Lecture: Analysis of Algorithms (CS583 - 002)\textsuperscript{1}

Amarda Shehu

Fall 2017

\textsuperscript{1}Some material adapted from Kevin Wayne’s Algorithm Class © Princeton
1 Single-source Shortest Path on Graph with Negative Weights
   - Bellman-Ford’s Algorithm

2 All-pairs Shortest Paths
   - Dynamic Programming Approach
   - Floyd-Warshall
What is the Big Deal with Negative Weights?

**Dijkstra**: can fail if there are negative weights.

Dijkstra selects vertex 3 immediately after 0. But shortest path from 0 to 3 is \(0 \rightarrow 1 \rightarrow 2 \rightarrow 3\).

**Re-weighting**: Adding a constant to every edge weight can also fail.

Adding 9 to each edge changes the shortest path because it adds 9 to each edge; wrong thing to do for paths with many edges.
What is the Big Deal with Negative Cycles?

**Negative weight cycle:** sum of weights in cycle is negative.

![Diagram of a negative weight cycle](image)

**Observation:** If some path from $s$ to $t$ contains a negative weight cycle, then there is no shortest $s \rightsquigarrow t$ path. Shortest path can be made arbitrarily shorter by spinning around cycle.
Bellman-Ford solves the single-source shortest path problem in the general case when the graph may have negative weights.

Gives up trying to find shortest paths when negative cycles are detected.

Unlike greedy Dijkstra, which depends on the structural assumptions derived from nonnegative weights, an iterative approach is employed that uses Dynamic Programming and so extends to the general case.

Instead of greedily selecting the minimum-key vertex from the priority queue (not yet processed) to relax, the algorithm relaxes all edges for a certain number of iterations.

The iterations allow the shortest-path estimates to accurately propagate throughout the graph.
A greedy approach would not work, given negative weights
Bellman-Ford pursues a Dynamics Programming approach
Key realization is that in the absence of negative cycles, any shortest path cannot have more than $|V| - 1$ edges in it
Let’s introduce a variable that keeps track of the shortest path from a source vertex $s$ to any vertex $v$
$L(i, v) =$ length of shortest $s \rightsquigarrow v$ path that has $\leq i$ edges
How does $L(i, v)$ depend on $i$?
Definition: \( L(i, v) = \) length of shortest \( s \leadsto v \) path \( P \) that has \( \leq i \) edges

- Case 1: \( P \) uses at most \( i - 1 \) edges
  - Then \( L(i, v) = L(i - 1, v) \)

- Case 2: \( P \) uses exactly \( i \) edges
  - Let \((u, v)\) be last edge in \( P \). Then, \textit{given that the last edge is} \((u, v)\), \( L \) has to select \( s \leadsto u \) shortest path using at most \( i - 1 \) edges followed by \((u, v)\)

\[
L(i, v) = \begin{cases} 
0 & \text{if } i = 0 \\
\min_{e=(u,v)} \{ L(i - 1, v), L(i - 1, u) + w(e) \} & \text{otherwise}
\end{cases}
\]

Termin.: When \( \not\exists \) negative cycles, \( L(n - 1, v) = \) length of shortest \( s \leadsto v \) path
How are Negative Cycles Detected?

**Lemma** If $L(n, v) = L(n - 1, v)$ for all vertices $v$, then there are no negative weight cycles in the graph.

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

**Proof:**

- Since $L(n, v) < L(n - 1, v)$, we know that $P$ has exactly $n$ edges.
- By the pigeonhole principle, $P$ must contain a directed cycle $C$.
- Deleting $C$ yields $s \rightsquigarrow v$ path with $< n$ edges. So, $w(C) < 0$.

**Note:** Any $(\ast, v)$ edge can be that last edge in the shortest path $s \rightsquigarrow v$. That is why the recursion is over the length of the path.
Pseudocode DP Approach

DP-Approach(G, s, weight)
1: for each node v ∈ V do
2:   \(L[0, v] \leftarrow \infty\)
3:   \(L[0, s] \leftarrow 0\)
4: for i = 1 to n − 1 do
5:   for each node v ∈ V do
6:     \(L[i, v] \leftarrow L[i - 1, v]\)
7:   for each edge (u, v) ∈ E do
8:     \(L[i, v] \leftarrow \min\{L[i - 1, v], L[i - 1, u] + \text{weight}(u, v)\}\)

Analysis: Running Time? Space?
Bellman-Ford(G, s, w)

1: \(d[v] \leftarrow \infty\) for all \(v \in V\)
2: \(d[s] \leftarrow 0\)
3: for \(i \leftarrow 1\) to \(|V| - 1\) do
4: for each edge \((u, v) \in E\) do
5: \(\text{Relax}(u, v, w)\)
6: for each edge \((u, v) \in E\) do
7: if \(d[v] > d[u] + w(u, v)\) then
8: \(\text{Negative cycles detected: return FALSE}\)
9: return TRUE

Efficient Implementation No need to maintain the matrix \(L(i, v)\). Maintain only row/iteration with changes. So, use an array \(d[v]\).
Why Bellman-Ford Implements DP Approach?

- Bellman-Ford performs $|V| - 1$ iterations, because the highest number of edges in any shortest path is $|V| - 1$.
- After each iteration $i$, $d[v]$ is no smaller than $L(i, v)$.
- The relaxation updates $d[v]$ for each vertex $v$.
- Essentially, after each iteration $i$, Bellman-Ford answers the question: if shortest path $s \leadsto v$ contains $i$ edges, what would be its weight?
- After $|V| - 1$ iterations, the shortest paths have been found.
- If after one more iteration, the weights can be made shorter, a negative cycle exists.
Bellman-Ford Trace

Graph $G = (V, E)$

![Graph Diagram]

Initialization

![Initialization Diagram]
Bellman-Ford Trace

Order of edge relaxation

Begin of pass 1
Bellman-Ford Trace

Pass 1 continued

Pass 1 continued
Outline of Today’s Class

- Single-source Shortest Path on Graph with Negative Weights
- All-pairs Shortest Paths

Bellman-Ford’s Algorithm

Bellman-Ford Trace

Pass 1 continued

![Graph with Bellman-Ford algorithm trace](image.png)
Bellman-Ford Trace

Pass 1 continued

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>-3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>∞</td>
<td>-3</td>
<td>8</td>
</tr>
<tr>
<td>∞</td>
<td>1</td>
<td>8</td>
<td>-3</td>
<td>1</td>
</tr>
</tbody>
</table>

Pass 1 continued

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>∞</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>7</td>
<td>6</td>
<td>-3</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>∞</td>
<td>-3</td>
<td>8</td>
</tr>
<tr>
<td>∞</td>
<td>1</td>
<td>8</td>
<td>-3</td>
<td>1</td>
</tr>
</tbody>
</table>
Bellman-Ford Trace

Pass 1 continued

Pass 1 continued
Bellman-Ford Trace

Pass 1 ends

Pass 2 begins
Bellman-Ford Trace

Pass 2 continued

Bellman-Ford’s Algorithm
Bellman-Ford Trace

Pass 2 continued
Bellman-Ford Trace

Pass 2 continued

---

Pass 2 continued

---
Bellman-Ford Trace

Pass 2 continued

Pass 2, 3, 4 end
All-pairs Shortest Paths

- **Problem:** Find shortest paths between all pairs.
- **Assumption:** There are no cycles of 0 or negative weight.
- **Approach:** Dynamic programming, similar to Bellman-Ford, keeps track of intermediate vertices in a path.
All-pairs Shortest Paths: Floyd-Warshall Algorithm

**Definition:** The vertices $v_2, v_3, \ldots, v_{l-1}$ are the intermediate vertices of the path $p = \langle v_1, v_2, \ldots, v_l \rangle$.

- Let $d_{i,j}^{(k)}$ be the weight of shortest path $i \leadsto j$ s.t. all intermediate vertices (if any) are in set $\{1, 2, \ldots, k\}$.

- Let $D^{(k)}$ be $n \times n$ matrix $[d_{i,j}^{(k)}]$.

- What is $d_{i,j}^{(0)}$?

- Claim: $d_{i,j}^n$ is weight of shortest weight path from $i$ to $j$. Our goal is to compute $D^n$. 

---

Amarda Shehu  Lecture: Analysis of Algorithms (CS583 - 002)
Observation 1: A shortest path does not contain the same vertex twice. Why?

Observation 2: For a shortest path from $i$ to $j$ s.t. that any intermediate vertices on the path are chosen from the set $\{1, 2, \ldots, k\}$, there are only two possibilities:

- **$k$ is not a vertex on the path**: Shortest path has weight $d_{i,j}^{(k-1)}$.
- **$k$ is a vertex on the path**: Shortest path has weight $d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}$. 
Consider a shortest path from $i$ to $j$ containing the vertex $k$. It consists of a subpath from $i$ to $k$ and a subpath from $k$ to $j$. Each subpath can only contain intermediate vertices in \{1, \ldots, k−1\}, and must be as short as possible.

Putting it all together:

$$d_{i,j}^{(k)} = \begin{cases} w(v_i, v_j) & \text{if } k = 0 \\ \min\{d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\} & k \geq 1 \end{cases}$$
Floyd-Warshall: The Algorithm

\[
\text{Floyd-Warshall}(G, w, n)
\]

1. \textbf{for} \( i = 1 \) to \( n \) \textbf{do}
   
2. \textbf{for} \( j = 1 \) to \( n \) \textbf{do}
   
3. \quad \( d[i, j] \leftarrow w[i, j] \)
   
4. \quad \text{pred}[i, j] = \text{NIL}
   
5. \textbf{for} \( k = 1 \) to \( n \) \textbf{do}
   
6. \quad \textbf{for} \( i = 1 \) to \( n \) \textbf{do}
   
7. \quad \quad \textbf{for} \( j = 1 \) to \( n \) \textbf{do}
   
8. \quad \quad \quad \textbf{if} \ d[i, k] + d[k, j] < d[i, j] \quad \textbf{then}
   
9. \quad \quad \quad \quad d[i, j] \leftarrow d[i, k] + d[k, j]
   
10. \quad \quad \quad \text{pred}[i, j] = k

- Keep \( d_{i,j}^{(k)} \) values in matrix \( D^{(k)} \)
- Compute \( D^{(0)}, D^{(1)}, \ldots, D^{(n)} \)
- \( D^{(k)} \leftarrow D^{(k-1)} \)
- \text{pred} \ can be used to extract paths (how?)
- Storage: \( \theta(|V|^2) \)
- Time: \( \theta(|V|^3) \)
Floyd-Warshall: Time and Space Complexity

- **Storage:** $\theta(|V|^2)$
- **Time:** $\theta(|V|^3)$
  - Filling each matrix cell requires computing a minimum value by iterating over all $v_k \in V$: $|V|$ iterations are needed to fill each cell
  - Matrix has $|V| \times |V|$ cells
  - Hence, $|V| \times |V| \times |V| = |V|^3$ operations

- Johnson’s algorithm improves time: $O(|V|^2 \cdot lg(|V|) + |V| \cdot |E|)$
  - best on sparse graphs: adjacency list representation
  - uses Bellman-Ford: smart reweighting of negative-weight edges
  - uses Dijkstra: after reweighting, from each vertex
Johnson’s Algorithm

- Add fake source $s$, connected by zero weights to all nodes in graph
- Run Bellman-Ford to get shortest paths from $s$ to any other vertex in $G$.
- Let $h(v)$ be the length of the shortest path from $s$ to $v \in V$
- Reweight all edges of $G$: $w(u, v) \leftarrow w(u, v) + h(u) - h(v)$
- Now run Dijkstra’s from each vertex to find all shortest paths

Figure: Why are edges reweighted this way?