Lecture 4: Dynamic Programming

Analysis and Design of Computer Algorithms

GONG Xiu-Jun

School of Computer Science and Technology, Tianjin University

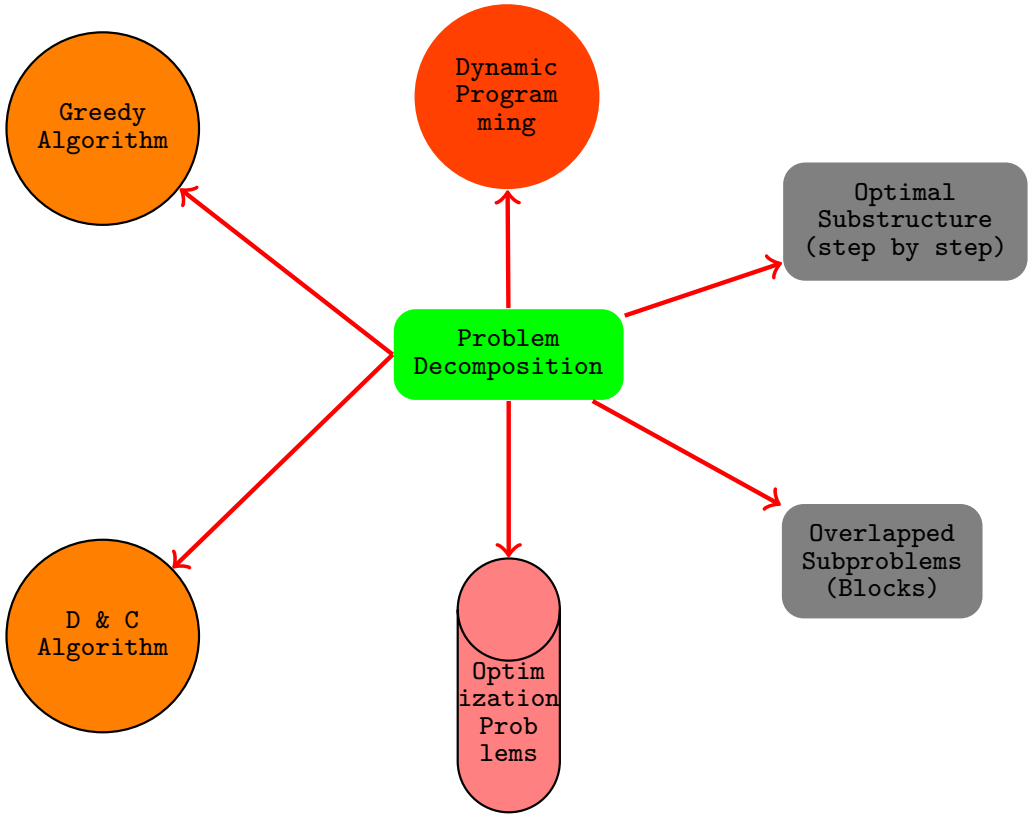
Email: gongxj@tju.edu.cn
Website: http://cs.tju.edu.cn/faculties/gongxj/course/algorithm
Discussion : http://groups.google.com/group/algorithm-practice-team

April 29, 2014

Dynamic Programming

- Aims and Objectives
  - Know the definition of DP
    - Optimal substructure and
    - Overlapped subproblems
  - prove DP applicable to a problem
  - Solving a real problem using DP
- Practice
  - 0/1 Knapsack
  - Matrix Multiplication Chains
  - All Pairs Shortest Paths
  - Maximum Non-crossing Subset nets
  - Longest Common Subsequence
  - Hidden Markov Model

Motivations

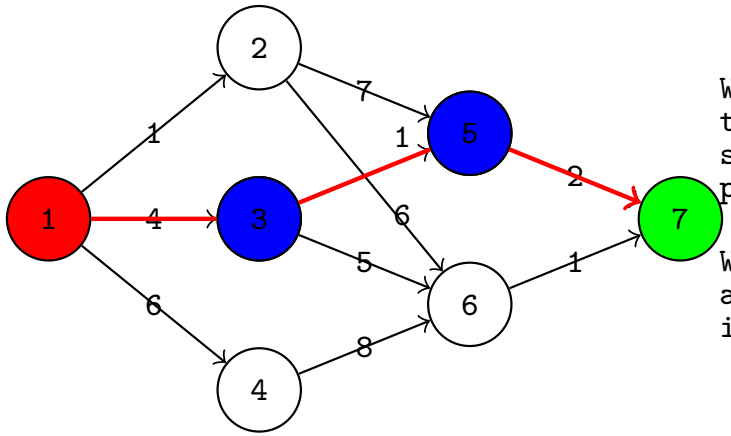


An exmple of task sheduling

In a weighted staged directed graph G, we want to search the shortest path from start node s to the terminal node e.

The sub-path from a node in the Global Shortest Path(GSP) to the terminal node is also shortest

For the node 1, whichever of its neighbor is in $\{2, 3, 4\}$, if it is in GSP, then sub-path of GSP from it to the terminal is also shortest



We call this property as the Optimal Substructure of subproblems in the procedure of problem decomposition.

We also should noted that there are some overlapped subproblems in seeking the shortest paths.

Define that $c(i)$ is the shortest distance from i to e, then

$c(e) = 0$ and $c(s)$ is the goal

$$c(i) = \min_{j \in N(i)} \{c(j) + cost(i, j)\}$$

where $N(i)$ is the set of neighbors of i

Solve $c(i)$ in reverse order

c(7) =0

c(6) =1, c(5) =2

c(4) =8+c(6)=9

$$c(3) = min\{1 + c(5), 5 + c(6)\}$$
$$= min\{3, 6\} = 3$$

c(2) =7, **c(1) =7**

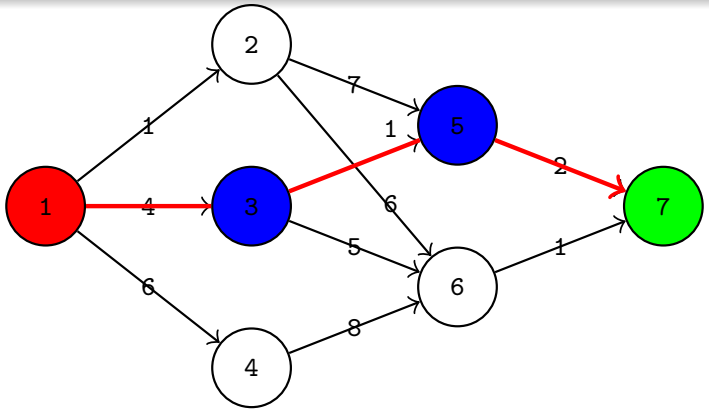
Definition

- The term dynamic programming was originally used in the 1940s by Richard Bellman (Bellman equation)
  - DP refers specifically to nesting smaller decision problems inside larger decisions.
  - Dynamic was chosen because it sounded impressive, not because it described how the method worked.
  - Programming referred to the use of the method to find an optimal programming.
- A method of solving complex problems by breaking them down into simpler steps (subproblems) in recursive manner.
  - the solution can be produced by combining solutions to subproblems;
  - the solution to each subproblem can be produced by combining solutions to sub-subproblems, etc;
  - the total number of subproblems arising recursively is polynomial.

Where DP appliable?

- Optimal substructure(Principle of Optimization)
  - A problem is said to have optimal substructure if an optimal solution can be constructed efficiently from optimal solutions to its subproblems.
  - In other words, a problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.
  - Subproblems must be only 'slightly' smaller than the larger problem
    - A constant additive factor
    - A multiplicative factor: D&C
- Overlapping subproblems
  - A problem said to have overlapping subproblems if the problem can be broken down into subproblems which are reused several times
- DP $\approx$ Recursion + Memorization

Procedures



1	Identifying subproblems	1-2 , 1-3 or 1-4 ?
2	Validating principle of optimization	What happen if not
3	Defining an optimal value function	Assumed that c(i) is the shortest distance from i to e
4	Deriving the recursive equation	$c(i) = \min_{j \in N(i)} \{ c(j) + cost(i,j) \}$
5	Solving the recursive equation	c(7) - c(6) - c(5) - c(4)- c(3) - c(2) -c(1)
6	Tracebacking the optimal solution	P(1) - P(3) -P(5) -P(7)

Probelm statements

Given a set of items, each with a mass and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit(capacity) and the total value is as large as possible.

Formal representation: Given $n > 0$, $c > 0$ and two n-tuples of positive numbers: $(w_1, w_2, \cdots, w_n)$ and $(p_1, p_2, \cdots, p_n)$.

we wish to determine the subset $T \subset \{1...n\}$ such that

$$\text{Maximize: } \sum_{i \in T} p_i$$

$$\text{Subjects to: } \sum_{i \in T} w_i \leq c$$

If used a n-tuple of indicator variables $(x_1, x_2, \cdots, x_n)$, then

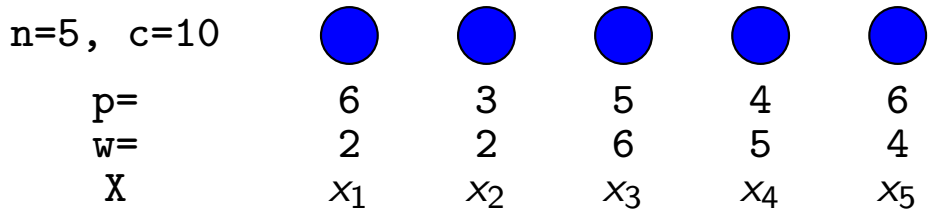
$$\text{Maximize: } \sum_{i=1}^{n} x_i * p_i$$

$$\text{Subjects to: } \sum_{i=1}^{n} x_i * w_i \leq c$$

Solutions

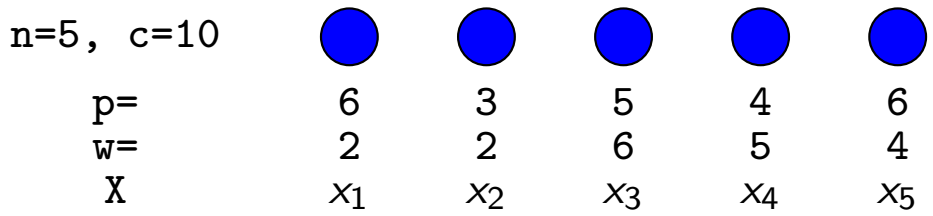
- Brute force:Try all 2^n possible subsets T
- Greedy
  - choose items maximizing value ?
  - choose items maximizing value/size
  - the second looks much great, but what if items dont exactly fit (non-divisible items)?
- any others?
  - traceback
  - branch and bound
  - ...
- **Dynamic programming** is one of great choice

Step 1. Identifying subproblems



- By fewer items: take an item i without consideration , what changed?
  - the number of items n becomes $n-1$, and
  - the capacities become c or $c-w_i$, why?
  - So the subproblem is constrained by two parameters.
- By smaller knapsack capacities,reduce the capacity, what changed?
  - the capacities become smaller c'
  - supposed that this is caused by removing an item i,$c' = c - w_i$, and thus the optimal value $f(c) = f(c-w_i) + v_i$
  - however, we donn't know which i,
  - $f(c) = \max_{i=1, w_i<c} f(c-w_i) + v_i$

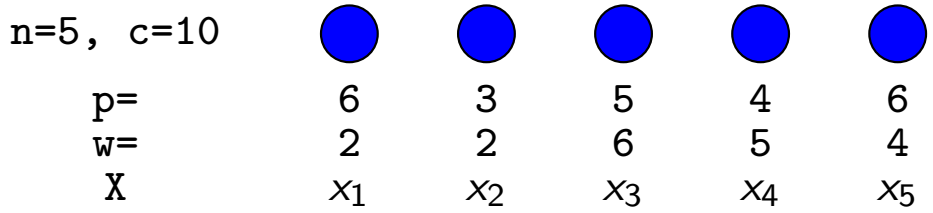
Step 2. Validating Principle of Optimization



[Optimal substructure] A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solutions to subproblems.

- Supposed that $(x_1, x_2, \cdots, x_n)$ is the optimal solution of original problem P^0.
- $(y_1, y_2, \cdots, y_{n-1})$ is the optimal solution of its subproblem P' with items $1, 2, \cdots, n-1$ and capacities $c - x_n * w_n$.
- We should show that $(y_1, y_2, \cdots, y_{n-1}, x_n)$ is no worse than $(x_1, x_2, \cdots, x_n)$ to P^0
- How? using a "cut-and-paste" technique (homework)

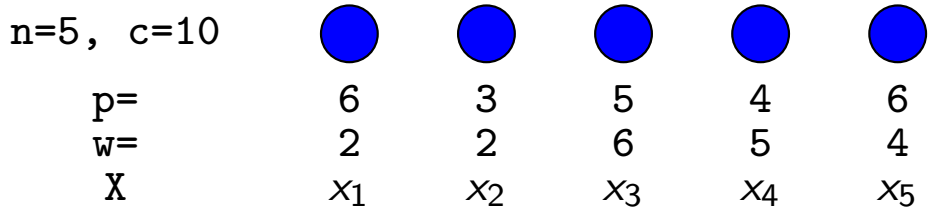
Step 3. Defining an optimal value function



We have known that a subproblem is constrained by two parameters: number of items and capacities.

- For a given subproblem with items $i, i+1, \cdots, n$ and capacities y, we define its optimal value by $f(i, y)$.
- $f(n, 0) = 0$, and $f(n, w_n) = p_n$ if $w_n < c$ else 0
- $f(1, c)$ is our goal, why?
- How to get $f(1, c)$ from $f(n, y)$?

Step 4. Deriving the recursive equation



To calculate $f(i, y)$ from $f(i+1, y)$, we only need know if item i could be included

- If $w_i > y$, item i couldn't be included, $f(i, y) = f(i+1, y)$.
- If $w_i \leq y$, item i might be included
  - if so, $f(i, y) = f(i+1, y - w_i) + p_i$
  - else, $f(i, y) = f(i+1, y)$, why?
  - finally, we use the larger one.

Step 5. Solve recursive equation

Now we have the recursive equation

$$f(i,y) = \begin{cases} f(i+1,y) & \text{if } 0 \le y < w_i \\ \max \begin{cases} f(i+1,y-w_i)+p_i \\ f(i+1,y) \end{cases} & \text{if } y \ge w_j \end{cases} \quad (1)$$

and the initial condition

$$f(n,y) = \begin{cases} p_n & \text{if } y \ge w_n \\ 0 & \text{if } 0 \le y < w_n \end{cases} \quad (2)$$

How to solve them?

- Recursive version
- Non-recursive version
- Tuple version

Recursive version

Algorithm 1: RKnapsack

Input: n,c,p[1..n],w[1..n]
Output: the optimal value $f(1, c)$

```
1 Function f(int i, int y)
2     if (i==n) then
3         return (y < w[n]?0 : p[n] );
4     if (y < w[i]) then
5         return f(i-1,y) ;
6     return max(f(i-1,y),f(i-1,y-w[i])+p[i]) ;
```

Algorithm complexity: We use $t(n)$ to denote the algorithm time complexity with n items

- $t(1) = a$
- At the best case(step4), $t(n) = t(n-1) + b$, so $t(n) = \Theta(n)$
- At the worst case(step6), $t(n) = 2t(n-1) + b$, so $t(n) = \Theta(2^n)$

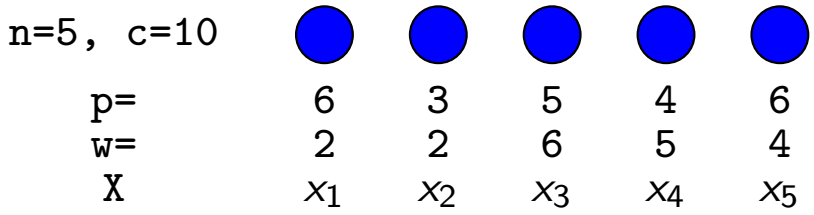
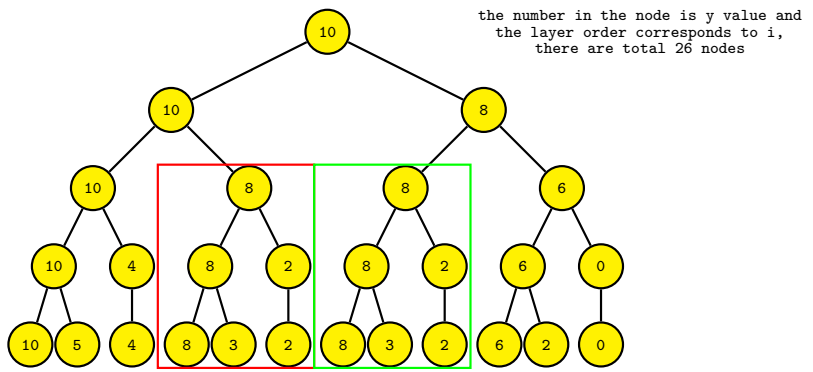


Figure : Recursive Call Relation Tree



If these repetitive calls are saved, the algorithm complexity should be reduced!

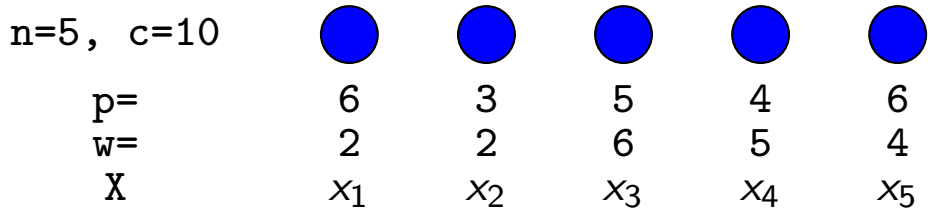
First attempt: using array

```
template<class T>
void Knapsack(T p[], int w[], int c, int n, T** f)
{// Compute f[i][y] for all i and y.

   // initialize f[n][]
   int yMax = min(w[n]-1,c);
   for (int y = 0; y <= yMax; y++)
      f[n][y] = 0;
   for (int y = w[n]; y <= c; y++)
      f[n][y] = p[n];

   // compute remaining f's
   for (int i = n - 1; i > 1; i--) {
      yMax = min(w[i]-1,c);
      for (int y = 0; y <= yMax; y++)
         f[i][y] = f[i+1][y];
      for (int y = w[i]; y <= c; y++)
         f[i][y] = max(f[i+1][y],
                       f[i+1][y-w[i]] + p[i]);
      }
   f[1][c] = f[2][c];
   if (c >= w[1])
      f[1][c] = max(f[1][c], f[2][c-w[1]] + p[1]);
}
```

Comments

- The algorithm saves all possible values using an array f, so that every $f(i, y)$ is calculated only once.
- It needs $\Theta(nc)$ extra space.
- Its time complexity is $\Theta(nc)$.
  - It is not polynomial: to describe c need $log_2 c$ bits
  - but pseudo-polynomial: exponential dependence on numerical inputs
- Its disadvantage
  - The capacity c must an integer
  - The complexity might still be very hight when c is large enough, for instance $c = 2^n$

Second attempt: using a tuple

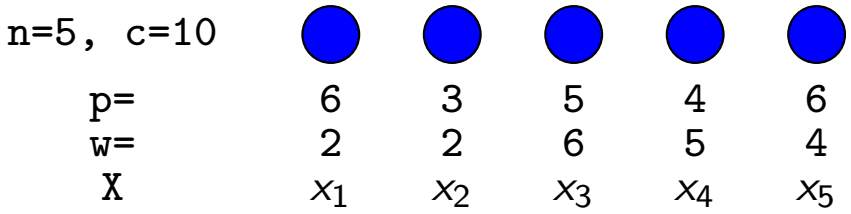


- Calculate f values using array
  - $f(5,0) = 0, \cdots, f(5,4) = 6, \cdots, f(5,10) = 6$
  - $f(4,0) = 0, \cdots, f(4,4) = 6, \cdots, f(4,9) = 10), f(4,10) = 10$
  - $\cdots$
- Save step points only for each i
  - Define a tuple (a, b), where $a = y$ and $b = f(i, y)$
  - (a, b) corresponds an optimal loading with capacity a and value b
  - Put all tuples into a set P_i
  - P_n can be got easily. $P_n = \{(0,0), (w_n, p_n)\}$
  - P_1 contains our goal! Why?

Tuple method: principle

- Let $Q = \{(s,t)|w_i \leq s < c, (s - w_i, t - p_i) \in P_{i+1}\}$
  - Q corresponds P_i in which item i **has been** selected
  - P_{i+1} corresponds P_i in which item i **has not been** selected
  - thus, $P_i = Q \bigcup P_{i+1}$
- Merge Q and P_{i+1} to get P_i
  - remove dominated tuples (a tuple (a,b) is dominated by (u,v) if $a > u$, but $b > v$)
  - remove repeated tuples
  - remove over capacity tuples in which $a > c$
- The complexity is still $O(2^n)$. The number of elements in P_i increase exponentially at worst case

Tuple method:an example



1.	P(5)=[(0,0), (4,6)]	Q=[(5,4),(9,10)]
2.	P(4)=[(0,0),**(4,6)**,(9,10)]	Q=[(6,5),(10,11)]
3.	P(3)=[(0, 0), **(4, 6)**, (9, 10), (10, 11)]	Q=[(2,3),(6,9)]
4.	P(2)=[(0,0), (2,3), (4,6), **(6,9)**, (9,10), (10,11)]	Q=[(2,6), (4,9), (6,12), (8,15)]
5.	P(1)=[(0,0), (2,6), (4,9), (6,12), **(8,15)**]	

The optimal value is 15 and the optimal solution is [1,1,0,0,1] by tracebacking

Problem Description

- Two matrices A×B with dimension (m,n) and (n,q) takes mnq multiplications.
- Let's consider three matrices: A×B×C with dimension (100,1) , (1,100) and (100,1) . The product can be done:
  - (A×B)×C takes 20000 multiplications.
  - A×(B×C)takes 200 multiplications.
  - Any other way? No
    - Associative law : (A× B)×C = A×(B× C)
    - Commutative law: A× B ≠ B× A
  - the order of multiplications makes a big difference in the final running time!
- Matrix Multiplication Chains. Suppose that we multiply q matrices
  - $M(1, q) = M_1 \times M_2 \times \cdots \times M_q$
  - There are $q + 1$ $(r_1, r_2, \cdots, r_q, r_{q+1})$numbers(parameters) to constrain their dimensions. Why?

Identifying subproblems

- The number of orders of multiplications equal to the number of parenthesization

$$P(q) = \begin{cases} 1 & \text{if } q = 2 \\ \sum_{k=1}^{q-1} p(k)p(q-k) & \text{if } q \geq 2 \end{cases} \quad \begin{aligned} T(q) &\approx O(4^q q^{\frac{3}{2}}) \\ &\approx O(2^q) \end{aligned}$$

- In fact, a particular parenthesization can be represented by a binary tree
  - The leaves corresponds to matrices
  - The root corresponds to the final product
  - Interior nodes are intermediate products
  - If a tree to be optimal, its subtrees must also be optimal.
- The binary tree representation suggests that
  - A subtree corresponds to a subproblem in MPC , so
  - **MPC satisfies the principle of optimization**

Define the optimal function

Define $c(i,j)$ is the cost of $M_i \times M_{i+1} \times \cdots \times M_j$

- $c(0,0) = 0$
- $c(i,i) = 0$
- $c(i,i+1) = r_i * r_j * r_{j+1}$
- $c(1,q)$ is our goal
- The recursive equation is

$$c(i,j) = min_{i \le k < j}\{c(i,k) + c(k,j) + r_k * r_j * r_{j+1}\} \qquad (3)$$

- Let kay(i,j) is the number minimizing above equation.
  - kay(i,i)=0
  - kay(i,i+1)=i

An exmple

Suppose that q =5 and r =(10 , 5 , 1 , 10 , 2 , 10), find the optimal orders of multiplications

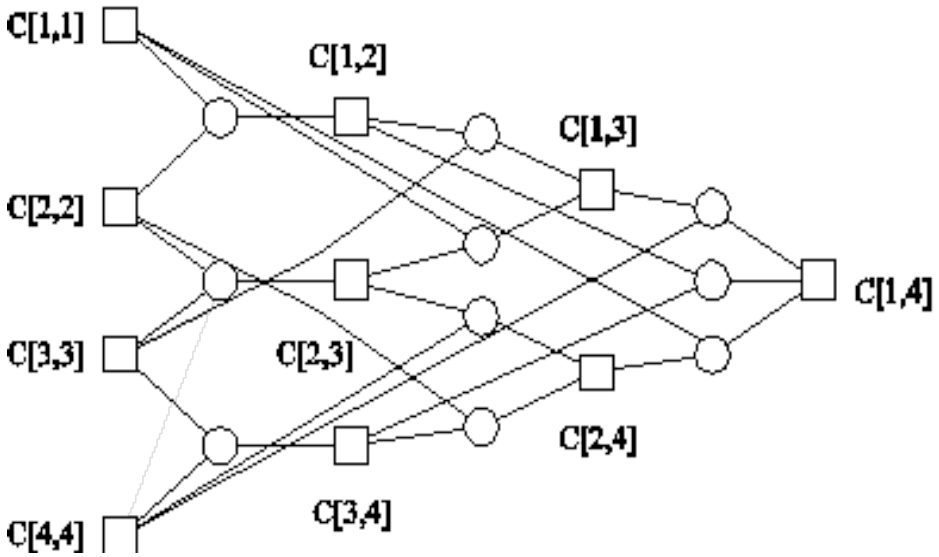
	$M(1,5) = M(1,2) \times M(3,5)$	$M(3,5) = M(3,4) \times M(5,5)$
c(1,5)=min	{c(1,1)+c(2,5)+500, c(1,2)+c(3,5)+100, c(1,3)+c(4,5)+1000, c(1,4)+c(5,5)+200}	c(1,5)=190, kay(1,5)=2 c(1,3)=150, kay(1,3)=2 c(1,4)=90, kay(1,4)=2
c(2,5)=min	{c(2,2)+c(3,5)+50, c(2,3)+c(4,5)+500, c(2,4)+c(5,5)+100}	c(2,5)=90, kay(2,5)=2
c(3,5)=min	{c(3,3)+c(4,5)+100, c(3,4)+c(5,5)+20} min{300,40} =40	c(3,5)=40,kay(3,5)=4
c(2,4)=min	{c(2,2)+c(3,4)+10, c(2,3)+c(4,4)+100} min{30,150} =30	c(2,4)=30,kay(2,4)=2

Recursive Algorithm for MPC

```
1   int RC(int i, int j)
2   {// Return c(i,j) and compute kay(i,j)=kay[i
        ][j].
3   // Avoid recomputations,check if already
        computed
4      if (c[i][j] > 0) return c[i][j];
5   // c[i][j] not computed before, compute now
6      if (i == j) return 0;  // one matrix
7      if (i == j - 1) {// two matrices
8                       kay[i][i+1] = i;
9                       c[i][j] = r[i]*r[i+1]*r[
                            i+2];
10                      return c[i][j];}
11  // more than two matrices
12  // set u to mini term for k = i
13     int u = RC(i,i) + RC(i+1,j) + r[i]*r[i
           +1]*r[j+1];
14     kay[i][j] = i;
15  // compute remaining min terms and update u
16     for (int k = i+1; k < j; k++) {
17        int t = RC(i,k) + RC(k+1,j) + r[i]*r[k
              +1]*r[j+1];
18        if (t < u) {// smaller min term
19                    u = t;
20                    kay[i][j] = k;}
21        }
22     c[i][j] = u;
23     return u;
24  }
```

```
1   void Traceback(int i, int j, int
        **kay)
2   {
3      if (i == j) return;
4      Traceback(i, kay[i][j], kay);
5      Traceback(kay[i][j]+1, j, kay
           );
6      cout << "Multiply␣M␣" << i <<
            "␣," << kay[i][j];
7      cout << "␣and␣M␣" << (kay[i][
           j]+1) << "␣," << j
8           << endl;
9   }
```

Revision recursive algorithm



The above figure suggest that $c(i,j)$ can be calculated in an iterative miner

$$c(i,i+s) = \min_{i \le k < s} \{c(i,k) + c(k,j) + r_i * r_k * r_{i+s+1}\}$$

$$s = 1, 2, \cdots, q$$

Iterative algorithm for MPC

```
1   void MatrixChain(int r[], int q, int **c, int **kay)
2   {//Compute costs and kay for all Mij's.
3   //initialize c[i][i],c[i][i+1],and kay[i][i+1]
4      for (int i = 1; i < q; i++) {
5         c[i][i] = 0;
6         c[i][i+1] = r[i]*r[i+1]*r[i+2];
7         kay[i][i+1] = i;
8         }
9      c[q][q] = 0;
10  //compute remaining c's and kay's
11     for (int s = 2; s < q; s++)
12        for (int i = 1; i <= q - s; i++) {
13           // min term for k = i
14           c[i][i+s] = c[i][i] + c[i+1][i+s]
15                       + r[i]*r[i+1]*r[i+s+1];
16           kay[i][i+s] = i;
17           // remaining mini terms
18           for (int k = i+1; k < i + s; k++) {
19              int t = c[i][k] + c[k+1][i+s]
20                      + r[i]*r[k+1]*r[i+s+1];
21              if (t < c[i][i+s]) {// smaller mini term
22                 c[i][i+s] = t;
23                 kay[i][i+s] = k;}
24              }
25           }
26  }
```

Problem statement

- **Input**: Given a directed graph $G = (V, E)$ and a matrix (a_{ij}) where

$$V = \{1, 2, \cdots, n\} \text{with edge weight function} W : E \to R$$

$$a_{ij} = \begin{cases} w(i,j) & \text{if } (i,j) \in E \\ 0 & \text{if } i = j \\ \infty & \text{otherwise} \end{cases}$$

- **Output**: A $n \times n$ matrix of shortest-path lengths $c(i, j)$
- **Assumption**: No negative-weight cycles

Subprobem identification by edges

- Define d_{ij}^m = weight of a shortest path from i to j that only uses at most m edges
- d_{ij}^{n-1} is our goal
- We have known that
  - $d_{ij}^0 = 0$ if $i = j$, and ∞ if $i \neq j$
  - $d_{ij}^1 = 0$ if $i = j$, and a_{ij} if $i \neq j$

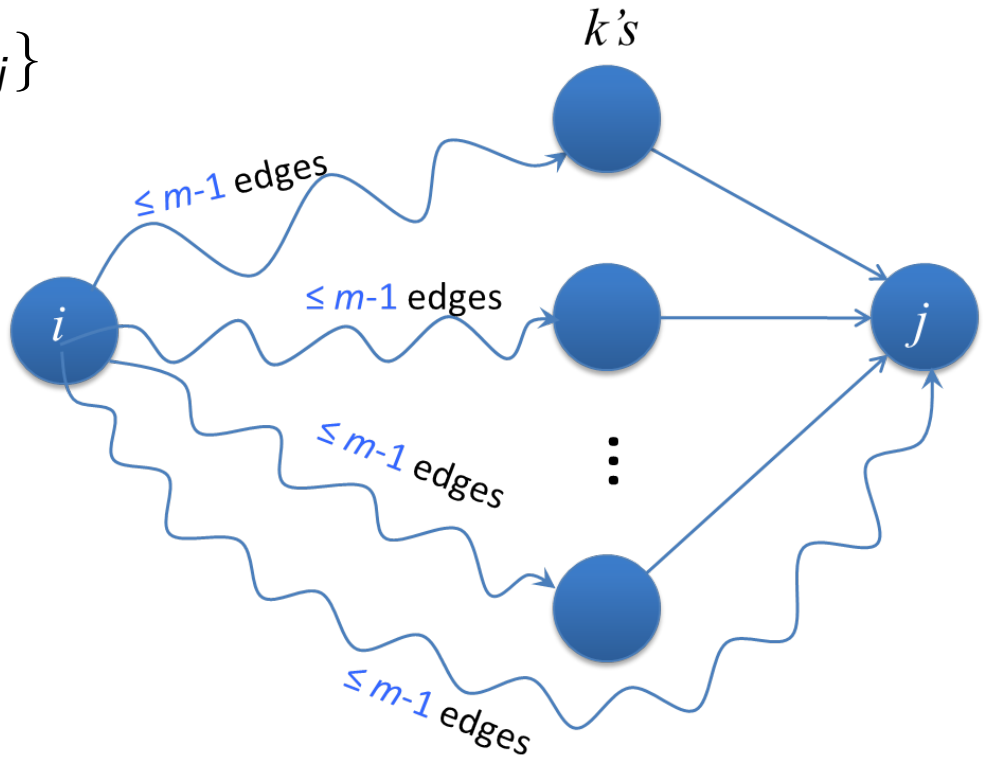
Theorem

For $m = 1, 2, \cdots, n-1$, we have

$$d_{ij}^m = \min_{1 \leq k < m} \{d_{ik}^{m-1} + a_{kj}\}$$

Proof

$$d_{ij}^{m} = \min_{1 \le k < m} \{ d_{ik}^{m-1} + a_{kj} \}$$



for k=1 to n

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

$d_{ij} = d_{ik} + a_{kj}$

Running time $O(n^4)$ - similar to n runs of Bellman-Ford algorithm

Subprolems identification by intermediate vertices

- Define d_{ij}^m = weight of a shortest path from i to j that only uses intermediate vertices from set $\{1, , m\}$ or
- Define d_{ij}^m = weight of a shortest path from i to j that the orders of intermediate vertices is no larger than m
- d_{ij}^n is our goal
- We have known that
  - $d_{ij}^0 = aij$
  - $d_{ik}^{k-1} = d_{ik}^k$
  - $d_{jk}^{k-1} = d_{jk}^k$

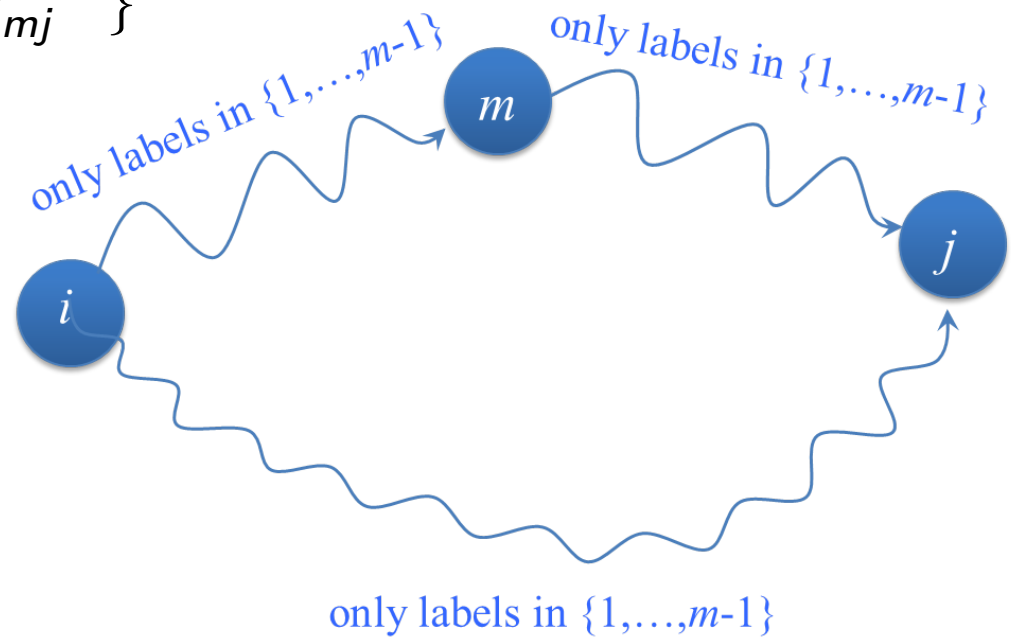
Theorem

For m = 1, 2, $\cdots$, n-1, we have

$$d_{ij}^m = \min\{d_{ij}^{m-1}, d_{im}^{m-1} + d_{mj}^{m-1}\}$$

Proof

$$d_{ij}^{m} = \min\{d_{ij}^{m-1}, d_{im}^{m-1} + d_{mj}^{m-1}\}$$



Running time $O(n^3)$ Known as Floyd-Warshall algorithm

Iterative algorithm for ASAP

```
1   template<class T>
2   void AdjacencyWDigraph<T>::AllPairs(T **c,
        int **kay)
3   {// All pairs shortest paths.
4    // Compute c[i][j] and kay[i][j] for all i
         and j.
5      // initialize c[i][j] = c(i,j,0)
6      for (int i = 1; i <= n; i++)
7         for (int j = 1; j <= n; j++) {
8            c[i][j] = a[i][j];
9            kay[i][j] = 0;
10           }
11     for (int i = 1; i <= n; i++)
12        c[i][i] = 0;
13
14     // compute c[i][j] = c(i,j,k)
15     for (int k = 1; k <= n; k++)
16        for (int i = 1; i <= n; i++)
17           for (int j = 1; j <= n; j++) {
18              T t1 = c[i][k];
19              T t2 = c[k][j];
20              T t3 = c[i][j];
21              if (t1 != NoEdge && t2 != NoEdge
                    &&
22                 (t3 == NoEdge || t1 + t2 < t3
                       )) {
23                 c[i][j] = t1 + t2;
24                 kay[i][j] = k;}
25           }
26  }
```

Kay[i][j] is used to store the largest vertex in the shortest path from i to j, so that traceback the shortest path

```
1   void outputPath(int **kay, int i,
        int j)
2   {// Actual code to output i to j
        path.
3      if (i == j) return;
4      if (kay[i][j] == 0) cout << j <<
           ' ';
5      else {outputPath(kay, i, kay[i][j
           ]);
6            outputPath(kay, kay[i][j],
                 j);}
7   }
8
9   template<class T>
10  void OutputPath(T **c, int **kay, T
        NoEdge,
11                              int i, int
                                    j)
12  {// Output shortest path from i to j
        .
13     if (c[i][j] == NoEdge) {
14        cout << "There is no path from
               " << i << " to "
15             << j << endl;
16        return;}
17     cout << "The path is" << endl;
18     cout << i << ' ';
19     outputPath(kay,i,j);
```

An example

Adjacent matrix

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

Distance matrix of shortest path

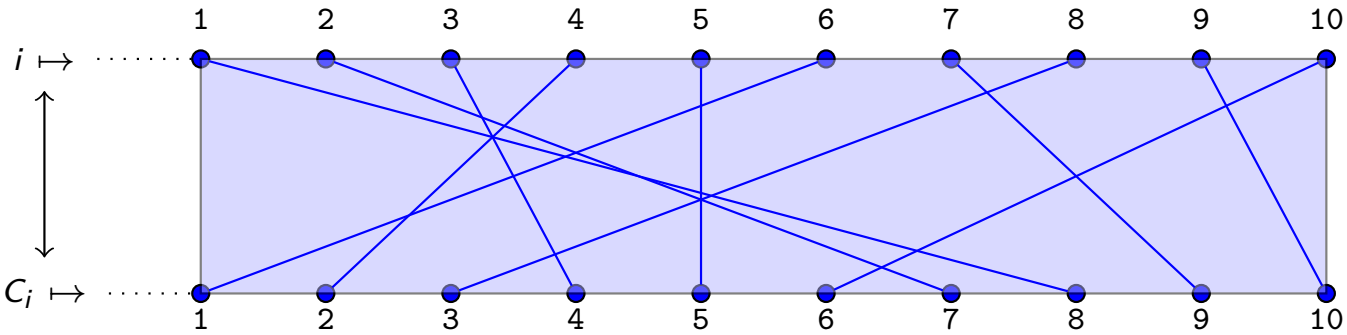
0	1	4	4	8
3	0	1	5	9
2	2	0	1	8
8	8	9	0	1
8	8	2	9	0

matrix kay_{ij}

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

- we can traceback the shortest paths from matrix (kay_{ij}), for example, from 1 to 5
  - kay(1,5)=4 , kay(4,5) =0 , $4 \rightarrow 5$ is a sub-path
  - kay(1,4)=3, kay(3,5)=0, $3 \rightarrow 4$ is a sub-path
  - kay(1,3)=2, kay(2,3)=0, $2 \rightarrow 3$ is a sub-path
  - kay(1,2)=0 , $1 \rightarrow 2$ is a sub-path
  - The final shortest path is $1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$

Problem statement



- A 1-1 map (i, c_i) is called a subnet
- Two subnets (i, c_i) and (j, c_j) are non-crossed if $i < j$ then $c_i < c_j$
- The set $MNS(i, j) = \{(u, c_u) | u \leq i, c_u \neq j\}$ is called non-crossing set if for $\forall(p, c_p)$ and $\forall(q, c_q) \in MNS(i, j)$ then (p, c_p) and (q, c_q) are non-crossed
- Our goal is to find a MNS(n,n) with the maximum number of elements
- Define size(i,j) $= |MNS(i, j)|$

Derive the recursive equation

- Our goal is to maximize size(n,n)
- We have known that

$$size(1,j) = \begin{cases} 0 & \text{if } j < c_1 \\ 1 & \text{if } j \geq c_1 \end{cases} \tag{4}$$

- We want to know the relationship between size(i,j) and size(i-1,j)
  - if $j < c_i$ then $(i, c_i) \notin MNS(i-1, j)$, thus size(i,j)=size(i-1,j)
  - if $j \geq c_i$, there are two cases
    - put (i, c_i) into MNS(i-1,j), but (i, c_i) might cross with items in MNS(i-1,j) or result in smaller size, we have size(i,j)=size(i-1,j)
    - put (i, c_i) into MNS(i-1,j), no crossing, then c_{i-1} must be less than $c_i - 1$,else crossed with (i, c_i) , thus we have size(i,j)=size(i-1,c_i -1)+1
    - finally, we choose the maximum of them!

$$size(i,j) = \begin{cases} size(i, j-1) & \text{if } j < c_i \\ \max\{size(i-1, j), size(i-1, c_i - 1) + 1\} & \text{if } j \geq c_i \end{cases}$$

Problem statements

Definition 1: Subsequence

Given a sequence X=$x_1 x_2 \cdots x_m$, another sequence Z=$z_1 z_2 \cdots z_k$ is a subsequence of X if there exists a strictly increasing sequence= $x_1 x_2 \cdots x_k$ of indices of X such that for all j=1,2,...k, we have $x_{i_j} = z_j$.

Example 1: If X=abcdefg, Z=abdg is a subsequence of X.

Definition 2: Common subsequence

Given two sequences X and Y, a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

Example 2: X=abcdefg and Y=aaadgfd. Z=adf is a common subsequence of X and Y

Definitions

Definition 3: Longest common subsequence:LCS

A longest common subsequence of X and Y is a common subsequence of X and Y with the longest length.

- Longest common subsequence may not be unique, for example, strings both acd and abd are LCS of abcd and acbd
- LCS has been successfully applied to many fields
  - Bioinformatics, e.g. long preserved regions in genomes
  - file comparison, e.g. diff
- Solutions
  - Brute force approach: $O(n2^m)$
  - DP approach: O(nm)

DP approach for LCS

- Let's consider the prefixes of x and y
  - x[1..i] ith prefix of x[1..m]
  - y[1..j] jth prefix of y[1..n]
- Subproblem: define c[i,j] $=|LCS(x[1..i], y[1..j])|$
- c[m,n] is our goal
- We have known that c[1,1]=1 if $x_1 = y_1$ otherwise 0
- To derive recursive relationship, we use the following theorem

Theorem: Let X=$x_1 x_2 \cdots x_m$ and Y=$y_1 y_2 \cdots y_n$ be two sequences, and Z=$z_1 z_2 \cdots z_k$ be any LCS of X and Y, then

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z[1..k-1] is an LCS of X[1..m-1] and Y[1..n-1].
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X[1..m-1] and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y[1..m-1].

Recursive equation

By the theorem, we can easily get the recursive equation

$$c[i,j] = \begin{cases} 0 & \text{if i=0 or j=0} \\ c[i-1,j-1]+1 & \text{if x[i]=y[j]} \\ \max\{c[i-1,j], c[i,j-1]\} & \text{otherwise} \end{cases} \quad (6)$$

```
1   LCS(X,Y,m,n,b)
2   for i=1 to m do
3           c[i,0]=0;
4   for j=0 to n do
5           c[0,j]=0;
6   for i=1 to m  do
7           for j=1 to n do
8           {//b[i,j] stores the directions.
9           if x[i] ==y[j]  then
10                  c[i,j]=c[i-1,j-1]+1;
11                  b[i,j]=1;  //1-diagonal,
12          else if c[i-1,j]>=c[i,j-1] then
13                          c[i,j]=c[i-1,j]
14                          b[i,j]=2;//2-up,
15                  else c[i,j]=c[i,j-1]
16                           b[i,j]=3; //3-
                                 forward.
17          }
```

LCS algorithm

```
1   PrintLCS(b,X,i,j)
2   i=m
3   j=n;
4   if i==0 or j==0 then exit;
5   if b[i,j]==1 then
6   {
7               i=i-1;
8               j=j-1;
9               print x[i];
10   }
11  if b[i,j]==2   i=i-1
12  if b[i,j]==3   j=j-1
13  Goto Step 3.
```

Print LCS algorithm