Finding Minimum in Monotone Matrices

Yanheng Wang

Consider the task of finding the minimum in a matrix $A \in \mathbb{R}^{m \times n}$. In practice A often possesses certain structure for exploitation; this note concerns with what people call monotonicity.

For each row $i \in [m]$, let $j(i)$ be the column that contains the leftmost smallest element in that row.

Definition 1. Matrix $A \in \mathbb{R}^{m \times n}$ is *monotone* if $j(1) \leqslant j(2) \leqslant \cdots \leqslant j(m)$.

If A is monotone then every submatrix formed by deleting some rows is also monotone. So we can compute the $j(i)$'s by divide-and-conquer on rows:

fn find-smallest(A)

> *{assume monotone $A \in \mathbb{R}^{m \times n}$; shall find $j(i)$ for every $i \in [m]$}*
> **if** $m \leqslant 2$ **then**
> > for $i \in [m]$, find $j(i)$ in brute-force
>
> **else**
> > let A' be the submatrix formed by even rows
> > find-smallest(A') *{so we get $j(2), j(4), \ldots$}*
> > **for** $k = 1, 2, \ldots, m/2$ **do**
> > > find $j(2k-1)$ by scanning entries $\{2k-1\} \times [j(2k-2), j(2k)]$

By monotonicity the for-loop costs $O(m+n)$ time only. Hence we have recursion

$$T(m,n) = T(m/2,n) + O(m+n).$$

Expanding it, we see

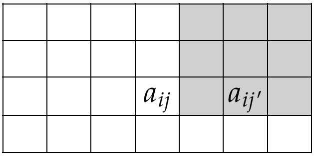
$$T(m,n) = \sum_{t=0}^{\log m} O\left(\frac{m}{2^t} + n\right) = O(m + n \log m).$$

The running time is especially good when $m \geqslant n$, that is when the matrix is "slim".

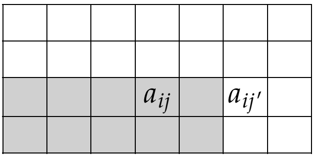
What about “fat” matrices for which $m < n$? (Think of $m = 3$ and $n = 100$ for example.) Well, in the end at most m columns can show up as $j(i)$'s. So our goal is to discard irrelevant columns quickly, thus making the matrix slim.

Here is a simple heuristic. Pick a row $i \in [m]$ and two columns $1 \leqslant j < j' \leqslant n$.

- If $a_{ij} \leqslant a_{ij'}$ then we know $j(1) \leqslant \cdots \leqslant j(i) \leqslant j$ by total monotonicity. So we may discard the box $[1, i] \times [j+1, n]$, meaning that these entries may never be minima of their respective rows.



- If $a_{ij} > a_{ij'}$ then we know $j(m) \geqslant \cdots \geqslant j(i) \geqslant j'$. So we may discard the box $[i, m] \times [1, j'-1]$.



By repeated applications of the heuristic, one can discard all irrelevant entries. But more strategy is needed as we care about efficiency.

For each column $j \in [n]$, we maintain an integer $d(j) \in [0, m]$ such that the topmost $d(j)$ entries in the column were already discarded. Initially $d(j) = 0$ and all columns are *active*.

In each iteration, we pick the leftmost active column j that maximises $d(j)$. Select row $i := d(j) + 1$ and the nearest active column $j' > j$. (If j' does not exist then we terminate.) Compare the entries a_{ij} and $a_{ij'}$ as above. In the first case we set $d(j') := i > d(j)$. In the second case we set $\alpha(j) = m$ and declare column j *dead*.

Note that we always make progress: either the largest $d(j)$ among active columns increases, or an active column becomes dead. So the number of iterations is at most $m + n$.

It is easy to argue inductively that

> After iteration t, we have only looked at columns $[t]$. Among the active columns $j \in [t]$, the value $d(j)$ strictly increases with j.

When the process terminates, all columns to the right of j are dead, so $t \geqslant n$. Hence $d(j)$ strictly increases across *all* active columns. Consequently, there can be at most m active columns. The dead columns can be safely discarded.

The process can be implemented with a stack:

```
fn reduce(A)
    {assume monotone A ∈ ℝ^{m×n}; shall discard all but m columns}
    initialise d(j) := 0 for all j ∈ [n]
    let S be an empty stack
    S.push(1)    {initially j = 1}
    j' := 2    {active column right next to j}
    repeat
        j := S.top()
        i := d(j) + 1
        if a_{ij} ⩽ a_{ij'} then
            d(j') := i
            S.push(j')
            j' := j' + 1
        else
            d(j) := m; declare j dead
            S.pop()
    until j' > n
    return the active columns of A
```

Now that the fat A is trimmed to a slim A', we want to apply find-smallest(A'). However, the submatrix might or might not be monotone. This is why we strengthen Definition 1 as follows:

Definition 2. A matrix is *totally monotone* if every submatrix is monotone (in the sense of Definition 1).

Exercise. Show that every vector is totally monotone.

Definition 2 allows us to recurse on the submatrix, applying reduction whenever necessary. The final algorithm, named after its inventors Shor, Moran, Aggarwal, Wilber and Klawe, is summarised below:

Algorithm SMAWK(A)

{*assume totally monotone $A \in \mathbb{R}^{m\times n}$; shall find $j(i)$ for every $i \in [m]$*}
if $m \leqslant 2$ **then**
  for $i \in [m]$, find $j(i)$ in brute-force
else
  if $m < n$ **then**
    $A := \text{reduce}(A)$
  let A' be the submatrix formed by even rows
  find-smallest(A') {*so we get $j(2), j(4), \ldots$*}
  for $k = 1, 2, \ldots, m/2$ **do**
    find $j(2k-1)$ by scanning entries $\{2k-1\} \times [j(2k-2), j(2k)]$

So the time recursion writes

$$T(m,n) = T(m/2, \min\{m,n\}) + O(m+n).$$

Suppose we start with a slim matrix. Within $\log(m/n) \leqslant m/n$ recursive calls the matrix becomes fat. Each call spends time $O(n)$ in the for-loop, so they cost time $O(m)$ altogether.

As soon as we have reached a fat matrix, the reduction comes into play. It ensures that n shrinks almost synchronously with m, so the recursion is essentially $T(m+n) = T((m+n)/2) + O(m+n)$, which has solution $T(m+n) = O(m+n)$. Putting the two phases together, the total running time is thus $T(m,n) = O(m+n)$.

Finally we take a closer look at Definition 2. Though restrictive, it still covers a wide range of matrices, for example Monge matrices.

Definition 3. A matrix $A = (a_{ij})$ is *Monge* if $a_{ij} + a_{i+1,j+1} \leqslant a_{i,j+1} + a_{i+1,j}$ for all i, j.

Example. Let $f : [m] \to \mathbb{R}$ and $g : [n] \to \mathbb{R}$ be two functions. The matrix defined by $a_{ij} := f(i) + g(j)$ is Monge.

Example. Take m and n points on two parallel lines, respectively. Define a_{ij} as the distance from the i-th point on the first line to the j-th point on the second line. The Monge property follows from triangle inequality.

Exercise. Show that a matrix $A = (a_{ij})$ is Monge iff $a_{ij} + a_{i'j'} \leqslant a_{ij'} + a_{i'j}$ for all $i < i'$ and $j < j'$.

Lemma 4. Every Monge matrix is totally monotone.

Proof. Suppose A is not totally monotone. Then there exists rows $i < i'$ such that $j := j(i') < j(i) =: j'$. In particular, $a_{ij} > a_{ij'}$ and $a_{i'j} \leqslant a_{i'j'}$, thus $a_{ij} + a_{i'j'} > a_{ij'} + a_{i'j}$, contradicting the definition of Monge matrices. □