Dynamic Programming

Jianlin Cheng, PhD

Computer Science Department

University of Missouri, Columbia

Fall, 2013

References

- Princeton’s class notes on dynamic programming
- Berkeley’s class notes on dynamic programming
- RPI’s class notes on dynamic programming

Dynamic Programming History

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

Dynamic programming = planning over time

Secretary of Defense was hostile to
mathematical research

Bellman sought an impressive name to avoid
confrontation

Algorithmic Paradigms

- **Greed**: build up a solution incrementally, myopically optimizing some local criterion (hill climbing)
- **Divide-and-conquer**: Break up a problem into two sub-problems, solve each-sub problem independently, and combine solutions to sub-problems to form solution to original problem (binary search)
- **Dynamic programming**: break up a problem into a series of *overlapping* sub-problems, and build up solutions to larger and larger sub-problems.

Dynamic Programming Applications

Areas

- Bioinformatics
- Control theory
- Information theory
- Operations research
- Computer science: theory, graphics, AI, systems…

Some famous dynamic programming algorithms

- Unix *diff* command for comparing two files
- Viterbi for hidden Markov models
- Smith-Waterman for sequence alignment
- Bellman-Ford for shortest path routing in networks
- Cocke-Kasami-Younger for parsing context free grammars

Dynamic Programming – A First Example

Fibonacci Numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …
- F(0) = 0, F(1) = 1
- F(n) = F(n-1) + F(n-2)

Dynamic Programming - A First Example

Fibonacci Numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …
- F(0) = 0, F(1) = 1
- F(n) = F(n-1) + F(n-2)

Computing the Fibonacci Numbers

- Each n^{th} number is a function of previous solutions
- A recursive solution:

```
Fib(n)
1. if n < 0 then RETURN "undefined"
2. if n ≤ 1 then RETURN n
3. RETURN Fib(n-1) + Fib(n-2)
```

What's the drawback to this solution?

Dynamic Programming - A First Example

Fibonacci Numbers

- 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- $F(0) = 0$, $F(1) = 1$
- $F(n) = F(n-1) + F(n-2)$

Computing the Fibonacci Numbers

- Each n^{th} number is a function of previous solutions
- A recursive solution:

```
Fib(n)
1. if n < 0 then RETURN "undefined"
2. if n ≤ 1 then RETURN n
3. RETURN Fib(n-1) + Fib(n-2)
```

What's the drawback to this solution?

- Complexity is exponential

Dynamic Programming - A First Example

Computing Fibonacci Numbers - Can we do better than exponential?

- Yes - "Memoization"
- Each time you encounter a new subproblem and compute the result, store it so that you never need to recompute that subproblem

- Each subproblem is computed just once, and is based on the results of smaller subproblems
  - This leads naturally to converting the recursive solution to an iterative solution

```
FibDynProg(n)
1. Fib[0] = 0
2. Fib[1] = 1
3. for i=2 to n do
4.    Fib[i] = Fib[i-1] + Fib[i-2]
5. RETURN Fib[n]
```

Knapsack Problem

Knapsack problem.

- Given n objects and a "knapsack."
- Item i weighs $w_i > 0$ kilograms and has value $v_i > 0$.
- Knapsack has capacity of W kilograms.
- Goal: fill knapsack so as to maximize total value.

Ex: { 3, 4 } has value 40.

W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Greedy: repeatedly add item with maximum ratio v_i / w_i.

Ex: { 5, 2, 1 } achieves only value = 35 $\Rightarrow$ greedy not optimal.

Dynamic Programming: False Start

Def. OPT(i) = max profit subset of items 1, …, i.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 }

- Case 2: OPT selects item i.
  - accepting item i does not immediately imply that we will have to reject other items
  - without knowing what other items were selected before i, we don't even know if we have enough room for i

Conclusion. Need more sub-problems!

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, ..., i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, ..., i-1 } using weight limit w

- Case 2: OPT selects item i.
  - new weight limit = $w - w_i$
  - OPT selects best of { 1, 2, ..., i-1 } using this new weight limit

$$OPT(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w_i > w \\ \max\{ OPT(i-1, w), \quad v_i + OPT(i-1, w - w_i) \} & \text{otherwise} \end{cases}$$

Dynamic Programming: Adding a New Variable

Def. OPT(i, w) = max profit subset of items 1, …, i with weight limit w.

- Case 1: OPT does not select item i.
  - OPT selects best of { 1, 2, …, i-1 } using weight limit w
- Case 2: OPT selects item i.
  - new weight limit = w – w_i
  - OPT selects best of { 1, 2, …, i-1 } using this new weight limit

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i) \,\} & \text{otherwise} \end{cases}$$

Solve OPT(i, w) for every i and w gradually, starting from lowest i and w Until reaching the largest i and w.

Knapsack Problem: Bottom-Up

Knapsack. Fill up an n-by-W array.

```
Input: n, w₁,…,w_N, v₁,…,v_N

for w = 0 to W
   M[0, w] = 0

for i = 1 to n
   for w = 1 to W
      if (w_i > w)
         M[i, w] = M[i-1, w]
      else
         M[i, w] = max {M[i-1, w], v_i + M[i-1, w-w_i ]}

return M[n, W]
```

Knapsack Algorithm

W + 1 →

n + 1 ↓

	0	1	2	3	4	5	6	7	8	9	10	11
ϕ	0	0	0	0	0	0	0	0	0	0	0	0
{ 1 }	0	1	1	1	1	1	1	1	1	1	1	1
{ 1, 2 }	0	1	6	7	7	7	7	7	7	7	7	7
{ 1, 2, 3 }	0	1	6	7	7	18	19	24	25	25	25	25
{ 1, 2, 3, 4 }	0	1	6	7	7	18	22	24	28	29	29	40
{ 1, 2, 3, 4, 5 }	0	1	6	7	7	18	22	28	29	34	34	40

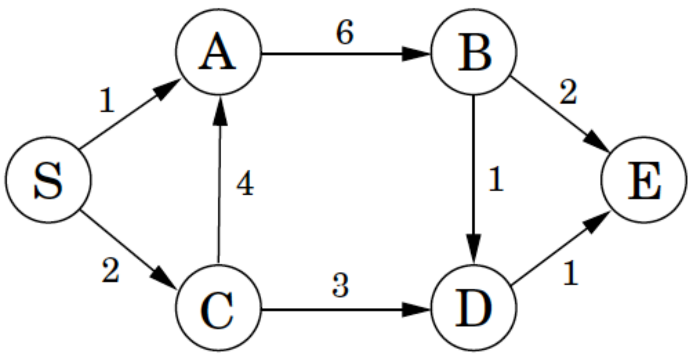
OPT: { 4, 3 }
value = 22 + 18 = 40

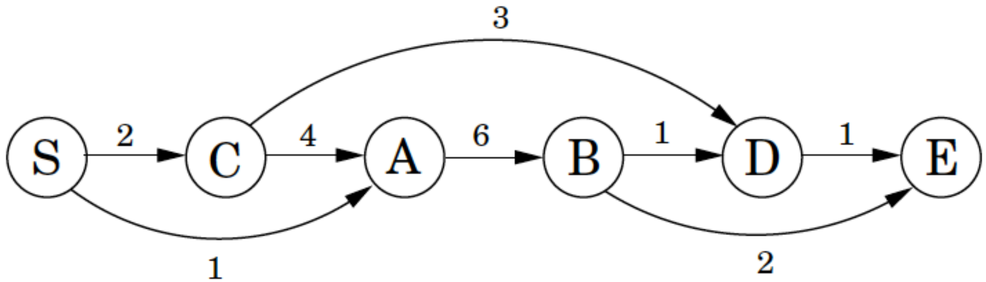
W = 11

Item	Value	Weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

Fill the matrix row by row

Shortest path from one node to all other nodes in a directed graph





A directed graph
Find shortest path from S to other nodes

Linearization of the graph: nodes are arranged on a line and all edges go from left to right.

Find shortest path to D

- dist(v): the distance of the shortest path to any node v.
- Find dist(D) assuming the shortest distances to all the nodes listed before D are known
- Then dist(D) = ?

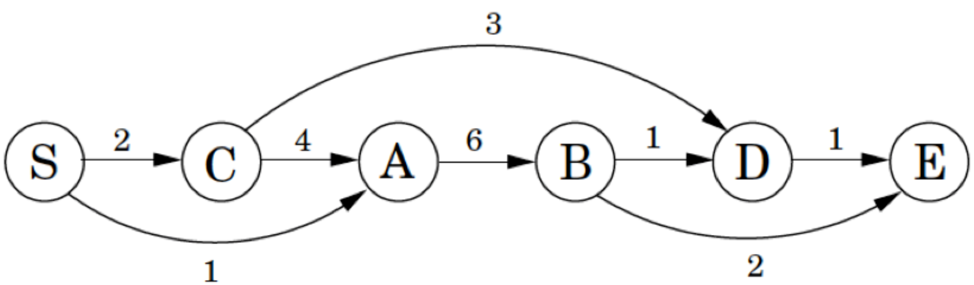
Find shortest path to D

- dist(v): the distance of the shortest path to any node v.
- Find dist(D) assuming the shortest distances to all the nodes listed before D are known
- Then dist(D) = min{dist(B) + 1, dist(C) + 3}.

Algorithm

initialize all dist$(\cdot)$ values to ∞
dist$(s) = 0$
for each $v \in V \backslash \{s\}$, in linearized order:
    dist$(v) = \min_{(u,v)\in E}\{$dist$(u) + l(u,v)\}$

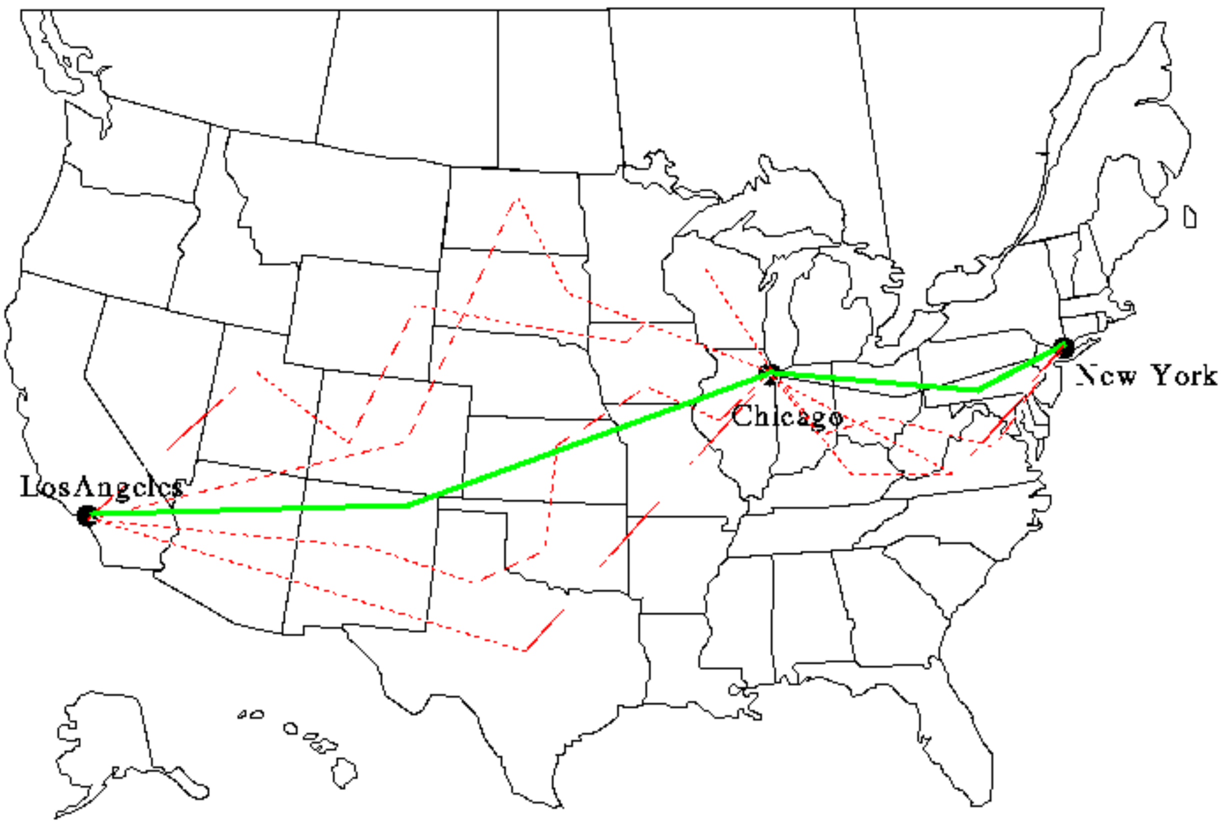
Example



Distance	S	C	A	B	D	E
Initialization	**0**	**∞**	**∞**	**∞**	**∞**	**∞**
		2				
			1			
				6		
					5	
						6

Application

TRIVIAL EXAMPLE OF BELLMAN'S OPTIMALITY PRINCIPLE



String Alignment

A natural measure of the distance between two strings is the extent to which they can be *aligned*, or matched up. Technically, an alignment is simply a way of writing the strings one above the other. For instance, here are two possible alignments of SNOWY and SUNNY:

S	–	N	O	W	Y
S	U	N	N	–	Y

Cost: 3

–	S	N	O	W	–	Y
S	U	N	–	–	N	Y

Cost: 5

Mismatch cost: 1; gap cost: 1

Widely used in bioinformatics, natural language processing, speech recognition

Edit Distance and Alignment

- The - indicates a gap; any number of these can be placed in either string. The cost of an alignment is the number of columns in which the letters differ.
- And the edit distance between two strings is the cost of their best possible alignment.
- Do you see that there is no better alignment of SNOWY and SUNNY than the one shown here with a cost of 3?

S	—	N	O	W	Y
S	U	N	N	—	Y

Meaning of Edit Distance

- Edit distance is so named because it can also be thought of as the minimum number of edits: **insertions**, **deletions**, and **substitutions** of characters needed to transform the first string into the second.
- For instance, the alignment shown on the left corresponds to three edits: insert U, substitute O -> N, and delete W.

S	—	N	O	W	Y
S	U	N	N	—	Y

Dynamic Programming

- When solving a problem by dynamic programming, the most crucial question is, **What are the subproblems?** As long as they are chosen so as to have the property as follows.
- There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to smaller subproblems, that is, subproblems that appear earlier in the ordering.
- it is an easy matter to write down the algorithm: iteratively solve one subproblem after the other, in order of increasing size.
- **Our goal is to find the shortest edit distance between two strings x[1,m] and y[1,n]. What is a good subproblem?**

Dynamic Programming

- How about looking at the edit distance between some prefix of the first string, x[1, i], and some prefix of the second, y[1, j]? Call this subproblem E(i; j) Our final objective, then, is to compute E(m; n).

The subproblem $E(7, 5)$.

E X P O N E N T I A L

P O L Y N O M I A L

For this to work, we need to somehow express $E(i,j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1\cdots i]$ and $y[1\cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{array}{c} x[i] \\ - \end{array} \quad \textbf{or} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \textbf{or} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

For this to work, we need to somehow express $E(i,j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1\cdots i]$ and $y[1\cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{matrix} x[i] \\ - \end{matrix} \quad \text{or} \quad \begin{matrix} - \\ y[j] \end{matrix} \quad \text{or} \quad \begin{matrix} x[i] \\ y[j] \end{matrix}$$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1\cdots i-1]$ with $y[1\cdots j]$. But this is exactly the subproblem $E(i-1,j)$! We seem to be getting somewhere. In the second case, also with cost 1, we still need to align $x[1\cdots i]$ with $y[1\cdots j-1]$. This is again another subproblem, $E(i,j-1)$. And in the final case, which either costs 1 (if $x[i]\neq y[j]$) or 0 (if $x[i]=y[j]$), what's left is the subproblem $E(i-1,j-1)$. In short, we have expressed $E(i,j)$ in terms of three *smaller* subproblems $E(i-1,j)$, $E(i,j-1)$, $E(i-1,j-1)$. We have no idea which of them is the right one, so we need to try them all and pick the best:

$$E(i,j) = \min\{1+E(i-1,j),\ 1+E(i,j-1),\ \text{diff}(i,j)+E(i-1,j-1)\}$$

where for convenience diff(i,j) is defined to be 0 if $x[i]=y[j]$ and 1 otherwise.

For this to work, we need to somehow express $E(i, j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1 \cdots i]$ and $y[1 \cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{array}{c} x[i] \\ - \end{array} \quad \text{or} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \text{or} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1 \cdots i-1]$ with $y[1 \cdots j]$. But this is exactly the subproblem $E(i-1, j)$! We seem to be getting somewhere. In the second case, also with cost 1, we still need to align $x[1 \cdots i]$ with $y[1 \cdots j-1]$. This is again another subproblem, $E(i, j-1)$. And in the final case, which either costs 1 (if $x[i] \neq y[j]$) or 0 (if $x[i] = y[j]$), what's left is the subproblem $E(i-1, j-1)$. In short, we have expressed $E(i, j)$ in terms of three *smaller* subproblems $E(i-1, j)$, $E(i, j-1)$, $E(i-1, j-1)$. We have no idea which of them is the right one, so we need to try them all and pick the best:

$$E(i, j) \;=\; \min\{1 + E(i-1, j),\ 1 + E(i, j-1),\ \text{diff}(i, j) + E(i-1, j-1)\}$$

where for convenience $\text{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

For this to work, we need to somehow express $E(i, j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1 \cdots i]$ and $y[1 \cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{array}{c} x[i] \\ - \end{array} \quad \text{or} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \text{or} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1 \cdots i-1]$ with $y[1 \cdots j]$. But this is exactly the subproblem $E(i-1, j)$! We seem to be getting somewhere. In the second case, also with cost 1, we still need to align $x[1 \cdots i]$ with $y[1 \cdots j-1]$. This is again another subproblem, $E(i, j-1)$. And in the final case, which either costs 1 (if $x[i] \neq y[j]$) or 0 (if $x[i] = y[j]$), what's left is the subproblem $E(i-1, j-1)$. In short, we have expressed $E(i, j)$ in terms of three *smaller* subproblems $E(i-1, j)$, $E(i, j-1)$, $E(i-1, j-1)$. We have no idea which of them is the right one, so we need to try them all and pick the best:

$$E(i, j) \;=\; \min\{1 + E(i-1, j),\; 1 + E(i, j-1),\; \text{diff}(i, j) + E(i-1, j-1)\}$$

where for convenience $\text{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

For this to work, we need to somehow express $E(i, j)$ in terms of smaller subproblems. Let's see—what do we know about the best alignment between $x[1 \cdots i]$ and $y[1 \cdots j]$? Well, its rightmost column can only be one of three things:

$$\begin{array}{c} x[i] \\ - \end{array} \quad \text{or} \quad \begin{array}{c} - \\ y[j] \end{array} \quad \text{or} \quad \begin{array}{c} x[i] \\ y[j] \end{array}$$

The first case incurs a cost of 1 for this particular column, and it remains to align $x[1 \cdots i-1]$ with $y[1 \cdots j]$. But this is exactly the subproblem $E(i-1, j)$! We seem to be getting somewhere. In the second case, also with cost 1, we still need to align $x[1 \cdots i]$ with $y[1 \cdots j-1]$. This is again another subproblem, $E(i, j-1)$. And in the final case, which either costs 1 (if $x[i] \neq y[j]$) or 0 (if $x[i] = y[j]$), what's left is the subproblem $E(i-1, j-1)$. In short, we have expressed $E(i, j)$ in terms of three *smaller* subproblems $E(i-1, j)$, $E(i, j-1)$, $E(i-1, j-1)$. We have no idea which of them is the right one, so we need to try them all and pick the best:

$$E(i, j) \quad = \quad \min\{1 + E(i-1, j),\ 1 + E(i, j-1),\ \text{diff}(i, j) + E(i-1, j-1)\}$$

where for convenience $\text{diff}(i, j)$ is defined to be 0 if $x[i] = y[j]$ and 1 otherwise.

An Example

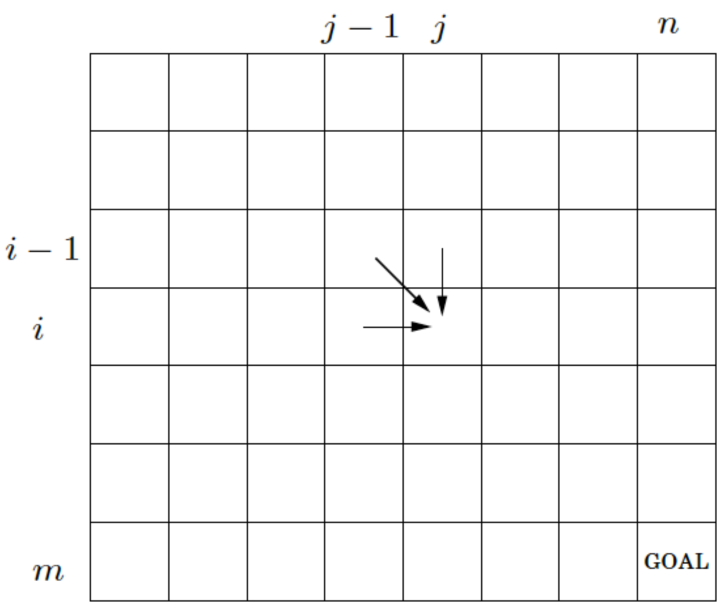
For instance, in computing the edit distance between EXPONENTIAL and POLYNOMIAL, subproblem $E(4,3)$ corresponds to the prefixes EXPO and POL. The rightmost column of their best alignment must be one of the following:

$$\begin{array}{c} \texttt{O} \\ - \end{array} \quad \text{or} \quad \begin{array}{c} - \\ \texttt{L} \end{array} \quad \text{or} \quad \begin{array}{c} \texttt{O} \\ \texttt{L} \end{array}$$

Thus, $E(4,3) = \min\{1 + E(3,3),\ 1 + E(4,2),\ 1 + E(3,2)\}$.

Figure 6.4 (a) The table of subproblems. Entries $E(i-1, j-1)$, $E(i-1, j)$, and $E(i, j-1)$ are needed to fill in $E(i, j)$. (b) The final table of values found by dynamic programming.

(a)

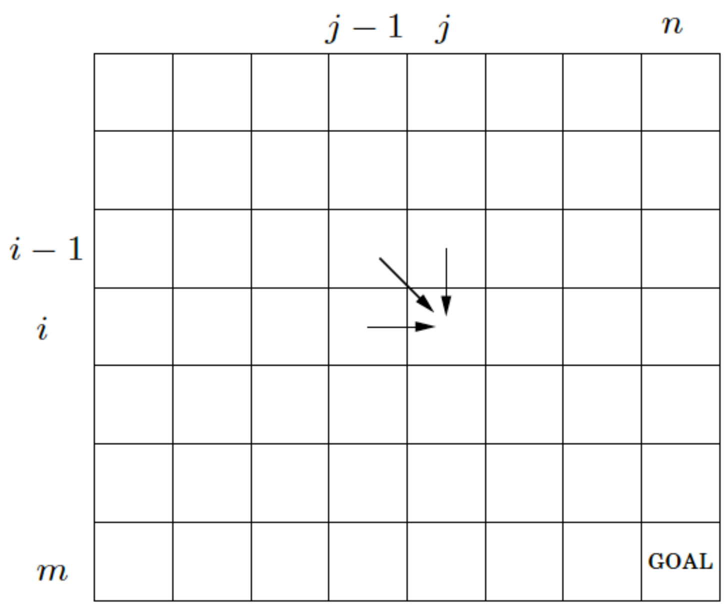


(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Figure 6.4 (a) The table of subproblems. Entries $E(i-1,j-1)$, $E(i-1,j)$, and $E(i,j-1)$ are needed to fill in $E(i,j)$. (b) The final table of values found by dynamic programming.

(a)



(b)

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

An Example

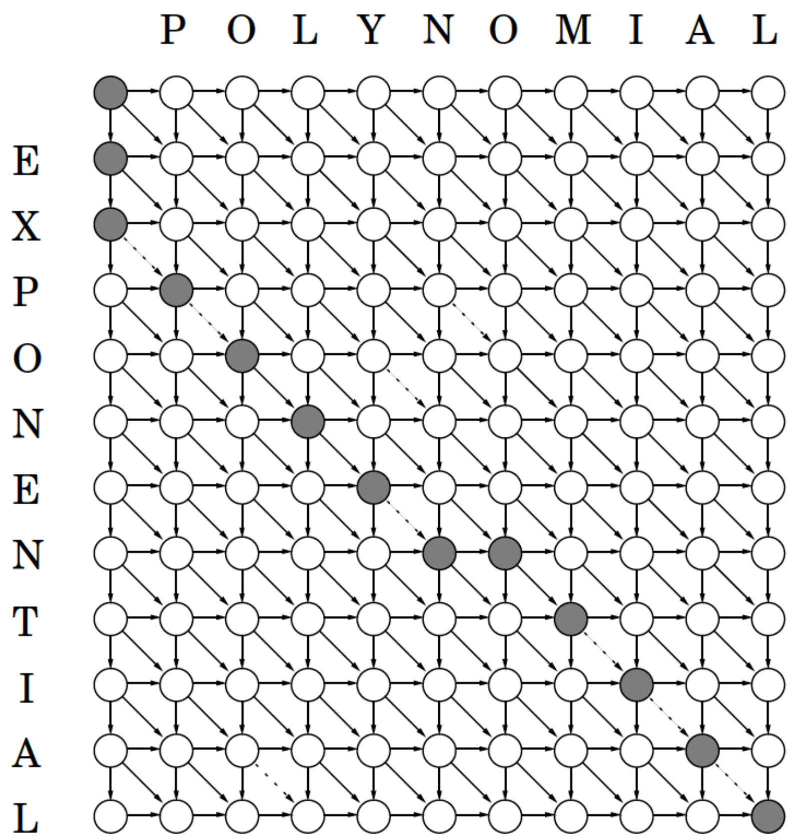
And in our example, the edit distance turns out to be 6:

E	X	P	O	N	E	N	—	T	I	A	L
—	—	P	O	L	Y	N	O	M	I	A	L

Underlying DAG

- Every dynamic program has an underlying dag structure: think of **each node as representing a subproblem**, and **each edge as a precedence constraint on the order in which the subproblems can be tackled**.
- Having nodes $u_1; : : : ; u_k$ point to v means. subproblem v can only be solved once the answers to $u_1; : : : ; u_k$ are known..

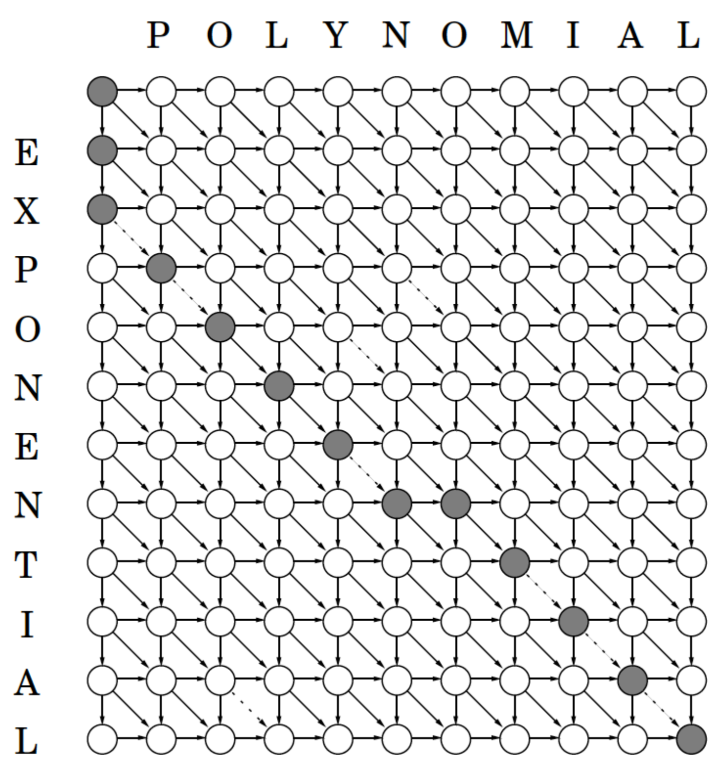
The underlying dag, and a path of length 6.



		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

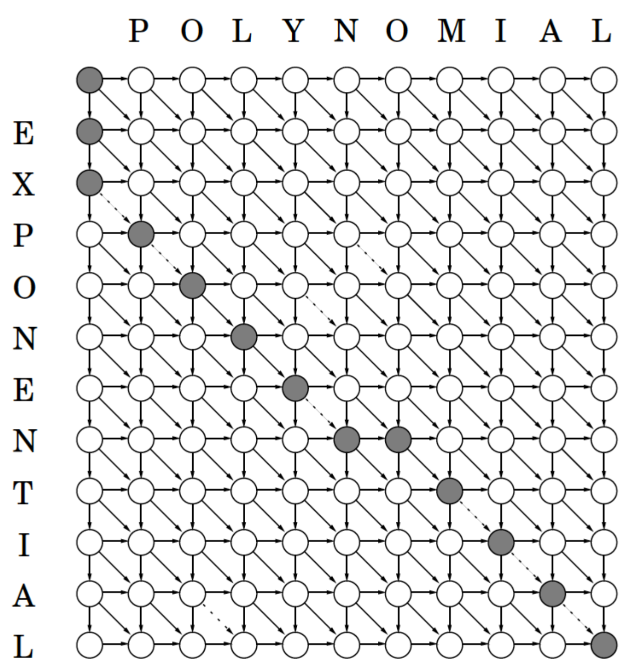
- In our present edit distance application, the nodes of the underlying dag correspond to subproblems, or equivalently, to positions (i; j) in the table. Its edges are the precedence constraints, of the form (i-1; j) -> (i; j), (i; j -1) -> (i; j), and (i-1; j-1) -> (i; j)

The underlying dag, and a path of length 6.



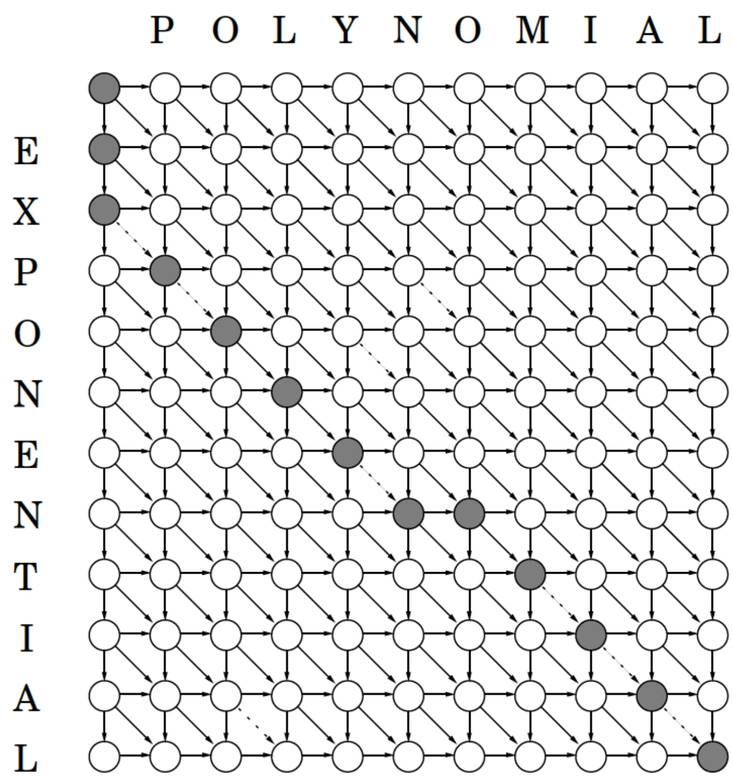
- In fact, we can take things a little further and put weights on the edges so that the edit distances are given by shortest paths in the dag!
- To see this, set all edge lengths to 1, except for (i - 1; j - 1) -> (i; j) : x[i] = y[j] (shown dotted in the figure), whose length is 0.

The underlying dag, and a path of length 6.



- The final answer is then simply the distance between nodes s = (0; 0) and t = (m; n).
- One possible shortest path is shown, the one that yields the alignment we found earlier.

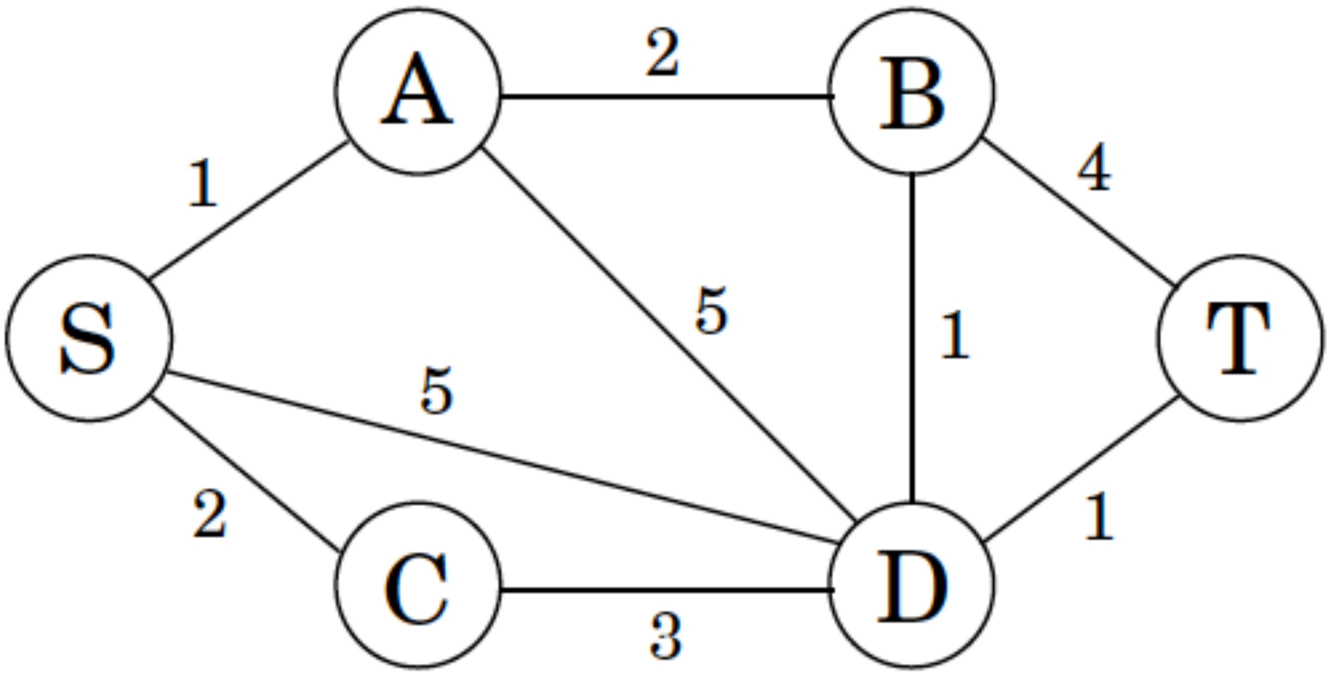
The underlying dag, and a path of length 6.



More Advanced Shortest Path

- Suppose then that we are given a graph G with lengths on the edges, along with two nodes **s** and **t** and an integer **k**, and we want the shortest path from **s** to **t** that uses at most **k** edges.

Find shortest path from S to T with at most 3 edges, 4 edges?



Updating Rule

In dynamic programming, the trick is to choose subproblems so that all vital information is remembered and carried forward. In this case, let us define, for each vertex v and each integer $i < k$, $\texttt{dist}(v, i)$ to be *the length of the shortest path from s to v that uses i edges.* The starting values $\texttt{dist}(v, 0)$ are ∞ for all vertices except s, for which it is 0. And the general update equation is, naturally enough,

$$\texttt{dist}(v, i) = \min_{(u,v) \in E} \{\texttt{dist}(u, i-1) + \ell(u, v)\}.$$

How to implement it?

Updating Matrix

	0	1	2	….	K-1	K
S						
U1						
U2						
….						
…						
V						

Shortest Paths Between All Pair of Nodes

One idea comes to mind: the shortest path $u \to w_1 \to \cdots \to w_l \to v$ between u and v uses some number of intermediate nodes—possibly none. Suppose we disallow intermediate nodes altogether. Then we can solve all-pairs shortest paths at once: the shortest path from u to v is simply the direct edge (u, v), if it exists. What if we now gradually expand the *set of permissible intermediate nodes*? We can do this one node at a time, updating the shortest path lengths at each stage. Eventually this set grows to all of V, at which point all vertices are allowed to be on all paths, and we have found the true shortest paths between vertices of the graph!

More concretely, number the vertices in V as $\{1, 2, \ldots, n\}$, and let $\texttt{dist}(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates. Initially, $\texttt{dist}(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and is ∞ otherwise.

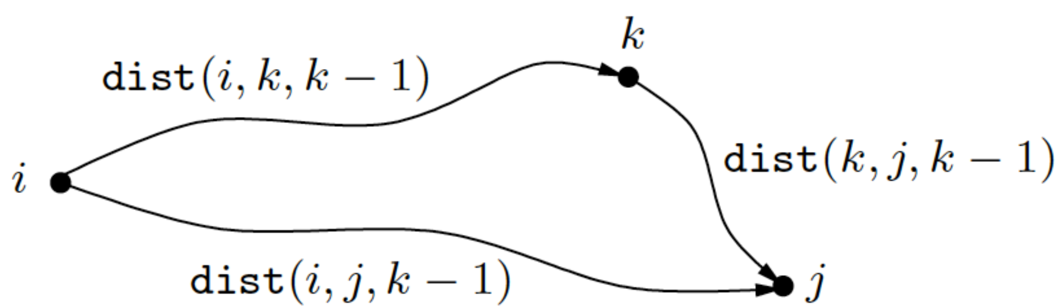
Shortest Paths Between All Pair of Nodes

One idea comes to mind: the shortest path $u \to w_1 \to \cdots \to w_l \to v$ between u and v uses some number of intermediate nodes—possibly none. Suppose we disallow intermediate nodes altogether. Then we can solve all-pairs shortest paths at once: the shortest path from u to v is simply the direct edge (u, v), if it exists. What if we now gradually expand the *set of permissible intermediate nodes*? We can do this one node at a time, updating the shortest path lengths at each stage. Eventually this set grows to all of V, at which point all vertices are allowed to be on all paths, and we have found the true shortest paths between vertices of the graph!

More concretely, number the vertices in V as $\{1, 2, \ldots, n\}$, and let $\texttt{dist}(i, j, k)$ denote the length of the shortest path from i to j in which only nodes $\{1, 2, \ldots, k\}$ can be used as intermediates. Initially, $\texttt{dist}(i, j, 0)$ is the length of the direct edge between i and j, if it exists, and is ∞ otherwise.

Subproblem and Updating Rule

What happens when we expand the intermediate set to include an extra node k? We must reexamine all pairs i, j and check whether using k as an intermediate point gives us a shorter path from i to j. But this is easy: a shortest path from i to j that uses k along with possibly other lower-numbered intermediate nodes goes through k just once (why? because we assume that there are no negative cycles). And we have already calculated the length of the shortest path from i to k and from k to j using only lower-numbered vertices:



Thus, using k gives us a shorter path from i to j if and only if

$$\texttt{dist}(i,k,k-1) + \texttt{dist}(k,j,k-1) \; < \; \texttt{dist}(i,j,k-1),$$

in which case $\texttt{dist}(i,j,k)$ should be updated accordingly.

Floyd-Warshall Algorithm

Here is the Floyd-Warshall algorithm—and as you can see, it takes $O(|V|^3)$ time.

```
for i = 1 to n:
    for j = 1 to n:
        dist(i, j, 0) = ∞

for all (i, j) ∈ E:
    dist(i, j, 0) = ℓ(i, j)
for k = 1 to n:
    for i = 1 to n:
        for j = 1 to n:
            dist(i, j, k) = min{dist(i, k, k − 1) + dist(k, j, k − 1),  dist(i, j, k − 1)}
```