Dynamic Design Technique

Adnan YAZICI
Dept. of Computer Engineering
Middle East Technical Univ.
Ankara - TURKEY

Dynamic Programming Design Technique

- Dynamic programming can be viewed as a generalization of the Divide and Conquer approach.
- As you know, Divide and Conquer design technique partitions the problem into independent sub-problems of the same type, solve the sub-problems recursively, and then combine their solutions to solve the original problem.
- In contrast, Dynamic programming is applicable when the sub-problems are not_independent, that is, when sub-problems share subsubproblems.
- In this context, Divide and Conquer approach does more work than necessary, repeatedly solving the common subsubproblems.

Dynamic Programming Design Technique

- Dynamic Programming differs from the Greedy Design Technique since the greedy method produces only one feasible solution, which may or may not be optimal.
- While Dynamic Programming produces all possible sub-problems at most once, one of which guaranteed to be optimal.
- Optimal solutions to sub-problems are retained in a table, thereby avoiding the work of recomputing the answer every time a sub-problem is encountered.
- The use of these tabulated values makes it natural to recast the recursive equations into an iterative program.

Dynamic Programming Design Technique

• As mentioned, in DP, the problem is also divided into sub-problems of the same type.

• Often subdivision can be expressed in terms of a recurrence of the general form.

$$T(n) = f[T(n-1)]$$

• That is, the sate of the problem solution at stage n is related to the state of the solution at stage n-1.

• The function f in the recurrence is the *Principle of Optimality* (also coined by Bellman), that can be stated as follows:

> *An optimal sequence of decisions has the property that whatever the initial state and initial decision are, the remaining decision sequence must constitute an optimal decision sequence with regard to the state resulting from the first decision.*

• Another way to say this: *No matter how we got here, let's do the optimal thing from here on out*.

Dynamic Programming Design Technique

- Dynamic Programming is typically applied to optimization problems. The development of a dynamic programming algorithm can be broken into a sequence of four steps:
  1. *Characterize the structure of an optimal solution*
  2. *Recursively define the value of an optimal solution*
  3. *Compute the value of an optimal solution in a bottom-up fashion and retain the results in a structure (i.e., table)*
  4. *Construct an optimal solution from computed information*
- Steps 1-3 form the basis of a DP solution to a problem. Step 4 can be omitted if only the value of an optimal solution is required.

Dynamic Programming Design Technique

When should we look for a DP solution to a problem?

- *Optimal substructure*: we say that a problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to sub-problems. Whenever a problem exhibits optimal substructure, it is a good clue that DP might apply.
- For a DP to be applicable, the space of sub-problems must be "small" in the sense that a recursive algorithm for the problem solves the same sub-problems over and over, rather than always generating new sub-problems. Typically, the total number of distinct sub-problems is a polynomial in the input size.
- So whenever a recursion tree for the natural recursive solution to a problem contains the same problem repeatedly and the total number of different sub-problems is small, it is a good idea to see if DP can be made to work.

Examples for DP Approach

Example-1: The 0-1 Knapsack Problem (BNP)

Problem Instance: positive integers w_1, …, w_n, p_1, …, p_n, M (we have n objects, where object i has weight w_i and profit p_i, and M is the capacity of the knapsack.)

Feasible Solution: a vector $(x_1,\ldots x_n)$, where $x_i=0$ or $x_i=1$ and $\sum_{0\le i\le n} w_i x_i \le M$. In other words, the knapsack capacity cannot be exceeded. Note that we allow a fraction of an object to be placed into the knapsack.

Objective Function: $P = \sum_{0\le i\le n} p_i x_i$, where P is the profit associated with $(x_1,\ldots x_n)$

Optimal Solution: the maximum profit feasible solution.

Dynamic Programming Design Technique

- 0-1 knapsack problem is NP-Complete. We will see that a DP approach to this problem yields an algorithm whose complexity depends on the size of the knapsack capacity. Unfortunately, the capacity is often large in problems of practical interest.

1. Characterize the structure of an optimal solution:

Consider x_n: it is either 0 or 1.

If $x_n = 0$, then the best profit we can attain is whatever we get from the objects 1,2,…,(n-1).

If $x_n = 1$, we have a profit of p_n plus whatever we can get from the other objects, but now the capacity is reduced to $M-w_n$ (clearly $w_n \leq M$ must hold in this case).

The optimal solution is the better of these two feasible solutions (i.e., the principle of optimality applies).

Dynamic Programming Design Technique

2. Recursively define the value of an optimal solution: Let us now state this in mathematical terms:

For $0 \le m \le M$ and for $1 \le j \le n$, define P(j,m) to be the profit of the optimal solution using objects 1,2,..,j and capacity M. Then, we have the following recurrence relation:

$P(n,M) = P(n-1, M)$, if $w_n > M$

$P(n,M) = \max \{P(n-1,M), p_n + P(n-1, M-w_n)\}$, if $w_n \le M$

More generally, for $2 \le j \le n$ and $0 \le m \le M$,

$P(j,m) = P(j-1, m)$, if $w_j > m$

$P(j,m) = \max \{P(j-1,m), p_j + P(j-1, m-w_j)\}$, if $w_j \le m$

Also we have the boundary conditions

$P(1,m) = p_1$, if $w_1 \le m$

$P(1,m) = 0$, if $w_1 > m$

The optimal solution is P(n,M)

Dynamic Programming Design Technique

- We could compute the DP solution by writing a recursive function in a HLL. However, this top-down implementation would be very inefficient. To see this, let T(n) be the time required to calculate P(n,M) recursively.
- In general, we can write the following recurrence relation.

  $T(n) = 2.T(n-1) + c$; subject to $T(1) = 1$.
- For solution of this recurrence, the first few terms of this recurrence are;

  $T(2) = 2 + c$, $T(3) = 4 + 3c$, $T(4) = 8 + 7c$

  … ……

  $T(2^k) = 2^{k-1} + c.2^{k-1} - c$

  $T(n) = 2^n (1+c)/2 – c$

Therefore, T(n) is $\Theta(2^n)$.

- It is important to note that the total number of sub-problems is only n.M. This is the key observation: If $n.M < 2^n$, it would be less work to solve every sub-problem.

Dynamic Programming Design Technique

3. Compute the value of an optimal solution in a bottom-up fashion

- Dynamic programming provides a method whereby the maximum number of subproblems evaluated is n.M (which hopefully is less than 2^n).
- Also, it is possible that efficient implementations will avoid many of the "obviously" non-optimal subproblems, so less than M.n need be evaluated.
- First, solve all subproblems with j=1, then all subproblems with j=2, …, and finally all subproblems with j=n.
- We use the same recurrence relations as before, but we are solving bottom-up.
- It is easy to keep track of the progress of solutions using a two-dimensional array so that we can look up the answer to a subproblem whenever it is required later (this is important).

Dynamic Programming Design Technique

Example instances:

Weights: 2,3,5,8,13,16

Profit: 1,2,3,5,7,10, Capacity, M = 30.

For this example, the following two-dimensional array of subproblem solutions is generated. To see how the numbers are generated, when j=1, the initial conditions apply, so

$P(1,m) = 1$, if $2 \leq m$

$P(1,m) = 0$, if $2 > m$

When j=2, the recurrence becomes

$P(2,m) = P(1,m)$ if $m < 3$

$P(2,m) = \max \{P(1,m), 2 + P(1, m-3)\}$, if $3 \leq m$

$[P(j,m) = P(j-1, m)$, if $w_j > m$

$P(j,m) = \max \{P(j-1,m), p_j + P(j-1, m-w_j)\}$, if $w_j \leq m]$

- The optimal profit is 18, but what are the contents of the knapsack? This can be constructed from the table.

Dynamic Programming Design Technique

m	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
j=1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
j=2	0	0	1	2	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
j=3	0	0	1	2	2	3	3	4	5	5	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
j=4	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	9	10	10	11	11	11	11	11	11	11	11	11	11	11	11	11
j=5	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	10	10	10	11	11	11	12	12	13	14	14	15	15	16	17	17
j=6	0	0	1	2	2	3	3	4	5	5	6	7	7	8	8	10	10	10	11	12	12	13	13	14	15	15	16	17	17	18	**18**

$$P(j,m) = P(j-1, m), \text{ if } w_j > m$$
$$P(j,m) = \max \{P(j-1,m), p_j + P(j-1, m-w_j)\}, \text{ if } w_j \leq m$$

Dynamic Programming Design Technique

4. *Construct an optimal solution from computed information: The optimal solution is*

- P(6,30) = 18. Since 18 > P(5,30), object 6 must be included (so $x_6 = 1$). Therefore, $p_6 = 10$ contributes to the overall profit. The 18 was computed from P(6,30) = 10 + P(5,30-16), so we backtrack in the table to find P(5,14) = 18 – 10 = 8.
- Since P(5,14) = P(4,14), object 5 is excluded (so $x_5 = 0$).
- Since P(4,14) > P(3,14), object 4 must be included (so $x_4 = 1$). Therefore, $p_4 = 5$ contributes to the overall profit. The 8 was computed from P(4,14) = 5 + P(3,14-8), so we backtrack in the table to find P(3,6) = 8 – 5 = 3.
- Since P(3,6) = P(2,6), object 3 is excluded (so $x_3 = 0$).
- Since P(2,6) > P(1,6), object 2 must be included (so $x_2 = 1$). Therefore, $p_2 = 2$ contributes to the overall profit. The 3 was computed from P(2,6) = 3 + P(1,6-3), so we backtrack in the table to find P(1,3)= 3–2 = p_1, so object 1 is included, so $x_1 = 1$.
- Therefore, the optimal knapsack is X= (1,1,0,1,0,1), yielding a weight of (2+3+8+16) = 29, and a profit of (1+2+5+10) = 18.
- Note that if the optimal knapsack is not required (only optimal profit is desired), then only two rows of the matrix need be retained.

Examples for DP Approach

Example-2: Matrix Chain Products

- Given the matrices M_1, M_2,…, M_n, it is desired to calculate $M_1 \bullet M_2 \bullet \ldots \bullet M_n$. The matrices must be conformable; that is, the number of columns of M_i must be the same as the number of rows of M_{i+1}.
- The overall product is calculated by multiplying any two matrices at a time. A parenthesization of the sequences M_1, M_2,…, M_n determines how the product pairs are grouped.
- We can perform the multiplications using any parethesization, but we cannot alter the left-to-right order of the n matrices. So, the problem can be formalized as follows:

Example-2: Matrix Chain Products

Problem Instance: a sequence of n matrices M_1, M_2,…, M_n, where the no. of columns of M_i = no. of rows of M_{i+1}, for $0 \leq i \leq n-1$.

Feasible Solution: any parenthesization of M_1,M_2,…, M_n.

Objective Function: Given parenthesization, the cost is the total number of scalar multiplications required to produce $M_1 \bullet M_2 \bullet \ldots \bullet M_n$

Optimal Solution: minimum cost.

Dynamic Programming Design Technique

1. Characterize the structure of an optimal solution:

Given the sequence A, B, C, and D, there are five different parethesizations:

((A• B)•C)•D

(A• (B• C))•D (A• B)•(C•D)

A• ((B•C)•D) A• (B•(C•D))

Which is the best parethesization?

- Note that it can be shown that in general there are (2n-2)!/((n!).((n-1)!) different parethesizations of n matrices.
- This number is called the $(n-1)^{st}$ *Catalan number*, which arises often in counting problems.
- It can be shown that (2n-2)!/((n!).((n-1)!) is $\Theta(4^n/n^{3/2})$ using the *Sterling's* formula.
- This growth rate is enormous, therefore exhaustive enumeration is out of the question for any reasonable sized n.

Dynamic Programming Design Technique

- A reasonable cost measure is the total number of scalar multiplications required. When multiplying a mxp matrix by a pxr matrix, a total of m.p.r scalar multiplications are required when using the standard formula

  $$C_{ij} = \sum_{1 \le k \le p} a_{ik}.b_{kj};\ 1 \le i \le m,\ 1 \le j \le r$$

- We ignore the possibility of using Strassen's method or another to improve the performance for multiplying two matrices.
- Consider the following example: A, B, C, and D, having dimensions 5x3, 3x1, 1x4, and 4x6, respectively.
- By exhaustive enumeration:

  $((A \bullet B) \bullet C) \bullet D = 5.3.1 + 5.1.4 + 5.4.6 = 155$

  $(A \bullet (B \bullet C)) \bullet D = 3.1.4 + 5.3.4 + 5.4.6 = 192$

  $(A \bullet B) \bullet (C \bullet D) = 5.3.1 + 1.4.6 + 5.1.6 = 69$

  $A \bullet ((B \bullet C) \bullet D) = 3.1.4 + 3.4.6 + 5.3.6 = 174$

  $A \bullet (B \bullet (C \bullet D)) = 1.4.6 + 3.1.6 + 5.3.6 = 132$

Dynamic Programming Design Technique

2. Recursively define the value of an optimal solution

- Let's see how DP can yield a dramatic improvement in the computational complexity.
- If we are multiplying n matrices $M_1 \bullet M_2 \bullet \ldots \bullet M_n$ the last multiplication must be of the form $(M_1 \bullet M_2 \bullet \ldots \bullet M_i) \bullet (M_{i+1} \bullet M_2 \bullet \ldots \bullet M_n)$ for some $i \in \{1,2,\ldots,n-1\}$.
- For any particular choice of i, the best we can do is to find the optimal parenthesizations for $M_1 \bullet M_2 \bullet \ldots \bullet M_i$ and $M_{i+1} \bullet M_2 \bullet \ldots \bullet M_n$.
- If these subproblems have costs s_1 and s_2, then in terms of scalar multiplications *total cost* $= s_1 + s_2 + r_1 c_i c_n$
  where r_1= no. of rows of M_1, c_i = no. of columns of M_i and no. of rows of M_{i+1}, and c_n = no. of columns of M_n.

Dynamic Programming Design Technique

2. Recursively define the value of an optimal solution

- The optimal solution is obtained by finding the value of i that minimizes the total cost.
- That is, if C(j,k) denotes the minimum number of scalar multiplications required to multiply $M_1 \bullet M_2 \bullet \ldots \bullet M_n$, then we have

  $C(1,n) = \min \{C(1,i) + C(i+1,n) + r_1 c_i c_n ; 1 \le i \le n-1$
- Or more generally, the number of scalar multiplications required to multiply $M_j \bullet M_{j+1} \bullet \ldots \bullet M_k$ are;

  $C(j,k) = \min \{C(j,i) + C(i+1,k) + r_j c_i c_k ; j \le i \le k-1$
- The boundary conditions are:

  $C(j,j) = 0; 1 \le i \le n$

Dynamic Programming Design Technique

3. Compute the value of an optimal solution in a bottom-up fashion:

To implement the DP algorithm, we proceed as follows:

- We start with the boundary conditions where we calculate C(j,k) for j=k (product of 1 matrix).
- Next, we calculate C(j,k) for k = j+1 (products of 2 matrices), etc.
- Finally, we determine C(1,n) (the product of n matrices).
- The table includes the optimum results of each subsolution of the example problem.

Dynamic Programming Design Technique

The sequence of the calculations is as follows:

$C(1,1) = C(2,2) = C(3,3) = C(4,4) = 0$

$C(1,2) = 5.3.1 = 15$

$C(2,3) = 3.1.4 = 12$

$C(3,4) = 1.4.6 = 24$

$$C(j,k) = min\ \{C(j,i) + C(i+1,k) + r_j c_i c_k\};\ j \le i \le k-1$$

$C(1,3) = \min\ \{C(1,1) + C(2,3) + r_1c_1c_3,\ C(1,2) + C(3,3) + r_1c_2c_3\}$

$C(1,3) = \min\ \{0 + 12 + 5.3.4,\ 15 + 0 + 5.1.4\} = \min\ \{72,35\} = 35$

$C(2,4) = \min\ \{C(2,2) + C(3,4) + r_2c_2c_4,\ C(2,3) + C(4,4) + r_2c_3c_4\}$

$C(2,4) = \min\ \{0 + 24 + 3.1.6,\ 12 + 0 + 3.4.6\} = \min\ \{42,84\} = 42$

$C(1,4) = \min\ \{C(1,1) + C(2,4) + r_1c_1c_4,\ C(1,2) + C(3,4) + r_1c_2c_4,$
$C(1,3) + C(4,4) + r_1c_3c_4\}$

$C(1,4) = \min\ \{0 + 42 + 5.3.6,\ 15 + 24 + 5.1.6,\ 35 + 0 + 5.4.6\} =$
$\min\ \{132,69,155\} = 69.$

Dynamic Programming Design Technique

3. Compute the value of an optimal solution in a bottom-up fashion:

j	k=1, A	2, B	3, C	4, D
A	0 *A*	15 *(A.B)*	35 *(A.B).C*	69 *(A.B)(C.D)*
B		0 *B*	12 *(B.C)*	42 *B.(C.D)*
C			0 *C*	24 *(C.D)*
D				0 *D*

4. Construct an optimal solution from computed information: The optimal solution is (AB)*(CD), which can constructed by starting from the optimum solution for n matrices to 1 matrix. Look into what suboptimal solutions are used to result in the optimum solution at the end.

Dynamic Programming Design Technique

Analysis of the DP solution:

- If we were to implement these recurrence relations recursively, it can be shown that the complexity is $\Theta(2^n)$, which is much better than $\Theta(4^n/n^{3/2})$, but still exponential.
- The solution using DP design approach is to ask: how many C(j,k)'s are there to be calculated? Since $1 \leq j \leq k \leq n$, the total number is $n(n+1)/2$, which is $\Theta(n^2)$.
- Each C(j,k) can be calculated in time is $\Theta(n)$ if all "previous" C(j,i) and C(i,k) are known. So the overall algorithm is $\Theta(n^3)$, which is a dramatic improvement.

Dynamic Programming Design Technique

- As seen from the solution above, diagonal s contains the elements m_{ij} such that j-i = s. The diagonal s=0 therefore contains the elements m_{ii}, $1 \le i \le n$, corresponding to the "products" M_i. Here there is no multiplication to be done, $m_{ij} = 0$ for every i.
- The diagonal s=1 contains the elements $m_{i,i+1}$, $1 \le i \le n$, corresponding to the products of $M_i\ M_{i+1}$.
- Finally when s>1 the diagonal s contains the elements $m_{i,i+s}$ corresponding to the products of $M_i\ M_{i+1} \ldots M_{i+s}$.
- Now we have a choice: we can make the first cut in the product after any matrices $M_i\ M_{i+1} \ldots M_{i+s}$.
- To find the optimum, we choose the cut that minimizes the required number of scalar multiplications.
- For s>0 there are n-s elements to be computed in the diagonal s; for each of these we must choose between s possibilities given by the different values of k.
- The execution time of the algorithm is therefore in the exact order of

$$\sum_{1 \le s \le (n-1)} (n-s)s = n \sum_{1 \le s \le (n-1)} s - \sum_{1 \le s \le (n-1)} s^2$$
$$= n^2(n-1)/2 - n(n-1)(2n-1)/6 = (n^3 - n)/6$$

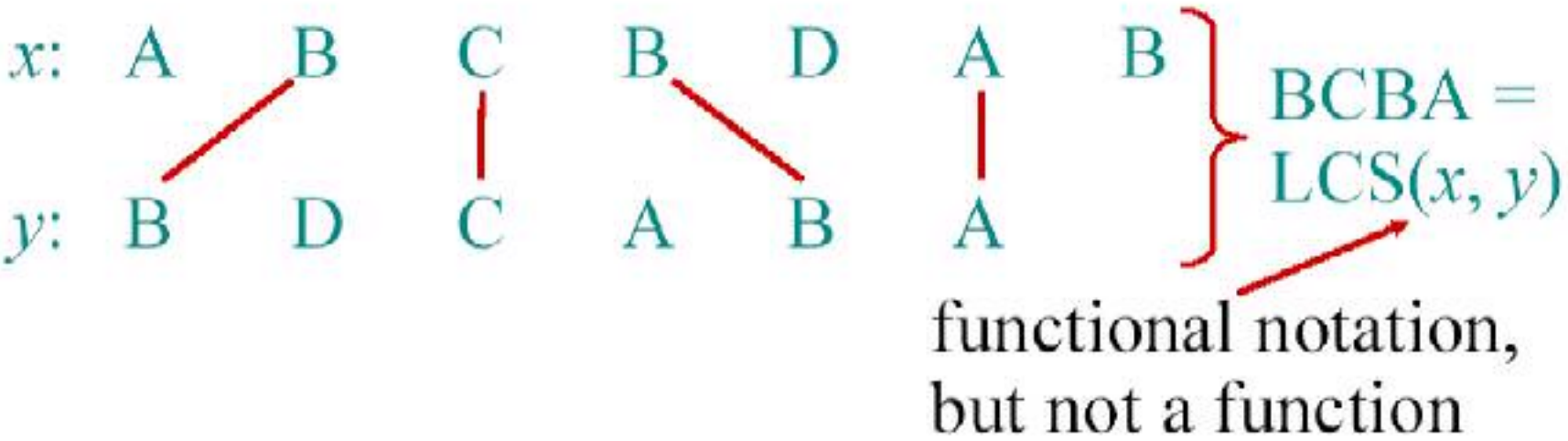
Therefore, the execution of the algorithm is thus in $\Theta(n^3)$.

Dynamic Programming Design Technique

Example: *Longest Common Subsequence (LCS)*

- Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.

"a" *not* "the"



Brute-force LCS algorithm

Check every subsequence of $x[1 \,..\, m]$ to see if it is also a subsequence of $y[1 \,..\, n]$.

Analysis

- Checking = $O(n)$ time per subsequence.
- 2^m subsequences of x (each bit-vector of length m determines a distinct subsequence of x).

Worst-case running time = $O(n2^m)$
= exponential time.

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

```
LCS(x, y, i, j)
    if c[i, j] = NIL
        then if x[i] = y[j]
            then c[i, j] ← LCS(x, y, i–1, j–1) + 1
            else c[i, j] ← max{LCS(x, y, i–1, j),
                              LCS(x, y, i, j–1)}
```

same as before

Time = $\Theta(mn)$ = constant work per table entry.
Space = $\Theta(mn)$.

Dynamic-programming algorithm

IDEA:

Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.

Exercise: $O(\min\{m, n\})$.

		A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Example: All-Pairs Shortest Path in Graphs

Problem Instance: a graph G in which each edge e of G has been assigned a non-negative weight (or cost), w(e).

Feasible Solution: For all unordered pairs of distinct vertices v, u, a path u-v is required.

Objective Function: $\sum_{1 \le i \le k}$ Dist(i)

Optimal Solution: The shortest feasible set of u-v paths.

All-pairs shortest paths

Input: Digraph $G = (V, E)$, where $|V| = n$, with edge-weight function $w : E \to \mathbb{R}$.

Output: $n \times n$ matrix of shortest-path lengths $\delta(i, j)$ for all $i, j \in V$.

IDEA #1:

- Run Bellman-Ford once from each vertex.
- Time = $O(V^2E)$.
- Dense graph $\Rightarrow$ $O(V^4)$ time.

Good first try!

Dynamic programming

Consider the $n \times n$ adjacency matrix $A = (a_{ij})$ of the digraph, and define

$d_{ij}^{(m)}$ = weight of a shortest path from i to j that uses at most m edges.

Claim: We have

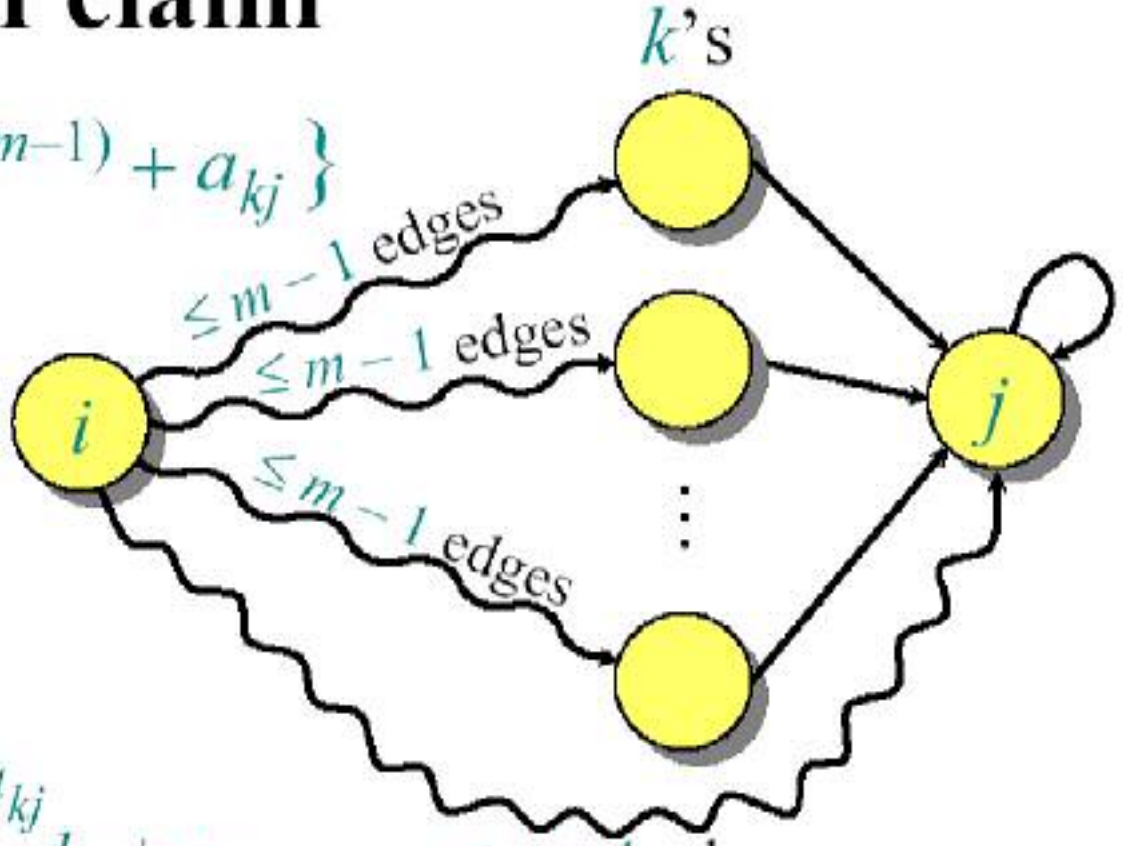
$$d_{ij}^{(0)} = \begin{cases} 0 & \text{if } i = j, \\ \infty & \text{if } i \neq j; \end{cases}$$

and for $m = 1, 2, \ldots, n-1$,

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}.$$

Proof of claim

$$d_{ij}^{(m)} = \min_k \{ d_{ik}^{(m-1)} + a_{kj} \}$$



Relaxation!

for $k \leftarrow 1$ **to** n

  do if $d_{ij} > d_{ik} + a_{kj}$

    then $d_{ij} \leftarrow d_{ik} + a_{kj}$

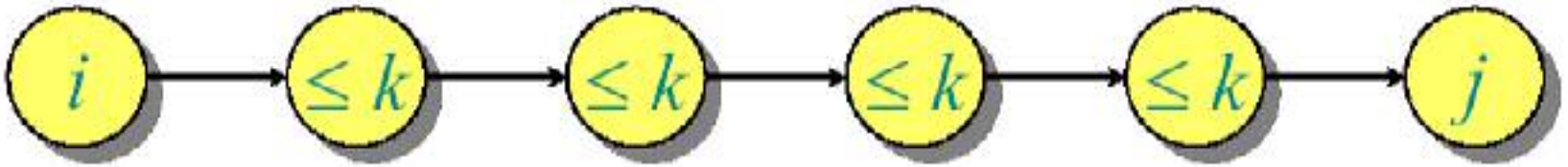
Note: No negative-weight cycles implies

$$\delta(i, j) = d_{ij}^{(n-1)} = d_{ij}^{(n)} = d_{ij}^{(n+1)} = \ldots$$

Floyd-Warshall algorithm

Also dynamic programming, but faster!

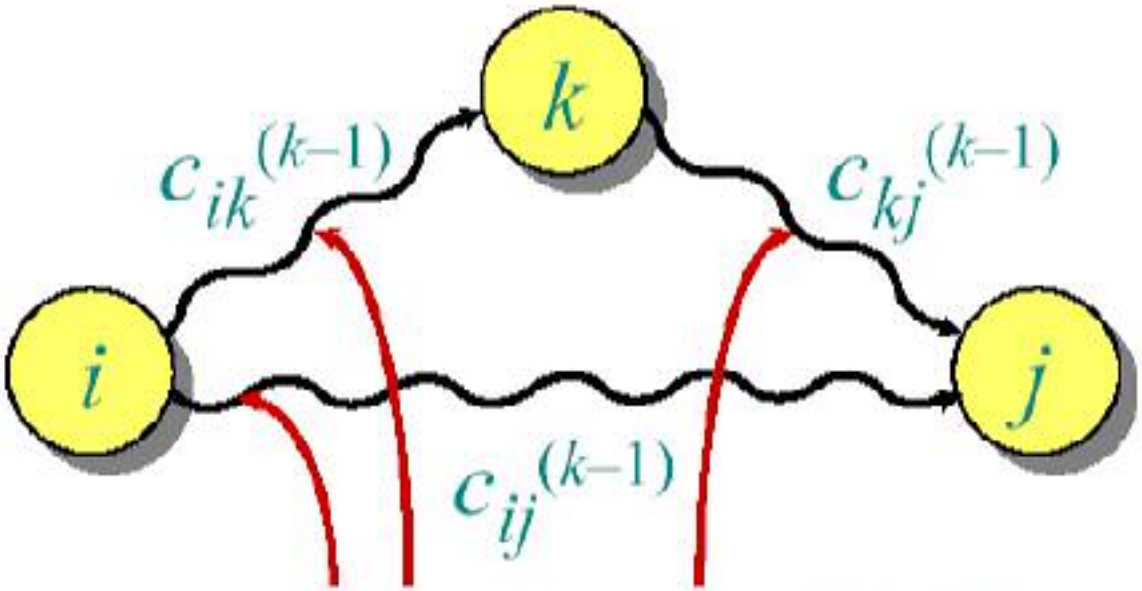
Define $c_{ij}^{(k)}$ = weight of a shortest path from i to j with intermediate vertices belonging to the set $\{1, 2, \ldots, k\}$.



Thus, $\delta(i, j) = c_{ij}^{(n)}$. Also, $c_{ij}^{(0)} = a_{ij}$.

Floyd-Warshall recurrence

$$c_{ij}^{(k)} = \min_k \{c_{ij}^{(k-1)}, c_{ik}^{(k-1)} + c_{kj}^{(k-1)}\}$$



intermediate vertices in $\{1, 2, \ldots, k\}$

All-Pairs Shortest Path in Graphs

Recursively define the value of an optimal solution

If G has n vertices, and if $1 \le k \le n$, define $S^k(i,j)$ to be the length of the shortest i-j path in G using only vertices from the set $\{1 \ldots k\}$.

For convenience, define $S^k(i,i) = 0$ for $1 \le k \le n$ and $1 \le i \le n$.

If we know $S^k(i,j)$, how can we calculate $S^{k+1}(i,j)$?

Based on the fact developed above, if P is the shortest i-j path containing intermediate vertices from $\{1,\ldots k+1\}$, then we have two possibilities:

- If $(k+1) \notin P$ then $S^{k+1}(i,j) = S^k(i,j)$
- If $(k+1) \in P$ then P consists of a shortest i-(k+1) path joined to a shortest (k+1)-j path.

If (k+1) leads to a shorter path then we place it in P, otherwise we exclude it.

Thus we have the following recurrence:

$S^{k+1}(i,j) = \min \{S^k(i,j), S^k(i,k+1) + S^k(k+1,j)\}$ for $0 \le k \le n$, $1 \le i,j \le n$.

The initial conditions are:

$S^k(i,j) = M(i,j)$ if $i \ne j$

$S^k(i,j) = 0$ if $i = j$

All-Pairs Shortest Path in Graphs

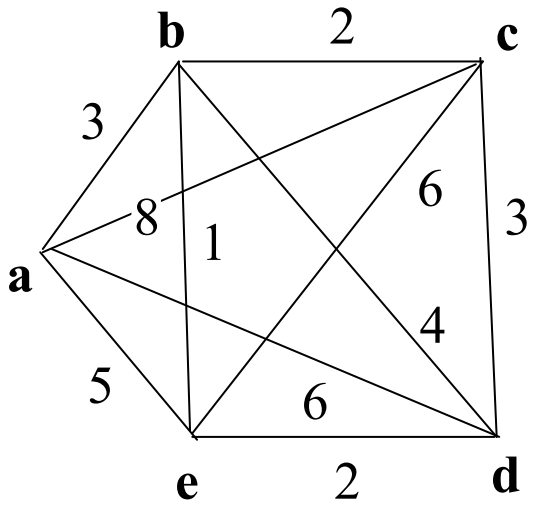
3. Compute the value of an optimal solution in a bottom-up fashion:

The pseudo-code for Floyd's algorithm is as follows:

Procedure Floyd (W[1:n,1:n],P[1:n,1:n],S[1:n,1:n])
Input: W[1:n,1:n], weight matrix for a weighted digraph G
Output: P[1:n,1:n], matrix implementing shortest paths
S[1:n,1:n]), distance matrix where S[u,v] is the length (cost) of a shortest path u to v in G)

```
for i = 1 to n do{
      for j = 1 to n do{        // Intialize P and S
            P[i,j] = 0;
            S[i,j] = W[i,j]}}
For k = 1 to n do               // update S and P using the recurrence relation
      For i = 1 to n do
           For j = 1 to n do{
           If S[i,j] > S[i,k] + S[k,j] {
                       P[i,j] = k
                       S[i,j] = S[i,k] + S[k,j]}
                       }}
EndFloyd
```

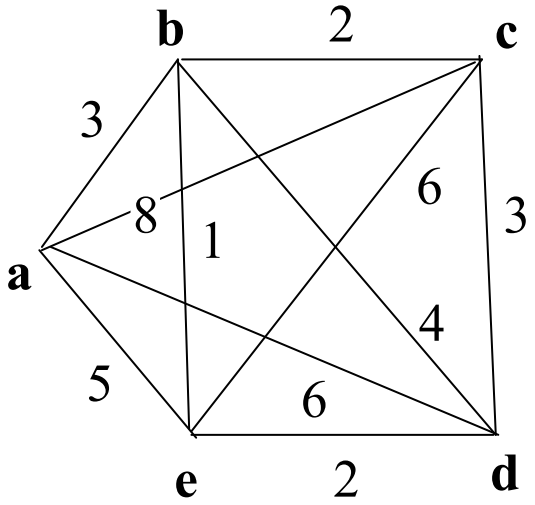
All-Pairs Shortest Path in Graphs



M=

0	3	8	6	5
3	0	2	4	1
8	2	0	3	6
6	4	3	0	2
5	1	6	2	0

All-Pairs Shortest Path in Graphs

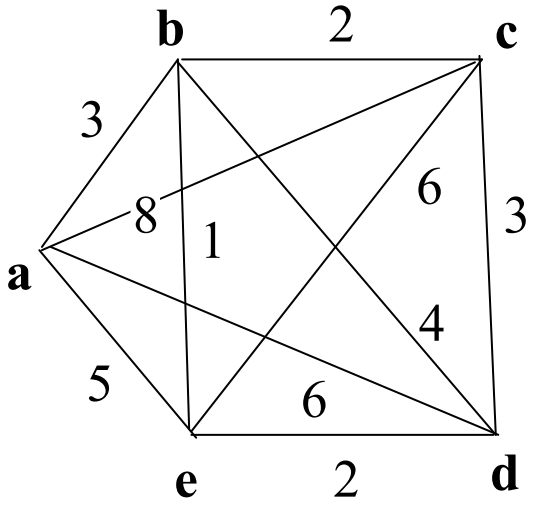


$S^0 = S^1 =$

0	3	8	6	5
3	0	2	4	1
8	2	0	3	6
6	4	3	0	2
5	1	6	2	0

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

All-Pairs Shortest Path in Graphs

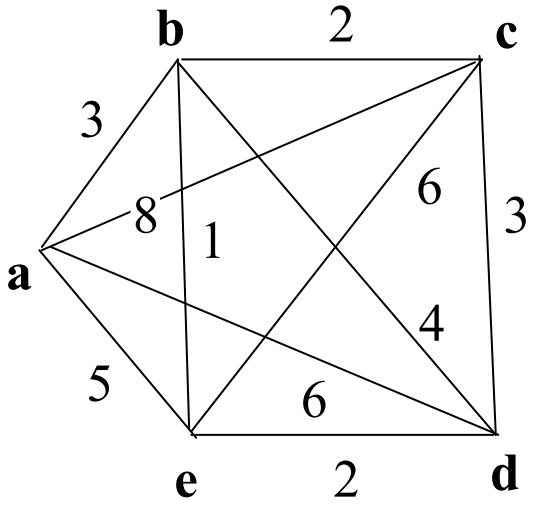


$S^2 = S^3 = S^4 =$

0	3	*5*	6	*4*
3	0	2	4	1
5	2	0	3	*3*
6	4	3	0	2
4	1	*3*	2	0

$$\begin{pmatrix} 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 \end{pmatrix}$$

All-Pairs Shortest Path in Graphs

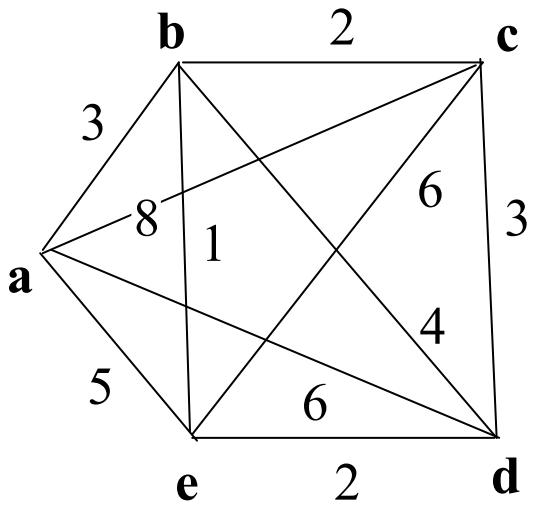


$S^5=$

0	3	5	6	4
3	0	2	*3*	1
5	2	0	3	3
6	*3*	3	0	2
4	1	3	2	0

$$\begin{pmatrix} 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 5 & 0 \\ 2 & 0 & 0 & 0 & 2 \\ 0 & 5 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 \end{pmatrix}$$

All-Pairs Shortest Path in Graphs



$S^5=$

0	3	5	6	4
3	0	2	*3*	1
5	2	0	3	3
6	*3*	3	0	2
4	1	3	2	0

$$\begin{pmatrix} 0 & 0 & 2 & 0 & 2 \\ 0 & 0 & 0 & 5 & 0 \\ 2 & 0 & 0 & 0 & 2 \\ 0 & 5 & 0 & 0 & 0 \\ 2 & 0 & 2 & 0 & 0 \end{pmatrix}$$

4. Construct an optimal solution from computed information:
So, for example, the shortest a-e path is 4 and the shortest d-e path is 2. The actual path can be determined by tracing back through the matrices. Whenever an element changes, then vertex k is on the path. So, for example, the a-e path is become 4 when k was b, so the path is a-b-e.

Pseudocode for Floyd-Warshall

```
for k ← 1 to n
    do for i ← 1 to n
           do for j ← 1 to n
                  do if c_ij > c_ik + c_kj
                        then c_ij ← c_ik + c_kj
```

} ***relaxation***

Notes:

- Okay to omit superscripts, since extra relaxations can't hurt.
- Runs in $\Theta(n^3)$ time.
- Simple to code.
- Efficient in practice.

Example: Travel Salesman Problem, (TSP)

Problem Instance: a graph G in which each edge has a non-negative weight (or cost).

Feasible Solution: one cycle, which passes through every vertex exactly once (a Hamiltonian Circuit)

Objective Function: $c(H) = \sum_{e \in H} c(e)$, where H is a Hamiltonian Circuit of G and c(e) is the cost of edge e.

Optimal Solution: the minimum cost Hamiltonian Circuit.

This problem is NP-Complete, so we do not expect a polynomial-time algorithm.

Recursively define the value of an optimal solution

To pose a DP algorithm, we must decompose the problem instance. Suppose we have a graph G with n vertices and w(i,j) is the weight (or cost) of edge ij. Any Hamiltonian circuit H can be considered to start at vertex 1. Let k be the last vertex in H before we return to vertex 1 (so $2 \leq k \leq n$).

Then the optimal Hamiltonian circuit consists of:

1. an optimal path from vertex 1 to vertex k, passing through all the vertices.
2. an edge from k to 1.

Next, consider the vertex j preceding k in the path from 1 to k. Note that $j \in \{2,..n\} - \{k\}$. The optimal path from 1 to k can be decomposed into:

- an optimal path 1 to j
- the edge from j to k

This process can be repeated and we can write the associated recurrence relations.

Let S be a subset of {2…n} and let k∈ S. Define P(S,k) to be the cost of the optimal path from vertex 1 to vertex k, in which the intermediate vertices are precisely those in S-{k}. Then, the optimal Hamiltonian circuit cost is:

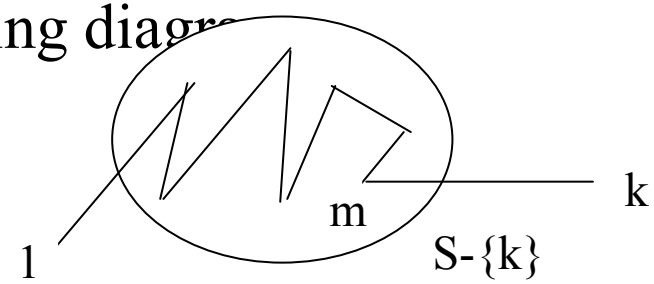
min{P(2…n}, k) + w(k,1)}, 2≤ k ≤ n

The P(S,k)'s can be calculated from the following recurrence relations:

P(S,k) = min {P(S-{k},m) + w(m,k)}, if |S| > 1, m ∈ S – {k}

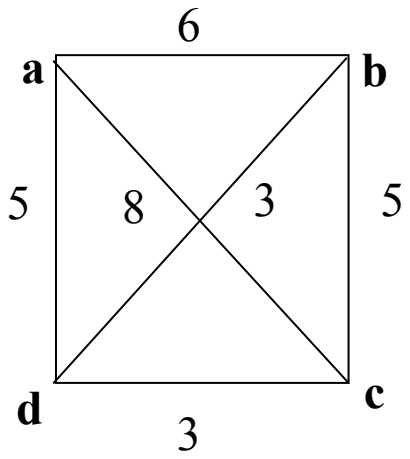
P(S,k) = w(1,k), if S = {k}

m represents the intermediate vertices in S, generating a path from the first node to k going through m nodes (an edge from each m to k exists). For a particular m, the situation is depicted in the following diagr



Compute the value of an optimal solution in a bottom-up fashion:

First, we calculate all P(S,k) with |S| = 1, then all those with |S| = 2, …, and finally all those with |S| = n-1. Let's work out an example before analyzing the complexity of this approach.



S	k	P(S,k)
{b}	b	6
{c}	c	8
{d}	d	5
{b,c}	b	13 (8+5)
{b,c}	c	11 (6+5)
{b,d}	b	8 (5+3)
{b,d}	d	9 (6 +3)
{c,d}	c	8 (5+3)
{c,d}	d	11 (8 +3)
{b,c,d}	b	13 (min{8 + 5 (b-c)=13, 11 + 3 (b-d) = 14})
{b,c,d}	c	12 (min{8+5 (c-b)=13, 9+3 (c-d) = 12})
{b,c,d}	d	14 (min{13+3 (d-b)=16, 11+3 (d-c) = 14})

The minimum cost Hamiltonian circuit is then given by Min {13 +6, 12+8, 14 +5}
= min {19,20,19} = 19.

Construct an optimal solution from computed information:

There are two solutions:
(1-4-3-2-1 and 1-2-3-4-1) = 19.

Analysis of the DP algorithm for TSP:

We now consider the complexity of this algorithm. First, we must count the number of P(S,k)'s that must be calculated. For each S of cardinality j, there are j possibilities for k. Number of distinct sets S of size j (cardinality j) not including 1 is C(n-1,j). So, each S can be any j-subset of {2…n}, and the number of these is C(n,j) = n! / (j!(n-j)!).

Hence the total number of P(S,k)'s to calculate is:

$\sum_{0\le j \le n-1} j.C(n-1,j)$, where for each j-subset there are (n-1) choices for j.

To calculate this sum, we use the binomial theorem:

$(1+x)^{n-1} = \sum_{0\le j \le n-1} C(n-1,j)\, x^j$

Differentiating both sides gives:

$(n-1)(1+x)^{n-2} = \sum_{0\le j \le n-1} j.C(n-1,j)\, x^{j-1}$

$= \sum_{0\le j \le n-1} j.C(n-1,j)\, x^{j-1}$

Now, substitute x = 1, and we get

$(n-1).2^{n-2} = \sum_{0\le j \le n-1} j.C(n-1,j)$, which is equal to the total the number of P(S,k)'s.

Hence, the total number of P(S,k)'s is $(n-1).\ 2^{n-2}$

This is $\Theta(n.2^{n-2})$, so there are an exponential number to calculate.

Calculating one P(S,k) require finding the minimum of at most n quantities. Therefore, the entire algorithm is $\Theta(n^2.2^{n-2})$. The total number of Hamiltonian circuits is (n-1)! The growth rate of $n^2.2^{n-2}$ is much smaller than that of the factorial function (which we can prove using Sterling's formula). That is, this is better than enumerating all (n-1)! Different tours to find the best one. So, we have traded on exponential growth for a much smaller exponential growth. The most serious drawback of this DP algorithm solution is the space needed, which is $O(n.2^n)$.