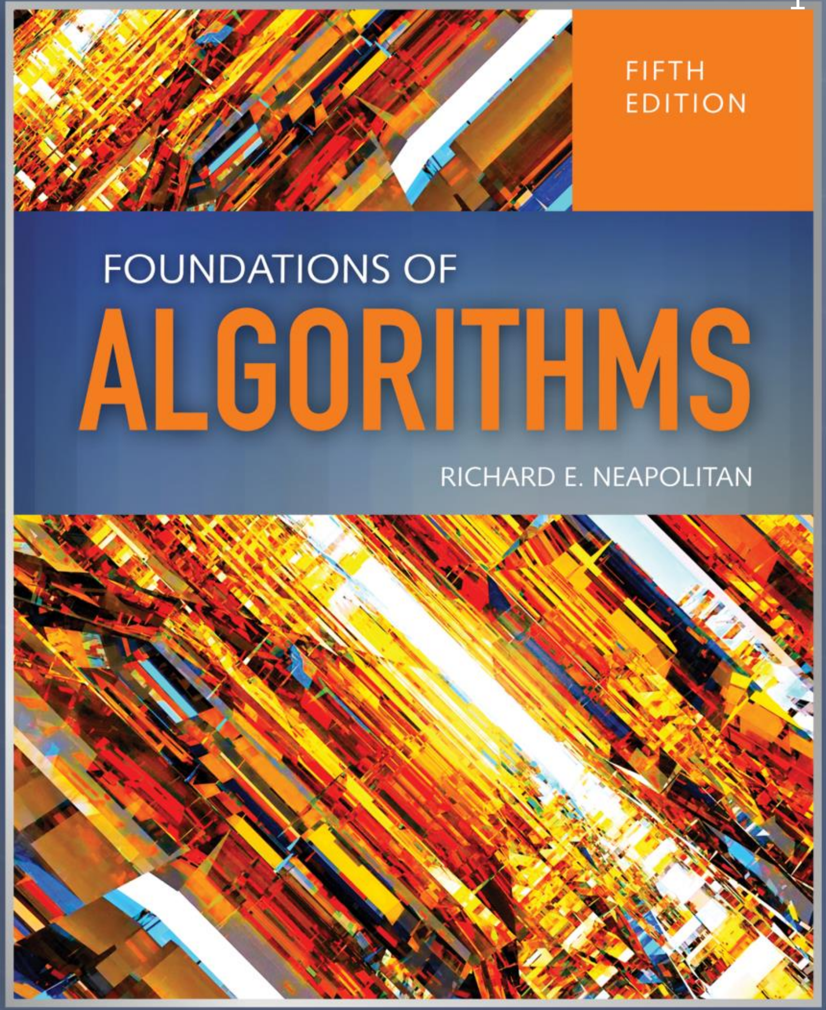
Dynamic Programming

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

- Dynamic Programming is a general algorithm design technique for solving problems defined by recurrences with overlapping subproblems

- Invented by American mathematician Richard Bellman in the 1950s to solve optimization problems and later assimilated by CS

- “Programming” here means “planning”

- Main idea:
  - set up a recurrence relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

- Iterative solution to the Fibonacci Sequence

Example 1: Fibonacci numbers

- Recall definition of Fibonacci numbers:

$$F(n) = F(n\text{-}1) + F(n\text{-}2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- Computing the n^{th} Fibonacci number recursively (top-down):

$$F(n)$$

$$F(n\text{-}1) \quad + \quad F(n\text{-}2)$$

$$F(n\text{-}2) \quad + \quad F(n\text{-}3) \qquad\qquad F(n\text{-}3) \quad + \quad F(n\text{-}4)$$

...

Example 1: Fibonacci numbers (cont.)

Computing the n^{th} Fibonacci number using bottom-up iteration and recording results:

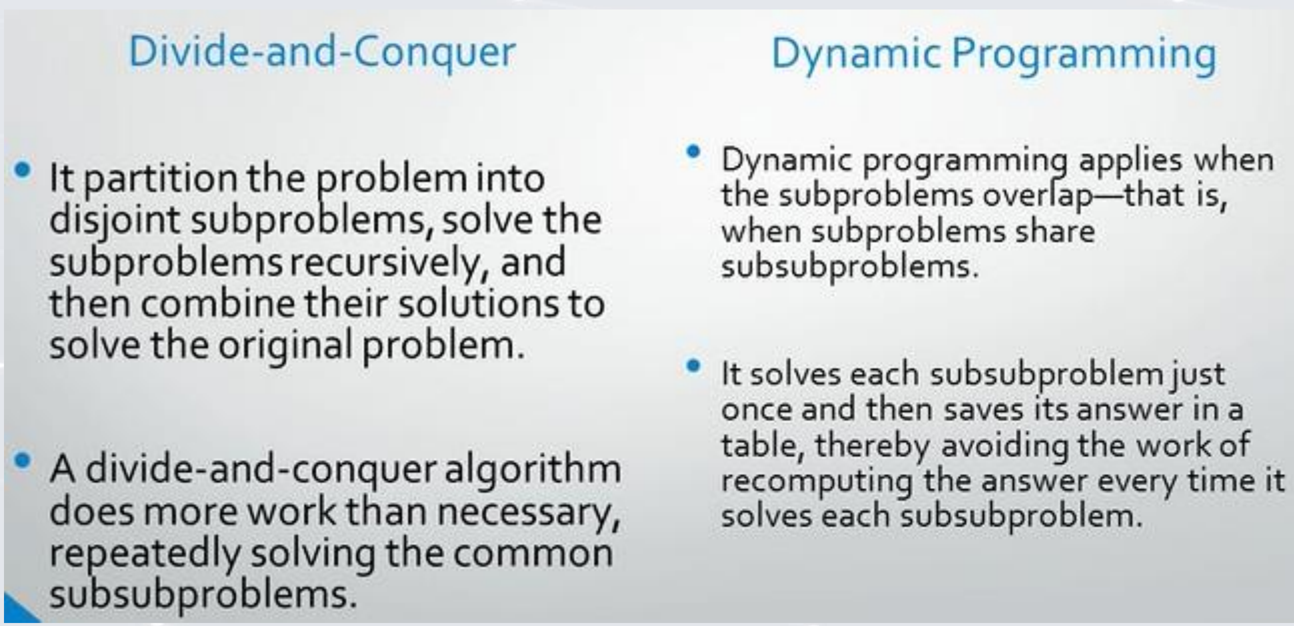
$F(0) = 0$
$F(1) = 1$
$F(2) = 1+0 = 1$
…
$F(n-2) =$
$F(n-1) =$
$F(n) = F(n-1) + F(n-2)$

0	1	1	. . .	$F(n-2)$	$F(n-1)$	$F(n)$

Efficiency:
- time
- space

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 change-making problem [DPV, Exercise 6.17]

Change-Making Problem

Input: Positive integers $1 = x_1 < x_2 < \cdots < x_n$ and v

Task: Given an unlimited supply of coins of denominations $x_1, \ldots, x_n$, find the minimum number of coins needed to sum up to v.

Key question of dynamic programming: *What are the subproblems?*

For $0 \leq u \leq v$, compute the minimum number of coins needed to make value u, denoted as $C[u]$

For $u = v$, $C[u]$ is the solution of the original problem.

Optimal substructure: for $u \geq 1$, one has

$$C[u] = 1 + \min\{\, C[u - x_i] : 1 \leq i \leq n \;\wedge\; u \geq x_i \,\}\,.$$

$C[u]$ can be computed from the values of $C[u']$ with $u' < u$.

Pseudocode for the Change-Making Problem

$\textsc{Change-Making}(x_1, \ldots, x_n; v)$

Input: Positive integers $1 = x_1 < x_2 < \cdots < x_n$ and v

Output: Minimum number of coins needed to sum up to v

$C[0] = 0$
for $u = 1$ **to** v
    $C[u] = 1 + \min\{ C[u - x_i] : 1 \leq i \leq n \ \wedge \ u \geq x_i \}$
return $C[v]$

Running time analysis

The array $C[1..v]$ has length v, and each entry takes $O(n)$ time to compute. Hence running time is $O(nv)$.

The Binomial Coefficient

Binomial coefficients are coefficients of the binomial formula:
$(a + b)^n = C(n,0)a^n b^0 + \ldots + C(n,k)a^{n-k}b^k + \ldots + C(n,n)a^0 b^n$

$$C(n,k) = \binom{n}{k} = \frac{n!}{k!\,(n-k)!} \qquad \text{for } 0 \le k \le n.$$

For values of n and k that are not small, we cannot compute the binomial coefficient directly from this definition because $n!$ is very large even for moderate values of n. In the exercises we establish that

$$\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k} & 0 < k < n \\ 1 & k = 0 \text{ or } k = n. \end{cases} \qquad (3.1)$$

Recurrence: $C(n,k) = C(n\text{-}1,k) + C(n\text{-}1,k\text{-}1)$ for $n > k > 0$
$C(n,0) = 1, \quad C(n,n) = 1$ for $n \ge 0$

Algorithm 3.1 Binomial Coefficient

Binomial Coefficient Using Divide-and-Conquer

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k, where $k \le n$.

Outputs: *bin*, the binomial coefficient $\binom{n}{k}$.

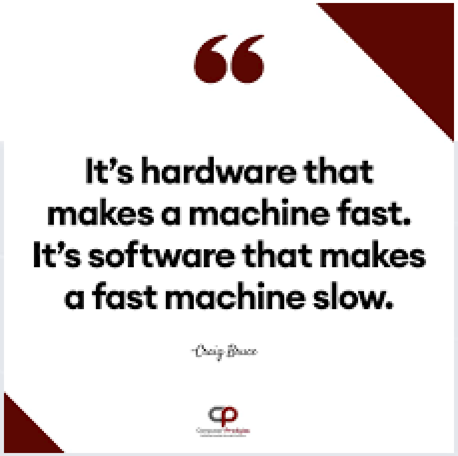
```
int bin (int n, int k)
{
   if (k == 0 || n == k)
      return 1;
   else
      return bin(n-1, k - 1)+bin(n - 1, k);
}
```

Number of terms computed by recursive bin

Like Algorithm 1.6 (nth Fibonacci Term, Recursive), this algorithm is very inefficient. In the exercises you will establish that the algorithm computes

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

terms to determine $\binom{n}{k}$.



Dynamic Programming Solution to the Binomial Coefficient Problem

The steps for constructing a dynamic programming algorithm for this problem are as follows:

1. *Establish* a recursive property. This has already been done in Equality 3.1. Written in terms of B, it is

$$B\binom{i}{j} = B[i][j] = \begin{cases} B[i-1][j-1] + B[i-1][j] & 0 < j < i \\ 1 & j = 0 \quad \text{or} \quad j = i. \end{cases}$$

2. Solve an instance of the problem in a *bottom-up* fashion by computing the rows in B in sequence starting with the first row.

- At each iteration, the values needed for that iteration have already been computed

Example

Compute $B[4][2] = \binom{4}{2}$.

Compute row 0: {This is done only to mimic the algorithm exactly.}
{The value $B[0][0]$ is not needed in a later computation.}

$$B[0][0] = 1$$

Compute row 1:

$$B[1][0] = 1$$
$$B[1][1] = 1$$

Compute row 2:

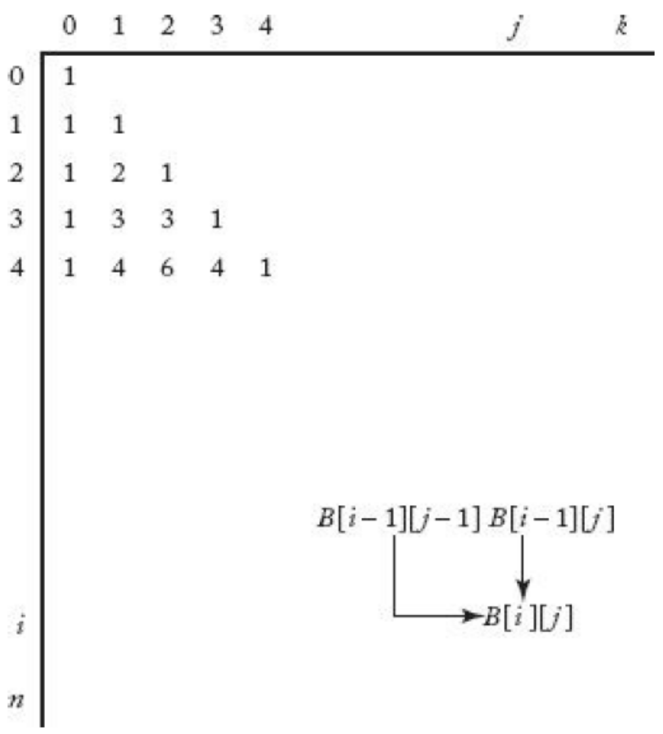
$$B[2][0] = 1$$
$$B[2][1] = B[1][0] + B[1][1] = 1 + 1 = 2$$
$$B[2][2] = 1$$

Compute row 3:

$$B[3][0] = 1$$
$$B[3][1] = B[2][0] + B[2][1] = 1 + 2 = 3$$
$$B[3][2] = B[2][1] + B[2][2] = 2 + 1 = 3$$

Compute row 4:

$$B[4][0] = 1$$
$$B[4][1] = B[3][0] + B[3][1] = 1 + 3 = 4$$
$$B[4][2] = B[3][1] + B[3][2] = 3 + 3 = 6$$



Algorithm 3.2

Binomial Coefficient Using Dynamic Programming

Problem: Compute the binomial coefficient.

Inputs: nonnegative integers n and k, where $k \leq n$.

Outputs: *bin*2, the binomial coefficient $\binom{n}{k}$.

```
int bin2 (int n, int k)
{
  index i, j;
  int B[0..n][0..k];

  for (i = 0; i <= n; i++)
     for (j = 0; j <= minimum(i,k); j++)
        if (j == 0 || j == i)
           B[i][j] = 1;
        else
           B[i][j] = B[i-1][j-1] + B[i-1][j];
  return B[n][k];
}
```

Algorithm 3.2 Binomial Coefficient using Dynamic Programing

- The work done by bin2 as a function of n and k

for-j loop. The following table shows the number of passes for each value of i:

i	0	1	2	3	$\cdots$	k	$k+1$	$\cdots$	n
Number of passes	1	2	3	4	$\cdots$	$k+1$	$k+1$	$\cdots$	$k+1$

The total number of passes is therefore given by

$$1+2+3+4+\cdots+k+\underbrace{(k+1)+(k+1)\cdots+(k+1)}_{n-k+1 \text{ times}}.$$

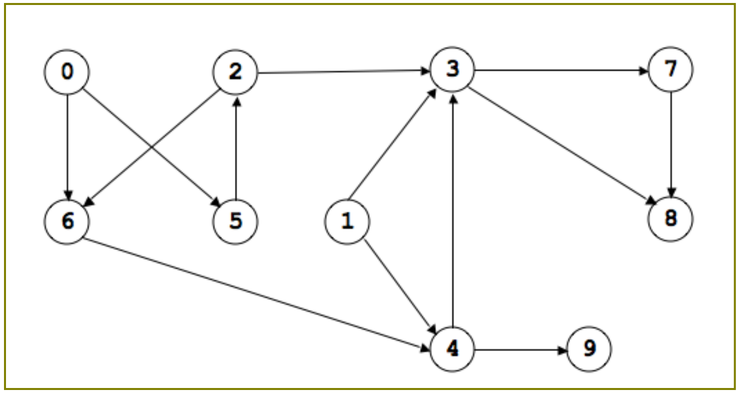
Applying the result in Example A.1 in Appendix A, we find that this expression equals

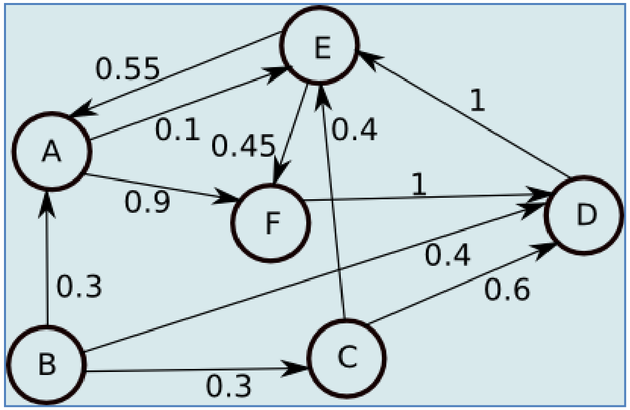
$$\frac{k(k+1)}{2}+(n-k+1)(k+1)=\frac{(2n-k+2)(k+1)}{2}\in\Theta(nk).$$

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

Graphs – an introduction



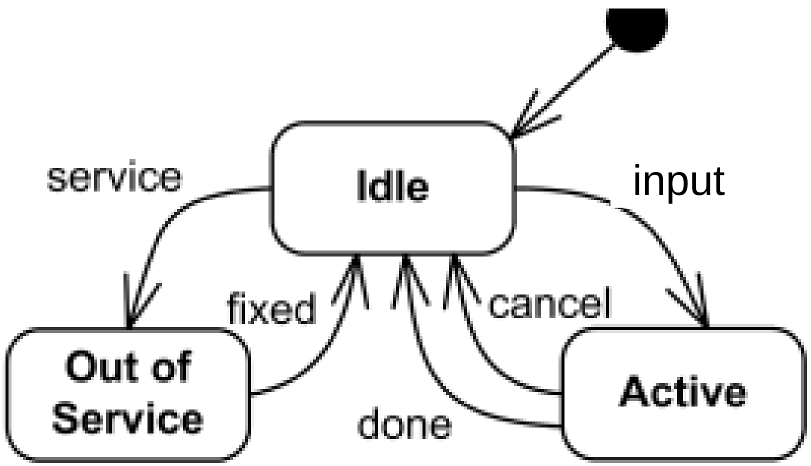


Aims

- General Terminology
  - Vertices, Edges, Connectedness, Path, Cycle etc.
- SubTypes
  - DiGraphs, Trees
- Representation techniques
  - Adjacency List, Adjacency Matrix

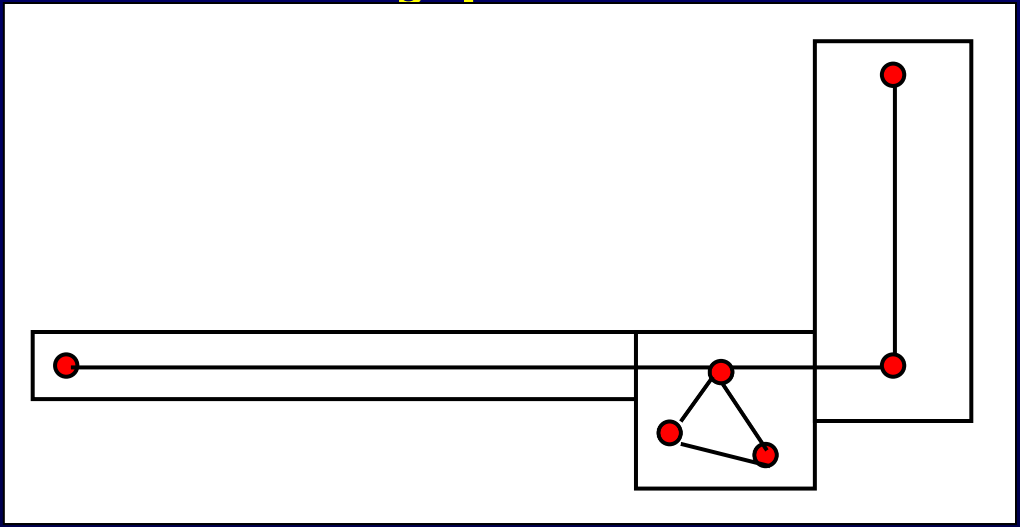
Introduction

- Graph Theory – discrete mathematics
- Used to represent relationships between objects, E.g.
  – route plan (places and connections), network, pipeline.
    - Cheapest way to lay cables between towns
  – state machines (states, transitions)



  – Game - board states (positions of pieces, possible moves)
    - Plan a strategy to win a game – e.g. find quickest checkmate

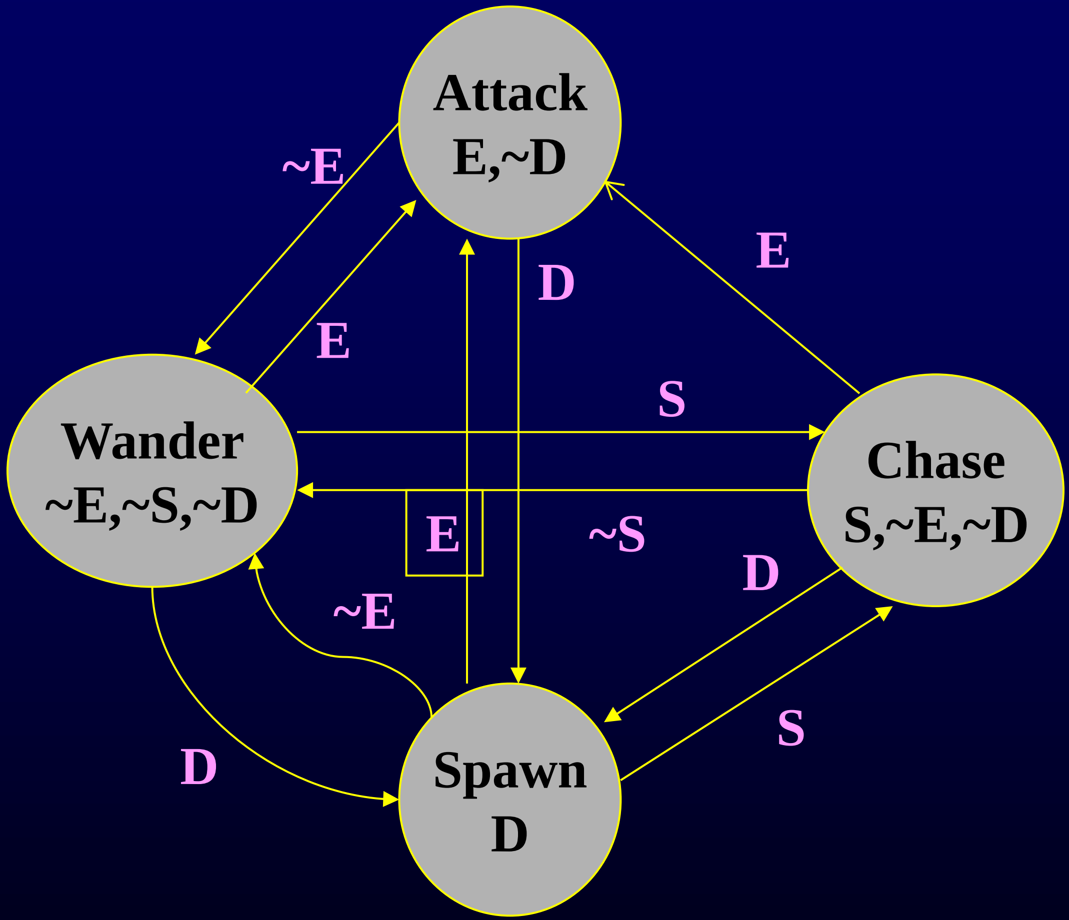
Waypoints



- Points in the graph/map.
- The points are connected to form a graph.
- To find a path through this map, you get to the closest point and then follow the lines to the target.

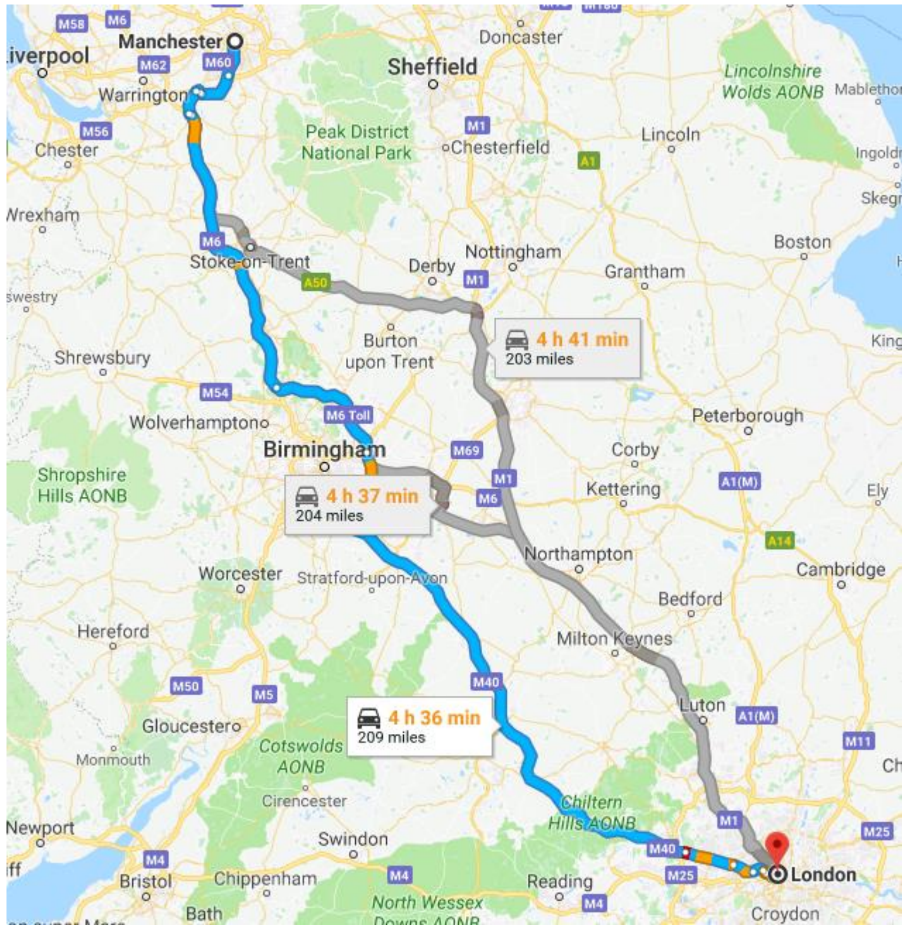
Finite State Machine - FSM

Example FSM



- States:
  - E: enemy in sight
  - S: sound audible
  - D: dead
- Events:
  - E: see an enemy
  - S: hear a sound
  - D: die
- Action performed:
  - On each transition
  - On each update in some states (e.g. attack)

Fastest Route



google map

Fastest Route

google map

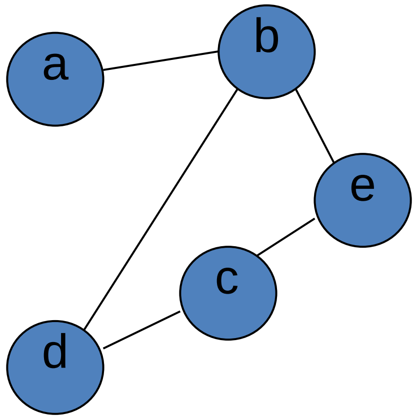
Graph: a set of points (cities) and lines connecting the points (roads)

Graphs and Terminology

- Graph G is the data structure specified by the pair $G = < V, E >$
  - V set of **vertices** (or nodes)
  - E set of **(unordered)** pairs on V called **edges** (or arcs) :
    - $(E \subset V \times V)$

e.g.

$G_1 = < a, b, c, d, e, (a,b), (b,e), (e,c), (c,d), (b,d)>$



- **Connectedness**

  If $(a,b) \in E$ then there is an edge between a and b;

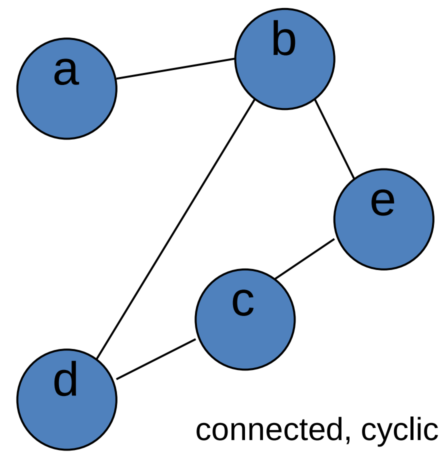
  a and b are **adjacent**;

  a and b are **connected**;

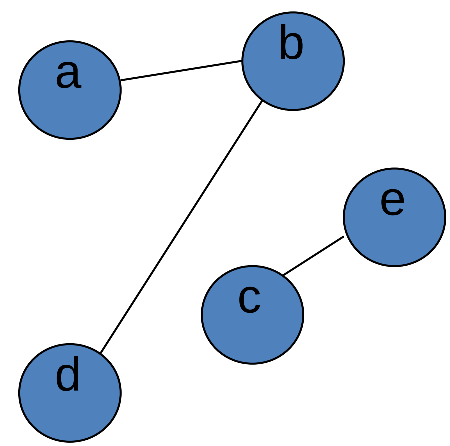
  there is a **path** between a and b.

Graphs and Terminology

- If there's a path from **a** to **b**, and a path from **b** to e, then there is a path from **a** to e.
  - vertices on path must be distinct (unless first = last)
- A graph G is **connected** if there is a **path** between any given pair of vertices.
  - Otherwise its an **unconnected** graph
- A path from a vertex to at least one other node and back to itself is a **closed path** or **cycle**.
- A graph G with at least one **cycle** is a **cyclic** graph.
  - Otherwise **acyclic**



connected, cyclic



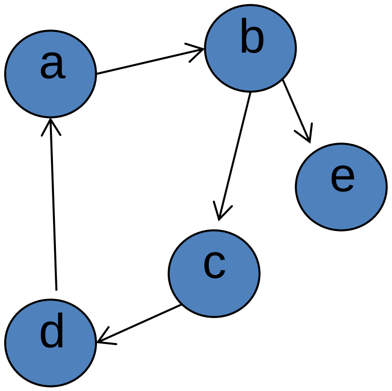
unconnected

Directed graph (digraph)

- A **Directed graph** G is the data structure specified by the pair G = < V, E >
  - V is the set of **vertices** (or nodes)
  - E is the set of **ordered** pairs on V
  - ($E \subset V \times V$ is a binary relation on V)

- If $(a,b) \in E$ then there is an edge from a to b
  - but not necessarily an edge from b to a.

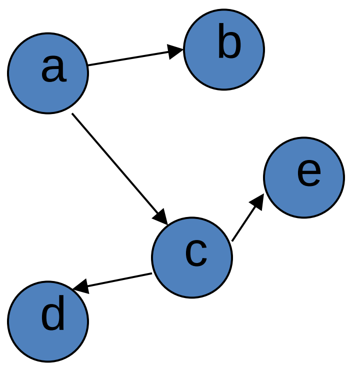
- On a diagram (digraph) an edge is represented by an arrow.

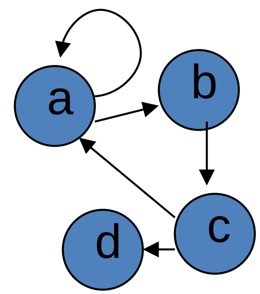


Directed graph (digraph)

- Draw digraphs
  1. G1=<a,b,c,d,e, (a,b),(a,c),(c,d),(c,e)>
  2. G2=<a,b,c,d, (a,a),(a,b),(b,c),(c,a),(c,d)>

  and comment on them.





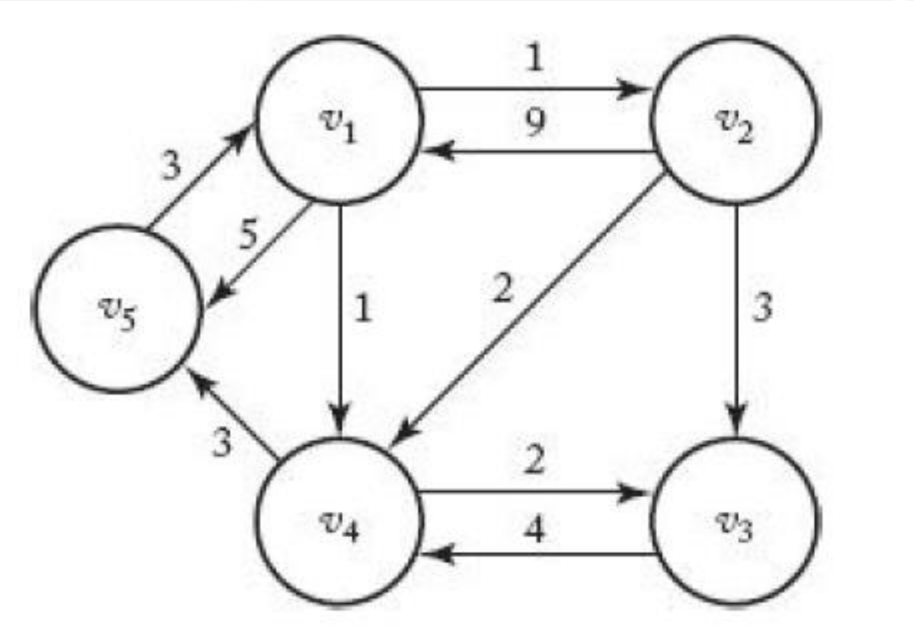
- Unconnected - no directed path between e,d
- Weakly connected – undirected paths for all pairs of nodes
- Acyclic
- Binary Tree
- Longest path = 3 nodes

- Connected
- Cyclic
- Trivial path
- No longest path - cyclic

Shortest Path Problem

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

Weighted Directed Graph



Simple path – never passes through the same vertex twice

Shortest path must be simple path !

Here are three simple paths from v_1 to v_3

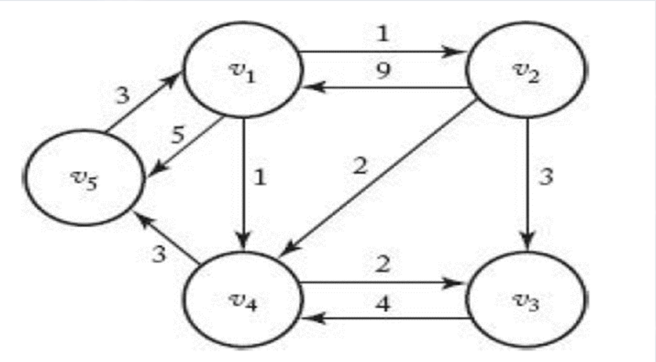
length[v1, v2, v3] = 1+ 3 = 4
length [v1, v4, v3] = 1 + 2 = 3
length [v1, v2, v4, v3] = 1 + 2 + 2 = 5

[v1, v4, v3] is the **shortest path** from v1 to v3

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

	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, where
  - **D^k [i,j]** = **length of the shortest path from v_i to v_j** using only vertices in the set $\{v_1, v_2, ... v_k\}$ as intermediate vertices

- D^n [i,j]= length of the shortest path from v_i to v_j
- D^0 [i,j]= the weight on the edge from v_1 to v_j
- We have established

$$D^0 = W \text{ and } D^n = D$$

Dynamic Programming Steps

To determine D from W, we need only find a way to obtain D^N from D^0 using the following two steps:

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

$$\underset{W}{D^0}, D^1, D^2, \ldots, \underset{D}{D^N}$$

We accomplish step 1 by considering 2 cases…

Establish a recursive Property

Two Cases to consider (details following two slides)

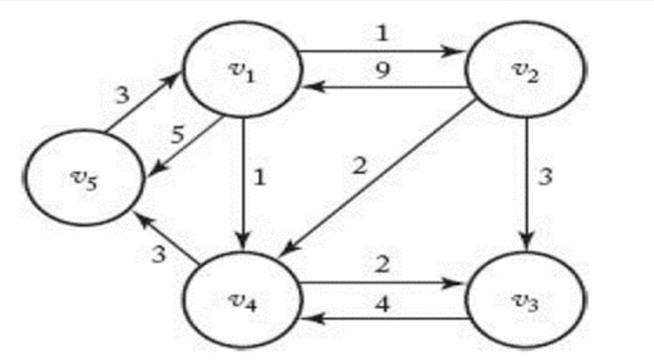
D^k [i,j] = minimum (case 1, case 2)

= 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 $\{v_1, v_2, \ldots, v_k\}$ as the intermediate vertex does not use v_k

  Then $D^k[i,j] = D^{(k-1)}[i,j]$

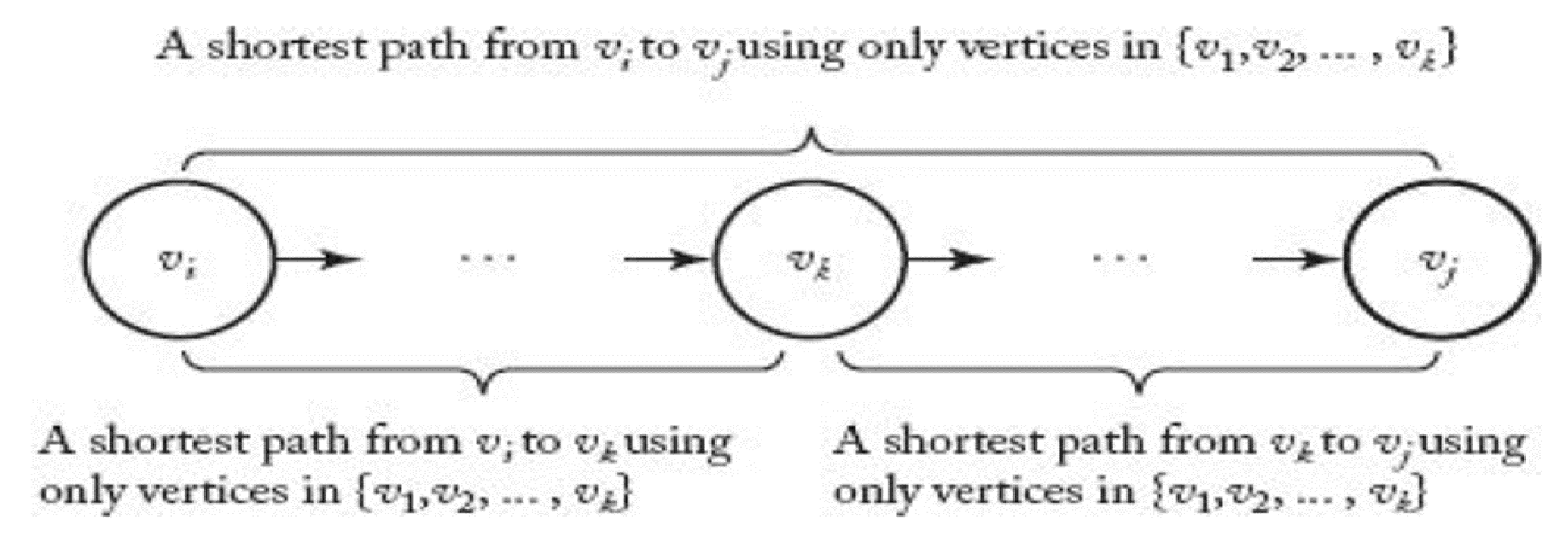


Example

$D^5[1,3] = D^4[1,3] = 3$, because when we include vertex v_5, the shortest path from v_1 to v_3 is still $[v_1, v_4, v_3]$.

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 vertices do use v_k

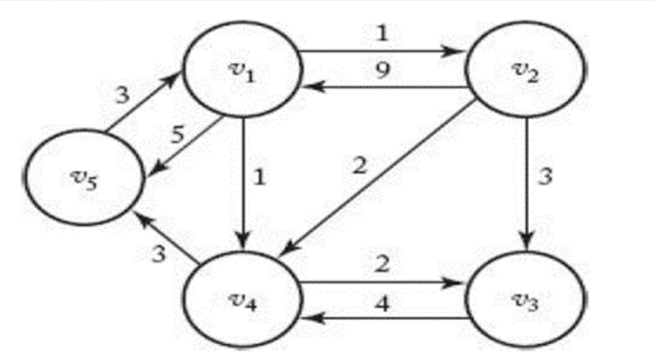


- Path = $v_i, \ldots, v_k, \ldots, v_j$ where $v_i, \ldots, v_k$ consists only of vertices in $\{v_1, v_2, \ldots, v_{k-1}\}$ as intermediates: **Cost of path = $D^{(k-1)}[i,k]$**
- And where $v_k, \ldots, v_j$ consists only of vertices in $\{v_1, v_2, \ldots V_{k-1}\}$ as intermediates: **Cost of path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

Case 2, cont.

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

- Therefore $D^{(k)}[i,j] = D^{(k-1)}[i,k] + D^{(k-1)}[k,j]$



Example

$D^2[5,3] = 7 = 4+3 = D^1[5,2] + D^1[2,3]$

Floyd's Algorithm for Shortest Paths – Algorithm 3.3

Floyd's Algorithm for Shortest Paths

Problem: Compute the shortest paths from each vertex in a weighted graph to each of the other vertices. The weights are nonnegative numbers.

Inputs: A weighted, directed graph and n, the number of vertices in the graph. The graph is represented by a two-dimensional array W, which has both its rows and columns indexed from 1 to n, where $W[i][j]$ is the weight on the edge from the ith vertex to the jth vertex.

Outputs: A two-dimensional array D, which has both its rows and columns indexed from 1 to n, where $D[i][j]$ is the length of a shortest path from the ith vertex to the jth vertex.

```
void floyd (int n
            const number W[][],
            number D[][])
{
  index i, j, k;
  D = W;
  for (k = 1; k <= n; k++)
    for (i = 1; i <= n; i++)
      for (j = 1; j <= n; j++)
        D[i][j] = minimum(D[i][j], D[i][k] + D[k][j]);
}
```

Basic operation: The instruction in the **for**-j loop.
Input size: n, the number of vertices in the graph.

$$T(n) = n \times n \times n = n^3 \in \Theta(n^3).$$

any
om

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 substances
- 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

Suppose we want to multiply a 2×2 matrix times a 3×4 matrix as follows:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} \times \begin{bmatrix} 7 & 8 & 9 & 1 \\ 2 & 3 & 4 & 5 \\ 6 & 7 & 8 & 9 \end{bmatrix} = \begin{bmatrix} 29 & 35 & 41 & 38 \\ 74 & 89 & 104 & 83 \end{bmatrix}$$

In general, to multiply an $i \times j$ matrix times a $j \times k$ matrix using the standard method, it is necessary to do

$$i \times j \times k \text{ elementary multiplications.}$$

Consider the multiplication of the following four matrices:

$$\begin{array}{ccccccc} A & \times & B & \times & C & \times & D \\ 20 \times 2 & & 2 \times 30 & & 30 \times 12 & & 12 \times 8 \end{array}$$

$$\begin{array}{ll} A(B(CD)) & 30 \times 12 \times 8 + 2 \times 30 \times 8 + 20 \times 2 \times 8 = 3,680 \\ (AB)(CD) & 20 \times 2 \times 30 + 30 \times 12 \times 8 + 20 \times 30 \times 8 = 8,880 \\ A((BC)D) & 2 \times 30 \times 12 + 2 \times 12 \times 8 + 20 \times 2 \times 8 = 1,232 \\ ((AB)C)D & 20 \times 2 \times 30 + 20 \times 30 \times 12 + 20 \times 12 \times 8 = 10,320 \\ (A(BC))D & 2 \times 30 \times 12 + 20 \times 2 \times 12 + 20 \times 12 \times 8 = 3,120 \end{array}$$

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

Chained-Matrix Multiplication

Suppose we have the following six matrices:

$$\begin{array}{ccccccccccc} A_1 & \times & A_2 & \times & A_3 & \times & A_4 & \times & A_5 & \times & A_6 \\ 5\times 2 & & 2\times 3 & & 3\times 4 & & 4\times 6 & & 6\times 7 & & 7\times 8 \\ d_0 \quad d_1 & & d_1 \quad d_2 & & d_2 \quad d_3 & & d_3 \quad d_4 & & d_4 \quad d_5 & & d_5 \quad d_6 \end{array}$$

To multiply A_4, A_5, and A_6, we have the following two orders and numbers of elementary multiplications:

$$\begin{aligned} (A_4A_5)\,A_6 \text{ Number of multiplications} &= d_3 \times d_4 \times d_5 + d_3 \times d_5 \times d_6 \\ &= 4\times 6\times 7 + 4\times 7\times 8 = 392 \\ A_4\,(A_5A_6) \text{ Number of multiplications} &= d_4 \times d_5 \times d_6 + d_3 \times d_4 \times d_6 \\ &= 6\times 7\times 8 + 4\times 6\times 8 = 528 \end{aligned}$$

Therefore,

$$M\,[4]\,[6] = minimum\,(392, 528) = 392.$$

M[i,j] = minimum number of multiplications needed to multiply A_i through A_j

Chained-Matrix Multiplication

- Principle of Optimality applies

The optimal order for multiplying six matrices must have one of these factorizations:

1. $A_1 (A_2A_3A_4A_5A_6)$
2. $(A_1A_2) (A_3A_4A_5A_6)$
3. $(A_1A_2A_3) (A_4A_5A_6)$
4. $(A_1A_2A_3A_4) (A_5A_6)$
5. $(A_1A_2A_3A_4A_5) A_6$

number of multiplications for the kth factorization is the minimum number needed to obtain each factor plus the number needed to multiply the two factors. This means that it equals

$$M[1][k] + M[k+1][6] + d_0 d_k d_6.$$

We have established that

$$M[1][6] = \underset{1 \le k \le 5}{minimum}(M[1][k] + M[k+1][6] + d_0 d_k d_6).$$

When multiplying n matrices, then for $1 \le i \le j \le n$

$$M[i][j] = \underset{i \le k \le j-1}{minimum}(M[i][k] + M[k+1][j] + d_{i-1} d_k d_j), \text{ if } i < j.$$

$$M[i][i] = 0.$$

Chained-Matrix Multiplication

The steps in the dynamic programming algorithm follow

Compute diagonal 0:

$$M[i][i] = 0 \qquad \text{for } 1 \le i \le 6.$$

Compute diagonal 1:

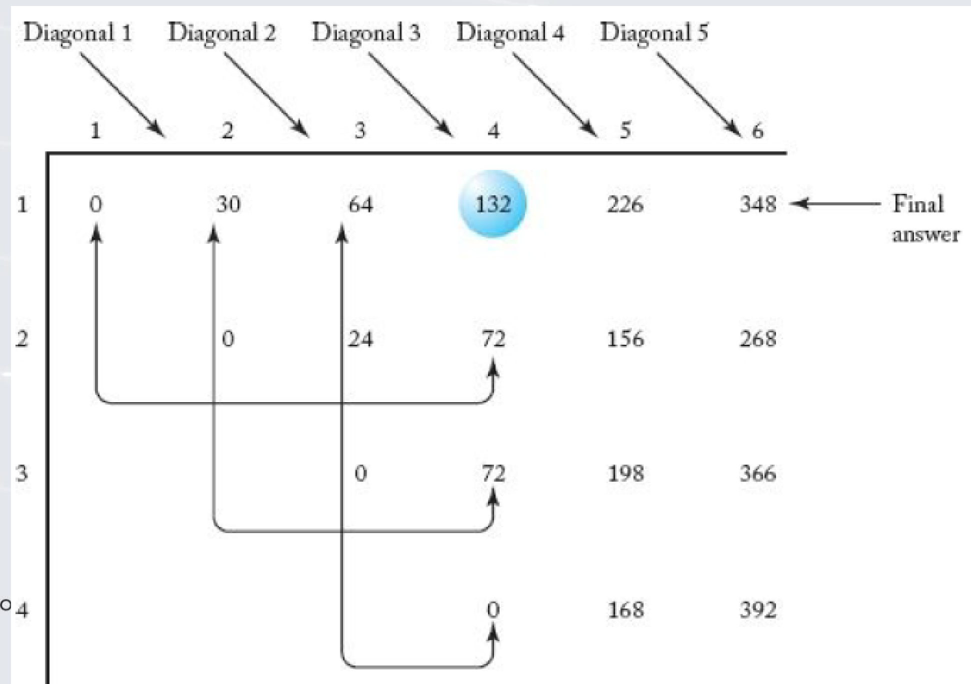
$$\begin{aligned} M[1][2] &= \underset{1 \le k \le 1}{minimum}(M[1][k] + M[k+1][2] + d_0 d_k d_2) \\ &= M[1][1] + M[2][2] + d_0 d_1 d_2 \\ &= 0 + 0 + 5 \times 2 \times 3 = 30. \end{aligned}$$

Compute diagonal 2:

$$\begin{aligned} M[1][3] &= \underset{1 \le k \le 2}{minimum}(M[1][k] + M[k+1][3] + d_0 d_k d_3) \\ &= minimum(M[1][1] + M[2][3] + d_0 d_1 d_3, \\ &\qquad M[1][2] + M[3][3] + d_0 d_2 d_3) \\ &= minimum(0 + 24 + 5 \times 2 \times 4, 30 + 0 + 5 \times 3 \times 4) = 64. \end{aligned}$$

Compute diagonal 3:

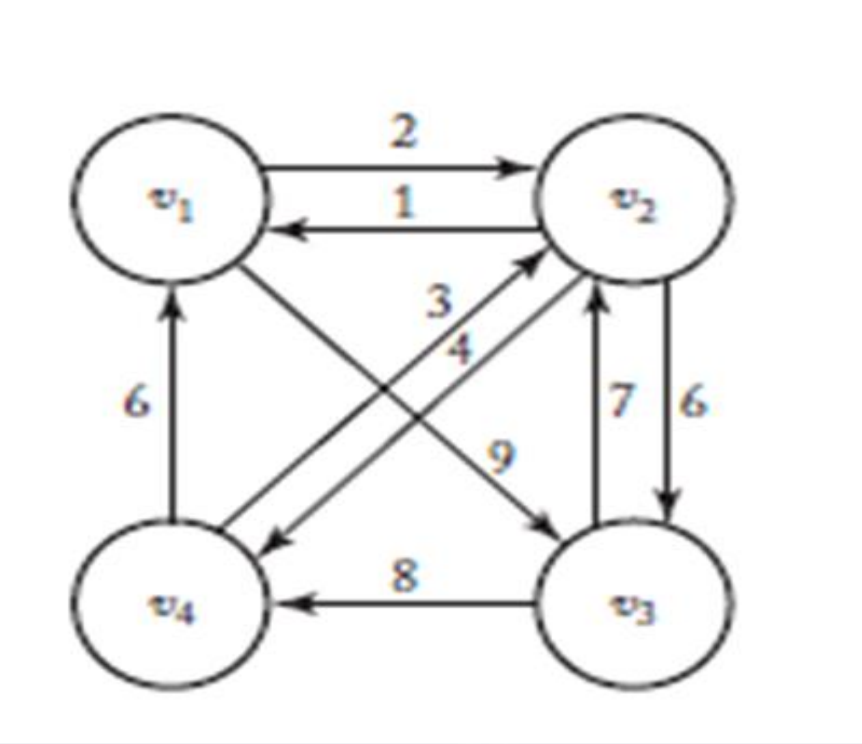
$$\begin{aligned} M[1][4] &= \underset{1 \le k \le 3}{minimum}(M[1][k] + M[k+1][4] + d_0 d_k d_4) \\ &= minimum(M[1][1] + M[2][4] + d_0 d_1 d_4, \\ &\qquad M[1][2] + M[3][4] + d_0 d_2 d_4, \\ &\qquad M[1][3] + M[4][4] + d_0 d_3 d_4) \\ &= minimum(0 + 72 + 5 \times 2 \times 6, 30 + 72 + 5 \times 3 \times 6, \\ &\qquad 64 + 0 + 5 \times 4 \times 6) = 132. \end{aligned}$$



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

Example



Tour (Hamiltonian circuit) in a directed graph – path from a vertex to itself that passes through each of the other vertices only once

Optimal tour in a weighted, directed graph – path of minimum length

$$length[v_1, v_2, v_3, v_4, v_1] = 22$$
$$length[v_1, v_3, v_2, v_4, v_1] = 26$$
$$length[v_1, v_3, v_4, v_2, v_1] = 21$$

Dynamic Programming Algorithm for Traveling Salesperson Problem

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

Dynamic Programming vs Divide-and-Conquer

- DP is an *optimization* technique and is applicable only to problems with *optimal substructure*.
  D&C is not normally used to solve optimization problems.
- Both DP and D&C split the problem into parts, find solutions to the parts, and combine them into a solution of the larger problem.
  - In D&C, the subproblems are *significantly smaller* than the original problem (e.g. half of the size, as in MERGE-SORT) and “do not overlap” (i.e. they do not share sub-subproblems).
  - In DP, the subproblems are not significantly smaller and are overlapping.
- In D&C, the dependency of the subproblems can be represented by a tree. In DP, it can be represented by a directed path from the smallest to the largest problem (or, more accurately, by a *directed acyclic graph*, as we will see later in the course).