

Speeding up dynamic programming

Ngo Minh Duc

School of Computing

National University of Singapore

Email: ducnm0@gmail.com

This is a note as part of my CS2306S's project report on data structures and algorithms.

Abstract

Dynamic programming is a widely used approach in designing algorithms. In this note, we summarize some techniques for speeding up dynamic programming algorithms. We also discuss some programming tasks that these techniques can be applied.

1. Quadrangle Inequalities

1.1. The problem

Let w be an $n \times n$ matrix. We need to compute an $n \times n$ dynamic programming table f satisfying the following properties:

- $f_{i,i} = w_{i,i} = 0$
- $f_{i,j} = w_{i,j} + \min_{i < k \leq j} \{f_{i,k-1} + f_{k,j}\}$ for $i < j$ (1.1)

(We don't consider values of $f_{i,j}$ or $w_{i,j}$ for $i > j$)

This recurrence relation appears in a number of dynamic programming problems. Usually $f_{1,n}$ is the desired result.

By applying (1.1) directly, one can compute f in $O(n^3)$ time. This section shows that under a certain condition of w , we can compute f in $O(n^2)$ time.

The following results are stated and proved by Yao [1].

Definition 1.1 If w satisfies $w_{i,j} + w_{i',j'} \leq w_{i',j} + w_{i,j'}$, $\forall i \leq i' \leq j \leq j'$, w is said to satisfy the *Quadrangle Inequalities (QI)*.

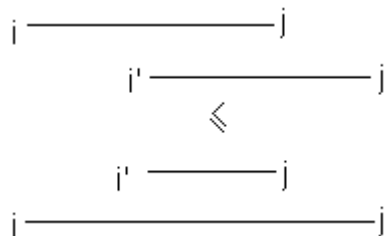


Figure 1. Quadrangle Inequalities

Definition 1.2 If $w_{i,j} \leq w_{i',j'}$, $\forall i' \leq i \leq j \leq j'$, w is said to be *monotone*.

Definition 1.3 Define $k_{i,j} = \min\{t \mid f_{i,j} = w_{i,j} + f_{i,t-1} + f_{t,j}\}$ where $i < j$. Specially, define $k_{i,i} = i$. In other words, $k_{i,j}$ is the minimum index k such that the minimum in (1.1) is achieved.

Theorem 1.1 If w is monotone and satisfies QI then f also satisfies QI.

Moreover, $k_{i,j-1} \leq k_{i,j} \leq k_{i+1,j}$ for $i < j$.

For a proof of theorem 1.1, readers may refer to [1].

Corollary 1.1 If w is monotone and satisfies QI, then f can be computed in $O(n^2)$.

Proof Consider the following algorithm QI to compute f :

QI()

```

1. for i ← 1 to n do
2.   f[i,i] ← 0
3.   k[i,i] ← i
4. for l ← 2 to n do
5.   for i ← 1 to n-l+1 do
6.     j ← i+l-1
7.     for t ← k[i,j-1] to k[i+1,j] do
8.       v ← w[i,j] + f[
i,t-1] + f[t,j]
9.       if v < f[i,j] then
10.        f[i,j] = v
11.        k[i,j] = t

```

According to theorem 1.1, $k_{i,j-1} \leq k_{i,j} \leq k_{i+1,j}$ for $i < j$, so QI correctly compute f .

We now show that QI has $O(n^2)$ complexity. Each time QI enters the loop in lines 5-11, the total time QI enters the loop in lines 7-11 is:

$$\sum_{i=1}^{n-l+1} (k_{i+1,i+l-1} - k_{i,i+l-2} + 1) = k_{n-l+2,n} - k_{1,l-1} + n - l + 1 = O(n)$$

We can conclude that QI has $O(n^2)$ complexity. ■

1.2. Applications

1.2.1. Optimal Cut <https://vn.spoj.pl/problems/OPTCUT/>

This problem is a direct application of the above result. The dynamic programming function is:

$$f_{ij} = a_i + a_{i+1} + \dots + a_j + \min_{i < k \leq j} \{f_{i,k-1} + f_{kj}\}$$

Therefore, we can use algorithm QI to solve the problem in $O(n^2)$ time.

1.2.2. Optimal Binary Search Trees

We need to construct a binary search tree with n keys $a_1 < a_2 < \dots < a_n$. We are also given $2n+1$ probabilities p_1, p_2, \dots, p_n and q_0, q_1, \dots, q_n , in which:

- p_i = probability that key a_i is searched
- q_i = probability that the search argument lies between key a_i and key a_{i+1}

Our goal is to minimize the expected number of comparisons in the search, namely:

$$\sum_{i=1}^n p_i (1 + \text{depth}(a_i)) + \sum_{i=0}^n q_i \text{depth}(i^{\text{th}} \text{ leaf})$$

To solve this problem, we define f_{ij} to be the optimal cost to build the binary search tree with keys a_i, a_{i+1}, \dots, a_j . Observe that f_{ij} satisfies the recurrence relation (1.1):

$$f_{ij} = w_{ij} + \min_{i < k \leq j} (f_{i,k-1} + f_{kj})$$

in which

$$w_{ij} = p_{i+1} + \dots + p_j + q_i + q_{i+1} + \dots + q_j$$

Since w_{ij} is mononote and satisfies the quadrangle inequalities (in fact as equalities), we can apply algorithm QI() to compute f in $O(n^2)$ time.

2. Shortest path in directed acyclic graphs - a special case

2.1. The problem

Given a directed acyclic graph of n vertices $1, 2, \dots, n$ and edges (i, j) where $i < j$. Let c_{ij} be the cost of the edge (i, j) . We need to find a path from 1 to n of minimum cost.

The following simple dynamic programming algorithm will solve the problem in $O(n^2)$ time.

Let F_i be the shortest path from i to n . Let $F_i(j) = c_{ij} + F_j$ for $i < j$. We have the recursive relation:

$$\begin{cases} F(n) = 0 \\ F(i) = \min_{i < j \leq n} \{F_i(j)\} \quad \forall i < n \end{cases}$$

P.Brucker [2] have shown that under the following condition of the cost matrix c , the problem can be solved in linear time:

$$c_{ik} - c_{ij} = f(i)h(j, k) \quad \forall i < j < k \quad (2.1)$$

where f is a non-increasing function and $h(j, k) > 0 \quad \forall j, k$

In this section, we show P.Brucker's results and discuss its applications in some programming tasks.

Recall the dynamic programming formula $F(i) = \min_{i < j \leq n} \{F_i(j)\} \quad \forall i < n$ (2.2)

The idea is that under condition (2.1), we can exclude some $F_i(j)$ that will never be a minimum candidate.

Observe that for $i < j < k$,

$$F_i(j) \leq F_i(k)$$

$$\Leftrightarrow c_{ij} + F_j \leq c_{ik} + F_k$$

$$\Leftrightarrow F_j - F_k \leq c_{ik} - c_{ij}$$

$$\Leftrightarrow F_j - F_k \leq f(i)h(j, k) \quad (\text{by 2.1})$$

$$\Leftrightarrow \frac{F_j - F_k}{h(j, k)} \leq f(i) \quad (\text{since } h(j, k) > 0)$$

The left-hand-side of the last inequality does not depend on i , so we can define

$$g(j, k) = \frac{F_j - F_k}{h(j, k)} \quad \text{for } j < k$$

Thus, we have proved the following lemma.

Lemma 2.1 $F_i(j) \leq F_i(k) \Leftrightarrow g(j, k) \leq f(i)$

The following two lemmas state sufficient conditions to discard a vertex (for the reason we do not need that vertex to compute the shortest path).

Lemma 2.2 If $g(j, k) \leq f(i)$ for some $i < j < k$, then

$$F_{i'}(j) \leq F_{i'}(k) \quad \forall 1 \leq i' \leq i$$

Proof Since f is given as a non-increasing function, we have $f(i) \leq f(i') \forall 1 \leq i' \leq i$. Thus

$g(j, k) \leq f(i) \leq f(i')$, and the result follows from lemma 2.1. ■

Lemma 2.2 implies that if at step i (when we compute F_i), we have

$g(j, k) \leq f(i)$ for some $i < j < k$, then the vertex k can be discarded from the graph.

Lemma 2.3 If $g(j, k) \leq g(k, l)$ for $j < k < l$, then for all i such that $1 \leq i \leq j$, either $F_i(j) \leq F_i(k)$ or $F_i(l) \leq F_i(k)$

Proof If $f(i) \geq g(j, k)$, then $F_i(j) \leq F_i(k)$ (by Lemma 2.1). If $f(i) < g(j, k)$, then $f(i) < g(k, l)$; thus, $F_i(l) < F_i(k)$ (by Lemma 2.1). ■

Lemma 2.3 implies that if at step j , we have $g(j, k) \leq g(k, l)$ for $j < k < l$, then the vertex k can also be discarded from the graph.

The algorithm

Based on the above lemmas, we will construct a linear time algorithm. Similar to the original dynamic programming approach, we compute F_i from n down to 1.

At each step, we maintain a list of vertices that are still not discarded. The vertices are stored in a double-ended queue $Q = (i_r, i_{r-1}, \dots, i_2, i_1)$ where $i_r < i_{r-1} < \dots < i_1$. The head of the queue is i_1 and the tail of the queue is i_r .

By lemma 2.3, we see that Q should satisfy $g(i_r, i_{r-1}) > g(i_{r-1}, i_{r-2}) > \dots > g(i_2, i_1)$, otherwise some vertex in Q has been discarded before.

When we compute F_i , follow the below two steps:

- Using $f(i)$, discard unnecessary vertices at the head of the queue

If $g(i_2, i_1) \leq f(i)$, discard i_1 from the queue. This can be done since by lemma 2.2, we do not need i_1 to compute $F_{i'}$ for all $1 \leq i' \leq i$. Continue this step until for some $t \geq 1$, $g(i_r, i_{r-1}) > g(i_{r-1}, i_{r-2}) > \dots > g(i_{t+1}, i_t) > f(i)$, or $r = t$ and Q only contains i_t .

By lemma 2.1, we have $F_i(i_r) > F_i(i_{r-1}) > \dots > F_i(i_{t+1}) > F_i(i_t)$. Thus, $F_i(i_t) = \min_{j \in Q} \{F_i(j)\}$.

Since the vertices not in the queue have been discarded at some point (and therefore not needed to compute F_i), we must have $F_i = F_i(i_t)$.

- When inserting i , discard unnecessary vertices at the tail of the queue

If $g(i, i_r) \leq g(i_r, i_{r-1})$, discard i_r from the queue. This can be done by lemma 2.3. Continue this step until for some $v \leq r$, $g(i, i_v) > g(i_v, i_{v-1})$, or Q only contains i_v . Insert i as the new tail of the queue.

Analysis Each vertex i is inserted to and discarded from the queue at most once. Therefore, the algorithm's complexity is $O(n)$.

Pseudocode (We assume the double-ended queue Q is 0-based index)

```

1.  $F[n] \leftarrow 0$ 
2.  $Q.push\_back(n)$ 
3. for  $i \leftarrow n-1$  downto 1 do
4.   while  $Q.size() > 1$  AND  $g(Q[1], Q[0]) \leq f(i)$  do
5.      $Q.pop\_front()$ 
6.    $F[i] \leftarrow c_{iQ[0]} + F[Q[0]]$ 
7.   while  $Q.size() > 1$  AND
      $g(i, Q[Q.size()-1]) \leq g(Q[Q.size()-1], Q[Q.size()-2])$  do
9.      $Q.pop\_back()$ 
10.   $Q.push\_back(i)$ 
11. return  $F[1]$ 

```

2.2. Generalization

Observe that lemma 2.2 and 2.3 only depend on lemma 2.1 but not on the explicit formula of g . Thus, Brucker's algorithm will work as long as we can define a function g satisfying lemma 2.1.

2.3. Applications

2.3.1. Leaves <http://vn.spoj.pl/problems/NKLEAVES/en/>

Solution

First, let's reverse the order of the leaves, so that our notations will be consistent with the problem in section 2.1. We now move the leaves to the right, and the last pile will now be at coordinate n .

Let w_i be the weight of leaf i and c_{ij} be the total cost to gather leaves $i..j$ into a pile at position

$$j. \text{ We have: } c_{ij} = \sum_{x=i}^j (j-x)w_x = j \sum_{x=i}^j w_x - \sum_{x=i}^j xw_x$$

$$\text{Define } s_i = \sum_{x=i}^n w_x \text{ and } p_i = \sum_{x=i}^n xw_x, \text{ we have: } c_{ij} = j(s_i - s_{j+1}) + p_{j+1} - p_i$$

For $i < k < l$, we have:

$$\begin{aligned}
c_{il} - c_{ik} &= l(s_i - s_{l+1}) + p_{l+1} - p_i - k(s_i - s_{k+1}) - p_{k+1} + p_i \\
&= (l-k)s_i - ls_{l+1} + ks_{k+1} + p_{l+1} - p_{k+1} \quad (2.3)
\end{aligned}$$

Let F_{ij} be the minimum cost to gather leaves $i..n$ into j piles. Define $F_{ij}(k) = c_{ik} + F_{k+1, j-1}$

We have:

$$F_{ij} = \min_{i \leq k \leq n} \{F_{ij}(k)\} \quad \forall 1 \leq i \leq n, j > 1$$

The base case is $F_{i,1} = c_{i,n} \quad \forall 1 \leq i \leq n$ and $F_{n+1,j} = \infty \quad \forall j > 0$

An obvious algorithm to compute F takes $O(n^2k)$ time (where n is the number of leaves and k is number of needed piles). We will apply Brucker's algorithm to reduce the complexity to $O(nk)$. To define a function g as shown in section 2.2, we should begin by solving the inequality $F_{ij}(k) \leq F_{ij}(l)$ for $i \leq k < l$.

We have:

$$F_{ij}(k) \leq F_{ij}(l)$$

$$\Leftrightarrow c_{ik} + F_{k+1,j-1} \leq c_{il} + F_{l+1,j-1}$$

$$\Leftrightarrow F_{k+1,j-1} - F_{l+1,j-1} \leq c_{il} - c_{ik} = (l-k)s_i - ls_{l+1} + ks_{k+1} + p_{l+1} - p_{k+1} \quad (\text{by 2.3})$$

$$\Leftrightarrow \frac{F_{k+1,j-1} - F_{l+1,j-1} + ls_{l+1} - ks_{k+1} - p_{l+1} + p_{k+1}}{l-k} \leq s_i$$

At a specific step j when we need to compute all the values $F_{1j}, F_{2j}, \dots, F_{nj}$, the left hand side of the above inequality only depends on k and l, so we can define:

$$g(k, l) = \frac{F_{k+1,j-1} - F_{l+1,j-1} + ls_{l+1} - ks_{k+1} - p_{l+1} + p_{k+1}}{l-k}$$

and obtain the following result: $F_{ij}(k) \leq F_{ij}(l) \Leftrightarrow g(k, l) \leq s_i$

Since s_i is non-increasing, this is exactly lemma 2.1. Thus, we can apply Brucker's algorithm to compute all the values $F_{1j}, F_{2j}, \dots, F_{nj}$ in $O(n)$ time.

We need to loop through $j=1, 2, \dots, k$ where k is the number of piles needed. Therefore, the total complexity is $O(nk)$.

2.3.2. Batch Scheduling (IOI 2002)

This problem is a direct application of Brucker's algorithm.

References

1. *Efficient dynamic programming using quadrangle inequalities*. Yao, F. Frances. 1980.
2. *Efficient algorithms for some path partitioning problems*. Brucker, Peter. 1-3, 1995, Discrete Applied Mathematics, Vol. 62.