W4231: Analysis of Algorithms

11/11/99

- Dynamic Programming

Shortest Paths

Let $G = (V, E)$ be a directed graph, and let $w_{u,v} \geq 0$ be the weight of edge (u, v). Let $V = \{1, \ldots, n\}$.

Idea for recursive solution to find shortest path between s and t, $s, t \in V$: the shortest path between s and t is

- either the shortest of the paths from s to t not passing through n
- or the shortest path from s to n concatenated with the shortest path between n and t.

Recursive Version

```
SP-REC(s, t, V, E, w)
  if |V| = 0 then return w_{s,t}
  else
    k = |V|
    l := SP (s, t, V − {k}, E, w)
    l' := SP (s, k, V − {k}, E, w) + SP(k, t, V − {k}, E, w)
    return min{l, l'}
```

Algorithm SP(s, t, V, E, w) finds the length of the shortest of the paths between s and t, among those that can pass through the vertices of V. (If $(u, v) \notin E$, then assume $w_{u,v} = \infty$ in the implementation.)

Iterative Version — General Idea

Define a $n \times n \times (n + 1)$ matrix M.

Design an algorithm that will fill the matrix so that $M[s, t, k]$ is the length of the shortest of the paths between s and t, among those that can pass through the vertices $\{1, \ldots, k\}$.

Once the matrix is filled, the length of the shortest path from s to t is in $M[s, t, n]$.

So, once the matrix is filled, we have the length of the shortest paths between any two vertices.

Filling the Matrix

We know what is $M[s, t, 0]$: this is just the length of the edge from s and t, if it exists. (Otherwise we set it to ∞ by convention.)

The value of $M[s, t, k]$ can then be computed based on precomputed values of $M[\cdot, \cdot, k - 1]$ since

$$M[s, t, k] = \min\{M[s, t, k-1],\ M[s, k, k-1] + M[k, t, k-1]\}$$

Algorithm

```
SP-DP(V, E, w)
  create n × n × (n + 1) matrix M
  for u := 1 to n
    for v := 1 to n
      if (u, v) ∈ E then M[u, v, 0] := w_{u,v}
      else M[u, v, 0] = ∞
  for k := 1 to n
    for u := 1 to n
      for v := 1 to n
        M[u, v, k] := min{ M[u, v, k − 1],
                           M[u, k, k − 1] + M[k, v, k − 1] }
```

Runs in time $O(n^3)$.

Reconstructing the actual paths

Together with M, we also create a matrix P of same dimension, such that $P[u,v,k]$ says which vertex comes before v in the shortest path from u to v among those that can pass through $\{1,\ldots,k\}$.

Once we have the final M and P, we can reconstruct the shortest path from u to v backwards. Start from v, and look up $v' = P[u,v,n]$. Then look up $v'' = P[u,v',n]$ and so one. Eventually we get to u, by way of a shortest path.

Algorithm that also computes P

```
SP-DP-2(V, E, w)
  create n × n × (n+1) matrices M and P
  for u := 1 to n
    for v := 1 to n
      if (u,v) ∈ E then M[u,v,0] := w_{u,v}; P[u,v,0] = u
      else M[u,v,0] = ∞; P[u,v,0] = u
  for k := 1 to n
    for u := 1 to n
      for v := 1 to n
        l_1 = M[u,v,k−1]; l_2 = M[u,k,k−1] + M[k,v,k−1]
        if l_1 ≤ l_2 then
          M[u,v,k] := M[u,v,k−1]; P[u,v,k] := P[u,v,k−1]
        else
          M[u,v,k] := M[u,k,k−1] + M[k,v,k−1]; P[u,v,k] = k
```


Knapsack

You are packing your knapsack for a long hiking. You have a choice of n items to put in. Item i has volume v_i and is worth an advantage c_i if you can take it with you.

The knapsack has volume B.

Choose a subset S of elements that fit into the knapsack and maximize $\sum_{i\in S} c_i$.

Formal description of the problem

Given: n items of "cost" $c_1, c_2, \ldots, c_n$ (positive integers), and of "volume" $v_1, v_2, \ldots, v_n$ (positive integers); a volume value B (for *b*ound).

Find a subset S of the items such that

$$\sum_{i\in S} v_i \leq B \tag{1}$$

and such that the total cost $\sum_{i\in S} c_i$ is maximized.

Dynamic programming algorithm

Construct a $n \times (B+1)$ table $M[\cdot,\cdot]$.

For every $1 \leq k \leq n$ and $0 \leq B' \leq B$,

$M[k,B']$ contains the cost of an optimum solution for the instance that uses only a subset of the first k elements (of volume $v_1,\ldots,v_k$ and cost $c_1,\ldots,c_k$) and with volume B'.

- $M[1,B'] = c_1$ if $B' \geq v_1$; $M[1,B'] = 0$ otherwise.
- for every $k, B' \geq 1$,

$$M[k,B'] = \max\{M[k-1,B'] \,,\; M[k-1,B'-v_k] + c_k\}$$

Second matrix

The *cost* of an optimum solution is reported in $M[n,B]$.

To *construct* an optimum solution, we also build a Boolean matrix $C[\cdot,\cdot]$ of the same size.

For every k and B', $C[k,B'] = True$ iff there is an optimum solution that packs a subset of the first k items in volume B' so that item k *is* included in the solution.

Example

Consider an instance with 9 items and a bag of size 15. The costs and the volumes of the items are as follows:

Item	1	2	3	4	5	6	7	8	9
Cost	2	3	3	4	4	5	7	8	8
Volume	3	5	7	4	3	9	2	11	5

Tables $M[\cdot,\cdot]$ and $C[\cdot,\cdot]$

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k=9$	0	0	7	7	7	11	11	15	15	15	19	19	19	21	23	23
$k=8$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k=7$	0	0	7	7	7	11	11	11	13	15	15	15	17	17	18	18
$k=6$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k=5$	0	0	0	4	4	4	6	8	8	8	10	10	11	11	11	13
$k=4$	0	0	0	2	4	4	4	6	6	7	7	7	9	9	9	9
$k=3$	0	0	0	2	2	3	3	3	5	5	5	5	6	6	6	8
$k=2$	0	0	0	2	2	3	3	3	5	5	5	5	5	5	5	5
$k=1$	0	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2

B'	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$k=9$	0	0	0	0	0	0	0	1	1	0	1	1	1	1	1	1
$k=8$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k=7$	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1
$k=6$	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
$k=5$	0	0	0	1	0	0	1	1	1	1	1	1	1	1	1	1
$k=4$	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1
$k=3$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
$k=2$	0	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1
$k=1$	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1

Solution

Optimum: 23
Item 9 Cost 8 Volume 5
Item 7 Cost 7 Volume 2
Item 5 Cost 4 Volume 3
Item 4 Cost 4 Volume 4

Approximate string matching

When you run a spell checker on a text, and it finds a word not in the dictionary, it normally proposes a choice of possible corrections.

If it finds `stell` it will suggest:

`tell`, `swell`, `stull`, `still`, `steel`, `steal`, `stall`, `spell`, `smell`, `shell`, and `sell`.

Edit distance

How do you decide the “closeness” between two strings?

The distance between two strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is the minimum number of “errors” (edit operations) needed to transform x into y.

Possible operations are:

- insert a character.
  $insert(x, i, a) = x_1 x_2 \cdots x_i a x_{i+1} \cdots x_n$.
- delete a character.
  $delete(x, i) = x_i x_2 \cdots x_{i-1} x_{i+1} \cdots x_n$.
- modify a character.
  $modify(x, i, a) = x_1 x_2 \cdots x_{i-1} a x_{i+1} \cdots x_n$.

Example

$x = aabab$ and $y = babb$.

One possibility

a	a	b	a	b		x
b	a	a	b	a	b	x' = insert(x,0,b)
b	a	b	a	b		x" = delete (x',2)
b	a	b	b			y = delete (x",4)

And also

a	a	b	a	b	x
a	b	a	b		x' = delete (x,1)
b	a	b			x" = delete(x',1)
b	a	b	b		y = insert (x",3,b)

Can you do better?

Computing edit distance

To transform $x_1 \cdots x_n$ into $y_1 \cdots y_m$ we have three choices:

- put y_m at the end: $x \to x_1 \cdots x_n y_m$ and then transform $x_1 \cdots x_n$ into $y_1 \cdots y_{m-1}$.
- delete x_n: $x \to x_1 \cdots x_{n-1}$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_m$.
- change x_n into y_m (if they are different): $x \to x_1 \cdots x_{n-1} y_m$ and then transform $x_1 \cdots x_{n-1}$ into $y_1 \cdots y_{m-1}$.

Dynamic programming table

Given strings $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$.

Define $(n+1) \times (m+1)$ matrix $M[\cdot,\cdot]$.

For every $0 \le i \le n$ and $0 \le j \le m$, $M[i,j]$ is the minimum number of operations to transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

Recursive definition of the table

- $M[0,j] = j$ because the only way to transform the empty string into $y_1 \cdots y_j$ is to add the j characters $y_1, \ldots, y_j$.
- $M[i,0] = i$ for similar reasons.

- For $i, j \ge 1$,

$$\begin{aligned} M[i,j] \quad = \min\{ \quad & M[i-1,j]+1, \\ & M[i,j-1], \\ & M[i-1,j-1] + change(x_i,y_j)\} \end{aligned}$$

where $change(x_i,y_j) = 1$ if $x_i \ne y_j$ and $change(x_i,y_j) = 0$ o/w.

Example

Consider again $x = aabab$ and $y = babb$

	λ	b	a	b	b
λ	0	1	2	3	4
a	1	1	1	2	3
a	2	2	1	2	3
b	3	2	2	1	2
a	4	3	2	2	2
b	5	4	3	2	2

Optimum solution?

Algorithm and running time

The table has $\Theta(nm)$ entries, each one computable in constant time.

One can construct an auxiliary table $Op[\cdot,\cdot]$ such that $Op[\cdot,\cdot]$ specifies what is the first operation to do in order to optimally transform $x_1 \cdots x_i$ into $y_1 \cdots y_j$.

Longest common subsequence

A subsequence of a string is obtained by taking a string and possibly deleting elements.
If $x_1 \cdots x_n$ is a string and $1 \le i_1 < i_2 < \cdots < i_k \le n$ is a strictly increasing sequence of indices, then $x_{i_1} x i_2 \cdots x_{i_k}$ is a subsequence of x.

E.g. **art** is a subsequence of **a**lgo**r**i**t**hm.

Given strings x and y we want to find the longest string that is a subsequence of both.

E.g. **art** is the longest common subsequence of **a**lgo**r**i**t**hm and p**ar**achu**t**e.

Reducing to a smaller subproblem

The length of the l.c.s. of $x = x_1 \cdots x_n$ and $y = y_1 \cdots y_m$ is either

- The length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_m$ or;
- The length of the l.c.s. of $x_1 \cdots x_n$ and $y_1 \cdots y_{m-1}$ or;
- $1 +$ the length of the l.c.s. of $x_1 \cdots x_{n-1}$ and $y_1 \cdots y_{m-1}$, if $x_n = y_m$.

Definition of the matrix

For every $0 \le i \le n$ and $0 \le j \le m$, $M[i,j]$ contains the length of the l.c.s. between $x_1 \cdots x_i$ and $y_1 \cdots y_j$.

-$M[i,0] = 0$
-$M[0,j] = 0$
-and

$$M[i,j] = \max\{ \begin{array}{l} M[i-1,j] \\ M[i,j-1] \\ M[i-1,j-1] + eq(x_i,y_j)\} \end{array}$$

where $eq(x_i,y_j) = 1$ if $x_i = y_j$, $eq(x_i,y_j) = 0$ o/w.

Example

	λ	p	a	r	a	c	h	u	t	e
λ	0	0	0	0	0	0	0	0	0	0
a	0	0	1	1	1	1	1	1	1	1
l	0	0	1	1	1	1	1	1	1	1
g	0	0	1	1	1	1	1	1	1	1
o	0	0	1	1	1	1	1	1	1	1
r	0	0	1	2	2	2	2	2	2	2
i	0	0	1	2	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	3	3
h	0	0	1	2	2	2	3	3	3	3
m	0	0	1	2	2	2	3	3	3	3