1. SMAWK

Nesta seção discutiremos o algoritmo SMAWK. Ele é conhecido pela sua aplicação no problema de encontrar o vértice mais distante de cada vértice num poígono convexo em tempo linear [1]. Ao final desta seção serão explicadas esta e outras aplicações deste algoritmo.

Dada uma matriz $A \in \mathbb{Q}^{n \times m}$, listamos os casos de uso deste algoritmo:

- Se A é totalmente monótona convexa ou côncava nas linhas podemos encontrar os índices de mínimos e máximos das linhas em tempo $\mathcal{O}(n+m)$ e
- se A é totalmente monótona convexa ou côncava nas colunas podemos encontrar os índices de mínimos e máximos das colunas em tempo $\mathcal{O}(n+m)$.

Apresentaremos o caso onde A é totalmente monótona convexa nas linhas e estamos interessados nos índices de mínimos. Na Subseção 1.6 explicamos como reduzir os problemas elencados para este caso.

1.1. **Técnica Primordial.** Para facilitar a compreensão do algoritmo SMAWK, iremos apresentar uma técnica parecida com a Divisão e Conquista apresentada na Seção **??** e mostrar uma otimização desta técnica que leva ao algoritmo SMAWK.

Dada uma matriz $A \in \mathbb{Q}^{n \times m}$ totalmente monótona convexa por linhas, queremos encontrar o menor índice de mínimo de cada uma das linhas de A. Se para uma dada linha i onde $i > 0$ e $i < n$ conhecermos os índices l e r de mínimos das linhas $i-1$ e $i+1$, respectivamente, já que A tem os índices de mínimos das linhas crescente (por ser totalmente monótona) basta buscar o índice de mínimo da linha i no intervalo entre l e r (inclusive). Além disso, se i é a primeira linha da matriz podemos considerar $l = 1$ ou se i é a última linha da matriz podemos considerar $r = n$ sem perder a validade do fato de que basta buscar entre l e r.

Após realizar as observações acima note que, já que A é totalmente monótona, remover qualquer linha de A mantém a total monotonicidade e não altera o índice de mínimo de outra linha. Com esta observação, concluímos que podemos remover todas as linhas pares da matriz, resolver o problema recursivamente para a matriz resultante e utilizar este resultado para calcular os índices de interessa para as linhas pares da matriz.

Com uma análise similar à realizada para a técnica da Divisão e Conquista é fácil concluir que uma implementação desta técnica que consiga remover as linhas pares da matriz (e adicionar elas de volta) em tempo $\mathcal{O}(1)$ resolve o problema em tempo $\mathcal{O}((n+m)\lg(n))$, assim como a técnica da divisão e conquista.

1.2. **Reduce.** Queremos agilizar a técnica apresentada acima. Para isso, vamos adicionar uma nova hipótese. Vamos supor que a matriz A é quadrada, ou seja, $n = m$. Lembre que a cada passo, removemos as $\lfloor n/2 \rfloor$ linhas pares da matriz gerando uma nova matriz A', resolvemos o problema

recursivamente para A' e usamos a solução de A' para resolver para as linhas restantes de A. O problema é que quando removemos linhas da nossa A, ela deixa de ser quadrada e passa a ser uma matriz com mais colunas do que linhas, isto é, $m \geq n$. Chamamos de ótimas as células da matriz que são mínimo de alguma linha e as colunas que contém o índice de mínimo de pelo menos uma linha. Note que uma matriz contém no máximo n colunas ótimas, pois cada linha faz com que exatamente uma célula seja ótima. Queremos remover colunas não ótimas de uma matriz A com mais colunas do que linhas fazendo com que A se torne quadrada.

Vamos desenvolver o algoritmo REDUCE a partir de um índice de linha k e de algumas invariantes:

(1) Vale $1 \leq k \leq n$,
(2) apenas colunas não ótimas foram removidas da matriz e
(3) toda célula em uma coluna de índice menor ou igual a k que possua índice de linha menor do que índice de coluna é não ótima. A Figura 1.1 representa, em azul, a célula de índice k, k e, em preto, as células que, segundo esta invariante, são não ótimas.

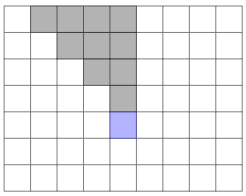

FIGURA 1.1. Invariante 3 do REDUCE.

Vamos comparar $A[k][k]$ com $A[k][k+1]$ e considerar dois casos. Em cada um dos casos, concluiremos que algumas células da matriz A são não ótimas. A Figura 1.2 mostra, hachuradas em vermelho, as células que são descobertas não ótimas quando $A[k][k] > A[k][k+1]$ e, com linhas verticais verdes, as células que são descobertas não ótimas quando $A[k][k] \leq A[k][k+1]$. Vamos provar estas implicações.

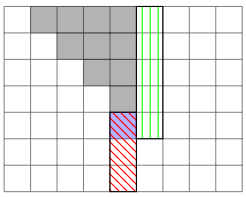

FIGURA 1.2. Casos do REDUCE.

Se $A[k][k] > A[k][k+1]$, as entradas com índice de linha maior ou igual a k na coluna k são não ótimos. Primeiramente, a célula (k, k) é não ótima como consequência direta da desigualdade. Agora, suponha que existe alguma

linha $i > k$ tal que $A[i][k] \leq A[i][k+1]$. Pela total monotonicidade convexa por linhas de A, isso implica em $A[k][k] \leq A[k][k+1]$, um absurdo.

Se $A[k][k] \leq A[k][k+1]$, as células da coluna $k+1$ com índices de linha menores ou iguals a k são não ótimas. A célula $(k, k+1)$ é não ótima pela desigualdade apresentada. Supojnha que existe alguma linha $i < k$ tal que $A[i][k] > A[i][k+1]$. Pela contrapositiva da total monotonicidade, temos $A[k][k] > A[k][k+1]$, um absurdo.

Com estas observações estamos prontos para deduzir um algoritmo que elimina exatamente $m - n$ colunas de A.

Algoritmo 1.3 Algoritmo REDUCE

```
1:  função REDUCE(A)
2:      k ← 1
3:      while A tem mais linhas do que colunas do
4:          se A[k][k] > A[k][k+1] então
5:              Remove a coluna k
6:              k ← max(1, k − 1)
7:          senão
8:              se k = n então
9:                  Remove a coluna k + 1
10:             senão
11:                 k ← k + 1
12:     devolve A
```

É fácil ver que as invariantes são válidas neste algoritmo. Olhamos para o primeiro passo, $k = 1$, nenhuma coluna foi removida ainda e não há elementos com índices de linha e coluna menores do que k, logo, as Invariantes 1, 2 e 3 valem. Em todo passo do loop, $A[k][k+1]$ existe, pois $k \leq n$ e $n \leq m$. Consideramos o caso onde $A[k][k] > A[k][k+1]$, a Invariante 1 sempre se mantém trivialmente, já provamos que a coluna k é não ótima neste caso, portanto, a Invariante 2 se mantém mesmo após a remoção da coluna k. Agora, se $k = 1$, vale a 3 por vacuidade e, no caso contrário, já que a k decresce, a Invariante 3 também se mantém.

Em outro caso, valem $A[k][k] \leq A[k][k+1]$ e $k = n$. Foi provado que os elemntos de linhas menores ou iguals a k na coluna coluna $k+1$ são não ótimos, porém, isto representa toda a coluna $k+1$, assim, remover ela mantém a Invariante 2. As outras duas invariantes se mantém trivialmente. Agora, falta considerar o caso onde $A[k][k] \leq A[k][k+1]$ e $k < n$. Neste caso, foi provado, novamente, que as células com índices menores ou iguais a k na coluna $k+1$ são inválidos. Estes são exatemente os elementos com índices de linhas menores do que índices de colunas na coluna $k+1$, o que faz com que a Invariante 3 se mantenha ao incrementarmos o valor de k. As outras duas invariantes se mantém trivialmente neste caso.

Além disso, o algoritmo, a cada passo, incrementa k ou remove uma coluna de A. Sabemos que k nunca passa de n e, já que a matriz tem m colunas, não

podemos remover mais do que m colunas. Supondo que a cada remoção de coluna k seja decrementado, chegamos a uma quantidade máxima de $2m+n$ passos. Supondo que as remoções sejam feitas em tempo constante, o tempo de cada passo é constante, portanto, atingimos uma complexidade de $\mathcal{O}(m)$ operações no algoritmo REDUCE, já que $n \leq m$.

1.3. **SMAWK.**

1.4. **Análise.**

1.5. **Implementação.**

1.6. **Generalizações.**

Referências

[1] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1):195–208, 1987.

[2] Wolfgang Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The knuth-yao quadrangle-inequality speedup is a consequence of total monotonicity. *ACM Trans. Algorithms*, 6(1):17:1–17:22, December 2009.

[3] Zvi Galil and Kunsoo Park. Dynamic programming with convexity, concavity and sparsity. *Theoretical Computer Science*, 92(1):49 – 76, 1992.

[4] F. Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, STOC '80, pages 429–435, New York, NY, USA, 1980. ACM.