\text{dist}[s] = 0\]

Update($u, v$)
   \[\text{if } \text{dist}[v] > \text{dist}[u] + w(u, v) \text{ then}\]
   \[\text{dist}[v] = \text{dist}[u] + w(u, v)\]
   \[\text{prev}[v] = u\]
   \[\text{end if}\]
Bellman-Ford

\[
\text{Bellman-Ford}(G = (V, E, w), s)
\]

\[
\text{Initialize}(G, s)
\]

\[
\text{for } i = 1, \ldots, n - 1 \text{ do}
\]

\[
\text{for } (u, v) \in E \text{ do}
\]

\[
\text{Update}(u, v)
\]

\[
\text{end for}
\]

\[
\text{end for}
\]

Running time? Space?
Detecting negative cycles
Detecting negative cycles

1. \( \text{dist}(v) \) goes to \(-\infty\) for every \( v \) on the cycle.
2. Any shortest \( s-v \) path can have at most \( n - 1 \) edges.
3. Update all edges \( n \) times (instead of \( n - 1 \)): if \( \text{dist}(v) \) changes for any \( v \in V \), then there is a negative cycle.
1. Shortest paths in graphs with non-negative edge weights (Dijkstra’s algorithm)
   - Implementations
   - Graphs with negative edge weights: why Dijkstra fails

2. Single-source shortest paths (negative edges): Bellman-Ford
   - A DP solution
   - An alternative formulation of Bellman-Ford

3. All-pairs shortest paths (negative edges): Floyd-Warshall
All pairs shortest-paths

- **Input:** a directed, weighted graph $G = (V, E, w)$ with real edge weights
- **Output:** an $n \times n$ matrix $D$ such that

$$D[i, j] = \text{length of shortest path from } i \text{ to } j$$
Solving all pairs shortest-paths

1. Straightforward solution: run Bellman–Ford once for every vertex \(O(n^2m)\) time.
2. Improved solution: Floyd-Warshall’s dynamic programming algorithm \(O(n^3)\) time.
Consider a shortest $s$-$t$ path $P$.

This path uses some intermediate vertices: that is, if $P = (s, v_1, v_2, \ldots, v_k, t)$, then $v_1, \ldots, v_k$ are intermediate vertices.

For simplicity, relabel the vertices in $V$ as $\{1, 2, 3, \ldots, n\}$ and consider a shortest $i$-$j$ path where intermediate vertices may only be from $\{1, 2, \ldots, k\}$.

**Goal:** compute the length of a shortest $i$-$j$ path for every pair of vertices $(i, j)$, using $\{1, 2, \ldots, n\}$ as intermediate vertices.
Rename \{s, a, b, c\} as \{1, 2, 3, 4\}
Examples of shortest paths

Shortest \((1, 2)\)-path using {} or \{1\} is \(P\).
Shortest \((1, 2)\)-path using \{1,2,3,4\} is \(P\).

Shortest \((1, 3)\)-path using {} or \{1\} is \(P'\).
Shortest \((1, 3)\)-path using \{1,2\} or \{1,2,3\} is \(P_1\).
Shortest \((1, 3)\)-path using \{1,2,3,4\} is \(P_1\).
Consider a shortest $i$-$j$ path $P$ where intermediate nodes may only be from the set of nodes $\{1, 2, \ldots, k\}$.

**Fact:** any subpath of $P$ must be shortest itself.
A useful observation

Focus on the last node $k$ from the set $\{1, 2, \ldots, k\}$. Either

1. $P$ completely avoids $k$: then a shortest $i$-$j$ path with intermediate nodes from $\{1, \ldots, k\}$ is the same as a shortest $i$-$j$ path with intermediate nodes from $\{1, \ldots, k - 1\}$.

2. Or, $k$ is an intermediate node of $P$.

Decompose $P$ into an $i$-$k$ subpath $P_1$ and a $k$-$j$ subpath $P_2$.

i. $P_1, P_2$ are shortest subpaths themselves.
ii. All intermediate nodes of $P_1, P_2$ are from $\{1, \ldots, k - 1\}$. 
Subproblems

Let

\[ OPT_k(i, j) = \text{cost of shortest } i - j \text{ path } P \text{ using} \]
\[ \{1, \ldots, k\} \text{ as intermediate vertices} \]

1. Either \( k \) does not appear in \( P \), hence

\[ OPT_k(i, j) = OPT_{k-1}(i, j) \]

2. Or, \( k \) appears in \( P \), hence

\[ OPT_k(i, j) = OPT_{k-1}(i, k) + OPT_{k-1}(k, j) \]
Hence

\[
OPT_k(i, j) = \begin{cases} 
  w(i, j), & \text{if } k = 0 \\
  \min \begin{cases} 
    OPT_{k-1}(i, j) \\
    OPT_{k-1}(i, k) + OPT_{k-1}(k, j)
  \end{cases}, & \text{if } k \geq 1
\end{cases}
\]

We want \( OPT_n(i, j) \).

Time/space requirements?
Floyd-Warshall on example graph

Let $D_k[i, j] = OPT_k(i, j)$.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$D_0 =

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$D_1 =

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>5</th>
<th>∞</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$D_2 =

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

$D_3 =

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>4</th>
<th>3</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>∞</td>
<td>0</td>
<td>-1</td>
<td>-2</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
A single \( n \times n \) dynamic programming table \( D \), initialized to \( w(i, j) \) (the adjacency matrix of \( G \)).

Let \( \{1, \ldots, k\} \) be the set of intermediate nodes that may be used for the shortest \( i-j \) path.

After the \( k \)-th iteration, \( D[i, j] \) contains the length of some \( i-j \) path that is no larger than the length of the shortest \( i-j \) path using \( \{1, \ldots, k\} \) as intermediate nodes.
The Floyd-Warshall algorithm

Floyd-Warshall\((G = (V, E, w))\)

\[
\text{for } k = 1 \text{ to } n \text{ do}
\]
\[
\text{for } i = 1 \text{ to } n \text{ do}
\]
\[
\text{for } j = 1 \text{ to } n \text{ do}
\]
\[
D[i, j] = \min(D[i, j], D[i, k] + D[k, j])
\]
\[
\text{end for}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]

- Running time: \(O(n^3)\)
- Space: \(\Theta(n^2)\)