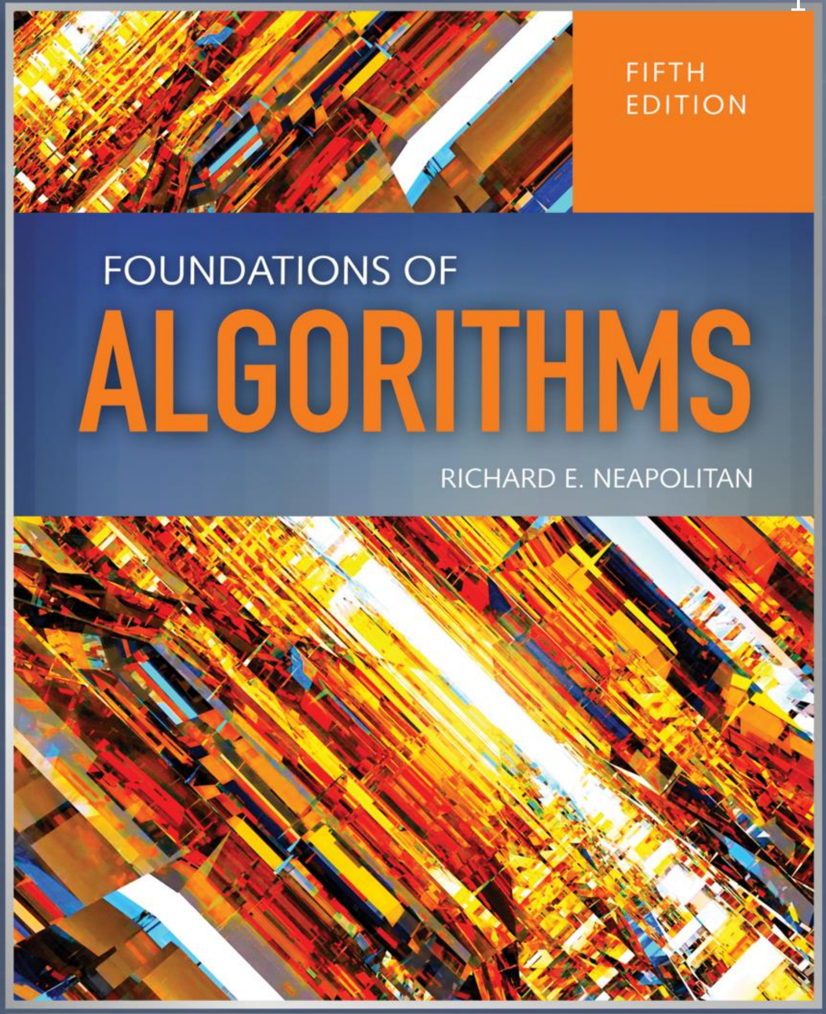
Dynamic Programming

Chapter 3



Objectives

- Describe the Dynamic Programming Technique
- Contrast the Divide and Conquer and Dynamic Programming approaches to solving problems
- Identify when dynamic programming should be used to solve a problem
- Define the Principle of Optimality
- Apply the Principle of Optimality to solve Optimization Problems

Divide and Conquer

- Top-down approach to problem solving
- Blindly divide problem into smaller instances and solve the smaller instances
- Technique works efficiently for problems where smaller instances are unrelated
- Inefficient solution to problems where smaller instances are related
- Recursive solution to the Fibonacci sequence

Dynamic Programming

- Bottom-up approach to problem solving
- Instance of problem divided into smaller instances
- Smaller instances solved first and stored for later use by solution to solve larger instances
- Look it up instead of re-compute
- Iterative solution to the Fibonacci Sequence

Steps to develop a dynamic programming algorithm

1. Establish a recursive property that gives the solution to an instance of the problem
2. Compute the value of an optimal solution in a bottom-up fashion by solving smaller instances first

The Binomial Coefficient

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

The Recursive Property

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k} \quad 0 < k < n$$

$$\binom{n}{k} = 1 \quad k = 0 \text{ or } k = n$$

Algorithm 3.1 Binomial Coefficient

- Divide-and-Conquer
- Recursive
- Very inefficient – like recursive Fibonocci

Number of terms computed by recursive bin

$$2\binom{n}{k} - 1$$

Dynamic Programming Solution to the Binomial Coefficient Problem

- Using the recursive property, construct an array B containing solutions to smaller instances

Construct Array B such that:

$$B[i,j] = \binom{i}{j}$$

Establish a recursive property and solve bottom-up

- B[i , j] = B[i – 1, j – 1] + B [i – 1,j] where 0 < j < I
- B[i , j] = 1 where j = 0 or j = I
- Solve an instance of the problem bottom up - compute rows in B in sequence starting with row 1
- At each iteration, the values needed for that iteration have already been computed

Algorithm 3.2 Binomial Coefficient using Dynamic Progra,ing

- bin2
- The work done by bin2 as a function of n and k
- bin2 ε θ(nk)

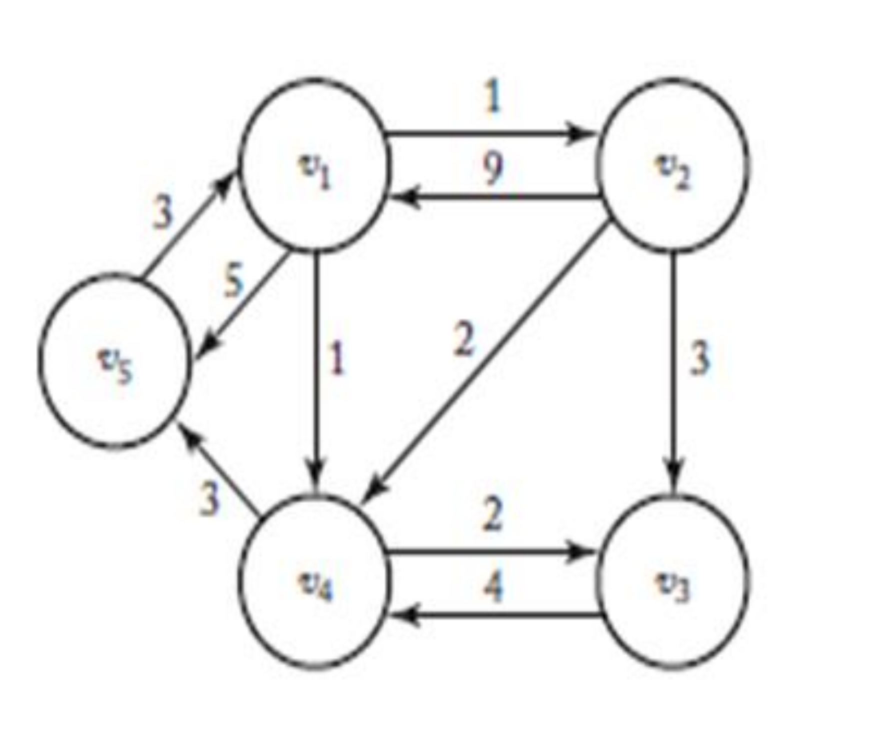
Optimization Problem

- Multiple candidate solutions
- Candidate solution has a value associated with it
- Solution to the instance is a candidate solution with an optimal value
- Minimum/Maximum

Shortest Path Problem

- Optimization Problem
- Candidate Solution: path from oe vertex to another
- Value of candidate solution: length of the path
- Optimal value – minimum length
- Possible multiple shortest paths

Weighted Directed Graph



Brute Force

- For every vertex, determine lengths of all paths from that vertex to every other vertex and compute minimum lengths
- Complete Graph G
  - (n-2)!

Adjacency Matrix M

- W[i,j] = weight of the path from vi->vj if there is an edge
- W[i,j] = ∞ if there is no edge from vi->vj
- W[i,j] = 0 if i = j

Figure 3.3

	1	2	3	4	5
1	0	1	∞	1	5
2	9	0	3	2	∞
3	∞	∞	0	4	∞
4	∞	∞	2	0	3
5	3	∞	∞	∞	0

W

	1	2	3	4	5
1	0	1	3	1	4
2	8	0	3	2	5
3	10	11	0	4	7
4	6	7	2	0	3
5	3	4	6	4	0

D

Dynamic Programming Solution to the all-pairs shortest path

- n vertices in the graph
- Create a sequence of n+1 arrays D^k where $0 <= k <=n$
- $D^k [i,j]$ = length of a shortest path from v_i to v_j using only vertices in the set $\{v_1 , v_2 , ... v_k \}$
- $D^n [i,j]$= length of the shortest path from v_i to v_j
- $D^0 = W$ and $D^n = D$

Dynamic Programming Steps

- Establish a recursive property to compute D^k from $D^{(k-1)}$
- Solve an instance of the problem in bottom-up fashion by repeating the process for k=1 to n

Establish a recursive Property

- Two Cases to consider (details next two slides)
- D^k [i,j] = minimum (case 1, case 2)
  - = minimum ($D^{(k-1)}$ [i,j], $D^{(k-1)}$ [i,k] + $D^{(k-1)}$ [k,j])

Case 1

- At least one shortest path from v_i to v_j uses only vertices in set $\{v_1 , v_2 , . \ \ . . v_k \}$ as intermediate vertex does not use v_k
  - $D^k [i,j] = D^{(k-1)} [i,j]$

Case 2

All shortest paths from v_i to v_j uses only vertices in set $\{v_1, v_2, \ldots v_k\}$ as intermediate vertex does use v_k

- Path = $v_i, \ldots, v_k, \ldots, v_j$ where $v_i, \ldots, v_k$ consists only of vertices in $\{v_1, v_2, \ldots V_{k-1}\}$ as intermediates: Cost of path = $D^{(k-1)}[i,k]$
- And where $v_k, \ldots, v_j$ consists only of vertices in $\{v_1, v_2, \ldots V_{k-1}\}$ as intermediates: Cost of path = $D^{(k-1)}[k,j]$

Floyd's Algorithm for Shortest Paths – Algorithm 3.3

- Every-case time complexity
- Basic operation – instructions in the for j loop
- Input size : lVl
- $T(n) = n^3$

Does Dynamic Programming Apply to all Optimization Problems?

- No
- The Principle of Optimality
  - An optimal solution to an instance of a problem always contains optimal solution to all subinstances
- Shortest Paths Problem
  - If v_k is a node on an optimal path from v_i to v_j then the sub-paths v_i to v_k and v_k to v_j are also optimal paths

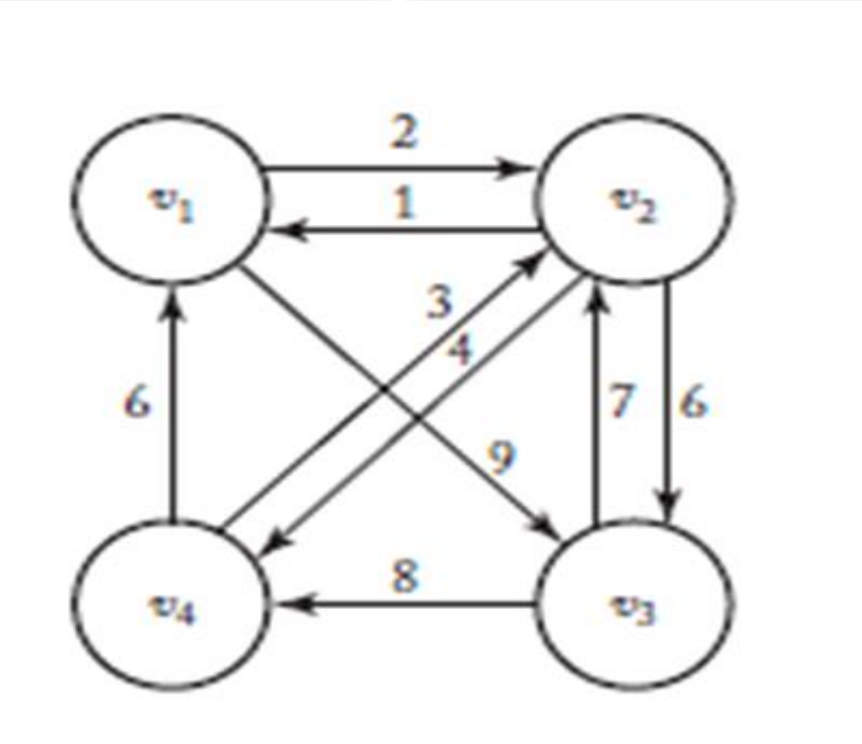
Chained-Matrix Multiplication

- Optimal order for chained-matrix multiplication dependent on array dimensions
- Consider all possible orders and take the minimum: $t_n > 2^{n-2}$
- Principle of Optimality applies
- Develop Dynamic Programming Solution
- M[i,j] = minimum number of multiplications needed to multiply A_i through A_j

Traveling Salesperson Problem

- Sales trip – n cities
- Each city connects to some of the other cities by a road
- Minimize travel time – determine a shortest route that starts at the salesperson's home city, visits each city once, and ends at home city
- Represent instance of the problem with a weighted graph

Figure 3.16



Dynamic Programming Algorithm for Traveling Sales Person Problem

- $\Theta(n2^n)$
- Inefficient solution using dynamic programming
- Problem NP-Complete