Dynamic Programming

T. M. Murali

September 26, October 1, 3, 8, 2018
Algorithm Design Techniques

Goal: design efficient (polynomial-time) algorithms.
Algorithm Design Techniques

1 Goal: design efficient (polynomial-time) algorithms.

2 Greedy
   ▶ Pro: natural approach to algorithm design.
   ▶ Con: many greedy approaches to a problem. Only some may work.
   ▶ Con: many problems for which no greedy approach is known.
Algorithm Design Techniques

1. **Goal:** design efficient (polynomial-time) algorithms.

2. **Greedy**
   - **Pro:** natural approach to algorithm design.
   - **Con:** many greedy approaches to a problem. Only some may work.
   - **Con:** many problems for which *no* greedy approach is known.

3. **Divide and conquer**
   - **Pro:** simple to develop algorithm skeleton.
   - **Con:** conquer step can be very hard to implement efficiently.
   - **Con:** usually reduces time for a problem known to be solvable in polynomial time.
Algorithm Design Techniques

1. Goal: design efficient (polynomial-time) algorithms.

2. Greedy
   - Pro: natural approach to algorithm design.
   - Con: many greedy approaches to a problem. Only some may work.
   - Con: many problems for which no greedy approach is known.

3. Divide and conquer
   - Pro: simple to develop algorithm skeleton.
   - Con: conquer step can be very hard to implement efficiently.
   - Con: usually reduces time for a problem known to be solvable in polynomial time.

4. Dynamic programming
   - More powerful than greedy and divide-and-conquer strategies.
   - *Implicitly* explore space of all possible solutions.
   - Solve multiple sub-problems and build up correct solutions to larger and larger sub-problems.
   - Careful analysis needed to ensure number of sub-problems solved is polynomial in the size of the input.
History of Dynamic Programming

- Bellman pioneered the systematic study of dynamic programming in the 1950s.
Bellman pioneered the systematic study of dynamic programming in the 1950s.

The Secretary of Defense at that time was hostile to mathematical research.

Bellman sought an impressive name to avoid confrontation.

- “it’s impossible to use dynamic in a pejorative sense”
- “something not even a Congressman could object to” (Bellman, R. E., *Eye of the Hurricane, An Autobiography*).
Applications of Dynamic Programming

- Computational biology: Smith-Waterman algorithm for sequence alignment.
- Operations research: Bellman-Ford algorithm for shortest path routing in networks.
- Control theory: Viterbi algorithm for hidden Markov models.
- Computer science (theory, graphics, AI, ...): Unix diff command for comparing two files.
Web Search for “dynamic programming”

- How do they know “Dynamic” and “Dymanic” are similar?
Sequence Similarity

- Given two strings, measure how similar they are.
- Given a database of strings and a query string, compute the string most similar to query in the database.

Applications:
- Online searches (Web, dictionary).
- Spell-checkers.
- Computational biology
- Speech recognition.
- Basis for Unix `diff`.
Defining Sequence Similarity

- “ocurrance” (wrong) vs “occurrence” (right).

\[
\begin{align*}
\text{o-currance} \\
\text{occurrence} \\
\text{o-curr-ance} \\
\text{occurre-nce}
\end{align*}
\]
Defining Sequence Similarity

- “ocurrance” (wrong) vs “occurrence” (right).

<table>
<thead>
<tr>
<th>o-currance</th>
<th>occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>o-curr-ance</td>
<td>occurre-nce</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>a-bbbaa--b-bbaab</td>
<td>ababaa-abb-bbba-b</td>
</tr>
</tbody>
</table>
Defining Sequence Similarity

- “ocurrance” (wrong) vs “occurrence” (right).

- Edit distance model: how many changes must you make to one string to transform it into another?
- Changes allowed are deleting a letter, adding a letter, changing a letter.
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings $x = x_1x_2x_3 \ldots x_m$ and $y = y_1y_2 \ldots y_n$.

Indices $\{1, 2, \ldots, m\}$ and $\{1, 2, \ldots, n\}$ represent positions in $x$ and $y$. 

Cost of an alignment is the sum of gap and mismatch penalties:

- Gap penalty $\delta > 0$ for every unmatched index.
- Mismatch penalty $\alpha$ if $(i, j) \in M$ and $x_i \neq y_j$.

Output: compute an alignment of minimal cost.
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings \( x = x_1 x_2 x_3 \ldots x_m \) and \( y = y_1 y_2 \ldots y_n \).

Indices \( \{1, 2, \ldots, m\} \) and \( \{1, 2, \ldots, n\} \) represent positions in \( x \) and \( y \).

A \textit{matching} of \( x \) and \( y \) is a set \( M \) of ordered pairs such that

1. in each pair \((i, j)\), \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \) and
2. no index from \( x \) (respectively, from \( y \)) appears as the first (respectively, second) element in more than one ordered pair.
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings \( x = x_1x_2x_3 \ldots x_m \) and \( y = y_1y_2 \ldots y_n \).

Indices \( \{1, 2, \ldots, m\} \) and \( \{1, 2, \ldots, n\} \) represent positions in \( x \) and \( y \).

A matching of \( x \) and \( y \) is a set \( M \) of ordered pairs such that
1. in each pair \( (i, j) \), \( 1 \leq i \leq m \) and \( 1 \leq j \leq n \) and
2. no index from \( x \) (respectively, from \( y \)) appears as the first (respectively, second) element in more than one ordered pair.

A matching \( M \) is an alignment if there are no “crossing pairs” in \( M \): if \( (i, j) \in M \) and \( (i', j') \in M \) and \( i < i' \) then \( j < j' \).
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings \( x = x_1 x_2 x_3 \ldots x_m \) and \( y = y_1 y_2 \ldots y_n \).

Indices \( \{1, 2, \ldots, m\} \) and \( \{1, 2, \ldots, n\} \) represent positions in \( x \) and \( y \).

A matching of \( x \) and \( y \) is a set \( M \) of ordered pairs such that

1. in each pair \((i, j)\), \(1 \leq i \leq m\) and \(1 \leq j \leq n\) and
2. no index from \( x \) (respectively, from \( y \)) appears as the first (respectively, second) element in more than one ordered pair.

A matching \( M \) is an alignment if there are no “crossing pairs” in \( M \): if \((i, j) \in M \) and \((i', j') \in M \) and \( i < i' \) then \( j < j' \).

An index is not matched if it does not appear in the matching.

Cost of an alignment is the sum of gap and mismatch penalties:

- **Gap penalty**   Penalty \( \delta > 0 \) for every unmatched index.
- **Mismatch penalty**   Penalty \( \alpha_{x_i, y_j} > 0 \) if \((i, j) \in M \) and \( x_i \neq y_j \).
Proposed by Needleman and Wunsch in the early 1970s.

Input: two strings \( x = x_1 x_2 x_3 \ldots x_m \) and \( y = y_1 y_2 \ldots y_n \).

Indices \( \{1, 2, \ldots, m\} \) and \( \{1, 2, \ldots, n\} \) represent positions in \( x \) and \( y \).

A matching of \( x \) and \( y \) is a set \( M \) of ordered pairs such that

1. in each pair \((i, j)\), \(1 \leq i \leq m\) and \(1 \leq j \leq n\) and
2. no index from \( x \) (respectively, from \( y \)) appears as the first (respectively, second) element in more than one ordered pair.

A matching \( M \) is an alignment if there are no “crossing pairs” in \( M \): if \((i, j) \in M \) and \((i', j') \in M \) and \( i < i' \) then \( j < j' \).

An index is not matched if it does not appear in the matching.

Cost of an alignment is the sum of gap and mismatch penalties:

- Gap penalty: Penalty \( \delta > 0 \) for every unmatched index.
- Mismatch penalty: Penalty \( \alpha_{x_i, y_j} > 0 \) if \((i, j) \in M \) and \( x_i \neq y_j \).

Output: compute an alignment of minimal cost.
Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index $m \in x$ and index $n \in y$. What are the possibilities?
Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index \( m \in x \) and index \( n \in y \). What are the possibilities?
  - \((m, n)\) could be paired in the matching \( M \).
  - Neither \( n \) nor \( m \) may be matched.
  - Only \( m \) may not be matched.
  - Only \( n \) may not be matched.

```
o - c u r r a n c e
  
  o c c u r r e n c e

  o c c u r r e n c e

  o c c u r r e n c e

  o c c u r r e n c e

  o - c u r r a n c e

  o c c u r r e n c e

  o c c u r r e n c e

  o c c u r r e n c e
```
Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index $m \in x$ and index $n \in y$. What are the possibilities?
  - $(m, n)$ could be paired in the matching $M$.
  - Neither $n$ nor $m$ may be matched.
  - Only $m$ may not be matched.
  - Only $n$ may not be matched.
  - $(m, i)$ could be paired and $(j, n)$ could be paired where $i < n$ and $j < m$. Not possible in an alignment!

- Not matched with each other
Developing Intuition for Dynamic Programming

- How do we start formulating the dynamic program?
- Consider index \( m \in x \) and index \( n \in y \). What are the possibilities?
  - \((m, n)\) could be paired in the matching \( M \).
  - Neither \( n \) nor \( m \) may be matched.
  - Only \( m \) may not be matched.
  - Only \( n \) may not be matched.
  - \((m, i)\) could be paired and \((j, n)\) could be paired where \( i < n \) and \( j < m \). Not possible in an alignment!
- Claim: \((m, n) \notin M \implies m \in x \) not matched or \( n \in y \) not matched.
Dynamic Programming Approach

- \((m, n) \notin M \Rightarrow m \in x\) not matched or \(n \in y\) not matched.
- How should we define sub-problems?
Dynamic Programming Approach

- (m, n) \notin M \Rightarrow m \in x \text{ not matched or } n \in y \text{ not matched.}
- How should we define sub-problems?
- \textit{OPT}(i, j): \text{ cost of optimal alignment between } x = x_1x_2x_3\ldots x_i \text{ and } y = y_1y_2\ldots y_j.
  - (i, j) \in M:
Dynamic Programming Approach

- $(m, n) \notin M \Rightarrow m \in x$ not matched or $n \in y$ not matched.
- How should we define sub-problems?
- $OPT(i, j)$: cost of optimal alignment between $x = x_1x_2x_3 \ldots x_i$ and $y = y_1y_2 \ldots y_j$.
  - $(i, j) \in M$: $OPT(i, j) = \alpha_{x_iy_j} + OPT(i - 1, j - 1)$.
Dynamic Programming Approach

- (m, n) \notin M \Rightarrow m \in x \text{ not matched or } n \in y \text{ not matched.}

- How should we define sub-problems?

- \textit{OPT}(i, j): \text{ cost of optimal alignment between } x = x_1x_2x_3 \ldots x_i \text{ and } y = y_1y_2 \ldots y_j.
  - (i, j) \in M: \text{OPT}(i, j) = \alpha_{x_iy_j} + \text{OPT}(i - 1, j - 1).
  - i \text{ not matched:}
Dynamic Programming Approach

- **(m, n) \notin M \Rightarrow m \in x \text{ not matched or } n \in y \text{ not matched.}**
- **How should we define sub-problems?**
- **OPT(i, j):** cost of optimal alignment between \( x = x_1x_2x_3 \ldots x_i \) and \( y = y_1y_2 \ldots y_j \).
  - \((i, j) \in M: \text{OPT}(i, j) = \alpha_{x_iy_j} + \text{OPT}(i - 1, j - 1).\)
  - \(i \text{ not matched: } \text{OPT}(i, j) = \delta + \text{OPT}(i - 1, j).\)
Sequence Alignment Shortest Paths

Dynamic Programming Approach

- **(m, n) \notin M \Rightarrow m \in x \text{ not matched or } n \in y \text{ not matched.}**
- **How should we define sub-problems?**
- **OPT(i, j):** cost of optimal alignment between \(x = x_1x_2x_3 \ldots x_i\) and \(y = y_1y_2 \ldots y_j\).
  - \((i, j) \in M: \text{OPT}(i, j) = \alpha_{x_iy_j} + \text{OPT}(i - 1, j - 1).\)
  - \(i \text{ not matched: } \text{OPT}(i, j) = \delta + \text{OPT}(i - 1, j).\)
  - \(j \text{ not matched: } \text{OPT}(i, j) = \delta + \text{OPT}(i, j - 1).\)
Dynamic Programming Approach

- \( m, n \not\in M \Rightarrow m \in x \) not matched or \( n \in y \) not matched.
- How should we define sub-problems?
- \( OPT(i, j) \): cost of optimal alignment between \( x = x_1x_2x_3 \ldots x_i \) and \( y = y_1y_2 \ldots y_j \).
  - \((i, j) \in M\): \( OPT(i, j) = \alpha_{x_i y_j} + OPT(i-1, j-1) \).
  - \( i \) not matched: \( OPT(i, j) = \delta + OPT(i-1, j) \).
  - \( j \) not matched: \( OPT(i, j) = \delta + OPT(i, j-1) \).

\[
OPT(i, j) = \min (\alpha_{x_i y_j} + OPT(i-1, j-1), \delta + OPT(i-1, j), \delta + OPT(i, j-1))
\]
  - \((i, j) \in M\) if and only if minimum is achieved by the first term.
- What are the base cases?

\[
OPT(i, 0) = OPT(0, i) = \delta.
\]
**Dynamic Programming Approach**

- \((m, n) \notin M \Rightarrow m \in x\) not matched or \(n \in y\) not matched.

- How should we define sub-problems?

- \(OPT(i, j)\): cost of optimal alignment between \(x = x_1x_2x_3 \ldots x_i\) and \(y = y_1y_2 \ldots y_j\).
  - \((i, j) \in M\): \(OPT(i, j) = \alpha_{x_iy_j} + OPT(i - 1, j - 1)\).
  - \(i\) not matched: \(OPT(i, j) = \delta + OPT(i - 1, j)\).
  - \(j\) not matched: \(OPT(i, j) = \delta + OPT(i, j - 1)\).

\[
OPT(i, j) = \min (\alpha_{x_iy_j} + OPT(i - 1, j - 1), \delta + OPT(i - 1, j), \delta + OPT(i, j - 1))
\]

- \((i, j) \in M\) if and only if minimum is achieved by the first term.

- What are the base cases? \(OPT(i, 0) = OPT(0, i) = i\delta\).
Dynamic Programming Algorithm

$$\text{OPT}(i,j) = \min \left( \alpha_{x_i y_j} + \text{OPT}(i-1,j-1), \delta + \text{OPT}(i-1,j), \delta + \text{OPT}(i,j-1) \right)$$

Alignment($X,Y$)

Array $A[0 \ldots m, 0 \ldots n]$

Initialize $A[i,0] = i\delta$ for each $i$

Initialize $A[0,j] = j\delta$ for each $j$

For $j = 1, \ldots, n$

For $i = 1, \ldots, m$

Use the recurrence (6.16) to compute $A[i,j]$

Endfor

Endfor

Return $A[m,n]$
Dynamic Programming Algorithm

\[ \text{OPT}(i,j) = \min \left( \alpha_{x_i,y_j} + \text{OPT}(i-1,j-1), \delta + \text{OPT}(i-1,j), \delta + \text{OPT}(i,j-1) \right) \]

Alignment \((X,Y)\)

Array \(A[0...m,0...n]\)

Initialize \(A[i,0]=i\delta\) for each \(i\)

Initialize \(A[0,j]=j\delta\) for each \(j\)

For \(j=1,...,n\)

For \(i=1,...,m\)

Use the recurrence (6.16) to compute \(A[i,j]\)

Endfor

Endfor

Return \(A[m,n]\)

- Running time is \(O(mn)\). Space used in \(O(mn)\).
- \((i,j)\) is in the optimal alignment if the first term is the smallest.
Improving the Running Time

\[ \text{OPT}(i, j) = \min \left( \alpha_{x_iy_j} + \text{OPT}(i - 1, j - 1), \delta + \text{OPT}(i - 1, j), \delta + \text{OPT}(i, j - 1) \right) \]

- Key observation: Computing entry \((i, j)\) requires values only in previous row/column or in previous row in current column.
Improving the Running Time

\[ \text{OPT}(i, j) = \min \left( \alpha_{x_i,y_j} + \text{OPT}(i-1, j-1), \delta + \text{OPT}(i-1, j), \delta + \text{OPT}(i, j-1) \right) \]

- Key observation: Computing entry \((i, j)\) requires values only in previous row/column or in previous row in current column.

---

Space-Efficient-Alignment\((X,Y)\)

Array \(B[0 \ldots m, 0 \ldots 1]\)

Initialize \(B[i,0] = i\delta\) for each \(i\) (just as in column 0 of \(A\))

For \(j = 1, \ldots, n\)

\[ B[0,1] = j\delta \] (since this corresponds to entry \(A[0,j]\))

For \(i = 1, \ldots, m\)

\[ B[i,1] = \min [\alpha_{x_i,y_j} + B[i-1,0], \delta + B[i-1,1], \delta + B[i,0]] \]

Endfor

Move column 1 of \(B\) to column 0 to make room for next iteration:

Update \(B[i,0] = B[i,1]\) for each \(i\)

Endfor

- Can compute \(\text{OPT}(m, n)\) in \(O(mn)\) time and \(O(m + n)\) space.
Improving the Running Time

\[ \text{OPT}(i, j) = \min \left( \alpha_{x_i y_j} + \text{OPT}(i - 1, j - 1), \delta + \text{OPT}(i - 1, j), \delta + \text{OPT}(i, j - 1) \right) \]

Key observation: Computing entry \((i, j)\) requires values only in previous row/column or in previous row in current column.

---

Space-Efficient-Alignment\((X,Y)\)

Array \(B[0...m, 0...1]\)

Initialize \(B[i, 0] = i\delta\) for each \(i\) (just as in column 0 of \(A\))

For \(j = 1, ..., n\)

\(B[0, 1] = j\delta\) (since this corresponds to entry \(A[0, j]\))

For \(i = 1, ..., m\)

\(B[i, 1] = \min [\alpha_{x_i y_j} + B[i - 1, 0], \delta + B[i - 1, 1], \delta + B[i, 0]]\)

Endfor

Move column 1 of \(B\) to column 0 to make room for next iteration:

Update \(B[i, 0] = B[i, 1]\) for each \(i\)

Endfor

---

Can compute \(\text{OPT}(m, n)\) in \(O(mn)\) time and \(O(m + n)\) space.

Problem: How do we compute matched pairs in the optimal alignment?
Improving the Running Time

\[ \text{OPT}(i, j) = \min (\alpha_{x_iy_j} + \text{OPT}(i - 1, j - 1), \delta + \text{OPT}(i - 1, j), \delta + \text{OPT}(i, j - 1)) \]

- Key observation: Computing entry \((i, j)\) requires values only in previous row/column or in previous row in current column.

---

**Space-Efficient-Alignment(X,Y)**

Array \(B[0 \ldots m, 0 \ldots 1]\)

Initialize \(B[i, 0] = i\delta\) for each \(i\) (just as in column 0 of \(A\))

For \(j = 1, \ldots, n\)

\(B[0, 1] = j\delta\) (since this corresponds to entry \(A[0, j]\))

For \(i = 1, \ldots, m\)

\(B[i, 1] = \min (\alpha_{x_iy_j} + B[i - 1, 0], \delta + B[i - 1, 1], \delta + B[i, 0])\)

Endfor

Move column 1 of \(B\) to column 0 to make room for next iteration:

Update \(B[i, 0] = B[i, 1]\) for each \(i\)

Endfor

---

- Can compute \(\text{OPT}(m, n)\) in \(O(mn)\) time and \(O(m + n)\) space.
- Problem: How do we compute matched pairs in the optimal alignment?

Requires new ideas: combine divide and conquer with dynamic programming!
Graph-theoretic View of Sequence Alignment

- **Grid graph** $G_{xy}$:
  - $m + 1$ rows numbered from 0 to $m$ (corresponding to $x$).
  - $n + 1$ rows numbered from 0 to $n$ (corresponding to $y$).
  - Rows labelled by symbols in $x$ and columns labelled by symbols in $y$.
  - Node $(i, j)$ has three outgoing edges to $(i, j + 1)$, to $(i + 1, j)$, and to $(i + 1, j + 1)$.
  - Edges directed upward and to the right have cost $\delta$.
  - Edge directed from $(i, j)$ to $(i + 1, j + 1)$ has cost $\alpha x_{i+1} y_{j+1}$.

Figure 6.17 A graph-based picture of sequence alignment.
For every $i, j$, $f(i, j) = \text{minimum cost of a path in } G_{XY} \text{ from } (0, 0) \text{ to } (i, j)$. 

**Figure 6.17** A graph-based picture of sequence alignment.
Shortest Paths in Grid Graphs

For every $i, j$, $f(i, j) =$ minimum cost of a path in $G_{XY}$ from $(0, 0)$ to $(i, j)$.

Claim: $f(i, j) = \text{OPT}(i, j)$. 

Figure 6.17 A graph-based picture of sequence alignment.
Shortest Paths in Grid Graphs

For every $i, j$, $f(i, j) = \text{minimum cost of a path in } G_{XY} \text{ from } (0, 0) \text{ to } (i, j)$.

Claim: $f(i, j) = \text{OPT}(i, j)$.

Proof by induction on $i + j$: Use the fact that the last edge on the shortest path to $(i, j)$ must be either from $(i - 1, j - 1)$, $(i - 1, j)$ or $(i, j - 1)$.

Figure 6.17 A graph-based picture of sequence alignment.
For every $i, j$, $f(i, j) =$ minimum cost of a path in $G_{XY}$ from $(0, 0)$ to $(i, j)$.

Claim: $f(i, j) = \text{OPT}(i, j)$.

Proof by induction on $i + j$: Use the fact that the last edge on the shortest path to $(i, j)$ must be either from $(i - 1, j - 1)$, $(i - 1, j)$ or $(i, j - 1)$.

Diagonal edges in the shortest path are the matched pairs in the alignment.
Shortest Paths in $G_{xy}$

- *Corner-to-corner path*: path from $(0,0)$ to $(n,m)$.
- Given $i$ and $j$, what is the length $l(i,j)$ of the shortest corner-to-corner path through $(i,j)$?
Shortest Paths in $G_{xy}$

- **Corner-to-corner path**: path from $(0,0)$ to $(n,m)$.
- Given $i$ and $j$, what is the length $l(i,j)$ of the shortest corner-to-corner path through $(i,j)$?
  - One segment is the shortest path from $(0,0)$ to $(i,j)$ with cost $f(i,j)$.
  - The other segment is the shortest path from $(i,j)$ to $(m,n)$ with some cost.

How can we compute the cost of this path?
Shortest Paths in $G_{xy}$

- **Corner-to-corner path**: path from $(0, 0)$ to $(n, m)$.
- Given $i$ and $j$, what is the length $l(i, j)$ of the shortest corner-to-corner path through $(i, j)$?
  - One segment is the shortest path from $(0, 0)$ to $(i, j)$ with cost $f(i, j)$.
  - The other segment is the shortest path from $(i, j)$ to $(m, n)$ with some cost. How can we compute the cost of this path?
  - Define $g(i, j)$ as cost of the shortest path from $(i, j)$ to $(m, n)$.

\[
g(i, j) = \min(\alpha x_{i+1} + \gamma y_{j+1} + \text{OPT}(i+1, j+1), \delta + \text{OPT}(i+1, j), \delta + \text{OPT}(i, j+1))
\]

We can compute $g(i, j)$ for every $i$ and $j$ in $O(mn)$ time and $O(m+n)$ space using Backward-Space-Efficient-Alignment.
Shortest Paths in $G_{xy}$

- **Corner-to-corner path**: path from $(0, 0)$ to $(n, m)$.
- Given $i$ and $j$, what is the length $l(i, j)$ of the shortest corner-to-corner path through $(i, j)$?
  - One segment is the shortest path from $(0, 0)$ to $(i, j)$ with cost $f(i, j)$.
  - The other segment is the shortest path from $(i, j)$ to $(m, n)$ with some cost.
  - How can we compute the cost of this path?
  - Define $g(i, j)$ as cost of the shortest path from $(i, j)$ to $(m, n)$.

$$g(i, j) = \min \left( \alpha x_{i+1} y_{j+1} + \text{OPT}(i+1, j+1), \delta + \text{OPT}(i+1, j), \delta + \text{OPT}(i, j+1) \right)$$
Shortest Paths in $G_{xy}$

- **Corner-to-corner path**: path from $(0, 0)$ to $(n, m)$.
- Given $i$ and $j$, what is the length $l(i, j)$ of the shortest corner-to-corner path through $(i, j)$?
  - One segment is the shortest path from $(0, 0)$ to $(i, j)$ with cost $f(i, j)$.
  - The other segment is the shortest path from $(i, j)$ to $(m, n)$ with some cost.
  - How can we compute the cost of this path?
  - Define $g(i, j)$ as cost of the shortest path from $(i, j)$ to $(m, n)$.

$$g(i, j) = \min \left( \alpha x_{i+1} y_{j+1} + \text{OPT}(i+1, j+1), \delta + \text{OPT}(i+1, j), \delta + \text{OPT}(i, j+1) \right)$$

- We can compute $g(i, j)$ for every $i$ and $j$ in $O(mn)$ time and $O(m + n)$ space using Backward-Space-Efficient-Alignment.
Shortest Path Through \((i, j)\) in \(G_{xy}\)

- Claim: \(l(i, j) = f(i, j) + g(i, j)\).
Shortest Path Through \((i,j)\) in \(G_{xy}\)

- Claim: \(l(i,j) = f(i,j) + g(i,j)\).
- Shortest corner-to-corner path through \((i,j)\) must go from \((0,0)\) to \((i,j)\) and then from \((i,j)\) to \((m,n)\).
Shortest Path Through \((i, j)\) in \(G_{xy}\)

- Claim: \(l(i, j) = f(i, j) + g(i, j)\).
- Shortest corner-to-corner path through \((i, j)\) must go from \((0, 0)\) to \((i, j)\) and then from \((i, j)\) to \((m, n)\).
- Therefore, \(l(i, j) \geq f(i, j) + g(i, j)\).
Shortest Path Through \((i, j)\) in \(G_{xy}\)

- Claim: \(l(i, j) = f(i, j) + g(i, j)\).
- Shortest corner-to-corner path through \((i, j)\) must go from \((0, 0)\) to \((i, j)\) and then from \((i, j)\) to \((m, n)\).
- Therefore, \(l(i, j) \geq f(i, j) + g(i, j)\).
- Now consider the following corner-to-corner path: Shortest path from \((0, 0)\) to \((i, j)\) followed by the shortest path from \((i, j)\) to \((m, n)\). What is its length?
Shortest Path Through \((i, j)\) in \(G_{xy}\)

- Claim: \(l(i, j) = f(i, j) + g(i, j)\).
- Shortest corner-to-corner path through \((i, j)\) must go from \((0, 0)\) to \((i, j)\) and then from \((i, j)\) to \((m, n)\).
- Therefore, \(l(i, j) \geq f(i, j) + g(i, j)\).
- Now consider the following corner-to-corner path: Shortest path from \((0, 0)\) to \((i, j)\) followed by the shortest path from \((i, j)\) to \((m, n)\). What is its length? \(f(i, j) + g(i, j)\).
Claim: \( l(i,j) = f(i,j) + g(i,j) \).

Shortest corner-to-corner path through \((i,j)\) must go from \((0,0)\) to \((i,j)\) and then from \((i,j)\) to \((m,n)\).

Therefore, \( l(i,j) \geq f(i,j) + g(i,j) \).

Now consider the following corner-to-corner path: Shortest path from \((0,0)\) to \((i,j)\) followed by the shortest path from \((i,j)\) to \((m,n)\). What is its length? \( f(i,j) + g(i,j) \).

Therefore, \( l(i,j) \leq f(i,j) + g(i,j) \).
Shortest Paths Through Column $k$ in $G_{xy}$

- Fix arbitrary $k$ between 0 and $n$.
- Does the shortest corner-to-corner path pass through a node in column $k$?

Let $l^*$ be the length of the shortest corner-to-corner path.

$$l^* \leq f(q_k, k) + g(q_k, k).$$

Why?
Shortest Paths Through Column $k$ in $G_{xy}$

- Fix arbitrary $k$ between 0 and $n$.
- Does the shortest corner-to-corner path pass through a node in column $k$?
  - Yes, it must pass through exactly one such node, say $(q_k, k)$.
  - How can we compute $q_k$ given values of $f(i,j)$ and $g(i,j)$ for every node $(i,j)$.
Shortest Paths Through Column $k$ in $G_{xy}$

- Fix arbitrary $k$ between 0 and $n$.
- Does the shortest corner-to-corner path pass through a node in column $k$?
  - Yes, it must pass through exactly one such node, say $(q_k, k)$.
  - How can we compute $q_k$ given values of $f(i, j)$ and $g(i, j)$ for every node $(i, j)$.

$$q_k = \arg \min_{0 \leq i \leq m} \ f(i, k) + g(i, k)$$
Shortest Paths Through Column $k$ in $G_{xy}$

- Fix arbitrary $k$ between 0 and $n$.
- Does the shortest corner-to-corner path pass through a node in column $k$?
  - Yes, it must pass through exactly one such node, say $(q_k, k)$.
  - How can we compute $q_k$ given values of $f(i, j)$ and $g(i, j)$ for every node $(i, j)$.

\[ q_k = \arg \min_{0 \leq i \leq m} f(i, k) + g(i, k) \]

- Why should there be a shortest corner-to-corner path that passes through node $(q_k, k)$? Proof is very similar to previous path.
Fix arbitrary $k$ between 0 and $n$.

Does the shortest corner-to-corner path pass through a node in column $k$?
  ▶ Yes, it must pass through exactly one such node, say $(q_k, k)$.
  ▶ How can we compute $q_k$ given values of $f(i, j)$ and $g(i, j)$ for every node $(i, j)$.

$$q_k = \arg \min_{0 \leq i \leq m} f(i, k) + g(i, k)$$

Why should there be a shortest corner-to-corner path that passes through node $(q_k, k)$? Proof is very similar to previous path.
  ▶ Let $l^*$ be the length of the shortest corner-to-corner path.
Fix arbitrary \( k \) between 0 and \( n \).

Does the shortest corner-to-corner path pass through a node in column \( k \)?

- Yes, it must pass through exactly one such node, say \((q_k, k)\).
- How can we compute \( q_k \) given values of \( f(i, j) \) and \( g(i, j) \) for every node \((i, j)\).

\[
q_k = \arg \min_{0 \leq i \leq m} f(i, k) + g(i, k)
\]

Why should there be a shortest corner-to-corner path that passes through node \((q_k, k)\)? Proof is very similar to previous path.

- Let \( l^* \) be the length of the shortest corner-to-corner path.
- \( l^* \leq f(q_k, k) + g(q_k, k) \). Why?
Shortest Paths Through Column $k$ in $G_{xy}$

- Fix arbitrary $k$ between 0 and $n$.
- Does the shortest corner-to-corner path pass through a node in column $k$?
  - Yes, it must pass through exactly one such node, say $(q_k, k)$.
  - How can we compute $q_k$ given values of $f(i, j)$ and $g(i, j)$ for every node $(i, j)$.
    \[
    q_k = \arg \min_{0 \leq i \leq m} f(i, k) + g(i, k)
    \]
- Why should there be a shortest corner-to-corner path that passes through node $(q_k, k)$? Proof is very similar to previous path.
  - Let $l^*$ be the length of the shortest corner-to-corner path.
  - $l^* \leq f(q_k, k) + g(q_k, k)$. Why?
  - Shortest corner-to-corner path must use some node $p$ in column $k$. Therefore, $l^* = f(p, k) + g(p, k) \geq \min_{0 \leq i \leq m} f(i, k) + g(i, k)$. 

Motivation

- **Computational finance:**
  - Each node is a financial agent.
  - The cost $c_{uv}$ of an edge $(u, v)$ is the cost of a transaction in which we buy from agent $u$ and sell to agent $v$.
  - Negative cost corresponds to a profit.

- **Internet routing protocols**
  - Dijkstra’s algorithm needs knowledge of the entire network.
  - Routers only know which other routers they are connected to.
  - Algorithm for shortest paths with negative edges is decentralised.
  - We will not study this algorithm in the class. See Chapter 6.9.
Problem Statement

- Input: a directed graph \( G = (V, E) \) with a cost function \( c : E \rightarrow \mathbb{R} \), i.e., \( c_{uv} \) is the cost of the edge \((u, v) \in E\).

- A **negative cycle** is a directed cycle whose edges have a total cost that is negative.

- Two related problems:
  1. If \( G \) has no negative cycles, find the **shortest s-t path**: a path of from source \( s \) to destination \( t \) with minimum total cost.
  2. Does \( G \) have a **negative cycle**?
Problem Statement

- Input: a directed graph $G = (V, E)$ with a cost function $c : E \to \mathbb{R}$, i.e., $c_{uv}$ is the cost of the edge $(u, v) \in E$.
- A **negative cycle** is a directed cycle whose edges have a total cost that is negative.
- Two related problems:
  1. If $G$ has no negative cycles, find the **shortest s-t path**: a path of from source $s$ to destination $t$ with minimum total cost.
  2. Does $G$ have a **negative cycle**?

![Graph Example]

**Figure 6.20** In this graph, one can find $s$-$t$ paths of arbitrarily negative cost (by going around the cycle $C$ many times).
Approaches for Shortest Path Algorithm

1. Dijsktra’s algorithm.

2. Add some large constant to each edge.
Approaches for Shortest Path Algorithm

1. Dijsktra’s algorithm. Computes incorrect answers because it is greedy.

2. Add some large constant to each edge. Computes incorrect answers because the minimum cost path changes.

Figure 6.21 (a) With negative edge costs, Dijkstra’s Algorithm can give the wrong answer for the Shortest-Path Problem. (b) Adding 3 to the cost of each edge will make all edges nonnegative, but it will change the identity of the shortest s-t path.
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is simple (does not repeat a node)
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is \textit{simple} (does not repeat a node) and hence has at most $n - 1$ edges.
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is simple (does not repeat a node) and hence has at most $n - 1$ edges.
- How do we define sub-problems?
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is simple (does not repeat a node) and hence has at most $n-1$ edges.

How do we define sub-problems?

- Shortest $s$-$t$ path has $\leq n-1$ edges: how we can reach $t$ using $i$ edges, for different values of $i$?
- We do not know which nodes will be in shortest $s$-$t$ path: how we can reach $t$ from each node in $V$?
Dynamic Programming Approach

- Assume $G$ has no negative cycles.
- Claim: There is a shortest path from $s$ to $t$ that is *simple* (does not repeat a node) and hence has at most $n - 1$ edges.

How do we define sub-problems?
- Shortest $s$-$t$ path has $\leq n - 1$ edges: how we can reach $t$ using $i$ edges, for different values of $i$?
- We do not know which nodes will be in shortest $s$-$t$ path: how we can reach $t$ from each node in $V$?

Sub-problems defined by varying the number of edges in the shortest path and by varying the starting node in the shortest path.
Dynamic Programming Recursion

- \( OPT(i, v) \): minimum cost of a \( v-t \) path that uses at most \( i \) edges.
- \( t \) is not explicitly mentioned in the sub-problems.
- Goal is to compute \( OPT(n - 1, s) \).
Dynamic Programming Recursion

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses at most $i$ edges.
- $t$ is not explicitly mentioned in the sub-problems.
- Goal is to compute $OPT(n - 1, s)$.

Let $P$ be the optimal path whose cost is $OPT(i, v)$.

![Diagram](image-url)

**Figure 6.22** The minimum-cost path $P$ from $v$ to $t$ using at most $i$ edges.
Dynamic Programming Recursion

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses at most $i$ edges.
- $t$ is not explicitly mentioned in the sub-problems.
- Goal is to compute $OPT(n - 1, s)$.

Let $P$ be the optimal path whose cost is $OPT(i, v)$.

1. If $P$ actually uses $i - 1$ edges, then $OPT(i, v) = OPT(i - 1, v)$.
2. If first node on $P$ is $w$, then $OPT(i, v) = c_{vw} + OPT(i - 1, w)$.
**Dynamic Programming Recursion**

- $OPT(i, v)$: minimum cost of a $v$-$t$ path that uses at most $i$ edges.
- $t$ is not explicitly mentioned in the sub-problems.
- Goal is to compute $OPT(n - 1, s)$.

Let $P$ be the optimal path whose cost is $OPT(i, v)$.

1. If $P$ actually uses $i - 1$ edges, then $OPT(i, v) = OPT(i - 1, v)$.
2. If first node on $P$ is $w$, then $OPT(i, v) = c_{vw} + OPT(i - 1, w)$.

$$OPT(i, v) = \min \left ( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right )$$

**Figure 6.22** The minimum-cost path $P$ from $v$ to $t$ using at most $i$ edges.
Example of Dynamic Programming Recursion

$$OPT(i, v) = \min \left( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right) \]
Example of Dynamic Programming Recursion

\[
\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i - 1, w) \right) \right)
\]
Example of Dynamic Programming Recursion

\[
\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)
\]
Example of Dynamic Programming Recursion

\[
OPT(i, v) = \min \left( \OPT(i - 1, v), \min_{w \in V} (c_{vw} + \OPT(i - 1, w)) \right)
\]
Example of Dynamic Programming Recursion

\[
\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)
\]
Example of Dynamic Programming Recursion

$$OPT(i, v) = \min \left( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right)$$

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>t</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>a</td>
<td>∞</td>
<td>-3</td>
<td>-3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>∞</td>
<td>∞</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>8</td>
<td>4</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e</td>
<td>8</td>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example of Dynamic Programming Recursion

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i-1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i-1, w) \right) \right) \]
Example of Dynamic Programming Recursion

$$OPT(i, v) = \min\left( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right)$$
Example of Dynamic Programming Recursion

\[
OPT(i, v) = \min \left( OPT(i - 1, v), \min_{w \in V} (c_{vw} + OPT(i - 1, w)) \right)
\]
Example of Dynamic Programming Recursion

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i-1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i-1, w) \right) \right) \]
Example of Dynamic Programming Recursion

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} \left(c_{vw} + \text{OPT}(i - 1, w)\right) \right)$$
Example of Dynamic Programming Recursion

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} \left( c_{vw} + \text{OPT}(i - 1, w) \right) \right) \]
Alternate Dynamic Programming Formulation

- $OPT_{\leq}(i, v)$: minimum cost of a $v-t$ path that uses exactly $i$ edges. Goal is to compute

- $OPT(i, v)$: minimum cost of a $v-t$ path that uses exactly $i$ edges. Goal is to compute
Alternate Dynamic Programming Formulation

- $OPT_\leftarrow(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute

$$\min_{i=1}^{n-1} OPT_\leftarrow(i, s).$$
Alternate Dynamic Programming Formulation

- $OPT_{\geq}(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute

\[
\min_{i=1}^{n-1} OPT_{\geq}(i, s).
\]

- Let $P$ be the optimal path whose cost is $OPT_{\geq}(i, v)$. 

T. M. Murali September 26, October 1, 3, 8, 2018 Dynamic Programming
Alternate Dynamic Programming Formulation

- $OPT_{=}(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute

$$\min_{i=1}^{n-1} OPT_{=}(i, s).$$

- Let $P$ be the optimal path whose cost is $OPT_{=}(i, v)$.
  - If first node on $P$ is $w$, then $OPT_{=}(i, v) = c_{vw} + OPT_{=}(i - 1, w)$. 
Alternate Dynamic Programming Formulation

- $OPT_{\leq}(i, v)$: minimum cost of a $v$-$t$ path that uses exactly $i$ edges. Goal is to compute
  $$n-1 \min_{i=1}^{n} OPT_{\leq}(i, s).$$

- Let $P$ be the optimal path whose cost is $OPT_{\leq}(i, v)$.
  - If first node on $P$ is $w$, then $OPT_{\leq}(i, v) = c_{vw} + OPT_{\leq}(i - 1, w)$.
    $$OPT_{\leq}(i, v) = \min_{w \in V} (c_{vw} + OPT_{\leq}(i - 1, w))$$
Alternate Dynamic Programming Formulation

- \( OPT_{\leq}(i, v) \): minimum cost of a \( v-t \) path that uses exactly \( i \) edges. Goal is to compute
  \[
  \min_{i=1}^{n-1} OPT_{\leq}(i, s).
  \]

- Let \( P \) be the optimal path whose cost is \( OPT_{\leq}(i, v) \).
  - If first node on \( P \) is \( w \), then \( OPT_{\leq}(i, v) = c_{vw} + OPT_{\leq}(i - 1, w) \).
    \[
    OPT_{\leq}(i, v) = \min_{w \in V} (c_{vw} + OPT_{\leq}(i - 1, w))
    \]

- Compare the two desired solutions:
  \[
  \min_{i=1}^{n-1} OPT_{\leq}(i, s) = \min_{i=1}^{n-1} \left( \min_{w \in V} (c_{sw} + OPT_{\leq}(i - 1, w)) \right)
  \]
  \[
  OPT(n - 1, s) = \min \left( OPT(n - 2, s), \min_{w \in V} (c_{sw} + OPT(n - 2, w)) \right)
  \]
Bellman-Ford Algorithm

$$\text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right)$$

---

Shortest-Path($G, s, t$)

- $n =$ number of nodes in $G$
- Array $M[0 \ldots n-1, V]$
- Define $M[0, t] = 0$ and $M[0, v] = \infty$ for all other $v \in V$
- For $i = 1, \ldots, n-1$
  - For $v \in V$ in any order
    - Compute $M[i, v]$ using the recurrence (6.23)
  - Endfor
- Endfor
- Return $M[n - 1, s]$
Bellman-Ford Algorithm

\[ \text{OPT}(i, v) = \min \left( \text{OPT}(i - 1, v), \min_{w \in V} (c_{vw} + \text{OPT}(i - 1, w)) \right) \]

---

Shortest-Path\((G, s, t)\)

1. \(n = \) number of nodes in \(G\)
2. Array \(M[0 \ldots n - 1, V]\)
3. Define \(M[0, t] = 0\) and \(M[0, v] = \infty\) for all other \(v \in V\)
4. For \(i = 1, \ldots, n - 1\)
   - For \(v \in V\) in any order
     - Compute \(M[i, v]\) using the recurrence (6.23)
   - Endfor
5. Endfor
6. Return \(M[n - 1, s]\)

- Space used is \(O(n^2)\). Running time is \(O(n^3)\).
- If shortest path uses \(k\) edges, we can recover it in \(O(kn)\) time by tracing back through smaller sub-problems.
An Improved Bound on the Running Time

Suppose $G$ has $n$ nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?
An Improved Bound on the Running Time

Suppose $G$ has $n$ nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min \left( M[i - 1, v], \min_{w \in V} (c_{vw} + M[i - 1, w]) \right)$$
An Improved Bound on the Running Time

Suppose $G$ has $n$ nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min \left( M[i - 1, v], \min_{w \in N_v} (c_{vw} + M[i - 1, w]) \right)$$

- $w$ only needs to range over outgoing neighbours $N_v$ of $v$.
- If $n_v = |N_v|$ is the number of outgoing neighbours of $v$, then in each round, we spend time equal to

$$\sum_{v \in V} n_v =$$
An Improved Bound on the Running Time

Suppose $G$ has $n$ nodes and $m \ll \binom{n}{2}$ edges. Can we demonstrate a better upper bound on the running time?

$$M[i, v] = \min \left( M[i - 1, v], \min_{w \in N_v} (c_{vw} + M[i - 1, w]) \right)$$

- $w$ only needs to range over outgoing neighbours $N_v$ of $v$.
- If $n_v = |N_v|$ is the number of outgoing neighbours of $v$, then in each round, we spend time equal to
  $$\sum_{v \in V} n_v = m.$$

- The total running time is $O(mn)$. 
Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in N_v} \left( c_{vw} + M[i - 1, w] \right) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in N_v} (c_{vw} + M[i - 1, w]) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
- Observe that \( M[i, v] \) depends only on \( M[i - 1, *] \) and no other indices.
Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in N_v} \left( c_{vw} + M[i - 1, w] \right) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
- Observe that \( M[i, v] \) depends only on \( M[i - 1, \ast] \) and no other indices.
- Modified algorithm:
  1. Maintain two arrays \( M \) and \( M' \) indexed over \( V \).
  2. At the beginning of each iteration, copy \( M \) into \( M' \).
  3. To update \( M \), use

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} \left( c_{vw} + M'[w] \right) \right) \]
Improving the Memory Requirements

\[ M[i, v] = \min \left( M[i - 1, v], \min_{w \in N_v} (c_{vw} + M[i - 1, w]) \right) \]

- The algorithm uses \( O(n^2) \) space to store the array \( M \).
- Observe that \( M[i, v] \) depends only on \( M[i - 1, *] \) and no other indices.
- Modified algorithm:
  1. Maintain two arrays \( M \) and \( M' \) indexed over \( V \).
  2. At the beginning of each iteration, copy \( M \) into \( M' \).
  3. To update \( M \), use

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]

- Claim: at the beginning of iteration \( i \), \( M \) stores values of \( \text{OPT}(i - 1, v) \) for all nodes \( v \in V \).
- Space used is \( O(n) \).
Computing the Shortest Path: Algorithm

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]

How can we recover the shortest path that has cost \( M[v] \)?
Computing the Shortest Path: Algorithm

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]

- How can we recover the shortest path that has cost \( M[v] \)?
- For each node \( v \), compute and update \( f(v) \), the first node after \( v \) in the current shortest path from \( v \) to \( t \).
- Updating \( f(v) \):
Computing the Shortest Path: Algorithm

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]

- How can we recover the shortest path that has cost \( M[v] \)?
- For each node \( v \), compute and update \( f(v) \), the first node after \( v \) in the current shortest path from \( v \) to \( t \).
- Updating \( f(v) \): If \( x \) is the node that attains the minimum in \( \min_{w \in N_v} (c_{vw} + M'[w]) \),
Computing the Shortest Path: Algorithm

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]

- How can we recover the shortest path that has cost \( M[v] \)?
- For each node \( v \), compute and update \( f(v) \), the first node after \( v \) in the current shortest path from \( v \) to \( t \).
- Updating \( f(v) \): If \( x \) is the node that attains the minimum in \( \min_{w \in N_v} (c_{vw} + M'[w]) \), set
  - \( M[v] = c_{vx} + M'[x] \) and
  - \( f(v) = x \).
- At the end, follow \( f(v) \) pointers from \( s \) to \( t \).
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
**Example of Maintaining Pointers**

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

$$M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right)$$

Graph and matrix representation:

- Graph with nodes a, b, c, d, e, and t, with edges and costs.
- Matrix showing transitions and costs between states.
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Example of Maintaining Pointers

\[ M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right) \]
Computing the Shortest Path: Correctness

- **Pointer graph** $P(V, F)$: each edge in $F$ is $(v, f(v))$.
  - Can $P$ have cycles?
  - Is there a path from $s$ to $t$ in $P$?
  - Can there be multiple paths $s$ to $t$ in $P$?
  - Which of these is the shortest path?

```
0  1  2  3  4  5
---  ---  ---  ---  ---  ---
t  0  0  0  0  0  0
a  8 -3 -3 -4 -6 -6
b  8  8  0 -2 -2 -2
c  8  3  3  3  3  3
d  8  4  3  3  2  0
e  ∞ 2  0  0  0  0
```
Computing the Shortest Path: Cycles in $P$

$$M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right)$$

- Claim: If $P$ has a cycle $C$, then $C$ has negative cost.
Computing the Shortest Path: Cycles in $P$

$$M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right)$$

Claim: If $P$ has a cycle $C$, then $C$ has negative cost.

- Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M[w]$.
- Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right)

Claim: If \( P \) has a cycle \( C \), then \( C \) has negative cost.

- Suppose we set \( f(v) = w \). At this instant, \( M[v] = c_{vw} + M[w] \).
- Between this assignment and the assignment of \( f(v) \) to some other node, \( M[w] \) may itself decrease. Hence, \( M[v] \geq c_{vw} + M[w] \), in general.
- Let \( v_1, v_2, \ldots v_k \) be the nodes in \( C \) and assume that \((v_k, v_1)\) is the last edge to have been added.
- What is the situation just before this addition?
Computing the Shortest Path: Cycles in $P$

$M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right)$

Claim: If $P$ has a cycle $C$, then $C$ has negative cost.

- Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M[w]$.
- Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
- Let $v_1, v_2, \ldots v_k$ be the nodes in $C$ and assume that $(v_k, v_1)$ is the last edge to have been added.
- What is the situation just before this addition?
- $M[v_i] - M[v_{i+1}] \geq c_{v_iv_{i+1}}$, for all $1 \leq i < k - 1$.
- $M[v_k] - M[v_1] > c_{v_kv_1}$.
Computing the Shortest Path: Cycles in $P$

$$M[v] = \min \left( M'[v], \min_{w \in N_v} (c_{vw} + M'[w]) \right)$$

**Claim:** If $P$ has a cycle $C$, then $C$ has negative cost.

- Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M[w]$.
- Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
- Let $v_1, v_2, \ldots, v_k$ be the nodes in $C$ and assume that $(v_k, v_1)$ is the last edge to have been added.
- What is the situation just before this addition?
  - $M[v_i] - M[v_{i+1}] \geq c_{v_iv_{i+1}}$, for all $1 \leq i < k - 1$.
  - $M[v_k] - M[v_1] > c_{v_kv_1}$.
- Adding all these inequalities, $0 > \sum_{i=1}^{k-1} c_{v_iv_{i+1}} + c_{v_kv_1} = \text{cost of } C$. 

Corollary: if $G$ has no negative cycles that $P$ does not either.
Computing the Shortest Path: Cycles in $P$

\[ M[v] = \min \left( M'[v], \min_{w \in \mathcal{N}_v} (c_{vw} + M'[w]) \right) \]

- **Claim:** If $P$ has a cycle $C$, then $C$ has negative cost.
  - Suppose we set $f(v) = w$. At this instant, $M[v] = c_{vw} + M[w]$.
  - Between this assignment and the assignment of $f(v)$ to some other node, $M[w]$ may itself decrease. Hence, $M[v] \geq c_{vw} + M[w]$, in general.
  - Let $v_1, v_2, \ldots v_k$ be the nodes in $C$ and assume that $(v_k, v_1)$ is the last edge to have been added.
  - What is the situation just before this addition?
  - $M[v_i] - M[v_{i+1}] \geq c_{v_iv_{i+1}}$, for all $1 \leq i < k - 1$.
  - $M[v_k] - M[v_1] > c_{v_kv_1}$.
  - Adding all these inequalities, $0 > \sum_{i=1}^{k-1} c_{v_iv_{i+1}} + c_{v_kv_1} = \text{cost of } C$.

- **Corollary:** if $G$ has no negative cycles that $P$ does not either.
Computing the Shortest Path: Paths in $P$

- Let $P$ be the pointer graph upon termination of the algorithm.
- Consider the path $P_v$ in $P$ obtained by following the pointers from $v$ to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.
Computing the Shortest Path: Paths in $P$

- Let $P$ be the pointer graph upon termination of the algorithm.
- Consider the path $P_v$ in $P$ obtained by following the pointers from $v$ to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.
- Claim: $P_v$ terminates at $t$. 
Let $P$ be the pointer graph upon termination of the algorithm.

Consider the path $P_v$ in $P$ obtained by following the pointers from $v$ to $f(v) = v_1$, to $f(v_1) = v_2$, and so on.

Claim: $P_v$ terminates at $t$.

Claim: $P_v$ is the shortest path in $G$ from $v$ to $t$. 
Bellman-Ford Algorithm: One Array

\[ M[v] = \min \left( M[v], \min_{w \in N_v} (c_{vw} + M[w]) \right) \]

- We can prove algorithm’s correctness in this case as well.
Bellman-Ford Algorithm: Early Termination

\[ M[v] = \min \left( M[v], \min_{w \in N_v} (c_{vw} + M[w]) \right) \]

In general, after \( i \) iterations, the path whose length is \( M[v] \) may have many more than \( i \) edges.
Bellman-Ford Algorithm: Early Termination

\[ M[v] = \min \left( M[v], \min_{w \in N_v} (c_{vw} + M[w]) \right) \]

- In general, after \( i \) iterations, the path whose length is \( M[v] \) may have many more than \( i \) edges.
- Early termination: If \( M \) does not change after processing all the nodes, we have computed all the shortest paths to \( t \).