Lecture 6.1

- Dr. Anurag Sharma

Dynamic Programming

About the topic

- Dynamic programming is similar to the divide-and-conquer approach in that an instance of a problem is divided into smaller instances.
- In dynamic programming, we solve the small instances first, store the results, and look them up when we need them instead of recomputing them.

Cont.

- Dynamic programming is a bottom-up approach since the solution is constructed from the bottom up in the array.
- There are two steps in the development of this approach.
  - Establish the recursive property that gives the solution to an instance of the problem.
  - Solve an instance of the problem in a bottom-up fashion by solving smaller instances first.

Example

- The Binomial coefficient

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

- We cannot compute the binomial coefficient directly from this definition because $n!$ is very large, even for moderate values of n.
- Solution?
  - Eliminate the need to compute $n!$ or $k!$ by using the recursive property.

Recursive binomial coefficient

- $\binom{n}{k} = \begin{cases} \binom{n-1}{k-1} + \binom{n-1}{k}, & 0 < k < n \\ 1\,, & k = 0 \text{ or } k = n \end{cases}$
- Problem solved?
- Same instances are being solved in each recursion.
- Solve $\binom{4}{2}$
- Remember Fibonacci (and worst case complexities)?
  - It is always inefficient when an instance is divided into almost as large as original instance using D&C approach.

DP version of Binomial Coefficient

- These kinds of problems can be developed in a more efficient way using dynamic programming.
  - Establish the recursive property that gives the solution to an instance of the problem.

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

  - Solve an instance of the problem in a bottom-up fashion by solving smaller instances first.
    - How? See next slide.

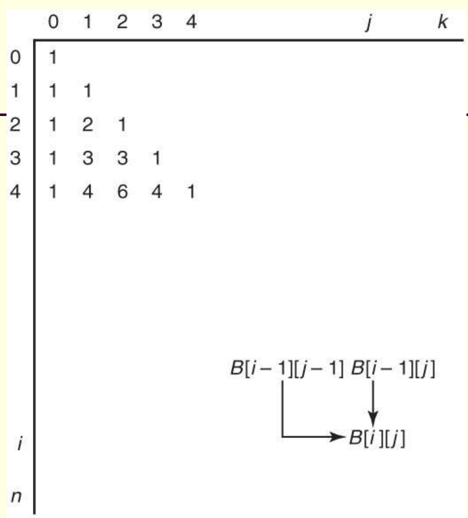


Figure 3.1: The array B used used to compute the binomial coefficient.

Algorithm for Binomial Coefficient

▶ Algorithm 3.2 **Binomial Coefficient Using Dynamic Programming**

Problem: Compute the binomial coefficient.

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

Outputs: $bin2$, 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];
}
```

What is T(n) for binomial coeff.?

- Look at the algorithm and analyze number of computations needed.
- The inner loop would require minimum of following values
- $T(n) = min(1,k) + min(2,k) + \cdots + min(k,k) + min(k+1,k) + \cdots + min(n,k)$
- $T(n) = 1 + 2 + \cdots + k + \sum_{i=1}^{n-k+1} k$
- $T(n) = k\frac{(k+1)}{2} + k(n-k+1)$
- $T(n) = \frac{1}{2}k^2 + \frac{1}{2}k + nk - k^2 + k$
- $T(n) = nk - \frac{1}{2}k^2 + \frac{3}{2}k$
- Since, $nk - \frac{1}{2}k^2 + \frac{3}{2}k \le nk + 2k^2 \le nk$
- $\therefore$ Big O order would be $O(nk)$

Example – 2

- Floyd's Algorithm for Shortest Path
  - To understand this let us first review graph theory.

Graph Theory

- In a pictorial representation of a graph, circles represent **vertices**, and a line from one circle to another represents an **edge** (sometimes also called an arc).
- If each edge has a direction associated with it, the graph is called a **directed graph**, or **digraph**.
- If the edges have values associated with them, the values are called **weights**, and the graph is called a **weighted graph**.

Cont.

- A **path** is a sequence of adjacent vertices in a graph, while a simple-path is a path but with distinct vertices (that is you can not pass through the same vertex twice).
- A **cycle** is a simple path with three or more vertices such that the last is adjacent to the first. A graph is said to be **acyclic** if it has no cycles and cyclic if it has one or more cycles.
- A **path** is called **simple** if it never passes through the same vertex twice. The **length** of a path in a weighted graph is the sum of the weights on the path. In an unweighted graph, it is the number of edges in the path.

A weighted, directed graph

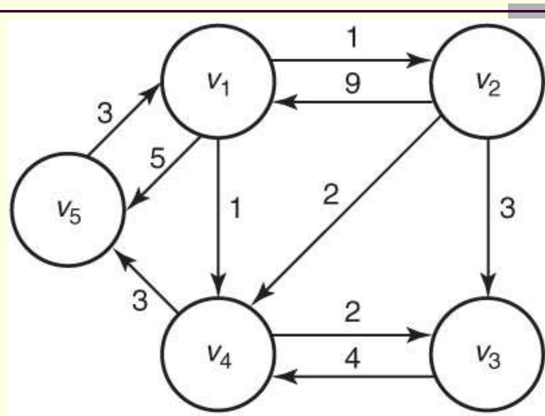


Figure 3.2: A weighted, directed graph.

Shortest Path Problem

- A problem that has many applications is finding the **shortest path** from each vertex to all other vertices
- Examples: Google map, telecommunication, & networking, Airline flight times etc.
- A shortest path must be a simple path
- The Shortest Paths problem is an **optimization problem**

Optimization Problem

- There can be more than one candidate solution to an instance of an optimization problem.
- Each candidate solution has a value associated with it, and a solution to the instance is any candidate solution that has an optimal value.
- Depending on the problem, the optimal value is either the maximum or minimum of these lengths.

Find shortest path

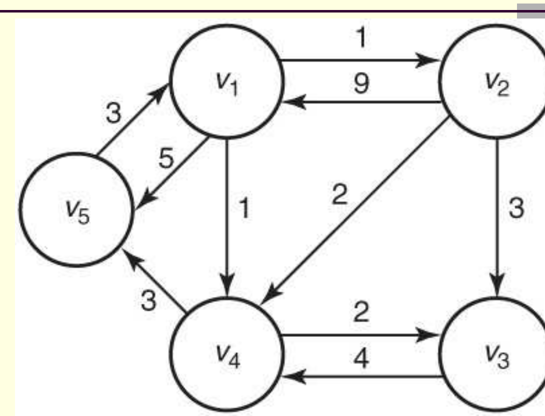


Figure 3.2: A weighted, directed graph.

	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

Figure 3.3: W represents the graph in figure 3.2 and D contains the lengths of the shortest paths. Our algorithm for the Shortest Paths problem computes the values in D from those in W.

Some examples

We will calculate some exemplary values of $D^{(k)}[i][j]$ for the graph in Figure 3.2.

$$D^{(0)}[2][5] = length[v_2, v_5] = \infty.$$

$$D^{(1)}[2][5] = minimum(length[v_2, v_5], length[v_2, v_1, v_5]) = minimum(\infty, 14) = 14.$$

$D^{(2)}[2][5] = D^{(1)}[2][5] = 14.$ {For any graph these are equal because a} {shortest path starting at v_2 cannot pass } {through v_2.}

$D^{(3)}[2][5] = D^{(2)}[2][5] = 14.$ {For this graph these are equal because} {including v_3 yields no new paths} {from v_2 to v_5.}

Shortest path formulation

- **Case 1:** At least one shortest path from v_i to v_j using only vertices in $\{v_1, v_2, \ldots, v_{k-1}\}$ as intermediate vertices;
  - Thus, $D^k[i][j] = D^{k-1}[i][j]$
- **Case 2:** All shortest paths from v_i to v_j using only vertices in $\{v_1, v_2, \ldots, v_k\}$ as intermediate vertices;
  - Thus, $D^k[i][j] = D^{k-1}[i][k] + D^{k-1}[k][j]$

"The principle of optimality states that an optimal solution to an instance of a problem always contains optimal solutions to all sub-instances."

Therefore:

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

It means, either the shortest path goes through k or without k.

Cont.

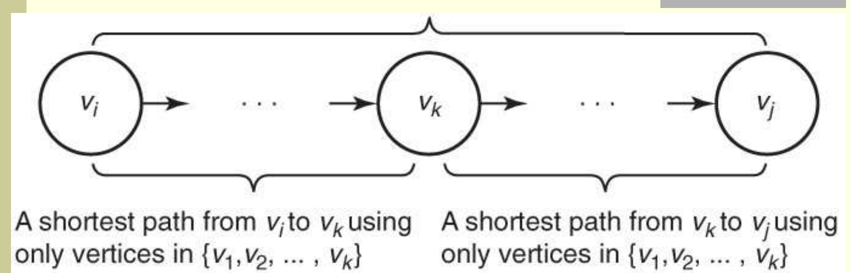


Figure 3.4: The shortest path uses vk

Floyd's Algorithm for Shortest Path

Input: n — number of vertices
a — adjacency matrix
Output: Transformed a that contains the shortest path lengths

for $k \leftarrow 0$ to $n-1$
  for $i \leftarrow 0$ to $n-1$
    for $j \leftarrow 0$ to $n-1$
      $a[i,j] \leftarrow \min(a[i,j],\ a[i,k]+a[k,j])$
    endfor
  endfor
endfor

Time complexity of FA?

- $T(n) = ?$
- $T(n) = n * n * n = n^3$
- What would be $T(n)$ of D&C version of Floyd's algorithm? Better or worse?

Wait! Where is the shortest path?

```
void floyd2 (int n,
             const number W[][],
                   number D[][],
                   index  P[][])
{
  index i, j, k;

  for (i = 1; i <= n; i++)
     for (j = 1; j <= n; j++)
        P[i][j] = 0;
  D = W;
  for (k = 1; k <= n; k++)
     for (i = 1; i <= n; i++)
        for (j = 1; j <= n; j++)
           if (D[i][k] + D[k][j] < D[i][j]){
               P[i][j] = k;
               D[i][j] = D[i][k] + D[k][j];
           }
}
```

	1	2	3	4	5
1	0	0	4	0	4
2	5	0	0	0	4
3	5	5	0	0	4
4	5	5	0	0	0
5	0	1	4	1	0

What is the shortest path from 5 to 3?

- Look at the path table:
- 5 – 3
- 5 – (4) – 3 ($v_k =\ 4$)
- 5 – (1) – 4 – 3
- => 3+1+2 = 6

Lecture 6.2

- Dr. Anurag Sharma

Dynamic Programming (TSP)

Travelling Salesman Problem

- Suppose a salesperson is planning a sales trip that includes 20 cities. Each city is connected to some of the other cities by a road.
- To minimize travel time, we want to determine a shortest route that starts at the salesperson's home city, visits each of the cities once, and ends up at the home city.
- This problem of determining a shortest route is called the **Traveling Salesperson problem (TSP)**.

TSP

- An instance of this problem can be represented by a weighted graph, in which each vertex represents a city.
- A tour in a directed graph is a path from a vertex to itself that passes through each of the other vertices only once.
- An optimal tour in a weighted, directed graph is such a path of minimum length.
- No one has ever found an algorithm for the Traveling Salesperson problem whose worst-case time complexity is better than exponential. Yet, no one has ever proved that the algorithm is not possible.

Principle of optimality

- "The principle of optimality states that an optimal solution to an instance of a problem always contains optimal solutions to all sub-instances."
- In case of shortest path problem if v_k is a vertex on an optimal path from v_i to v_j, then the subpaths* from v_i to v_k and from v_k to v_j must also be optimal. [*with all same nodes.]

Solve TSP?

- The TSP is to find an optimal tour when at least one tour exists
- One method is to apply the **Brute-force** approach – i.e. start with one city and consider each remaining city in turn, but this will yield a factorial time!
- However, **dynamic programming** can also be applied to this problem.
  - Use DP paradigm and principle of optimality to divide the problem using bottom up approach.

DP for TSP

- If v_k is the first vertex after v_1 on an optimal tour, the subpath of that tour from v_k to v_1 must be a shortest path from v_k to v_1 that passes through each of the other vertices exactly once
- Let

  W = adjacency matrix for a graph

  V = set of all the vertices

  A = a subset of V
- $D[vi]\ [A]$ = length of shortest path from v_i to v_1 passing through each vertex in A exactly once

Cont.

- $V - \{v_1, v_j\}$ contains all vertices except $v1$ and vj and since principle of optimality applies, length of an optimal tour =

$$\min_{2\le j\le n}(W[1][\,j\,] + D[v_j][V - \{v_1, v_j\}])$$

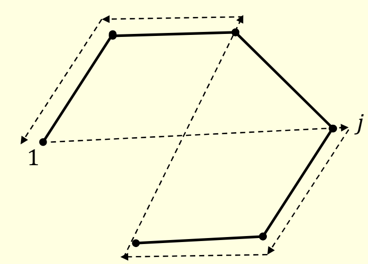
- In general for $i \neq 1$ and v_i not in A, $D[v_i][A]\ =$

$$\min_{j:\, v_j \in A}(W[i][j]\ +\ D[v_j][A - \{v_j\}],\text{ if } A \neq \emptyset$$

- $D[v_i][\emptyset] = W[i][1]$

Cont.

$$\min_{2\le j\le n}(W[1][\,j\,] + D[v_j][V - \{v_1, v_j\}])$$



Example

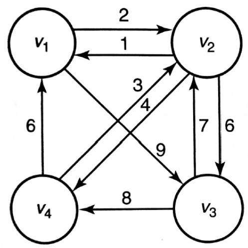


Figure 3.16 ● The optimal tour is $[v_1, v_3, v_4, v_2, v_1]$.

	1	2	3	4
1	0	2	9	∞
2	1	0	6	4
3	∞	7	0	8
4	6	3	∞	0

Figure 3.17 ● The adjacency matrix representation

Cont.

- $D[v_2][\phi] = 1$
- $D[v_3][\phi] = \infty$
- $D[v_4][\phi] = 6$

- $D[v_3][\{v_2\}] = 7 + 1 = 8$
- $D[v_4][\{v_2\}] = 4$

- $D[v_2][\{v_3\}] = 6 + \infty = \infty\ [v_2 - v_1 \Rightarrow v_2 - v_3 - v_1]$
- $D[v_4][\{v_3\}] = \infty$

- $D[v_2][\{v_4\}] = 4 + 6 = 10$
- $D[v_3][\{v_4\}] = 8 + 6 = 14\ \ [v_3 - v_1 \Rightarrow v_3 - v_4 - v_1]$

Cont.

- $D[v_4][\{v_2, v_3\}] = \min_{j:\, v_j \in \{v_2, v_3\}} \left(W[4][j] + D[v_j][\{v_2, v_3\} - \{v_j\}]\right)$

$= \min_{j:\, v_j \in \{v_2, v_3\}} \left((W[4][2] + D[v_2][\{v_2, v_3\} - \{v_2\}), (W[4][3] + D[v_3][\{v_2, v_3\} - \{v_3\}]\right)$

$= \min_{j:\, v_j \in \{v_2, v_3\}} \left((W[4][2] + \underline{D}[v_2][\{v_3\}]), (W[4][3] + \underline{D}[v_3][\{v_2\}])\right)$

$= \min(3 + \infty, \infty + 8) = \infty$

- And, $D[v_1][\{v_2, v_3, v_4\}] = \min_{j:\, v_j \in \{v_2, v_3, v_4\}} \left(W[1][j] + D[v_j][\{v_2, v_3, v_4\} - \{v_j\}]\right)$

$$= \min_{j:\, v_j \in \{v_2, v_3, v_4\}} \begin{pmatrix} W[1][2] + D[v_2][\{v_3, v_4\}, \\ W[1][3] + D[v_3][\{v_2, v_4\}, \\ W[1][4] + D[v_4][\{v_2, v_3\} \end{pmatrix}$$

$= \min(2 + 20, 9 + 12, \infty + \infty) = 21$

- ~~What is $D[v_3][\{v_2, v_4\}]$ pictorially?~~

Algorithm

```
void travel (int n,
             const number W[][],
             index P[][],
             number& minlength)
{
  index i, j, k;
  number D[1..n][subset of V − {v1}];

  for (i = 2; i <= n; i++)
     D[i][∅] = W[i][1];
  for (k = 1; k <= n − 2; k++)
     for (all subsets A ⊆ V − {v1} containing k vertices)
        for (i such that i ≠ 1 and vi is not in A){
           D[i][A] = minimum (W[i][j] + D[j][A − {vj}]);
                    j:vj ∈ A
           P[i][A] = value of j that gave the minimum;
        }
  D[1][V − {v1}] = minimum (W[1][j] + D[j][V − {v1, vj}]);
                   2 ≤ j ≤ n
  P[1][V − {v1}] = value of j that gave the minimum;
  minlength = D[1][V − {v1}];
}
```

Time complexity?

- $T(n) = ?$
- According to the for loops in the algorithm:
- First loop: $k = 1: \sim n$ i.e., n times
- Second loop: $\binom{n}{k}$ for every k
- Third loop: $\mathrm{n} - k \approx n$
- Roughly, $T(n) = (n) \sum_{i=1}^{n} k \binom{n}{k}$
- $T(n) = (n)n2^n = n^2 2^n$
- Big O order is $O(2^n)$ (better than $n!$)

What is $\sum_{k=0}^{n} k\binom{n}{k}$? (from https://math.stackexchange.com)

Since the binomial coefficients have the $n-k$ symmetry, we can put

$$\sum_{k=0}^{n}(n-k)\binom{n}{n-k}$$

thus

$$S_n = \sum_{k=0}^{n} k\binom{n}{k} = \sum_{k=0}^{n}(n-k)\binom{n}{n-k}$$

But the RHS is

$$n\sum_{k=0}^{n}\binom{n}{n-k} - \sum_{k=0}^{n} k\binom{n}{n-k}$$

Now

$$S_n = n\sum_{k=0}^{n}\binom{n}{k} - \sum_{k=0}^{n} k\binom{n}{k}$$

or

$$S_n = n\sum_{k=0}^{n}\binom{n}{k} - S_n$$

$$S_n = n2^n - S_n$$

$$2S_n = n2^n$$

$$S_n = n2^{n-1}$$