Dynamic Programming: Shortest Paths and DFA to Reg Expressions
18.1
Shortest Paths with Negative Length Edges
18.1.1

Why Dijkstra’s algorithm fails with negative edges
Single-Source Shortest Paths with Negative Edge Lengths

Problem statement

**Single-Source Shortest Path Problems**

**Input**: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.
Single-Source Shortest Path Problems

**Input**: A directed graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.
What are the distances computed by Dijkstra’s algorithm?

The distance as computed by Dijkstra algorithm starting from \( s \):

- **A** \( s = 0, x = 5, y = 1, z = 0 \).
- **B** \( s = 0, x = 1, y = 2, z = 5 \).
- **C** \( s = 0, x = 5, y = 1, z = 2 \).
- **D** IDK.
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail.
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail

![Graph 1](image1)

![Graph 2](image2)
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail.
Dijkstra’s Algorithm and Negative Lengths

With negative length edges, Dijkstra’s algorithm can fail.

False assumption: Dijkstra’s algorithm is based on the assumption that if $s = v_0 \rightarrow v_1 \rightarrow v_2 \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then $dist(s, v_i) \leq dist(s, v_{i+1})$ for $0 \leq i < k$. Holds true only for non-negative edge lengths.
Shortest Paths with Negative Lengths

Lemma 18.1.

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

2. False: $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$ for $1 \leq i < k$. Holds true only for non-negative edge lengths.

Cannot explore nodes in increasing order of distance! We need other strategies.
Lemma 18.1.

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$
2. False: $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$ for $1 \leq i < k$. Holds true only for non-negative edge lengths.

Cannot explore nodes in increasing order of distance! We need other strategies.
Shortest Paths with Negative Lengths

**Lemma 18.1.**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

2. **False:** $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$ for $1 \leq i < k$. Holds true only for non-negative edge lengths.

Cannot explore nodes in increasing order of distance! We need other strategies.
THE END

...  

(for now)
18.1.2

But wait! Things get worse: Negative cycles
Negative Length Cycles

Definition 18.2.
A cycle $C$ is a negative length cycle if the sum of the edge lengths of $C$ is negative.
Negative Length Cycles

**Definition 18.2.**

A cycle $C$ is a negative length cycle if the sum of the edge lengths of $C$ is negative.
Definition 18.2.
A cycle \( C \) is a negative length cycle if the sum of the edge lengths of \( C \) is negative.

What is the shortest path distance between \( s \) and \( t \)?
Reminder: Paths have to be simple...
Shortest Paths and Negative Cycles

Given $G = (V, E)$ with edge lengths and $s, t$. Suppose

1. $G$ has a negative length cycle $C$, and
2. $s$ can reach $C$ and $C$ can reach $t$.

Question: What is the shortest distance from $s$ to $t$?

Possible answers: Define shortest distance to be:

1. undefined, that is $-\infty$, OR
2. the length of a shortest simple path from $s$ to $t$. 
Shortest Paths and Negative Cycles

Given $G = (V, E)$ with edge lengths and $s, t$. Suppose
1. $G$ has a negative length cycle $C$, and
2. $s$ can reach $C$ and $C$ can reach $t$.

Question: What is the shortest distance from $s$ to $t$?

Possible answers: Define shortest distance to be:
1. undefined, that is $-\infty$, OR
2. the length of a shortest simple path from $s$ to $t$. 
Lemma 18.3.

If there is an efficient algorithm to find a shortest simple $s \rightarrow t$ path in a graph with negative edge lengths, then there is an efficient algorithm to find the longest simple $s \rightarrow t$ path in a graph with positive edge lengths.

Finding the $s \rightarrow t$ longest path is difficult. **NP-Hard**!
THE END

... (for now)
18.1.3

Restating problem of Shortest path with negative edges
Alternatively: Finding Shortest Walks

Given a graph $G = (V, E)$:

1. A **path** is a sequence of distinct vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$.

2. A **walk** is a sequence of vertices $v_1, v_2, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $1 \leq i \leq k - 1$. Vertices are allowed to repeat.

Define $\text{dist}(u, v)$ to be the length of a shortest walk from $u$ to $v$.

1. If there is a walk from $u$ to $v$ that contains negative length cycle then $\text{dist}(u, v) = -\infty$

2. Else there is a path with at most $n - 1$ edges whose length is equal to the length of a shortest walk and $\text{dist}(u, v)$ is finite

Helpful to think about walks
Shortest Paths with Negative Edge Lengths

Problems

Algorithmic Problems

**Input:** A directed graph $G = (V, E)$ with edge lengths (could be negative). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

**Questions:**

1. Given nodes $s, t$, either find a negative length cycle $C$ that $s$ can reach or find a shortest path from $s$ to $t$.

2. Given node $s$, either find a negative length cycle $C$ that $s$ can reach or find shortest path distances from $s$ to all reachable nodes.

3. Check if $G$ has a negative length cycle or not.
Shortest Paths with Negative Edge Lengths
In Undirected Graphs

**Note**: With negative lengths, shortest path problems and negative cycle detection in undirected graphs cannot be reduced to directed graphs by bi-directing each undirected edge. Why?

Problem can be solved efficiently in undirected graphs but algorithms are different and significantly more involved than those for directed graphs. One need to compute $T$-joins in the relevant graph. Pretty painful stuff.
THE END
...
(for now)
18.1.4
Applications of shortest path for negative weights on edges
Why negative lengths?

Several Applications

1. Shortest path problems useful in modeling many situations — in some negative lengths are natural
2. Negative length cycle can be used to find arbitrage opportunities in currency trading
3. Important sub-routine in algorithms for more general problem: minimum-cost flow
Negative cycles
Application to Currency Trading

Currency Trading

**Input**: $n$ currencies and for each ordered pair $(a, b)$ the exchange rate for converting one unit of $a$ into one unit of $b$.

**Questions**:

1. Is there an arbitrage opportunity?
2. Given currencies $s, t$ what is the best way to convert $s$ to $t$ (perhaps via other intermediate currencies)?

Concrete example:

1. 1 Chinese Yuan = **0.1116** Euro
2. 1 Euro = **1.3617** US dollar
3. 1 US Dollar = **7.1** Chinese Yuan.

Thus, if exchanging 1 $ → Yuan → Euro → $, we get: **0.1116 * 1.3617 * 7.1 = 1.07896$**.
Reducing Currency Trading to Shortest Paths

**Observation:** If we convert currency $i$ to $j$ via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of $i$ yields $\text{exch}(i, k_1) \times \text{exch}(k_1, k_2) \ldots \times \text{exch}(k_h, j)$ units of $j$.

Create currency trading directed graph $G = (V, E)$:

1. For each currency $i$ there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. edge length $\ell(v_i, v_j) = - \log(\text{exch}(i, j))$ can be negative

**Exercise:** Verify that

1. There is an arbitrage opportunity if and only if $G$ has a negative length cycle.
2. The best way to convert currency $i$ to currency $j$ is via a shortest path in $G$ from $i$ to $j$. If $d$ is the distance from $i$ to $j$ then one unit of $i$ can be converted into $2^d$ units of $j$. 
Reducing Currency Trading to Shortest Paths

**Observation:** If we convert currency $i$ to $j$ via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of $i$ yields $\text{exch}(i, k_1) \times \text{exch}(k_1, k_2) \ldots \times \text{exch}(k_h, j)$ units of $j$.

Create currency trading **directed** graph $G = (V, E)$:

1. **For each currency** $i$ there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. **edge length** $\ell(v_i, v_j) = -\log(\text{exch}(i, j))$ can be negative

**Exercise:** Verify that

1. There is an arbitrage opportunity if and only if $G$ has a negative length cycle.
2. The best way to convert currency $i$ to currency $j$ is via a shortest path in $G$ from $i$ to $j$. If $d$ is the distance from $i$ to $j$ then one unit of $i$ can be converted into $2^d$ units of $j$. 
Reducing Currency Trading to Shortest Paths

Observation: If we convert currency $i$ to $j$ via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of $i$ yields $\text{exch}(i, k_1) \times \text{exch}(k_1, k_2) \times \cdots \times \text{exch}(k_h, j)$ units of $j$.

Create currency trading directed graph $G = (V, E)$:

1. For each currency $i$ there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. edge length $\ell(v_i, v_j) = -\log(\text{exch}(i, j))$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if $G$ has a negative length cycle.
2. The best way to convert currency $i$ to currency $j$ is via a shortest path in $G$ from $i$ to $j$. If $d$ is the distance from $i$ to $j$ then one unit of $i$ can be converted into $2^d$ units of $j$. 
Reducing Currency Trading to Shortest Paths

Observation: If we convert currency $i$ to $j$ via intermediate currencies $k_1, k_2, \ldots, k_h$ then one unit of $i$ yields $\text{exch}(i, k_1) \times \text{exch}(k_1, k_2) \ldots \times \text{exch}(k_h, j)$ units of $j$.

Create currency trading directed graph $G = (V, E)$:

1. For each currency $i$ there is a node $v_i \in V$
2. $E = V \times V$: an edge for each pair of currencies
3. edge length $\ell(v_i, v_j) = -\log(\text{exch}(i, j))$ can be negative

Exercise: Verify that

1. There is an arbitrage opportunity if and only if $G$ has a negative length cycle.
2. The best way to convert currency $i$ to currency $j$ is via a shortest path in $G$ from $i$ to $j$. If $d$ is the distance from $i$ to $j$ then one unit of $i$ can be converted into $2^d$ units of $j$. 
Reducing Currency Trading to Shortest Paths

Math recall - relevant information

1. \( \log(\alpha_1 \times \alpha_2 \times \cdots \times \alpha_k) = \log \alpha_1 + \log \alpha_2 + \cdots + \log \alpha_k. \)
2. \( \log x > 0 \) if and only if \( x > 1 \).
THE END

...

(for now)
18.2
Bellman Ford Algorithm
18.2.1

Shortest path with negative lengths: The challenge
Shortest Paths with Negative Lengths

**Lemma 18.1.**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$
2. False: $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$ for $1 \leq i < k$. Holds true only for non-negative edge lengths.

Cannot explore nodes in increasing order of distance! We need other strategies.
Shortest Paths with Negative Lengths

Lemma 18.1.

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$
2. False: $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$ for $1 \leq i < k$. Holds true only for non-negative edge lengths.

Cannot explore nodes in increasing order of distance! We need other strategies.
Shortest Paths with Negative Lengths

**Lemma 18.1.**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

1. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$
2. False: $\text{dist}(s, v_i) \leq \text{dist}(s, v_k)$ for $1 \leq i < k$. Holds true only for non-negative edge lengths.

Cannot explore nodes in increasing order of distance! We need other strategies.
THE END

... (for now)
18.2.2
Shortest path via number of hops
Shortest Paths and Recursion

1. Compute the shortest path distance from \( s \) to \( t \) recursively?
2. What are the smaller sub-problems?

Lemma 18.2.

Let \( G \) be a directed graph with arbitrary edge lengths. If \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):

\[ s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \text{ is a shortest path from } s \text{ to } v_i \]

Sub-problem idea: paths of fewer hops/edges
Shortest Paths and Recursion

1. Compute the shortest path distance from \( s \) to \( t \) recursively?
2. What are the smaller sub-problems?

**Lemma 18.2.**

Let \( G \) be a directed graph with arbitrary edge lengths. If \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \) is a shortest path from \( s \) to \( v_k \) then for \( 1 \leq i < k \):

4. \( s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i \) is a shortest path from \( s \) to \( v_i \)

Sub-problem idea: paths of fewer hops/edges
Shortest Paths and Recursion

1. Compute the shortest path distance from $s$ to $t$ recursively?
2. What are the smaller sub-problems?

**Lemma 18.2.**

Let $G$ be a directed graph with arbitrary edge lengths. If $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k$ is a shortest path from $s$ to $v_k$ then for $1 \leq i < k$:

4. $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_i$ is a shortest path from $s$ to $v_i$

Sub-problem idea: paths of fewer hops/edges
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source \( s \).
Assume that all nodes can be reached by \( s \) in \( G \)
Assume \( G \) has no negative-length cycle (for now).

\( d(v, k) \): shortest walk length from \( s \) to \( v \) using at most \( k \) edges.

Note: \( \text{dist}(s, v) = d(v, n - 1) \). Recursion for \( d(v, k) \):

\[
d(v, k) = \min \left\{ \min_{u \in V} \left( d(u, k - 1) + \ell(u, v) \right), d(v, k - 1) \right\}
\]

Base case: \( d(s, 0) = 0 \) and \( d(v, 0) = \infty \) for all \( v \neq s \).
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.
Assume that all nodes can be reached by $s$ in $G$
Assume $G$ has no negative-length cycle (for now).

$d(v, k)$: shortest walk length from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = d(v, n - 1)$. Recursion for $d(v, k)$:

$$d(v, k) = \min \left\{ \min_{u \in V} \left( d(u, k - 1) + \ell(u, v) \right), \quad d(v, k - 1) \right\}$$

Base case: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$. 
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.
Assume that all nodes can be reached by $s$ in $G$
Assume $G$ has no negative-length cycle (for now).

$d(v, k)$: shortest walk length from $s$ to $v$ using at most $k$ edges.
Note: $\text{dist}(s, v) = d(v, n - 1)$. Recursion for $d(v, k)$:

$$d(v, k) = \min \left\{ \min_{u \in V} (d(u, k - 1) + \ell(u, v)), d(v, k - 1) \right\}$$

Base case: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$. 
Hop-based Recursion: Bellman-Ford Algorithm

Single-source problem: fix source $s$.
Assume that all nodes can be reached by $s$ in $G$
Assume $G$ has no negative-length cycle (for now).

$d(v, k)$: shortest walk length from $s$ to $v$ using at most $k$ edges.

Note: $\text{dist}(s, v) = d(v, n - 1)$. Recursion for $d(v, k)$:

$$d(v, k) = \min \left\{ \min_{u \in V} (d(u, k - 1) + \ell(u, v)), \ d(v, k - 1) \right\}$$

Base case: $d(s, 0) = 0$ and $d(v, 0) = \infty$ for all $v \neq s$. 
Example

```
  d  e  f
 / \ / \ / \
-8  2  \\
/ \ / \
1  0  5
/ / \
-3 b -3
/ / \\n8  1  8
/ / \\
6  4  3
/ / \\
|s| a | b | c | d | e | f |
```

Example

<table>
<thead>
<tr>
<th>round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

![Graph with nodes and edges labeled with weights]
Example

<table>
<thead>
<tr>
<th>round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
Example

A network diagram with nodes labeled $a$, $b$, $c$, $d$, $e$, $f$, and $s$. The edges are labeled with weights:

- $d$ to $a$: -3
- $d$ to $b$: 1
- $d$ to $c$: 5
- $b$ to $2$: 8
- $b$ to $c$: -1
- $2$ to $b$: 5
- $2$ to $c$: -3
- $a$ to $6$: 4
- $a$ to $c$: 3
- $6$ to $4$: 6
- $6$ to $3$: 3
- $s$ to $0$: 6
- $0$ to $s$: 3
- $e$ to $f$: 2
- $f$ to $e$: 0
- $∞$ to $d$, $e$, $f$
- $-1$ to $a$, $b$, $c$, $d$, $e$, $f$

A table with the following format:

<table>
<thead>
<tr>
<th>Round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>∞</td>
<td>9</td>
</tr>
</tbody>
</table>

Table continues with additional rows.
Example

**Network Diagram:**

- Nodes: a, b, c, d, e, f
- Edges with weights:
  - a to b: 4
  - a to c: 6
  - b to c: -1
  - b to f: 5
  - c to 0: 3
  - d to 2: -3
  - d to a: 1
  - e to 11: 2
  - f to 7: -8

**Table:**

<table>
<thead>
<tr>
<th>round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>7</td>
</tr>
</tbody>
</table>
Example

![Graph Diagram]

<table>
<thead>
<tr>
<th>round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
**Example**

![Graph](image)

<table>
<thead>
<tr>
<th>round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
Example

```
Example

<table>
<thead>
<tr>
<th>round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>-2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
```
### Example

![Graph](image)

<table>
<thead>
<tr>
<th>round</th>
<th>s</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>4</td>
<td>3</td>
<td>∞</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>∞</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>-1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>-2</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>9</td>
<td>7</td>
</tr>
</tbody>
</table>
THE END

... 

(for now)
18.2.3
The Bellman-Ford Algorithm
Bellman-Ford Algorithm

\[
\text{for each } u \in V \text{ do} \\
\quad d(u, 0) \leftarrow \infty \\
\quad d(s, 0) \leftarrow 0 \\
\text{for } k = 1 \text{ to } n - 1 \text{ do} \\
\quad \text{for each } v \in V \text{ do} \\
\quad \quad d(v, k) \leftarrow d(v, k - 1) \\
\quad \text{for each edge } (u, v) \in \text{in}(v) \text{ do} \\
\quad \quad d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \\
\text{for each } v \in V \text{ do} \\
\quad \text{dist}(s, v) \leftarrow d(v, n - 1)
\]

Running time: \(O(mn)\)  
Space: \(O(m + n^2)\)  
Space can be reduced to \(O(m + n)\).
Bellman-Ford Algorithm

\[
\begin{align*}
\text{for each } & \ u \in V \ \text{do} \\
& d(u, 0) \leftarrow \infty \\
& d(s, 0) \leftarrow 0 \\
\text{for } & \ k = 1 \ \text{to} \ n - 1 \ \text{do} \\
& \text{for each } \ v \in V \ \text{do} \\
& \quad d(v, k) \leftarrow d(v, k - 1) \\
& \quad \text{for each edge } \ (u, v) \in \text{in}(v) \ \text{do} \\
& \quad \quad d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \\
& \text{for each } \ v \in V \ \text{do} \\
& \quad \text{dist}(s, v) \leftarrow d(v, n - 1)
\end{align*}
\]

Running time: $O(mn)$  
Space: $O(m + n^2)$  
Space can be reduced to $O(m + n)$. 

Bellman-Ford Algorithm

for each \( u \in V \) do
    \( d(u, 0) \leftarrow \infty \)
    \( d(s, 0) \leftarrow 0 \)

for \( k = 1 \) to \( n - 1 \) do
    for each \( v \in V \) do
        \( d(v, k) \leftarrow d(v, k - 1) \)
        for each edge \((u, v) \in \text{in}(v)\) do
            \( d(v, k) = \min \{ d(v, k), d(u, k - 1) + \ell(u, v) \} \)

for each \( v \in V \) do
    \( \text{dist}(s, v) \leftarrow d(v, n - 1) \)

Running time: \( O(mn) \)  Space: \( O(m + n^2) \)
Space can be reduced to \( O(m + n) \).
Bellman-Ford Algorithm

for each \( u \in V \) do
  \( d(u, 0) \leftarrow \infty \)
  \( d(s, 0) \leftarrow 0 \)

for \( k = 1 \) to \( n - 1 \) do
  for each \( v \in V \) do
    \( d(v, k) \leftarrow d(v, k - 1) \)
  for each edge \( (u, v) \in in(v) \) do
    \( d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \)

for each \( v \in V \) do
  \( \text{dist}(s, v) \leftarrow d(v, n - 1) \)

Running time: \( O(mn) \) Space: \( O(m + n^2) \)

Space can be reduced to \( O(m + n) \).
Bellman-Ford Algorithm

```
for each \( u \in V \) do
    \( d(u, 0) \leftarrow \infty \)
    \( d(s, 0) \leftarrow 0 \)

for \( k = 1 \) to \( n - 1 \) do
    for each \( v \in V \) do
        \( d(v, k) \leftarrow d(v, k - 1) \)
        for each edge \( (u, v) \in i n(v) \) do
            \( d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \)

for each \( v \in V \) do
    \( \text{dist}(s, v) \leftarrow d(v, n - 1) \)
```

Running time: \( O(mn) \) Space: \( O(m + n^2) \)

Space can be reduced to \( O(m + n) \).
**Bellman-Ford Algorithm**

```plaintext
for each \( u \in V \) do
    \( d(u, 0) \leftarrow \infty \)
    \( d(s, 0) \leftarrow 0 \)

for \( k = 1 \) to \( n - 1 \) do
    for each \( v \in V \) do
        \( d(v, k) \leftarrow d(v, k - 1) \)
        for each edge \( (u, v) \in in(v) \) do
            \( d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \)

for each \( v \in V \) do
    dist(s, v) \leftarrow d(v, n - 1)
```

Running time: \( O(mn) \)  
Space: \( O(m + n^2) \)  
Space can be reduced to \( O(m + n) \).
Bellman-Ford Algorithm: Cleaner version

```python
for each $u \in V$ do
    \(d(u) \leftarrow \infty\)
\(d(s) \leftarrow 0\)

for $k = 1$ to $n - 1$ do
    for each $v \in V$ do
        for each edge $(u, v) \in \text{in}(v)$ do
            \(d(v) = \min\{d(v), d(u) + \ell(u, v)\}\)

for each $v \in V$ do
    \(\text{dist}(s, v) \leftarrow d(v)\)
```

Running time: \(O(mn)\)  Space: \(O(m + n)\)

Exercise: Argue that this achieves same results as algorithm on previous slide.
THE END

... (for now)
18.2.3.1

Correctness of the Bellman-Ford Algorithm
Bellman-Ford Algorithm: Modified for analysis

for each \( u \in V \) do
  \( d(u, 0) \leftarrow \infty \)
  \( d(s, 0) \leftarrow 0 \)

for \( k = 1 \) to \( n \) do
  for each \( v \in V \) do
    \( d(v, k) \leftarrow d(v, k - 1) \)
  for each edge \( (u, v) \in \text{in}(v) \) do
    \( d(v, k) = \min\{d(v, k), d(u, k - 1) + \ell(u, v)\} \)

for each \( v \in V \) do
  \( \text{dist}(s, v) \leftarrow d(v, n - 1) \)
Walks computed correctly

**Lemma 18.3.**

For each $v$, $d(v, k)$ is the length of a shortest walk from $s$ to $v$ with at most $k$ hops.

**Proof.**

Standard induction (left as exercise).
Bellman-Ford computes the shortest paths correctly

**Lemma 18.4.**

If $G$ does not have a negative length cycle reachable from $s \Rightarrow \forall v$: $d(v, n) = d(v, n - 1)$.

Also, $d(v, n - 1)$ is the length of the shortest path between $s$ and $v$.

**Proof.**

Shortest walk from $s$ to reachable vertex is a path [not repeated vertex] (otherwise $\exists$ neg cycle).

A path has at most $n - 1$ edges.

$\Rightarrow$ Len shortest walk from $s$ to $v$ with at most $n - 1$ edges

= Len shortest walk from $s$ to $v$

= Len shortest path from $s$ to $v$.

By **Lemma 18.3**: $d(v, n) = d(v, n - 1) = \text{dist}(s, v)$, for all $v$. □
Bellman-Ford computes the shortest paths correctly

**Lemma 18.4.**

If $G$ does not has a negative length cycle reachable from $s$ $\implies \forall v:$

$$d(v, n) = d(v, n - 1).$$

Also, $d(v, n - 1)$ is the length of the shortest path between $s$ and $v$.

**Proof.**

Shortest walk from $s$ to reachable vertex is a path [not repeated vertex] (otherwise $\exists$ neg cycle).

A path has at most $n - 1$ edges.

$\implies$ Len shortest walk from $s$ to $v$ with at most $n - 1$ edges

$= \text{Len shortest walk from } s \text{ to } v$

$= \text{Len shortest path from } s \text{ to } v$.

By Lemma 18.3: $d(v, n) = d(v, n - 1) = \text{dist}(s, v)$, for all $v$.  

Bellman-Ford computes the shortest paths correctly

**Lemma 18.4.**

If $G$ does not have a negative length cycle reachable from $s$ $\implies \forall v:$

$$d(v, n) = d(v, n - 1).$$

Also, $d(v, n - 1)$ is the length of the shortest path between $s$ and $v$.

**Proof.**

Shortest walk from $s$ to reachable vertex is a path [not repeated vertex] (otherwise $\exists$ neg cycle).

A path has at most $n - 1$ edges.

$\implies$ Len shortest walk from $s$ to $v$ with at most $n - 1$ edges

$=$ Len shortest walk from $s$ to $v$

$=$ Len shortest path from $s$ to $v$.

By **Lemma 18.3** : $d(v, n) = d(v, n - 1) = \text{dist}(s, v)$, for all $v$. $\square$
Bellman-Ford computes the shortest paths correctly

Lemma 18.4.

If $G$ does not have a negative length cycle reachable from $s$ $\iff \forall v:\ d(v, n) = d(v, n - 1)$.

Also, $d(v, n - 1)$ is the length of the shortest path between $s$ and $v$.

Proof.

Shortest walk from $s$ to reachable vertex is a path [not repeated vertex] (otherwise $\exists$ neg cycle).

A path has at most $n - 1$ edges.

$\implies$ Len shortest walk from $s$ to $v$ with at most $n - 1$ edges

$=$ Len shortest walk from $s$ to $v$

$=$ Len shortest path from $s$ to $v$.

By Lemma 18.3: $d(v, n) = d(v, n - 1) = \text{dist}(s, v)$, for all $v$. 

Bellman-Ford computes the shortest paths correctly

**Lemma 18.4.**

If $G$ does not has a negative length cycle reachable from $s$ \iff \forall v: d(v, n) = d(v, n - 1).

Also, $d(v, n - 1)$ is the length of the shortest path between $s$ and $v$.

**Proof.**

Shortest walk from $s$ to reachable vertex is a path [not repeated vertex] (otherwise $\exists$ neg cycle).

A path has at most $n - 1$ edges.

\implies Len shortest walk from $s$ to $v$ with at most $n - 1$ edges

= Len shortest walk from $s$ to $v$

= Len shortest path from $s$ to $v$.

By Lemma 18.3: $d(v, n) = d(v, n - 1) = \text{dist}(s, v)$, for all $v$. \qed
Bellman-Ford computes the shortest paths correctly

**Lemma 18.4.**

If $G$ does not has a negative length cycle reachable from $s \implies \forall v:\ d(v, n) = d(v, n - 1)$.

Also, $d(v, n - 1)$ is the length of the shortest path between $s$ and $v$.

**Proof.**

Shortest walk from $s$ to reachable vertex is a path [not repeated vertex] (otherwise $\exists$ neg cycle).

A path has at most $n - 1$ edges.

$\implies$ Len shortest walk from $s$ to $v$ with at most $n - 1$ edges

$=$ Len shortest walk from $s$ to $v$

$=$ Len shortest path from $s$ to $v$.

By **Lemma 18.3** : $d(v, n) = d(v, n - 1) = \text{dist}(s, v)$, for all $v$. 

\[ \square \]
THE END

...

(for now)
18.2.4
Bellman-Ford: Detecting negative cycles
Correctness: detecting negative length cycle

**Lemma 18.5.**

Suppose $G$ has a negative cycle $C$ reachable from $s$. Then there is some node $v \in C$ such that $d(v, n) < d(v, n - 1)$.

**Proof.**

Suppose not. Let $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_h \rightarrow v_1$ be negative length cycle reachable from $s$. $d(v_i, n - 1)$ is finite for $1 \leq i \leq h$ since $C$ is reachable from $s$. By assumption $d(v, n) \geq d(v, n - 1)$ for all $v \in C$; implies no change in $n$th iteration; $d(v_i, n - 1) = d(v_i, n)$ for $1 \leq i \leq h$. This means $d(v_i, n - 1) \leq d(v_{i-1}, n - 1) + \ell(v_{i-1}, v_i)$ for $2 \leq i \leq h$ and $d(v_1, n - 1) \leq d(v_n, n - 1) + \ell(v_n, v_1)$. Adding up all these inequalities results in the inequality $0 \leq \ell(C)$ which contradicts the assumption that $\ell(C) < 0$. 

□
Lemma 18.5.

Suppose $G$ has a negative cycle $C$ reachable from $s$. Then there is some node $v \in C$ such that $d(v, n) < d(v, n - 1)$.

Proof.

Suppose not. Let $C = v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_h \rightarrow v_1$ be negative length cycle reachable from $s$. $d(v_i, n - 1)$ is finite for $1 \leq i \leq h$ since $C$ is reachable from $s$. By assumption $d(v, n) \geq d(v, n - 1)$ for all $v \in C$; implies no change in $n$th iteration; $d(v_i, n - 1) = d(v_i, n)$ for $1 \leq i \leq h$. This means $d(v_i, n - 1) \leq d(v_{i-1}, n - 1) + \ell(v_{i-1}, v_i)$ for $2 \leq i \leq h$ and $d(v_1, n - 1) \leq d(v_n, n - 1) + \ell(v_n, v_1)$. Adding up all these inequalities results in the inequality $0 \leq \ell(C)$ which contradicts the assumption that $\ell(C) < 0$. □
Proof of Lemma 18.5 in more detail...

\[
d(v_1, n) \leq d(v_0, n - 1) + \ell(v_0, v_1)
\]
\[
d(v_2, n) \leq d(v_1, n - 1) + \ell(v_1, v_2)
\]
\[
\vdots
\]
\[
d(v_i, n) \leq d(v_{i-1}, n - 1) + \ell(v_{i-1}, v_i)
\]
\[
\vdots
\]
\[
d(v_k, n) \leq d(v_{k-1}, n - 1) + \ell(v_{k-1}, v_k)
\]
\[
d(v_0, n) \leq d(v_k, n - 1) + \ell(v_k, v_0)
\]
Proof of **Lemma 18.5** in more detail...

\[ d(v_1, n) \leq d(v_0, n) + \ell(v_0, v_1) \]
\[ d(v_2, n) \leq d(v_1, n) + \ell(v_1, v_2) \]
\[ \ldots \]
\[ d(v_i, n) \leq d(v_{i-1}, n) + \ell(v_{i-1}, v_i) \]
\[ \ldots \]
\[ d(v_k, n) \leq d(v_{k-1}, n) + \ell(v_{k-1}, v_k) \]
\[ d(v_0, n) \leq d(v_k, n) + \ell(v_k, v_0) \]
Proof of **Lemma 18.5** in more detail...

\[ d(v_1, n) \leq d(v_0, n) + \ell(v_0, v_1) \]
\[ d(v_2, n) \leq d(v_1, n) + \ell(v_1, v_2) \]

\[ \ldots \]
\[ d(v_i, n) \leq d(v_{i-1}, n) + \ell(v_{i-1}, v_i) \]

\[ \ldots \]
\[ d(v_k, n) \leq d(v_{k-1}, n) + \ell(v_{k-1}, v_k) \]
\[ d(v_0, n) \leq d(v_k, n) + \ell(v_k, v_0) \]

\[ \sum_{i=0}^{k} d(v_i, n) \leq \sum_{i=0}^{k} d(v_i, n) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0) \]
Proof of Lemma 18.5 in more detail...

\[ \sum_{i=0}^{k} d(v_i, n) \leq \sum_{i=0}^{k} d(v_i, n) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0) \]

\[ 0 \leq \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0). \]
Proof of **Lemma 18.5** in more detail...

\[
\sum_{i=0}^{k} d(v_i, n) \leq \sum_{i=0}^{k} d(v_i, n) + \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0)
\]

\[
0 \leq \sum_{i=1}^{k} \ell(v_{i-1}, v_i) + \ell(v_k, v_0) = \text{len}(C).
\]
Proof of Lemma 18.5 in more detail...

$C$ is a not a negative cycle. Contradiction.
Negative cycles can not hide

**Lemma 18.4** restated

If $G$ does not have a negative length cycle reachable from $s \Rightarrow \forall v:\ d(v, n) = d(v, n - 1)$.

Also, $d(v, n - 1)$ is the length of the shortest path between $s$ and $v$.

**Lemma 18.4** and **Lemma 18.5** put together are the following:

**Lemma 18.6.**

$G$ has a negative length cycle reachable from $s \iff$ there is some node $v$ such that $d(v, n) < d(v, n - 1)$. 
Bellman-Ford: Negative Cycle Detection

The official final version

```plaintext
for each \( u \in V \) do
    \( d(u) \leftarrow \infty \)
    \( d(s) \leftarrow 0 \)

for \( k = 1 \) to \( n - 1 \) do
    for each \( v \in V \) do
        for each edge \((u, v) \in in(v)\) do
            \( d(v) = \min\{d(v), d(u) + \ell(u, v)\} \)

(* One more iteration to check if distances change *)

for each \( v \in V \) do
    for each edge \((u, v) \in in(v)\) do
        if \( d(v) > d(u) + \ell(u, v) \)
            Output ``Negative Cycle''

for each \( v \in V \) do
    \( \text{dist}(s, v) \leftarrow d(v) \)
```
THE END

... (for now)
18.2.5
Variants on Bellman-Ford
Finding the Paths and a Shortest Path Tree

How do we find a shortest path tree in addition to distances?

- For each $v$ the $d(v)$ can only get smaller as algorithm proceeds.
- If $d(v)$ becomes smaller it is because we found a vertex $u$ such that $d(v) > d(u) + ℓ(u, v)$ and we update $d(v) = d(u) + ℓ(u, v)$. That is, we found a shorter path to $v$ through $u$.
- For each $v$ have a $\text{prev}(v)$ pointer and update it to point to $u$ if $v$ finds a shorter path via $u$.
- At end of algorithm $\text{prev}(v)$ pointers give a shortest path tree oriented towards the source $s$. 
Negative Cycle Detection

Given directed graph $G$ with arbitrary edge lengths, does it have a negative length cycle?

1. Bellman-Ford checks whether there is a negative cycle $C$ that is reachable from a specific vertex $s$. There may negative cycles not reachable from $s$.
2. Run Bellman-Ford $|V|$ times, once from each node $u$.
Negative Cycle Detection

Given directed graph $G$ with arbitrary edge lengths, does it have a negative length cycle?

1. Bellman-Ford checks whether there is a negative cycle $C$ that is reachable from a specific vertex $s$. There may be negative cycles not reachable from $s$.
2. Run Bellman-Ford $|V|$ times, once from each node $u$?
Negative Cycle Detection

1. Add a new node $s'$ and connect it to all nodes of $G$ with zero length edges. Bellman-Ford from $s'$ will find a negative length cycle if there is one. Exercise: why does this work?

2. Negative cycle detection can be done with one Bellman-Ford invocation.
THE END

... (for now)
18.3

Shortest Paths in DAGs
Shortest Paths in a DAG

Single-Source Shortest Path Problems

- **Input** A directed acyclic graph \( G = (V, E) \) with arbitrary (including negative) edge lengths. For edge \( e = (u, v) \), \( \ell(e) = \ell(u, v) \) is its length.

1. Given nodes \( s, t \) find shortest path from \( s \) to \( t \).
2. Given node \( s \) find shortest path from \( s \) to all other nodes.

Simplification of algorithms for DAGs

- No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
- Can order nodes using topological sort
Shortest Paths in a DAG

Single-Source Shortest Path Problems

**Input** A directed acyclic graph $G = (V, E)$ with arbitrary (including negative) edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.

Simplification of algorithms for DAGs

1. No cycles and hence no negative length cycles! Hence can find shortest paths even for negative length edges
2. Can order nodes using topological sort
Algorithm for DAGs

1. Want to find shortest paths from \( s \). Ignore nodes not reachable from \( s \).
2. Let \( s = v_1, v_2, v_{i+1}, \ldots, v_n \) be a topological sort of \( G \).

Observation:

1. shortest path from \( s \) to \( v_i \) cannot use any node from \( v_{i+1}, \ldots, v_n \)
2. can find shortest paths in topological sort order.
Algorithm for DAGs

1. Want to find shortest paths from $s$. Ignore nodes not reachable from $s$.
2. Let $s = v_1, v_2, v_{i+1}, \ldots, v_n$ be a topological sort of $G$

Observation:

1. shortest path from $s$ to $v_i$ cannot use any node from $v_{i+1}, \ldots, v_n$
2. can find shortest paths in topological sort order.
Algorithm for DAGs

```plaintext
for i = 1 to n do
    d(s, vi) = ∞
    d(s, s) = 0

for i = 1 to n - 1 do
    for each edge (vi, vj) in Adj(vi) do
        d(s, vj) = min{d(s, vj), d(s, vi) + ℓ(vi, vj)}

return d(s, ·) values computed
```

Correctness: induction on $i$ and observation in previous slide.

Running time: $O(m + n)$ time algorithm! Works for negative edge lengths and hence can find longest paths in a DAG.
Bellman-Ford and DAGs

Bellman-Ford is based on the following principles:
- The shortest walk length from $s$ to $v$ with at most $k$ hops can be computed via dynamic programming
- $G$ has a negative length cycle reachable from $s$ iff there is a node $v$ such that shortest walk length reduces after $n$ hops.

We can find hop-constrained shortest paths via graph reduction.

Given $G = (V, E)$ with edge lengths $\ell(e)$ and integer $k$ construction new layered graph $G' = (V', E')$ as follows.
- $V' = V \times \{0, 1, 2, \ldots, k\}$.
- $E' = \{((u, i), (v, i + 1)) \mid (u, v) \in E, 0 \leq i < k\}$,
  \[\ell((u, i), (v, i + 1)) = \ell(u, v)\]

Lemma 18.1.
Shortest path distance from $(u, 0)$ to $(v, k)$ in $G'$ is equal to the shortest walk from $u$ to $v$ in $G$ with exactly $k$ edges.
Layered DAG: Figure
THE END

...

(for now)
18.4
All Pairs Shortest Paths
18.4.1

Problem definition and what we can already do
Shortest Path Problems

Input: A (undirected or directed) graph $G = (V, E)$ with edge lengths (or costs). For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.
3. Find shortest paths for all pairs of nodes.
SSSP: Single-Source Shortest Paths

Single-Source Shortest Path Problems

- **Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.

Dijkstra’s algorithm for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $O(nm)$. 
SSSP: Single-Source Shortest Paths

Single-Source Shortest Path Problems

Input: A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Given nodes $s, t$ find shortest path from $s$ to $t$.
2. Given node $s$ find shortest path from $s$ to all other nodes.

Dijkstra’s algorithm for non-negative edge lengths. Running time: $O((m + n) \log n)$ with heaps and $O(m + n \log n)$ with advanced priority queues.

Bellman-Ford algorithm for arbitrary edge lengths. Running time: $O(nm)$. 
All-Pairs Shortest Paths

Using the shortest paths algorithms we already have...

**All-Pairs Shortest Path Problem**

- **Input**: A (undirected or directed) graph \( G = (V, E) \) with edge lengths. For edge \( e = (u, v) \), \( \ell(e) = \ell(u, v) \) is its length.
- Find shortest paths for all pairs of nodes.

Apply single-source algorithms \( n \) times, once for each vertex.

- Non-negative lengths: \( O(nm \log n) \) with heaps and \( O(nm + n^2 \log n) \) using advanced priority queues.
- Arbitrary edge lengths: \( O(n^2m) \).
  \( \Theta(n^4) \) if \( m = \Omega(n^2) \).

Can we do better?
All-Pairs Shortest Paths

Using the shortest paths algorithms we already have...

**All-Pairs Shortest Path Problem**

**Input** A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.

2. Arbitrary edge lengths: $O(n^2m)$.

$\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?
All-Pairs Shortest Paths

Using the shortest paths algorithms we already have...

All-Pairs Shortest Path Problem

Input A (undirected or directed) graph $G = (V, E)$ with edge lengths. For edge $e = (u, v)$, $\ell(e) = \ell(u, v)$ is its length.

1. Find shortest paths for all pairs of nodes.

Apply single-source algorithms $n$ times, once for each vertex.

1. Non-negative lengths. $O(nm \log n)$ with heaps and $O(nm + n^2 \log n)$ using advanced priority queues.

2. Arbitrary edge lengths: $O(n^2m)$.
   $\Theta(n^4)$ if $m = \Omega(n^2)$.

Can we do better?
THE END

... (for now)
18.4.2
All Pairs Shortest Paths: A recursive solution
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as \( v_1, v_2, \ldots, v_n \)

2. \( \text{dist}(i, j, k) \): length of shortest walk from \( v_i \) to \( v_j \) among all walks in which the largest index of an intermediate node is at most \( k \) (could be \( -\infty \) if there is a negative length cycle).

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
\]
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

2. $\text{dist}(i, j, k)$: length of shortest walk from $v_i$ to $v_j$ among all walks in which the largest index of an intermediate node is at most $k$ (could be $-\infty$ if there is a negative length cycle).

$$
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
$$
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

2. $\text{dist}(i, j, k)$: length of shortest walk from $v_i$ to $v_j$ among all walks in which the largest index of an intermediate node is at most $k$ (could be $-\infty$ if there is a negative length cycle).

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
\]
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

2. $\text{dist}(i, j, k)$: length of shortest walk from $v_i$ to $v_j$ among all walks in which the largest index of an intermediate node is at most $k$ (could be $-\infty$ if there is a negative length cycle).

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
\]
All-Pairs: Recursion on index of intermediate nodes

1. Number vertices arbitrarily as $v_1, v_2, \ldots, v_n$

2. $\text{dist}(i, j, k)$: length of shortest walk from $v_i$ to $v_j$ among all walks in which the largest index of an intermediate node is at most $k$ (could be $-\infty$ if there is a negative length cycle).

\[
\begin{align*}
\text{dist}(i, j, 0) &= 100 \\
\text{dist}(i, j, 1) &= 9 \\
\text{dist}(i, j, 2) &= 8 \\
\text{dist}(i, j, 3) &= 5
\end{align*}
\]
For the following graph, \( \text{dist}(i, j, 2) \) is...
All-Pairs: Recursion on index of intermediate nodes

\[
dist(i, k, k - 1) \\
\text{k} \\
dist(k, j, k - 1)
\]

\[
dist(i, j, k - 1)
\]

\[
dist(i, j, k) = \min \left\{ \begin{array}{ll}
dist(i, j, k - 1) \\
dist(i, k, k - 1) + dist(k, j, k - 1)
\end{array} \right\}
\]

Base case: \( \text{dist}(i, j, 0) = \ell(i, j) \) if \( (i, j) \in E \), otherwise \( \infty \)

Correctness: If \( i \to j \) shortest walk goes through \( k \) then \( k \) occurs only once on the path — otherwise there is a negative length cycle.
All-Pairs: Recursion on index of intermediate nodes

If $i$ can reach $k$ and $k$ can reach $j$ and $\text{dist}(k, k, k - 1) < 0$ then $G$ has a negative length cycle containing $k$ and $\text{dist}(i, j, k) = -\infty$.

Recursion below is valid only if $\text{dist}(k, k, k - 1) \geq 0$. We can detect this during the algorithm or wait till the end.

$$\text{dist}(i, j, k) = \min \begin{cases} 
\text{dist}(i, j, k - 1) \\
\text{dist}(i, k, k - 1) + \text{dist}(k, j, k - 1) 
\end{cases}$$
THE END

...(for now)
18.4.3
Floyd-Warshall algorithm
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

\[
d(i, j, k) = \min \begin{cases} 
  d(i, j, k - 1) \\
  d(i, k, k - 1) + d(k, j, k - 1) 
\end{cases}
\]

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    \( d(i, j, 0) = \ell(i, j) \)
    (* \( \ell(i, j) = \infty \) if \( (i, j) \notin E \), 0 if \( i = j \) *)

for \( k = 1 \) to \( n \) do
  for \( i = 1 \) to \( n \) do
    for \( j = 1 \) to \( n \) do
      \[ d(i, j, k) = \min \begin{cases} 
        d(i, j, k - 1), \\
        d(i, k, k - 1) + d(k, j, k - 1) 
      \end{cases} \]

for \( i = 1 \) to \( n \) do
  if \( \text{dist}(i, i, n) < 0 \) then
    Output \( \exists \) negative cycle in \( G \)

Running Time: \( \Theta(n^3) \).
Space: \( \Theta(n^3) \).
Correctness:
via induction and recursive definition
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

\[
d(i, j, k) = \min \left\{ \begin{array}{ll}
d(i, j, k - 1) \\
d(i, k, k - 1) + d(k, j, k - 1)
\end{array} \right.
\]

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    \( d(i, j, 0) = \ell(i, j) \)
    (* \( \ell(i, j) = \infty \) if \( (i, j) \notin E \), \( 0 \) if \( i = j \) *)

for \( k = 1 \) to \( n \) do
  for \( i = 1 \) to \( n \) do
    for \( j = 1 \) to \( n \) do
      \[
d(i, j, k) = \min \left\{ \begin{array}{ll}
d(i, j, k - 1), \\
d(i, k, k - 1) + d(k, j, k - 1)
\end{array} \right.
\]
  for \( i = 1 \) to \( n \) do
    if \( (\text{dist}(i, i, n) < 0) \) then
      Output \( \exists \) negative cycle in \( G \)

Running Time: \( \Theta(n^3) \).
Space: \( \Theta(n^3) \).
Correctness:
via induction and recursive definition
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

\[ d(i, j, k) = \min \left\{ \begin{array}{l}
    d(i, j, k - 1) \\
    d(i, k, k - 1) + d(k, j, k - 1)
\end{array} \right. \]

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    \( d(i, j, 0) = \ell(i, j) \)
    (\(* \ell(i, j) = \infty \) if \((i, j) \notin E, \ 0 \) if \( i = j \) \*)

for \( k = 1 \) to \( n \) do
  for \( i = 1 \) to \( n \) do
    for \( j = 1 \) to \( n \) do
      \[ d(i, j, k) = \min \left\{ \begin{array}{l}
        d(i, j, k - 1), \\
        d(i, k, k - 1) + d(k, j, k - 1)
      \end{array} \right. \]

for \( i = 1 \) to \( n \) do
  if (dist\( (i, i, n) < 0 \)) then
    Output \( \exists \) negative cycle in \( G \)

Running Time: \( \Theta(n^3) \).
Space: \( \Theta(n^3) \).
Correctness:
via induction and recursive definition
Floyd-Warshall Algorithm
for All-Pairs Shortest Paths

\[
d(i, j, k) = \min \left\{ d(i, j, k - 1), \ d(i, k, k - 1) + d(k, j, k - 1) \right\}
\]

for \( i = 1 \) to \( n \) do
  for \( j = 1 \) to \( n \) do
    \( d(i, j, 0) = \ell(i, j) \)  
    (\( \ell(i, j) = \infty \) if \((i, j) \notin E\), \(0\) if \(i = j\) *)
  
for \( k = 1 \) to \( n \) do
  for \( i = 1 \) to \( n \) do
    for \( j = 1 \) to \( n \) do
      \( d(i, j, k) = \min \left\{ d(i, j, k - 1), \ d(i, k, k - 1) + d(k, j, k - 1) \right\} \)

for \( i = 1 \) to \( n \) do
  if \( \text{dist}(i, i, n) < 0 \) then
    Output \( \exists \) negative cycle in \( G \)

Running Time: \( \Theta(n^3) \).
Space: \( \Theta(n^3) \).

Correctness:
via induction and recursive definition
Floyd-Warshall Algorithm: Finding the Paths

**Question:** Can we find the paths in addition to the distances?

1. Create a $n \times n$ array `Next` that stores the next vertex on shortest path for each pair of vertices.
2. With array `Next`, for any pair of given vertices $i, j$ can compute a shortest path in $O(n)$ time.
Floyd-Warshall Algorithm: Finding the Paths

**Question:** Can we find the paths in addition to the distances?

1. Create a $n \times n$ array $\text{Next}$ that stores the next vertex on shortest path for each pair of vertices.
2. With array $\text{Next}$, for any pair of given vertices $i, j$ can compute a shortest path in $O(n)$ time.
Floyd-Warshall Algorithm

Finding the Paths

for $i = 1$ to $n$ do
  for $j = 1$ to $n$ do
    $d(i, j, 0) = \ell(i, j)$
    (* $\ell(i, j) = \infty$ if $(i, j)$ not edge, 0 if $i = j$ *)
    $Next(i, j) = -1$
  Next ($i, j$) = $k$

for $k = 1$ to $n$ do
  for $i = 1$ to $n$ do
    for $j = 1$ to $n$ do
      if $(d(i, j, k - 1) > d(i, k, k - 1) + d(k, j, k - 1))$ then
        $d(i, j, k) = d(i, k, k - 1) + d(k, j, k - 1)$
        $Next(i, j) = k$
  Next ($i, j$) = $k$

for $i = 1$ to $n$ do
  if $(d(i, i, n) < 0)$ then
    Output that there is a negative length cycle in $G$

Exercise: Given Next array and any two vertices $i, j$ describe an $O(n)$ algorithm to find a $i-j$ shortest path.
THE END

... (for now)
18.5

Summary of shortest path algorithms
### Summary of results on shortest paths

<table>
<thead>
<tr>
<th>Single source</th>
<th>Dijkstra</th>
<th>Bellman Ford</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative edges</td>
<td>(O(n \log n + m))</td>
<td>(O(nm))</td>
</tr>
<tr>
<td>Edge lengths can be negative</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### All Pairs Shortest Paths

<table>
<thead>
<tr>
<th>No negative edges</th>
<th>(n \times) Dijkstra</th>
<th>(O(n^2 \log n + nm))</th>
</tr>
</thead>
<tbody>
<tr>
<td>No negative cycles</td>
<td>(n \times) Bellman Ford</td>
<td>(O(n^2 m) = O(n^4))</td>
</tr>
<tr>
<td>No negative cycles (*)</td>
<td>BF + (n \times) Dijkstra</td>
<td>(O(nm + n^2 \log n))</td>
</tr>
<tr>
<td>No negative cycles</td>
<td>Floyd-Warshall</td>
<td>(O(n^3))</td>
</tr>
<tr>
<td>Unweighted</td>
<td>Matrix multiplication</td>
<td>(O(n^{2.38}), O(n^{2.58}))</td>
</tr>
</tbody>
</table>
Summary of results on shortest paths

More details

(*): The algorithm for the case that there are no negative cycles, and doing all shortest paths, works by computing a potential function using Bellman-Ford and then doing Dijkstra. It is mentioned for the sake of completeness, but it outside the scope of the class.
THE END

...

(for now)
18.6

DFA to Regular Expression
Back to Regular Languages

We saw the following two theorems previously.

**Theorem 18.1.**

For every NFA $N$ over a finite alphabet $\Sigma$ there is DFA $M$ such that $L(M) = L(N)$.

**Theorem 18.2.**

For every regular expression $r$ over finite alphabet $\Sigma$ there is a NFA $N$ such that $L(N) = L(r)$.

We claimed the following theorem which would prove equivalence of NFAs, DFAs and regular expressions.

**Theorem 18.3.**

For every DFA $M$ over a finite alphabet $\Sigma$ there is a regular expression $r$ such that $L(M) = L(r)$. 
Back to Regular Languages

We saw the following two theorems previously.

**Theorem 18.1.**
For every NFA $N$ over a finite alphabet $\Sigma$ there is DFA $M$ such that $L(M) = L(N)$.

**Theorem 18.2.**
For every regular expression $r$ over finite alphabet $\Sigma$ there is a NFA $N$ such that $L(N) = L(r)$.

We claimed the following theorem which would prove equivalence of NFAs, DFAs and regular expressions.

**Theorem 18.3.**
For every DFA $M$ over a finite alphabet $\Sigma$ there is a regular expression $r$ such that $L(M) = L(r)$. 

DFA to Regular Expression

Given DFA $M = (Q, \Sigma, \delta, q_1, F)$ want to construct an equivalent regular expression $r$.

Idea:
- Number states of DFA: $Q = \{q_1, \ldots, q_n\}$ where $|Q| = n$.
- Define $L_{i,j} = \{w \mid \delta(q_i, w) = q_j\}$. Note $L_{i,j}$ is regular. Why?
- $L(M) = \bigcup_{q_i \in F} L_{1,i}$.
- Obtain regular expression $r_{i,j}$ for $L_{i,j}$.
- Then $r = \sum_{q_i \in F} r_{1,i}$ is regular expression for $L(M)$ – here the summation is the or operator.

Note: Using $q_1$ for start state is intentional to help in the notation for the recursion.
A recursive expression for $L_{i,j}$

Define $L_{i,j}^k$ be set of strings $w$ in $L_{i,j}$ such that the highest index state visited by $M$ on walk from $q_i$ to $q_j$ (not counting end points $i$ and $j$) on input $w$ is at most $k$.

**Claim:**

$$L_{i,i}^0 = \{ a \in \Sigma | \delta(q_i, a) = q_i \}^*$$

$$L_{i,j}^0 = L_{i,i}^0 \{ a \in \Sigma | \delta(q_i, a) = q_j \} L_{j,j}^0$$

if $i \neq j$

$$L_{i,j}^k = L_{i,j}^{k-1} \cup \left( L_{i,k}^{k-1} \cdot L_{k,k}^{k-1} \cdot L_{k,j}^{k-1} \right)$$

if $i \neq j$

$$L_{i,i}^k = \left( L_{i,i}^{k-1} \cup L_{i,k}^{k-1} \cdot L_{k,k}^{k-1} \cdot L_{k,i}^{k-1} \right)^*$$

$$L_{i,j} = L_{i,j}^n.$$
A recursive expression for $L_{i,j}$

Claim:

\[
L^0_{i,i} = \{ a \in \Sigma \mid \delta(q_i, a) = q_i \}^* \\
L^0_{i,j} = L^0_{i,i} \{ a \in \Sigma \mid \delta(q_i, a) = q_j \} L^0_{j,i} \\
L^k_{i,j} = L^k_{i,j} \cup \left( L^{k-1}_{i,k} \cdot L^{k-1}_{k,k} \cdot L^{k-1}_{k,j} \right) \\
L^k_{i,i} = \left( L^{k-1}_{i,i} \cup L^{k-1}_{i,k} \cdot L^{k-1}_{k,k} \cdot L^{k-1}_{k,i} \right)^* \\
L_{i,j} = L^n_{i,j}.
\]

Proof: by picture
A recursive expression for $L_{i,j}$

Claim:

\[
L^0_{i,i} = \{ a \in \Sigma \mid \delta(q_i, a) = q_i \}^*
\]

\[
L^0_{i,j} = L^0_{i,i} \{ a \in \Sigma \mid \delta(q_i, a) = q_j \} L^0_{j,j}
\]

if $i \neq j$

\[
L^k_{i,j} = L^k_{i,j} \cup \left( L^k_{i,k} \cdot L^k_{k,k} \cdot L^k_{k,j} \right)
\]

if $i \neq j$

\[
L^k_{i,i} = \left( L^k_{i,i} \cup L^k_{i,k} \cdot L^k_{k,k} \cdot L^k_{k,i} \right)^*
\]

\[
L_{i,j} = L^n_{i,j}.
\]

The desired language is

\[
L(M) = \bigcup_{q_i \in F} L_{1,i} = \bigcup_{q_i \in F} L^n_{1,i}
\]
A regular expression for $L(M)$

$$r_{i,i}^0 = \left( \sum_{a \in \Sigma: \delta(q_i, a) = q_i} a \right)^*$$

$$r_{i,j}^0 = r_{i,i}^0 \left( \sum_{a \in \Sigma: \delta(q_i, a) = q_j} a \right) r_{j,j}^0 \quad \text{if } i \neq j$$

$$r_{i,j}^k = r_{i,j}^{k-1} + r_{i,k}^{k-1} r_{k,k}^{k-1} r_{k,j}^{k-1} \quad i \neq j$$

$$r_{i,i}^k = \left( r_{i,i}^{k-1} + r_{i,k}^{k-1} \cdot r_{k,k}^{k-1} \cdot r_{k,i}^{k-1} \right)^*$$

$$r_{i,j} = r_{i,j}^n.$$

The desired regular expression is: $\text{reg-expression}(M) = \sum_{q_i \in F} r_{1,i} = \sum_{q_i \in F} r_{1,i}^n.$
Example

\[ r_{1,1}^0 = r_{2,2}^0 = b^* \]
\[ r_{1,2}^0 = r_{2,1}^0 = b^*ab^* \]

\[ r_{1,1}^1 = (r_{1,1}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,1}^0)^* = b^* \]
\[ r_{2,2}^1 = (r_{2,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,2}^0)^* = (b^* + b^*ab^* b^*b^*ab^*)^* = (b^* + ab^*a)^* \]
\[ r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,2}^0 = b^*ab^* + b^*b^*ab^* = b^*ab^*. \]
\[ r_{2,1}^1 = r_{2,1}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0 = b^*ab^* \]
\[ r_{1,1}^2 = (r_{1,1}^1 + r_{1,2}^1 r_{2,2}^1 r_{2,1}^1)^* = \cdots \]
\[ r_{2,2}^1 = \cdots \]
Example

\[ r_{1,1}^0 = r_{2,2}^0 = b^* \quad r_{1,2}^0 = r_{2,1}^0 = b^* ab^* \]

\[ r_{1,1}^1 = (r_{1,1}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,1}^0)^* = b^* \]
\[ r_{2,2}^1 = (r_{2,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,2}^0)^* = (b^* + b^* ab^* b^* b^* ab^*)^* = (b^* + ab^* a)^* \]
\[ r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,2}^0 = b^* ab^* + b^* b^* ab^* = b^* ab^* . \]
\[ r_{2,1}^1 = r_{2,1}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0 = b^* ab^* \]
\[ r_{1,1}^2 = (r_{1,1}^1 + r_{1,2}^1 r_{2,2}^1 r_{2,1}^1)^* = \cdots \]
\[ r_{2,2}^1 = \cdots \]
Example

\[ r_{1,1}^0 = r_{2,2}^0 = b^* \]
\[ r_{1,2}^0 = r_{2,1}^0 = b^*ab^* \]

\[ r_{1,1}^1 = (r_{1,1}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,1}^0)^* = b^* \]
\[ r_{2,2}^1 = (r_{2,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0)^* = (b^* + b^*ab^*b^*b^*ab^*)^* = (b^* + ab^*a)^* \]
\[ r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,2}^0 = b^*ab^* + b^*ab^* = b^*ab^*. \]
\[ r_{2,1}^1 = r_{2,1}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0 = b^*ab^* \]
\[ r_{1,1}^2 = (r_{1,1}^1 + r_{1,2}^1 r_{2,2}^1 r_{2,1}^1)^* = \cdots \]
\[ r_{2,2}^1 = \cdots \]
Example

\[ r_{1,1}^0 = r_{2,2}^0 = b^* \]
\[ r_{1,2}^0 = r_{2,1}^0 = b^* ab^* \]

\[ r_{1,1}^1 = (r_{1,1}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,1}^0)^* = b^* \]
\[ r_{2,2}^1 = (r_{2,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0)^* = (b^* + b^* ab^* b^* b^* ab^*)^* = (b^* + ab^* a)^* \]
\[ r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,2}^0 = b^* ab^* + b^* b^* ab^* = b^* ab^* . \]
\[ r_{2,1}^1 = r_{1,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0 = b^* ab^* \]
\[ r_{1,1}^2 = (r_{1,1}^1 + r_{1,2}^1 r_{2,2}^1 r_{2,1}^1)^* = \cdots \]
\[ r_{2,2}^1 \]
Example

\[ r_{1,1}^0 = r_{2,2}^0 = b^* \quad r_{1,2}^0 = r_{2,1}^0 = b^* ab^* \]

\[ r_{1,1}^1 = (r_{1,1}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,1}^0)^* = b^* \]
\[ r_{2,2}^1 = (r_{2,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0)^* = (b^* + b^* ab^* b^* b^* ab^*)^* = (b^* + ab^* a)^* \]
\[ r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,2}^0 = b^* ab^* + b^* b^* ab^* = b^* ab^*. \]
\[ r_{2,1}^1 = r_{1,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0 = b^* ab^* \]
\[ r_{1,1}^2 = (r_{1,1}^1 + r_{1,2}^1 r_{2,2}^1 r_{2,1}^1)^* = \cdots \]
\[ r_{2,2}^1 = \cdots \]
Example

\[ r_{1,1}^0 = r_{2,2}^0 = b^* \]
\[ r_{1,2}^0 = r_{2,1}^0 = b^* ab^* \]

\[ r_{1,1}^1 = (r_{1,1}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,1}^0)^* = b^* \]
\[ r_{2,2}^1 = (r_{2,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0)^* = (b^* + b^* ab^* b^* b^* ab^*)^* = (b^* + ab^* a)^* \]
\[ r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,2}^0 = b^* ab^* + b^* b^* ab^* = b^* ab^*. \]
\[ r_{2,1}^1 = r_{2,1}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0 = b^* ab^* \]
\[ r_{1,1}^2 = (r_{1,1}^1 + r_{1,2}^1 r_{2,2}^1 r_{2,1}^1)^* = \cdots \]
\[ r_{2,2}^1 = \cdots \]
Example

\[ r_{1,1}^0 = r_{2,2}^0 = b^* \quad \quad r_{1,2}^0 = r_{2,1}^0 = b^* ab^* \]

\[ r_{1,1}^1 = (r_{1,1}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,1}^0)^* = b^* \]
\[ r_{2,2}^1 = (r_{2,2}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,2}^0)^* = (b^* + b^* ab^* b^* b^* ab^*)^* = (b^* + ab^* a)^* \]
\[ r_{1,2}^1 = r_{1,2}^0 + r_{1,1}^0 r_{1,1}^0 r_{1,2}^0 = b^* ab^* + b^* b^* ab^* = b^* ab^*. \]
\[ r_{2,1}^1 = r_{2,1}^0 + r_{2,1}^0 r_{1,1}^0 r_{1,1}^0 = b^* ab^* \]
\[ r_{1,1}^2 = (r_{1,1}^1 + r_{1,2}^1 r_{2,2}^1 r_{2,1}^1)^* = \cdots \]
\[ r_{2,2}^1 = \cdots \]
Correctness

Similar to that of Floyd-Warshall algorithms for shortest paths via induction.

The length of the regular expression can be exponential in the size of the original DFA.
THE END

...  

(for now)
18.7
Dynamic Programming: Postscript
Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?
Dynamic Programming: Postscript

Dynamic Programming = Smart Recursion + Memoization

1. How to come up with the recursion?
2. How to recognize that dynamic programming may apply?
Some Tips

1. Problems where there is a natural linear ordering: sequences, paths, intervals, DAGs etc. Recursion based on ordering (left to right or right to left or topological sort) usually works.

2. Problems involving trees: recursion based on subtrees.

3. More generally:
   1. Problem admits a natural recursive divide and conquer
   2. If optimal solution for whole problem can be simply composed from optimal solution for each separate pieces then plain divide and conquer works directly
   3. If optimal solution depends on all pieces then can apply dynamic programming if interface/interaction between pieces is limited. Augment recursion to not simply find an optimum solution but also an optimum solution for each possible way to interact with the other pieces.
Examples

1. Longest Increasing Subsequence: break sequence in the middle say. What is the interaction between the two pieces in a solution?

2. Sequence Alignment: break both sequences in two pieces each. What is the interaction between the two sets of pieces?

3. Independent Set in a Tree: break tree at root into subtrees. What is the interaction between the subtrees?

4. Independent Set in an graph: break graph into two graphs. What is the interaction? Very high!

5. Knapsack: Split items into two sets of half each. What is the interaction?