Dynamic Programming

- I consider this to be the most important part of the course
- Goal is to be able to apply this technique to new problems
- Key concepts need to be highlighted so that we can start to see the structure of dynamic programming solutions

Dynamic Programming

- The most important algorithmic technique covered in this course.
- Key ideas
  - Express solution in terms of a polynomial number of sub problems
  - Order sub problems to avoid recomputation

PRINCIPLE OF OPTIMALITY

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

Where Dynamic Programming may not apply…

- Does not apply to optimal use of limited resources.

- Here optimal solution of an instance not obtained by combining optimal solutions of 2 or more sub-instances, if resources used in the sub-solutions add up to more than the total resources available.

GREEDY v/s Dynamic Programming

- In Greedy only one decision sequence is ever generated.

- In Dynamic Programming many decision sequences may be generated

Divide and Conquer v/s Dynamic Programming

- In Divide and Conquer subproblems are disjoint.
- In Dynamic Programming, subproblems are overlapping.

0/1 Knapsack problem.

```
0/1knapsack()
{
        // v[1..n] w[1..n] : inputs
        //C[1..N][1..W] : output
   for j=1 to w
                if j<w[1]   C[1,j]=0;
                else           C[1,j]= v[i];
   for i=2 to n
        for j= 1 to w
                C[i,j] = max { C[i-1, j] , C[i-1][j-w[i]] +v[i]};
}
```

Multistage Graphs

```
MSG() // K stage Graph.
{
    //c[i,j] is the cost matrix, cost[1..n] and path[1..k] output
   cost[n]=0;
    for (j = n-1 to 1)
   {
          //Let ‘r’ be a vertex such that  <j,r> is an edge of G
          //and c[j,r]+cost[r] is minimum;
         cost[j]= c[j,r] + cost [r];
         next [j]=r;
    }
    path[1]=1; path[k]=n;
    for i= 2 to k-1    path[i]=next[path[i-1]];
}
```

COURSE OUTCOMES

1. Analyze the average- and worst-case performance of algorithms,
2. Write Effective algorithm for any given problem.
3. Use the various strategies effectively,
4. Apply the concept of NP-completeness and be familiar with approximation algorithms

OBST

```
OBST (p,q,n)
{
//Initializations
    for (i=0 to n)
          sop[i][i] = q[i]; root[i][i]=0; cost[i][i]=0.0;
           if (i<n)
                    cost[i][i+1]= sop[i][i+1]=q[i]+q[i+1]+p[i+1];
                     root[i][i+1]=i+1;
```

OBST

```
OBST (Cont'd)
{
for t=2 to n
   for i= 0 to n-t
        j=i+t;
        sop[i][j]=sop[i][j-1] + p[j] + q[j];
        k=root[i][j] = findmin (cost, root, i,j);
       cost[i][j] = sop[i][j] + cost [i,k-1] +c[k,j];
}
}
```

The sequence alignment problem.

Applications :

1. Dictionary (ocurrance, occurrence)
2. Biology (2 strands of DNA codes X and Y, need to find a certain type of toxin, similar substring)

X and Y

x1,x2, x3,….xm

y1, y2,y3,…yn

Matching is an alignment if there are no “crossing pairs”

i.e. (i,j) {i’,j’) ε M then i< i’ and j<j’.

Eg.

stops-

-tops-

- 2 Penalties

1. Gap penalty δ
2. Mismatch penalty α

Motivation to use Dynamic Programming solution

In the optimal arrangement M, either (m,n) ε M or (m,n) does not belong to M. (That is, either the last symbols in the two strings are matched to each other, or they are’nt)

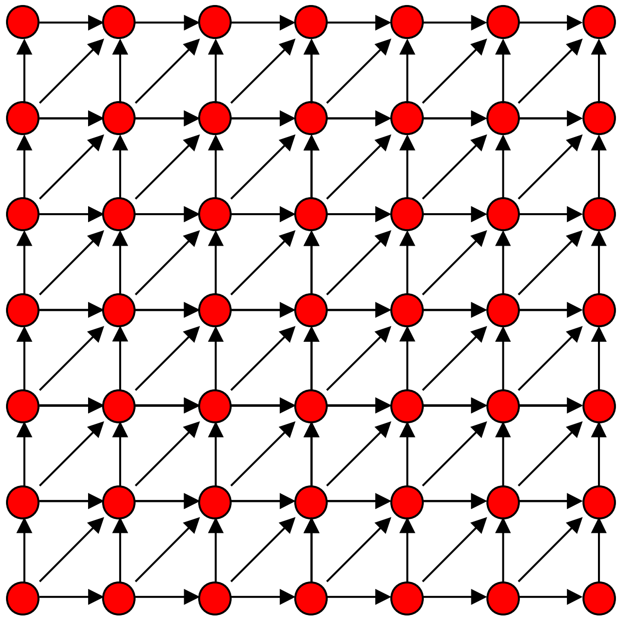
In an optimal arrangement M, atleast one of the following is true :

1. (m,n) ε M or
2. The mth position of X is not matched or
3. The nth position of Y is not matched.

$$
\begin{aligned}
opt(i,j) = \min(&\alpha_{xiyj} + opt(i-1,j-1), \\
&\delta + opt\ (i-1,j), \\
&\delta + opt\ (i.j-1)\ \ )
\end{aligned}
$$

```
Alignment (X,Y)
 Array A[0..m,0..n]
 Initialize A[i,0] = iδ for each i
 Initialize A[0,j] = jδ for each j
For j= 1 to n
     for i = 1 to m
            use equation to compute A[i,j]
```

Dynamic Programming Computation



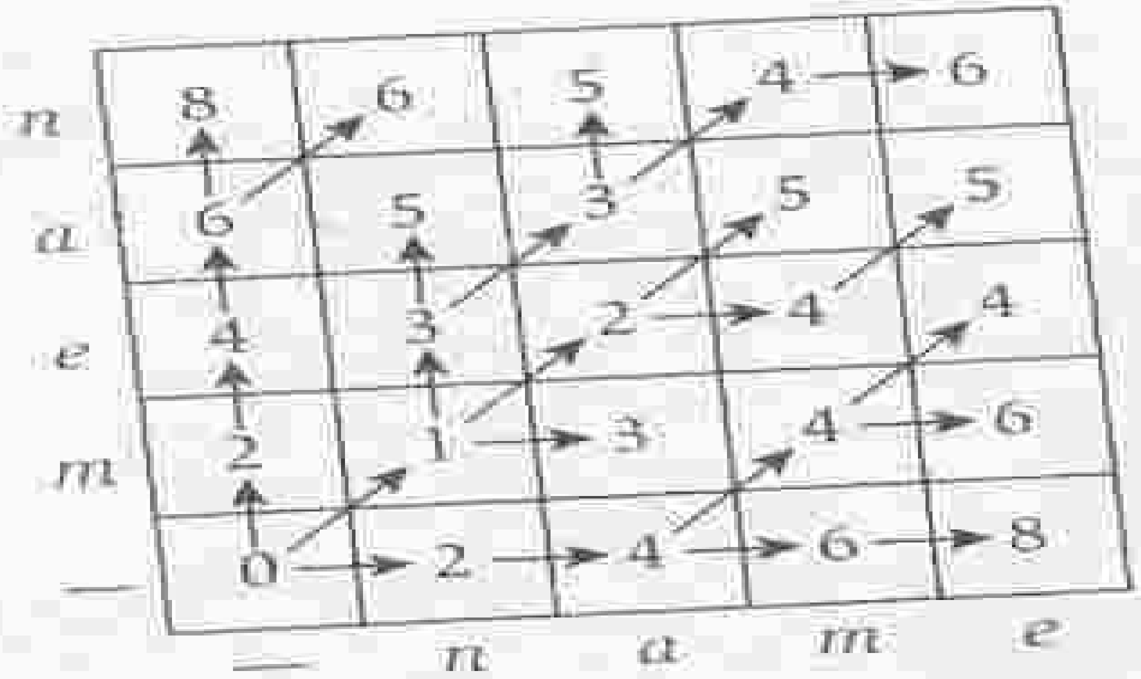


Figure 6.18 The OPT values for the problem of aligning the words *mean* to *name*.

in
a
c
c
i

```
Space Alignment problem (X,Y)
  Array B[0..m,0..1]
  Initialize B[i,0]=iδ for each I (Col 0 of A)
  For j = 1 to n
      B[0,1] = jδ
      for i = 1 to n
          B[i,1]= min ($\alpha_{xiyj}$ + B(i-1,0),
                δ + B (i-1,1), δ + B (i,0) )
      //Move col 1 of B to col 0 for next iteration
          Update B[i,0] = B[i,1] for each i
```

For i< m and j< n we have

$$g(i,j) = \min(\alpha_{x_{i+1}y_{j+1}} + opt(i+1,j+1),\ \delta + opt(i,j+1),\ \delta + opt(i+1,j))$$

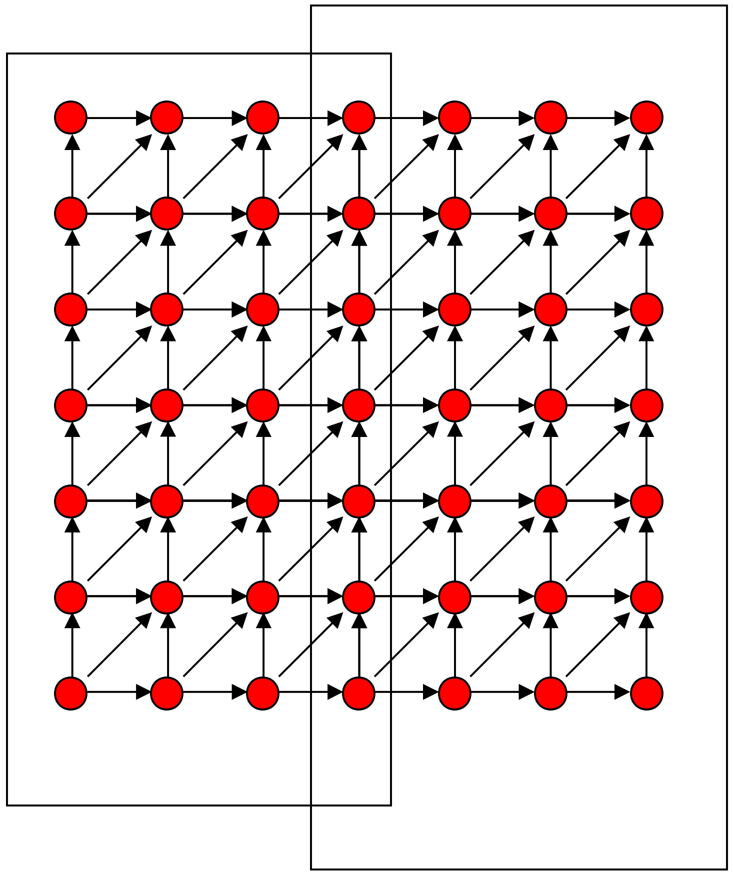
- The length of the shortest path in G_{XY} that passes through (i,j) is f(i,j) + g(i,j)

- Let k be any number in {0,….,n} and let q by an index that minimizes the quantity f(q,k) + g(q,k). Then there is a corner to corner path of minimum length that passes through node (q,k).

Divide and Conquer alignment (X,Y)

```
    Let m be the number of symbols in X
    Let n be the number of symbols in Y
    If m<=2 or n<=2 then
        Compute optimal alignment using Alignment(X,Y)
    Call Space-Efficient-Alignment (X,Y[1..n/2])
    Call Backward-Space-Efficient-Alignment(X,Y[n/2..n])
    Let q be index minimizing f(q,n/2) + g(q,n/2)
    Add (q,n/2) to the global list P
    Divide and Conquer alignment (X[1..q], Y[1..n/2])
    Divide and Conquer alignment (X[q..n],Y[n/2..n])
```

Dynamic Programming Computation



Dynamic Programming Computation