CSE 317: Design and Analysis of Algorithms
Dynamic Programming

Shahid Hussain

Fall 2025

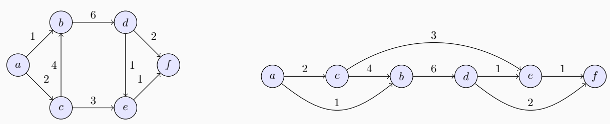
SMCS, IBA

Dynamic Programming

- The idea of dynamic programming is the following
- For a given problem, we define the notion of a subproblem and an ordering of subproblems from *"smallest"* to *"largest"*.
- (i) the number of subproblems is polynomial, and
- (ii) the solution of a subproblem can be easily (in polynomial time) computed from the solution of smaller subproblems
- If (i) and (ii) then we can design a polynomial algorithm for the initial problem

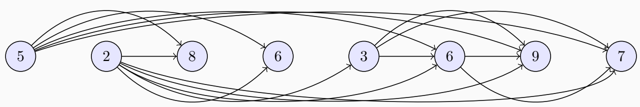
Shortest Paths in DAGS

- A graph G is called a *directed acyclic graph* (DAG) if it has no directed cycles
- For a given DAG G we can perform a topological sort of the vertices of G (linearization of G)



Longest Increasing Subsequences (LIS)

- Given a sequence of numbers $\langle a_1, a_2, \ldots, a_n \rangle$
- A *subsequence* is a sequence of numbers $\langle a_{i_1}, a_{i_2}, \ldots, a_{i_k} \rangle$ where $1 \leq i_1 < i_2 < \cdots < i_k \leq n$
- A subsequence is *increasing subsequence* if $a_{i_1} < a_{i_2} < \cdots < a_{i_k}$
- For example, given sequence $\langle 5, 2, 8, 6, 3, 6, 9, 7 \rangle$, the LIS is $\langle 2, 3, 6, 9 \rangle$
- Following graph $G = (V, E)$ is a DAG



Longest Increasing Subsequences (LIS)

Algorithm: LIS

Input: A sequence of numbers $\langle a_1, a_2, \ldots, a_n \rangle$

Output: The length of the longest increasing subsequence

$G(V, E) \leftarrow \text{CREATE-DAG}(a_1, a_2, \ldots, a_n)$ *// G is the DAG for the input sequence*

for $j \in \{1, 2, \ldots, n\}$

    $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$

return $\max\{L(j) : j \in \{1, 2, \ldots, n\}\}$

Longest Common Subsequence

- Let Σ be some fixed and finite alphabet
- A *string* X over Σ is a sequence of symbols from Σ i.e., $X = x_1x_2 \ldots x_n$, $n \geq 0$
- Let X be a sequence of length n over Σ
- We say that a *subsequence* of length k of X is a sequence $x_{i_1}x_{i_2} \ldots x_{i_k}$ such that $i_1 < i_2 < \cdots < i_k$
- Given two sequences X and Y lengths n and m, respectively
- A *common subsequence* of sequences X and Y is any sequence such that is common to both X and Y
- The problem *longest common subsequence* is to find such a common subsequence of maximum length

Longest Common Subsequence: Bruteforce approach

- A brute-force approach to solve such a problem would require to enumerate all subsequences of X and check if they are also common to Y
- We can see that there are $\Theta(2^n)$ subsequences of the sequence X and we can check whether it is also a subsequence in Y in linear time i.e, $\Theta(m)$
- Hence, brute-force algorithm would require $\Theta(m \cdot 2^n)$ time.

Longest Common Subsequence: Dynamic Programming

- We can solve this problem very efficiently using *dynamic programming*
- We can define the problem of finding the longest common subequence of X and Y of lengths n and m, respectively, as $\text{LCS}(n, m)$
- We can define a subproblem as $\text{LCS}(i, j)$ which finds the longest common subsequence of sequences ending at x_i and y_j i.e., between $x_1x_2 \ldots x_i$ and $y_1y_2 \ldots y_j$, for $0 \leq i \leq n$ and $0 \leq j \leq m$
- It is clear that $i = 0$ or $j = 0$ represent an empty sequence (correspondingly)
- So, we know that $\text{LCS}(i, 0) = 0$ as well as $\text{LCS}(0, j) = 0$ for all i and j

Longest Common Subsequence: Dynamic Programming

- We observe that there are two possible situations when comparing $x_1x_2\dots x_i$ and $y_1y_2\dots y_j$, that is either $x_i = y_j$ or $x_i \neq y_j$
- When $x_i = y_j$ these two characters must be included in any common subsequences that we may have found between $x_1x_2\dots x_{i-1}$ and $y_1y_2\dots y_{j-1}$, otherwise we need to check the longest common subsequences between $x_1x_2\dots x_{i-1}$ and $y_1y_2\dots y_j$ and between $x_1x_2\dots x_i$ and $y_1y_2\dots y_{j-1}$
- Therefore,

$$\mathrm{LCS}(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0, \\ \mathrm{LCS}(i-1,j-1)+1 & \text{if } i > 0 \text{ and } j > 0 \text{ and } a_i = b_j, \\ \max\{\mathrm{LCS}(i-1,j), \mathrm{LCS}(i,j-1)\} & \text{if } i > 0 \text{ and } j > 0 \text{ and } a_i \neq b_j. \end{cases}$$

Longest Common Subsequence: Dynamic Programming

Algorithm: TCS

Input: Two sequences X and Y of lengths n and m, respectively

Output: The length of the longest common subsequence of X and Y

```
for i = 0 to n: L(i,0) = 0
for j = 0 to m: L(0,j) = 0
for i = 1 to n
    for j = 1 to m
        if a_i = b_j then L(i,j) = L(i-1,j-1) + 1
        else L(i,j) = max{L(i,j-1), L(i-1,j)}
return L(n,m)
```


Longest Common Subsequence: Dynamic Programming

Theorem

An optimal solution to the longest common subsequence problem can be found in $\Theta(nm)$ time and $\Theta(\min\{n,m\})$ space.

Longest Common Subsequence: Example

- Let $X = \texttt{ABCBDAB}$ and $Y = \texttt{BDCABAB}$
- The subsequence BCBAB is common to both
- We can create following table:

	–	B	D	C	A	B	A	B
–	0	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1	1
B	0	1	1	1	1	2	2	2
C	0	1	1	2	2	2	2	2
B	0	1	1	2	2	3	3	3
D	0	1	2	2	2	3	3	3
A	0	1	2	2	3	3	4	4
B	0	1	2	2	3	4	4	**5**

Matrix Chain Multiplication

- Let us consider matrices, A, B, and C of dimensions $n \times 1$, $1 \times n$, and $n \times n$, respectively
- The dimensions of the product AB is $n \times n$
- So, the product $(AB)C$ requires $n^2 + n^3$ multiplications
- The dimensions of the product BC is $1 \times n$
- Therefore, the product $A(BC)$ requires $n^2 + n^2 = 2n^2$ multiplications
- Which clearly means computing $(AB)C$ is more expensive than $A(BC)$

Matrix Chain Multiplication

- Let us consider matrices $A_1, A_2, \ldots A_n$ with dimensions, $m_0 \times m_1$, $m_1 \times m_2$, $\ldots$, $m_{n-1} \times m_n$, respectively
- The problem is to find the *optimal parenthesization* of the product $A_1 A_2 \cdots A_n$ that minimizes the number of scalar multiplications
- The brute-force approach would require to enumerate all possible parenthesizations and compute the number of scalar multiplications
- For n matrices, there are 2^{n-1} possible parenthesizations
- We will use *dynamic programming* to solve this problem

Matrix Chain Multiplication: Dyanmic Programming

- Let us define the subproblem as follows
- Let $A_i A_{i+1} \cdots A_j$ be a subsequence of matrices $A_i A_{i+1} \cdots A_j$
- Let $B(i, j)$ denotes the subproblem of multiplying the matrices $A_i A_{i+1} \cdots A_j$ for $1 \leq i \leq j \leq n$
- The problem $B(1, n)$ represents the original problem
- Let us denote $C(i, j)$ as the number of scalar multiplications required to compute the matrix $A_i A_{i+1} \cdots A_j$

Matrix Chain Multiplication: Dyanmic Programming

- We can see that $C(i, j) = 0$ if $i = j$
- For $j > i$, we can see that

$$C(i, j) = \min_{i \leq k < j} \{C(i, k) + C(k+1, j) + m_{i-1} \cdot m_k \cdot m_j\}$$

- If the last operation of matrix multiplication divides the product $A_i A_{i+1} \cdots A_j$ into two subproducts $(A_i A_{i+1} \cdots A_k)(A_{k+1} A_{k+2} \cdots A_j)$ then to obtain the minimum number of element multiplications in the both subproducts and $m_{i-1} \cdot m_k \cdots m_j$ element multiplications to multiply the two subproducts
- We choose the k for which it minimizes the number of scalar multiplications
- Since there are $O(n^2)$ subproblems and each require $O(n)$ time to solve, the total time complexity is $O(n^3)$

Edit Distance

- Suppose we two strings X and Y over some fixed and finite alphabet Σ
- A natural measure of a distance between these strings is the degree to which they can be aligned, or matched up
- An alignment is a way of writing the strings one above the other.
- Let us consider two possible alignments of strings SNOWY and SUNNY.
- The symbol ''_'' indicates a "gap". Any number of gaps can be added to each string
- The cost of an alignment is the number of columns in which the letters differ. The edit distance between two strings is the cost of the best alignment

```
S _ N O W Y      _ S N O W _ Y
S U N N _ Y      S U N _ _ N Y
```


Edit Distance

- In the first alignment the cost is equal to 3
- And in the second alignment the cost is equal to 5
- In other words, the edit distance is the minimum number of *insertions*, *deletions* and *substitutions* of characters (letters) needed to transform the first string into the second one
- In the first example we insert U, substitute O $\to$ N and delete W

Edit Distance

- Let $X = x_1, \ldots, x_m$ and $Y = y_1, \ldots, y_n$
- For each $i \in \{0, 1, \ldots, m\}$ and $j \in \{0, 1, \ldots, n\}$, we consider the problem of finding the edit distance between $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$, and denote by $E(i, j)$ the considered distance
- If $i = 0$, the word $x_1, \ldots, x_i$ is the empty word ϵ. The same situation is for the case $j = 0$. It is clear that $E(i, 0) = i$ and $E(0, j) = j$ since the edit distance between the empty word ϵ and a nonempty word α of the length t is equal to t
- Let us consider the best alignment for $x_1, \ldots, x_i$ and $y_1, \ldots, y_j$ where $i > 0$ and $j > 0$. It is clear that in the rightmost column we can have one of the following three things:

$$\begin{array}{ccc} x_i & \_ & x_i \\ \_ & y_j & y_j \end{array}$$

Edit Distance

- In the first case, $E(i, j) = 1 + E(i - 1, j)$. In the second case, $E(i, j) = 1 + E(i, j - 1)$, and in the third case, $E(i, j) = \mathrm{diff}(i, j) + E(i - 1, j - 1)$, where $\mathrm{diff}(i, j) = 0$ if $x_i = y_j$ and $\mathrm{diff}(i, j) = 1$ if $x_i \neq y_j$. Therefore,

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

- We have $(m + 1) \times (n + 1)$ subproblems. If we know $E(i - 1, j)$, $E(i, j - 1)$, and $E(i - 1, j - 1)$ then to compute the value $E(i, j)$ it is necessary to make 3 operations of comparisons of numbers (1 to find the value $\mathrm{diff}(i, j)$, and 2 to find $\min$) and 3 operations of addition
- So, the considered algorithm makes $O(mn)$ operations of addition and comparison of numbers

Edit Distance

- To find the value $E(m,n)$ we should fill the table with $m+1$ rows labeled with numbers $0,1,\ldots,m$, and $n+1$ columns labeled with numbers $0,1,\ldots n$
- At the intersection of i-th row and j-th column we should have the number $E(i,j)$
- At the beginning, we can fill values $E(i,0)=i$ and $E(0,j)=j$, and after that row by row, from the left to the right we can fill out the table
- Note that it is not necessary to have the whole table in the memory: to fill the i-th row, $i>0$, it is enough to know values in the row $i-1$

Edit Distance

- Now we can find the optimal alignment if the considered table is filled
- If $E(m,n)=1+E(m-1,n)$, then in the optimal alignment the last column is $\begin{smallmatrix} x_m \\ - \end{smallmatrix}$
- If $E(m,n)=1+E(m,n-1)$, then in the optimal alignment the last column is $\begin{smallmatrix} - \\ y_n \end{smallmatrix}$
- If $E(m,n)=\mathrm{diff}(m,n)+E(m-1,n-1)$, then in the optimal alignment the last column is $\begin{smallmatrix} x_m \\ y_n \end{smallmatrix}$
- To find the next column we should consider: In the first case—the subproblem $E(m-1,n)$. In the second case—the subproblem $E(m,n-1)$ and in the third case - the subproblem $E(m-1,n-1)$, etc

Edit Distance: Example

		S	U	N	N	Y
	0	1	2	3	4	5
S	1	0	1	2	3	4
N	2	1	1	1	2	3
O	3	2	2	2	2	3
W	4	3	3	3	3	3
Y	5	4	4	4	4	3

x_i y_j		x_i _
	$i-1,j-1$	$i-1,j$
_ y_j	$i,j-1$ $i,j-1$	i,j i,j

```
S N O W Y   S _ N O W Y   S _ N O _ W Y
S U N N Y   S U N N _ Y   S U N _ N Y
```


Shortest Path: Floyd-Warshall Algorithm

- Let G be a complete directed graph with n vertices $v_1,v_2,\ldots,v_n$
- Each edge (v_i,v_j) has a label $d_{ij}\in\mathbb{R}\cup\{+\infty\}$
- We say that d_{ij} is the *length* of the edge (v_i,v_j). Clearly, $d_{ii}=0$ for $i=1,\ldots,n$
- The length of a directed path from v_i to v_j is equal to $+\infty$ if the length of at least one edge in the path is equal to $+\infty$
- It is possible to have negative lengths i.e., $d_{ij}<0$ however there is no directed cycle with negative length (no negative cycles)

Shortest Path: Floyd-Warshall Algorithm

- The minimum distance d_{ij}^* from v_i to v_j is equal to the minimum length of a directed path from v_i to v_j
- For a given $n \times n$ matrix $\mathbf{D} = [d_{ij}]$ of lengths of edges, we should construct the matrix $\mathbf{D}^* = [d_{ij}^*]$ of minimal distances
- For any $i, j \in \{1, \ldots, n\}$ and $k \in \{0, 1, \ldots, n\}$ let us consider the following subproblem to compute $d_{ij}^{(k)}$ which is the length of the shortest path from v_i to v_j in which only vertices $v_1, \ldots, v_k$ can be used as intermediate vertices. If $k = 0$ then $d_{ij}^{(0)} = d_{ij}$

Shortest Path: Floyd-Warshall Algorithm

- Let $\mathbf{D}^{(k)} = [d_{ij}^{(k)}]$. Then $\mathbf{D}^{(0)} = \mathbf{D}$ and $\mathbf{D}^{(n)} = \mathbf{D}^*$, we will sequentially compute $\mathbf{D}^{(1)}$, $\mathbf{D}^{(2)}$, ..., $\mathbf{D}^{(n)}$. Let us prove that for every i, j and $k > 0$

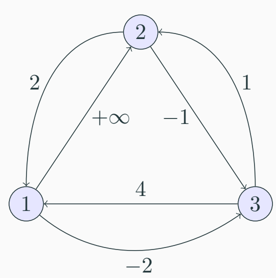
$$d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}.$$

- All directed paths from v_i to v_j that use only $v_1, \ldots, v_k$ as intermediate vertices can be divided into two sets A and B which do not pass through v_k and which pass through v_k, respectively
- The minimum length of path from A is equal to $d_{ij}^{(k-1)}$. Since there are no negative cycles in G, there exists a shortest path τ from B which passes through v_k exactly once.

Shortest Path: Floyd-Warshall Algorithm

- So τ can be divided into two paths: a path from v_i to v_k which use only vertices $v_1, \ldots, v_{k-1}$ (the minimum length of such a path is equal to $d_{ik}^{(k-1)}$ and a path from v_k to v_j which uses only vertices $v_1, \ldots, v_{k-1}$ (the minimum length of such a path is equal to $d_{kj}^{(k-1)}$
- Therefore, the length of τ is equal to $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$ and the considered equality holds
- If we know $\mathbf{D}^{(k-1)}$ then to compute $\mathbf{D}^{(k)}$ it is enough to make n^2 operations of additions and n^2 operations of comparisons of numbers
- Therefore, to compute $\mathbf{D}^{(n)} = \mathbf{D}^*$ it is enough to make $O(n^3)$ operations of addition and comparisons

Shortest Path: Floyd-Warshall Algorithm: Example



Shortest Path: Floyd-Warshall Algorithm: Example

For this example we compute the shortest path as following: for every i, j and $k > 0$, $d_{ij}^{(k)} = \min\left\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right\}$. If $i = k$ or $j = k$, then $d_{ij}^{(k)} = d_{ij}^{(k-1)}$.

$$\mathbf{D}^{(0)} = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & +\infty & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 4 & 1 & 0 \\ \hline \end{array}, \quad \mathbf{D}^{(1)} = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & +\infty & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 4 & 1 & 0 \\ \hline \end{array}$$

$$\mathbf{D}^{(2)} = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & +\infty & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 3 & 1 & 0 \\ \hline \end{array}, \quad \mathbf{D}^{(3)} = \begin{array}{c|ccc|} & 1 & 2 & 3 \\ \hline 1 & 0 & -1 & -2 \\ 2 & 2 & 0 & -1 \\ 3 & 3 & 1 & 0 \\ \hline \end{array} = \mathbf{D}^{*}$$