Algoritmos em matrizes monótonas e Monge convexas

Trabalho de conclusão de curso
Universidade de São Paulo
Bacharelado em Ciência da Computação, 2017

Victor Sena Molero
Orientadora: Cristina G. Fernandes

Trabalho fomentado pela bolsa PIBIC do CNPq.

Resumo

As matrizes Monge têm propriedades como a monotonicidade total e a monotonicidade, que são exploradas por uma série de algoritmos para a busca de máximos e mínimos em linhas e colunas destas matrizes. Algumas destas técnicas são populares pois são utilizadas para a agilização de algoritmos para problemas clássicos e aparecem com crescente frequência em competições de programação por sua utilidade na solução eficiente de problemas de programação dinâmica.

Busca-se neste trabalho formalizar, compilar e aprofundar os conhecimentos adquiridos sobre estes algoritmos por meio do estudo da literatura já existente a fim de facilitar o aprendizado destas técnicas. Além disso, procura-se generalizar os algoritmos inspirando-se em desafios propostos em competições, explorando suas limitações e modificando as formalizações encontradas na literatura para ampliar os escopos tradicionais.

Palavras-chave: Monge, programação competitiva, otimização, programação dinâmica, desigualdade quadrangular.

Sumário

Capítulo 1

Introdução

Neste trabalho vamos estudar algoritmos sobre matrizes Monge e matrizes com propriedades parecidas, como a monotonicidade total e a monotonicidade. As matrizes Monge foram estudadas inicialmente em 1781 por Gaspard Monge [15]. Ghys [19] escreveu um bom artigo sobre os estudos realizados por Monge. O problema pelo qual Monge se interessou consiste de duas regiões de igual área no plano. Uma das regiões está preenchida de terra e a outra está vazia. O objetivo é transportar toda a terra entre as regiões podendo dividir a terra em infinitas partes de forma a minimizar a soma, para cada porção de terra, do produto de sua área por sua distância percorrida. É interessante notar que Monge estudou também a versão em 3 dimensões deste problema.

Ao estudar este problema, Monge fez uma primeira observação simples, porém muito importante. Se duas partículas de terra, uma inicialmente em um ponto A e a outra em um outro ponto B, forem enviadas, respectivamente, para os pontos b e a, então os caminhos entre A e b e B e a não podem se cruzar em uma solução ótima. Se estes caminhos se cruzassem, seria menos custoso enviar a primeira porção para o ponto a e a segunda para o ponto b. Séculos mais tarde, em 1961, Hoffman [12] utilizou esta propriedade ao estudar problemas de transporte e creditou sua descoberta a Monge, cunhando o termo "Propriedade de Monge".

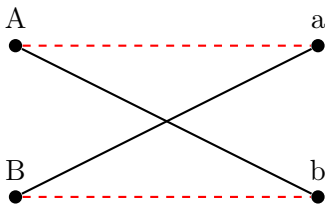

Figura 1.1: Desigualdade quadrangular. A soma dos tamanhos dos segmentos em preto é maior do que a soma dos tamanhos dos segmentos pontilhados em vermelho.

Como a Figura 1.1 indica, a propriedade de Monge possui uma interpretação geométrica simples e interessante: em um quadrilátero convexo, a soma dos tamanhos das diagonais supera a soma dos tamanhos de dois lados paralelos. Uma forma de modelar esta propriedade é a criação de matrizes Monge. Grande parte da importância das matrizes Monge é consequência de seu relacionamento natural com geometria.

Aggarwal, Klawe, Moran, Shor e Wilber [3], percebendo esta utilidade, estudaram as matrizes Monge e desenvolveram o algoritmo SMAWK para a busca de valores ótimos em linhas e colunas

de matrizes totalmente monótonas. Além disso, mostraram modelagens interessantes de problemas geométricos sobre matrizes deste tipo, resolvendo tais problemas com eficiência inédita. Estas descobertas incrementaram o interesse acadêmico sobre as matrizes Monge e desencadearam uma série de artigos que demonstram a utilidade delas. Burkard, Klinz e Rudolf [6] compilaram muito do conhecimento desenvolvido acerca do tópico. Esta introdução, até aqui, é baseada na introdução de tal compilação.

Com algumas destas descobertas em mente, Park [16], em sua tese, resumiu bem a motivação para o estudo das matrizes Monge. As propriedades destas matrizes permitem que certas entradas importantes sejam encontradas sem a necessidade de analisar toda a matriz e muitos problemas computacionais podem ser reduzidos a encontrar tais entradas neste tipo de matriz. A combinação destas duas afirmações faz com que estas matrizes muitas vezes permitam algoritmos especialmente eficientes para a solução de diversos problemas.

Além de suas aplicações geométricas, as matrizes Monge são úteis em diversas aplicações relacionadas a programação dinâmica [4, 6, 11]. Yao [17, 18] explicitou esta relação ainda antes do desenvolvimento do algoritmo SMAWK. Esta utilidade, que vai além das descobertas de Yao, fez com que as técnicas baseadas em matrizes Monge se tornassem populares em competições de programação como a Maratona de Programação e a ICPC.

Este trabalho explica alguns dos algoritmos conhecidos que se baseiam em propriedades relacionadas às matrizes Monge, implementa os algoritmos estudados e exemplifica algumas aplicações dos mesmos. O Capítulo 2 introduz as matrizes Monge, monótonas e totalmente monótonas, introduzindo também resultados importantes sobre elas que são utilizados durante todo o texto.

Os Capítulos 3 e 4 apresentam os algoritmos para busca de valores ótimos em linhas e colunas de matrizes Monge. O Capítulo 3 apresenta o método conhecido como otimização da divisão e conquista e o Capítulo 4 apresenta o algoritmo SMAWK, que já foi citado. Durante estes capítulos, é exemplificada a relação entre as matrizes Monge e programação dinâmica e alguns problemas são dados como exemplos. Um problema geométrico também é explicado e resolvido.

O Capítulo 5 introduz a otimização de Knuth-Yao, um método que agiliza certas soluções de programação dinâmica. Knuth [14] desenvolveu tal método e Yao [17, 18] notou sua relação com as matrizes Monge. Apresentamos o método demonstrando os resultados obtidos por Yao e aplicamos o conhecimento para resolver um problema.

O Capítulo 6 é focado em programação dinâmica. Ele apresenta um exemplo de solução em programação dinâmica que pode ser agilizado de maneira não trivial sem utilizar propriedades relacionadas a matrizes Monge. O objetivo deste capítulo é apresentar a possibilidade da utilização de estruturas de dados em programação dinâmica.

O Capítulo 7 apresenta uma estrutura de dados relacionada às matrizes Monge que é útil para agilizar algumas soluções de programação dinâmica de maneira parecida com a do Capítulo 6. O método apresentado encontra valores ótimos em linhas ou colunas de matrizes Monge onde as entradas não são todas conhecidas a priori que respeitam um certo formato. Este formato é natural para aplicações de programação dinâmica, o que demonstra a utilidade de tal técnica.

Finalmente, o Capítulo 8 documenta os exemplos de aplicações implementados que utilizam os algoritmos estudados e o Capítulo 9 contém uma apreciação subjetiva do autor acerca do trabalho.

1.1 Conteúdo

O conteúdo deste trabalho de conclusão de curso está acessível a partir do endereço http://linux.ime.usp.br/~victorsenam/mac0499. Lá estão disponíveis links para este texto, o pôster de apresentação do trabalho e o diretório de implementações. Além disso, o código fonte completo do trabalho está disponível em http://github.com/victorsenam/tcc.

1.2 Notação

Se i e j são dois inteiros, a expressão $[i\,..\,j]$ denota o conjunto $\{k \in \mathbb{Z} \mid i \leq k \leq j\}$. Além disso, se n é inteiro, $[n]$ denota $[1\,..\,n]$. Note que se $i > j$, o conjunto $[i\,..\,j]$ é o conjunto vazio. Se v é um vetor, podemos escrever $v[i\,..\,j]$ para denotar o vetor $(v_i, v_{i+1}, \ldots, v_j)$. O mesmo vale em termos de submatrizes. Se r e ℓ também são dois inteiros e A é uma matriz, podemos escrever $A[i\,..\,j][\ell\,..\,r]$ para denotar a submatriz de A que contém apenas as linhas de A em $[i\,..\,j]$ e as colunas em $[\ell\,..\,r]$. Por padrão, nestes dois últimos casos, os subvetores e as submatrizes geradas são reindexados pelos conjuntos $[j - i + 1]$ e $[j - i + 1] \times [r - \ell + 1]$, respectivamente.

Se A e B são conjuntos, podemos escrever A^B para denotar o conjunto de vetores com entradas em A indexados pelos elementos de B. Ainda se C também é um conjunto, $A^{B\times C}$ denota o conjunto das matrizes com entradas em A, com linhas indexadas por elementos de B e colunas indexadas por elementos de C. Por exemplo, se $v \in \mathbb{R}^{[3\,..\,5]}$ então v é um vetor com entradas reais v_3, v_4 e v_5. Definimos, ainda, para cada n e m inteiros e cada conjunto A, os conjuntos $A^n = A^{[n]}$ e $A^{n\times m} = A^{[n]\times[m]}$.

1.3 Implementações

As implementações em `C++` das técnicas discutidas neste trabalho podem ser encontradas no diretório de implementações na pasta `algoritmos`. O nome do arquivo é indicado no texto relacionado à técnica correspondente. Exemplos de uso dos algoritmos implementados podem ser encontrados na pasta `exemplos` e são documentados no Capítulo 8.

Trabalharemos com funções que buscam certos valores em uma matriz sem analisar cada uma das posições da matriz. Por causa deste fato, não faz sentido construir explicitamente as matrizes para poder aplicar os algoritmos, já que isso custaria mais tempo do que o próprio algoritmo. Em vez de implementar os algoritmos sobre a representação usual de matrizes em `C++`, com vetores bidimensionais, representaremos uma matriz com os seguintes elementos: uma função f e dois valores n e m. Estes três parâmetros representam a matriz A com n linhas e m colunas tal que, para cada $i \in [n]$ e $j \in [m]$, vale que $f(i, j) = A[i][j]$. Na prática, isso significa que as matrizes com as quais trabalhamos têm entradas que podem ser calculadas de maneira eficiente quando necessário. Assumimos que cada chamada à função f custa tempo $\mathcal{O}(1)$.

Nas implementações, trabalhamos com matrizes, vetores e funções indexados por 0. Durante o texto os algoritmos e as análises apresentados são feitos baseando-se em índices que começam em 1. Esta mudança pode causar diferenças sutis no código em relação aos algoritmos apresentados no texto.

Capítulo 2

Matrizes Monge e monotonicidade

Neste capítulo serão apresentados e explorados os conceitos de monotonicidade, convexidade e matrizes Monge. Além disso, alguns resultados referentes a estes conceitos serão demonstrados. As definições e os resultados desta seção são fundamentais para o desenvolvimento do restante do trabalho. Na Seção 2.1 apresentamos vários destes conceitos básicos. A Seção 2.2 apresenta as matrizes Monge e as relaciona com os conceitos básicos já apresentados. A Seção 2.3 define matrizes triangulares no contexto de matrizes Monge, adaptando a elas os conceitos já apresentados durante o capítulo.

2.1 Conceitos básicos

Definição 2.1 (Vetor monótono)**.** *Um vetor $a \in \mathbb{Q}^n$ é dito monótono quando vale uma das propriedades abaixo.*

- *Se, para todo $i, j \in [n]$, $i < j$ implica $a_i \leq a_j$, a é dito monótono crescente (ou só crescente).*
- *Se, para todo $i, j \in [n]$, $i < j$ implica $a_i \geq a_j$, a é dito monótono decrescente (ou só decrescente).*

Sabemos que a monotonicidade de vetores pode ser aproveitada para agilizar alguns algoritmos importantes. Por exemplo, a busca binária pode ser interpretada como uma otimização da busca sequencial para vetores monótonos.

Definição 2.2 (Função convexa)**.** *Seja $g : \mathbb{Q} \to \mathbb{Q}$ uma função.*

- *Se, para todo par de pontos $x, y \in \mathbb{Q}$ e $\lambda \in \mathbb{Q}$ que respeita $0 \leq \lambda \leq 1$, vale a desigualdade $g(\lambda x + (1-\lambda)y) \leq \lambda g(x) + (1-\lambda)g(y)$, então g é dita convexa.*
- *Se, para todo par de pontos $x, y \in \mathbb{Q}$ e $\lambda \in \mathbb{Q}$ que respeita $0 \leq \lambda \leq 1$, vale a desigualdade $g(\lambda x + (1-\lambda)y) \geq \lambda g(x) + (1-\lambda)g(y)$, então g é dita côncava.*

Proposição 2.3. *A função $g(x) = x^2$ é convexa.*

Demonstração. Tome quaisquer três valores x, y e $\lambda \in \mathbb{Q}$ de forma que $0 \leq \lambda \leq 1$. Queremos mostrar que $(\lambda x + (1-\lambda)y)^2 \leq \lambda x^2 + (1-\lambda)y^2$. Expandindo o lado esquerdo da

desigualdade, obtemos $\lambda^2x^2 + (1-\lambda)^2y^2 + 2\lambda(1-\lambda)xy \leq \lambda x^2 + (1-\lambda)y^2$, o que equivale a $(\lambda^2 - \lambda)x^2 + ((1-\lambda)^2 - (1-\lambda))y^2 + 2(\lambda - \lambda^2)xy \leq 0$, que é verdade uma vez que $(\lambda^2 - \lambda)(x^2 + y^2 - 2xy) = (\lambda^2 - \lambda)(x+y)^2 \leq 0$. □

É interessante definir convexidade também em termos de vetores.

Definição 2.4 (Vetor convexo)**.** *Seja $a \in \mathbb{Q}^n$ um vetor.*

- *Se, para todo $i, j, k \in [n]$, $i < j < k$ implica $a_j \leq \frac{(j-k)a_i + (i-j)a_k}{i-k}$, a é dito convexo e*
- *Se, para todo $i, j, k \in [n]$, $i < j < k$ implica $a_j \geq \frac{(j-k)a_i + (i-j)a_k}{i-k}$, a é dito côncavo.*

Assim como a monotonicidade, a convexidade também é usualmente explorada para agilizar algoritmos. Por exemplo, se um vetor é convexo, podemos encontrar o valor mínimo do vetor com uma busca ternária em vez de percorrer todo o vetor.

Definição 2.5. *Seja $A \in \mathbb{Q}^{n \times m}$. Definimos quatro vetores a seguir.*

- *O vetor de índices de máximos das linhas de A guarda na posição i o número* $\max\{j \in [m] \mid A[i][j] \geq A[i][j'] \text{ para todo } j' \in [m]\}$.
- *O vetor de índices de mínimos das linhas de A guarda na posição i o número* $\min\{j \in [m] \mid A[i][j] \leq A[i][j'] \text{ para todo } j' \in [m]\}$.
- *O vetor de índices de máximos das colunas de A guarda na posição j o número* $\max\{i \in [n] \mid A[i][j] \geq A[i'][j] \text{ para todo } i' \in [n]\}$.
- *O vetor de índices de mínimos das colunas de A guarda na posição j o número* $\min\{i \in [n] \mid A[i][j] \leq A[i'][j] \text{ para todo } i' \in [n]\}$.

Note que o máximo de uma linha (ou coluna) foi definido como o maior índice que atinge o máximo e o mínimo foi definido como o menor índice que atinge o mínimo. Esta escolha foi feita para simplificar o Lema 2.9 à frente, porém os algoritmos e resultados discutidos neste trabalho funcionam (com pequenas adaptações) para diversas definições distintas destes vetores.

Dada uma matriz, encontrar estes vetores é um problema central para este trabalho. Neste momento é interessante classificar algumas matrizes de acordo com propriedades que vão nos ajudar a calcular os vetores de mínimos e máximos de maneira especialmente eficiente.

A Figura 2.6 resume as relações de implicação da classificação que será realizada. Os conceitos ilustrados nela serão apresentados a seguir.

Definição 2.7 (Matriz monótona)**.** *Seja $A \in \mathbb{Q}^{n \times m}$ uma matriz. Se A tiver o vetor de índices de mínimos das linhas monótono, A é dita monótona nos mínimos das linhas. Valem também as definições análogas para máximos ou colunas e pode-se especificar monotonicidade crescente ou decrescente.*

Definição 2.8 (Matriz totalmente monótona)**.** *Seja $A \in \mathbb{Q}^{n \times m}$ uma matriz.*

- *Se $A[i'][j] \leq A[i'][j']$ implica $A[i][j] \leq A[i][j']$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, então A é totalmente monótona convexa nas linhas.*

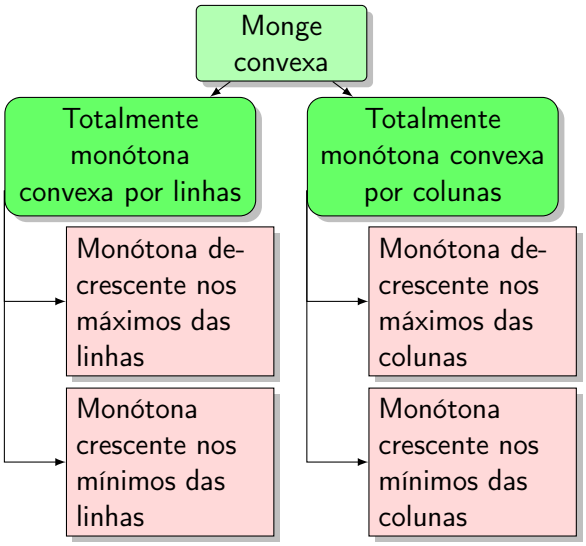

Figura 2.6: Comportamento dos vetores de índices ótimos em relação à convexidade.

- *Se* $A[i][j'] \leq A[i'][j']$ *implica* $A[i][j] \leq A[i'][j]$ *para todo* $i, i' \in [n]$ *e* $j, j' \in [m]$ *onde* $i < i'$ *e* $j < j'$*, então* A *é totalmente monótona convexa nas colunas.*

- *Se* $A[i'][j] > A[i'][j']$ *implica* $A[i][j] > A[i][j']$ *para todo* $i, i' \in [n]$ *e* $j, j' \in [m]$ *onde* $i < i'$ *e* $j < j'$*, então* A *é totalmente monótona côncava nas linhas.*

- *Se* $A[i][j'] > A[i'][j']$ *implica* $A[i][j] > A[i'][j]$ *para todo* $i, i' \in [n]$ *e* $j, j' \in [m]$ *onde* $i < i'$ *e* $j < j'$*, então* A *é totalmente monótona côncava nas colunas.*

O uso dos termos "convexa" e "côncava" em relação a matrizes durante o texto é explicado pelo Teorema 2.15. Note que se uma matriz é totalmente monótona, todas as suas submatrizes são totalmente monótonas no mesmo sentido.

Lema 2.9. *Se* $A \in \mathbb{Q}^{n \times m}$ *é uma matriz totalmente monótona convexa nas linhas, toda submatriz de* A *é monótona crescente nos mínimos das linhas e monótona decrescente nos máximos das linhas. Se* A *é totalmente monótona côncava nas linhas, toda submatriz de* A *é monótona decrescente nos mínimos das linhas e monótona crescente nos máximos das linhas. As afirmações valem similarmente em termos de colunas.*

Demonstração. Considere uma matriz A totalmente monótona convexa nas linhas. Sejam i e i' índices de linhas de A onde $i < i'$. Chamamos de j o índice de máximo da linha i e de j' o índice de máximo da linha i'. Queremos provar que os índices de máximo são decrescentes, portanto, vamos supor por absurdo que $j < j'$. Com isso, teremos que $A[i][j'] < A[i][j]$ e $A[i'][j] \leq A[i'][j']$. Porém, já que A é monótona convexa nas linhas, a segunda desigualdade implica que $A[i][j] \leq A[i][j']$, o que contradiz a primeira. Portanto, os índices de máximos são decrescentes.

Agora, considere novamente dois índices i e i' quaisquer de linhas de A onde $i < i'$. Denotamos por j o índice de mínimo da linha i' e por j' o índice de mínimo da linha i (note e inversão do nome dos índices). Vamos supor por absurdo que $j < j'$ e teremos que $A[i'][j] \leq A[i'][j']$ e $A[i][j'] < A[i][j]$. Novamente, usando o fato de que A é monótona convexa nas linhas, obtivemos uma contradição.

Finalmente, se A' é uma submatriz de A, então A' é totalmente monótona convexa nas linhas, portanto monótona crescente nos mínimos das linhas e monótona decrescente nos máximos das linhas.

As demonstrações no caso côncavo e nos casos relacionados a colunas são análogas. □

2.2 Matrizes Monge

Definição 2.10 (Monge convexidade)**.** *Seja $A \in \mathbb{Q}^{n\times m}$.*

1. *Se vale $A[i][j]+A[i'][j'] \leq A[i][j']+A[i'][j]$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, então A é dita Monge convexa.*

2. *Se vale $A[i][j]+A[i'][j'] \geq A[i][j']+A[i'][j]$ para todo $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, então A é dita Monge côncava.*

A desigualdade que define as matrizes Monge é conhecida pelos nomes "Propriedade de Monge" (em inglês, "Monge Property") [6] ou "Desigualdade Quadrangular" (em inglês, "Quadrangle Inequality") [4, 17]. O lema a seguir mostra que esta propriedade é mais forte do que a total monotonicidade.

Lema 2.11. *Se A é Monge convexa, A é totalmente monótona convexa tanto nas linhas quanto nas colunas. Se A é Monge côncava, A é totalmente monótona côncava tanto nas linhas quanto nas colunas.*

Demonstração. Seja A uma matriz Monge convexa. Suponha que vale, para certos $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, que $A[i'][j] \leq A[i'][j']$. Então, somamos esta desigualdade à definição de Monge convexa, obtendo que $A[i][j] \leq A[i][j']$, ou seja, A é totalmente monótona convexa nas linhas.

Por outro lado, se $A[i][j'] \leq A[i'][j]$ para certos $i, i' \in [n]$ e $j, j' \in [m]$ com $i < i'$ e $j < j'$, somamos esta desigualdade à da definição de Monge convexa e obtemos que $A[i][j] \leq A[i'][j]$. Assim, A é totalmente monótona convexa nas colunas.

A prova para o caso côncavo é análoga. □

Os algoritmos estudados durante este trabalho não utilizam diretamente a condição de Monge, apenas a total monotonicidade, a monotonicidade ou condições parecidas. Apesar disso, a condição de Monge é útil uma vez que implica na total monotonicidade e tem propriedades interessantes que facilitam a modelagem de problemas ou a prova de que alguns problemas têm propriedades necessárias para a aplicação dos algoritmos que nos interessam. O Teorema 2.12, por exemplo, mostra uma condição simples e de fácil visualização que é equivalente à condição de Monge.

Teorema 2.12. *Seja $A \in \mathbb{Q}^{n\times m}$.*

(a) Vale $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+1][j]$ para todo $i \in [n-1]$ e $j \in [m-1]$ se e somente se A é Monge convexa.

(b) Vale $A[i][j] + A[i+1][j+1] \geq A[i][j+1] + A[i+1][j]$ para todo $i \in [n-1]$ e $j \in [m-1]$ se e somente se A é Monge côncava.

Demonstração. Se $A \in \mathbb{Q}^{n\times m}$ é uma matriz Monge convexa, trivialmente vale, para todo par de $i \in [n-1]$ e $j \in [m-1]$, a desigualdade descrita pelo enunciado do teorema, isto é, $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+1][j]$. Vamos mostrar o outro lado do teorema.

Tomamos uma matriz $A \in \mathbb{Q}^{n\times m}$ e dois índices $i \in [n-1]$ e $j \in [m-1]$. Vamos provar, com indução em a, que $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+a][j]$ para todo $a \in [1\,..\,n-i]$. A base é o caso em que $a = 1$ e ela vale por hipótese. Para $a \in [2\,..\,n-i]$, assuma que a tese vale para $a-1$, ou seja, $A[i][j] + A[i+a-1][j+1] \leq A[i+a-1][j]$. Já que $i+a-1 \in [n-1]$, temos que $A[i+a-1][j] + A[i+a][j+1] \leq A[i+a-1][j-1] + A[i+a][j]$ e, somando as duas desigualdades, obtemos $A[i][j] + A[i+a][j+1] \leq A[i][j+1] + A[i+a][j]$, o que conclui a prova proposta neste parágrafo.

Agora tomamos novamente dois índices $i \in [n-1]$ e $j \in [m-1]$. Vamos provar que vale $A[i][j] + A[i+a][j+b] \leq A[i][j+b] + A[i+a][j+b]$ por indução em b para todo $a \in [1\,..\,n-i]$ e $b \in [1\,..\,n-j]$. A base desta indução é o caso em que $b = 1$ que foi provado no parágrafo anterior. Para $b \in [2\,..\,n-j]$, assumimos que a tese vale para $b-1$, ou seja, $A[i][j] + A[i+a][j+b] \leq A[i][j+b] + A[i+a][j+b]$. Então, pela prova do parágrafo anterior, vale que $A[i][j+b-1] + A[i+a][j+b] \leq A[i][j+b] + A[i+a][j+b-1]$ e, mais uma vez, somando as duas desigualdades, provamos que $A[i][j] + A[i+a][j+b] \leq A[i][j+b] + A[i+a][j]$. Com isso concluímos que A é Monge convexa.

A prova de (b) segue analogamente. □

Iremos discutir agora um problema que será resolvido com um algoritmo apresentado somente no Capítulo 4, o algoritmo SMAWK. Esse algoritmo não será explicado neste momento, mas será utilizado como caixa preta. Desta forma, poderemos introduzir de maneira gradual e motivada estes resultados que são importantes na aplicação prática dos conhecimentos discutidos aqui e demonstram a utilidade de pensar em matrizes Monge e não só na monotonicidade ou total monotonicidade.

Problema 2.13. *A função de custo c de cada vetor v é definida como $c(v) = \left(\sum_{i=1}^{|v|} v_i\right)^2$. Dados dois inteiros k e n e um vetor $v \in \mathbb{Q}_+^n$, particionar o vetor v em k subvetores de forma a minimizar a soma dos custos das partes. Formalmente, escolher um particionamento $p_0 = 1 \leq p_1 \leq p_2 \leq \cdots \leq p_k = n+1$ de v em subvetores que minimize $\sum_{i=1}^{k} c(v[p_{i-1}\,..\,p_i - 1])$.*

Definimos a matriz A onde $A[i][j] = \left(\sum_{\ell=1}^{j-1} v_\ell - \sum_{\ell=1}^{i-1} v_\ell\right)^2$ para todo $i, j \in [n+1]$. Quando $i \leq j$, vale que $A[i][j] = c(v[i\,..\,j-1])$. A matriz A não precisa ser explicitamente calculada. Pré-calculamos em $\mathcal{O}(n)$ o vetor a tal que $a_i = \sum_{\ell=1}^{i-1} v_\ell$ para todo $i \in [n+1]$. Com este vetor a, podemos calcular uma entrada i, j qualquer da matriz A em $\mathcal{O}(1)$, já que $A[i][j] = (a_j - a_i)^2$.

Podemos resolver o Problema 2.13 com programação dinâmica. Um subproblema de parâmetros ℓ e i é da forma: encontrar um melhor particionamento do vetor $v[i\,..\,n]$ em ℓ partes. Definimos a matriz $E \in \mathbb{Q}^{[k]\times[n+1]}$ de respostas desses subproblemas, assim, se ℓ e i definem um subproblema, o seu valor ótimo é guardado em $E[\ell][i]$. Vale a seguinte recorrência para E. Para todo $\ell \in [k]$ e $i \in [n+1]$,

$$E[\ell][i] = \begin{cases} A[i][n+1] & \text{se } \ell = 1 \text{ e} \\ \min\{A[i][j] + E[\ell-1][j] \mid j \in [i\,..\,n+1]\} & \text{caso contrário.} \end{cases}$$

Utilizando o fato de que sabemos calcular as entradas de A em $\mathcal{O}(1)$, fica fácil resolver a recorrência definida acima em tempo $\mathcal{O}(kn^2)$. Vamos simplificar a definição de E. Para todo $\ell \in [2\,..\,k]$, definimos a matriz B_ℓ que guarda, em cada entrada $i, j \in [n+1]$, o valor $A[i][j] + E[\ell-1][j]$. Perceba que para todo ℓ e i vale $E[\ell][i] = \min\{B_\ell[i][j] \mid j \in [i\,..\,n+1]\}$. Se $j < i$, é fácil mostrar que $E[j] \geq E[i]$ e, já que $A[i][i] = 0$ e $A[j][i] \geq 0$, temos $B_\ell[i][j] \geq B_\ell[i][i]$. Com isso, vale que $E[\ell][i] = \min\{B_\ell[i][j] \mid j \in [n+1]\}$.

O parágrafo acima mostra que $E[\ell][i]$ é o valor de mínimo da i-ésima linha da matriz B_ℓ. Com esta formulação, reduzimos o problema original a encontrar os mínimos das linhas de cada uma das matrizes B_ℓ com $\ell \in [2\,..\,k]$. O algoritmo SMAWK encontra mínimos de linhas de matrizes $(n+1) \times (n+1)$ totalmente monótonas por linhas em tempo $\mathcal{O}(n)$. Vamos mostrar que as matrizes B_ℓ são totalmente monótonas convexas por linhas para podermos aplicar o SMAWK.

Lema 2.14. *Sejam $A, B \in \mathbb{Q}^{n\times m}$ matrizes e $c \in \mathbb{Q}^m$ um vetor tais que $B[i][j] = A[i][j] + c[j]$ para todo $i \in [n]$ e $j \in [m]$. Se A é Monge convexa, B é Monge convexa. O mesmo vale se $c \in \mathbb{Q}^n$ e $B[i][j] = A[i][j] + c[i]$. Resultados análogos valem nos casos de concavidade.*

Demonstração. Suponha que A é Monge convexa. Vale, para quaisquer $i, i' \in [n]$ e $j, j' \in [m]$ onde $i < i'$ e $j < j'$, que $A[i][j] + A[i'][j'] \leq A[i'][j] + A[i][j']$. Somando $c[j] + c[j']$ nos dois lados, obtemos a desigualdade $A[i][j] + c[j] + A[i'][j'] + c[j'] \leq A[i'][j] + c[j'] + A[i][j'] + c[j]$ que equivale a $B[i][j] + B[i'][j'] \leq B[i'][j] + B[i][j']$. A prova para o caso onde $c \in \mathbb{Q}^n$ e $B[i][j] = A[i][j] + c[i]$ é análoga, bem como as provas para os casos côncavos. □

Suponha que a matriz A do Problema 2.13 é Monge convexa. Todas as matrizes B_ℓ se encaixam perfeitamente nas hipóteses do Lema 2.14 e, por isso, são Monge convexas, portanto, totalmente monótonas convexas por linhas. Basta provar que A é Monge convexa.

Teorema 2.15. *Sejam $A \in \mathbb{Q}^{n\times n}$ uma matriz, $w \in \mathbb{Q}_+^n$ um vetor e $g : \mathbb{Q} \to \mathbb{Q}$ uma função tais que para todo $i, j \in [n]$ vale $A[i][j] = g\left(\sum_{k=1}^{j} w_k - \sum_{k=1}^{i} w_k\right)$. Se g é convexa, A é Monge convexa. Similarmente, se g é côncava, A é Monge côncava.*

Antes de demonstrar este teorema, vamos usá-lo para provar que a matriz A do Problema 2.13 é Monge convexa. Considere a função $g(x) = x^2$. Vale, para todo $i, j \in [n]$, que $A[i][j] = g\left(\sum_{k=1}^{j} v_k - \sum_{k=1}^{i} v_k\right)$. Pela Proposição 2.3, g é convexa. Aplicamos o Teorema 2.15 para concluir que A é Monge convexa.

Com isso, já que o nosso problema se reduziu a encontrar, para todo $\ell \in [k]$, os mínimos das linhas da matriz B_ℓ e esta é Monge convexa, ela também é totalmente monótona convexa por linhas e podemos encontrar seus mínimos em $\mathcal{O}(n)$, resolvendo o problema todo em $\mathcal{O}(kn)$.

Resta provar o Teorema 2.15.

Demonstração do Teorema 2.15. Sejam A e g uma matriz e uma função quaisquer que respeitem as condições do enunciado do Teorema 2.15. Definimos, para todo par de índices $i, j \in [n]$, a função $s(i, i') = \sum_{k=1}^{i'} w_k - \sum_{k=1}^{i} w_k$. Note que desta maneira $A[i][j] = g(s(i, j))$. Sejam $i, i', j, j' \in [n]$ tais que $i < i'$ e $j < j'$. Vamos mostrar que vale a condição de Monge para estes índices, isto é, $A[i][j] + A[i'][j'] \leq A[i][j'] + A[i'][j]$. Definimos os valores $a = s(i, i')$, $b = s(j, j')$ e $z = s(j, i')$. É verdade que a condição de Monge para os índices fixados equivale à desigualdade $g(z + a) + g(z + b) \leq g(z) + g(z + a + b)$. Vamos provar que esta última vale.

Consideramos o caso em que $0 < a \leq b$. Temos que $z < z + a \leq z + b < z + a + b$. Definimos $\lambda = \frac{a}{a+b}$. Já que $0 \leq \lambda \leq 1$, $z + a = \lambda z + (1 - \lambda)(z + a + b)$ e $z + b = (1 - \lambda)z + \lambda(z + a + b)$, pela convexidade de g, obtemos que $g(z + a) \leq \lambda g(z) + (1 - \lambda)g(z + a + b)$ e também que $g(z + b) \leq (1 - \lambda)g(z) + \lambda g(z + a + b)$. Somando as duas desigualdades, concluímos que vale $g(z + a) + g(z + b) \leq g(z) + g(z + a + b)$. No caso em que $0 < b \leq a$ o mesmo resultado segue de maneira análoga. Finalmente, no caso onde $0 = a = b$ vale que $g(z) = g(z + a) = g(z + b) = g(z + a + b)$ e o resultado é trivial. □

2.3 Matrizes triangulares

Para alguns dos problemas e algoritmos discutidos neste trabalho, é interessante pensar em matrizes com entradas indefinidas. É possível definir uma matriz Monge com entradas arbitrárias indefinidas [6, Seção 9.4]. Vamos limitar quais posições podem ser indefinidas em uma matriz de forma a mantê-la útil e relacionada com os resultados já obtidos nesta seção.

Definição 2.16 (Matriz triangular)**.** *Seja c um inteiro. Uma matriz $A \in \mathbb{Q}^{n \times n}$ é triangular inferior em c se, para todo $i, j \in [n]$, $A[i][j]$ é indefinido se e somente se $i + c \leq j$. Similarmente, se, para todo $i, j \in [n]$, $A[i][j]$ é indefinido se e somente se $i + c \geq j$, a matriz A é triangular superior em c.*

Adaptaremos os conceitos e resultados apresentados neste capítulo para matrizes triangulares. Por simplicidade, vamos realizar estas adaptações em termos de matrizes triangulares inferiores no caso da convexidade. É fácil recriar as mesmas adaptações para matrizes triangulares superiores e para o caso da concavidade. Além disso, vamos comentar os conceitos apenas em termos de mínimos de linhas. As adaptações para máximos e colunas são análogas.

Várias das linhas de uma matriz triangular podem ser totalmente indefinidas. Isso faz com que o vetor de índices de mínimos das linhas da matriz seja mal definido. Vamos considerar que tal vetor está indexado apenas pelas linhas com pelo menos um valor definido. Se uma matriz não possui nenhum elemento definido nas linhas 1 e 2, por exemplo, como é o caso de uma triangular inferior em 2, consideramos que seu vetor de índices de mínimos das linhas está definido apenas para as posições em $[3 \,..\, n]$ onde n é a última linha da matriz.

Para que uma matriz $A \in \mathbb{Q}^{n \times n}$ triangular inferior em c seja totalmente monótona convexa nas linhas, é necessário que, para todo $i, i', j, j' \in [n]$ onde $i + c \leq i' + c \leq j \leq j'$, valha que $A[i'][j] \leq A[i'][j']$ implica em $A[i][j] \leq A[i][j']$. Isso quer dizer que toda submatriz de A que tem todas as entradas definidas é totalmente monótona convexas nas linhas. O Lema 2.9 vale para toda submatriz que tem todas as entradas definidas. É importante notar que os vetores de índices ótimos de matrizes totalmente monótonas triangulares **não** são necessariamente monótonos. Isso

vale apenas para os vetores de índices ótimos das submatrizes totalmente definidas desta triangular, que são totalmente monótonas.

De maneira similar, dizemos que uma matriz A triangular inferior em c de lado n é Monge convexa se a desigualdade quadrangular vale para todo $i, i', j, j' \in [n]$ onde $i + c \leq i' + c \leq j \leq j'$. O Lema 2.11 também vale para toda submatriz inteiramente definida de A. O Teorema 2.12, neste contexto, diz que, se $A[i][j] + A[i+1][j+1] \leq A[i][j+1] + A[i+1][j]$ sempre que $i, j \in [n-1]$ e $i + c + 1 \leq j$, então A é Monge convexa.

É útil perceber que, se $B \in \mathbb{Q}^n$ é uma matriz Monge qualquer, uma matriz $A \in \mathbb{Q}^n$ triangular tal que $A[i][j] = B[i][j]$ para toda entrada $A[i][j]$ definida de A também é Monge no mesmo sentido. Esta observação prova, imediatamente, o Lema 2.14 e o Teorema 2.15 para matrizes triangulares.

Capítulo 3

Divisão e conquista

Neste capítulo será apresentada uma técnica que chamamos de otimização por divisão e conquista. A ideia é citada por Aggarwal [3] e é um tópico recorrente em competições de programação, sendo conhecida como "Divide and Conquer Optimization" [1, 2] e geralmente aplicada a problemas de programação dinâmica.

Além disso, as hipóteses dos algoritmos decorrentes desta técnica são mais fracas do que as do algoritmo SMAWK, apresentado no Capítulo 4, portanto, todo problema para o qual o SMAWK pode ser aplicado, também pode ser resolvido com esta técnica. Ao final deste capítulo, apresentamos exemplos de aplicações em programação dinâmica e discutimos a interpretação desta técnica sob o olhar da programação competitiva.

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

- Se A é monótona nos mínimos das linhas, podemos encontrar os índices de mínimos das linhas em tempo $\mathcal{O}((n+m)\lg(n))$,
- se A é monótona nos máximos das linhas, podemos encontrar os índices de máximos das linhas em tempo $\mathcal{O}((n+m)\lg(n))$,
- se A é monótona nos mínimos das colunas, podemos encontrar os índices de mínimos das colunas em tempo $\mathcal{O}((n+m)\lg(m))$ e
- se A é monótona nos máximos das colunas, podemos encontrar os índices de máximos das colunas em tempo $\mathcal{O}((n+m)\lg(m))$.

Apresentaremos o caso em que A é crescente nos mínimos das linhas. É fácil manipular o algoritmo resultante para trabalhar com os outros casos.

3.1 Técnica

Dada uma matriz $A \in \mathbb{Q}^{n \times m}$ monótona crescente nos mínimos das linhas, queremos encontrar o vetor de índices de mínimos das linhas de A. Isto é, para todo $i \in [n]$, queremos encontrar

$$R[i] = \min\{j \mid A[i][j] \leq A[i][j'] \text{ para todo } j' \in [m]\}.$$

Se, para alguma linha i, encontrarmos o valor $R[i]$, sabemos, pela monotonicidade de A, que $R[i'] \leq R[i]$ para todo $i' < i$ e que $R[i'] \geq R[i]$ para todo $i' > i$, isto é, sabemos que os mínimos das outras linhas se encontram nas submatrizes $A[1\,..\,i-1][1\,..\,R[i]]$ e $A[i+1\,..\,n][R[i]\,..\,m]$. Seguindo o paradigma de divisão e conquista, vamos resolver o mesmo problema para estas submatrizes e, consequentemente, resolver o problema original. O Algoritmo 3.1 implementa esta ideia.

Algoritmo 3.1 Mínimos das linhas com divisão e conquista

```
1: função DivConq(A, r_s, r_t, c_s, c_t, R)
2:     se r_s ≤ r_t então
3:         ℓ ← ⌈(r_s + r_t)/2⌉
4:         Guarda em R[ℓ] o índice de mínimo da linha ℓ de A.
5:         DivConq(A, r_s, ℓ − 1, c_s, R[ℓ], R)
6:         DivConq(A, ℓ + 1, r_t, R[ℓ], c_t, R)
```

Note que, na linha 4, o mínimo só precisa ser buscado entre os índices c_s e c_t, inclusive, pois estamos resolvendo o problema para a submatriz $A[r_s\,..\,r_t][c_s\,..\,c_t]$. A implementação desta função em `C++` pode ser encontrada no arquivo `DivConq.cpp`.

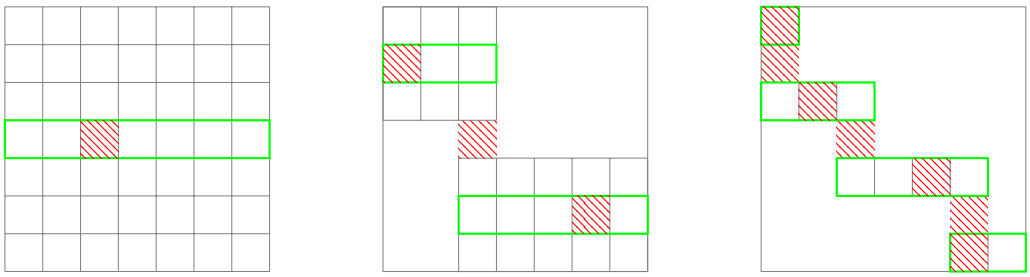

Figura 3.2: Progressão do algoritmo da divisão e conquista em uma matriz monótona. A cada chamada da função, as células circuladas em verde são percorridas linearmente. Os mínimos encontrados estão hachuradas em vermelho.

Sejam $r_s \leq r_t$ índices de linhas e $c_s \leq c_t$ índices de colunas de uma matriz A. Definimos os valores $n = r_t - r_s + 1$ e $m = c_t - c_s + 1$. Denotamos por $T(n, m)$ o tempo gasto por DivConq$(A, r_s, r_t, c_s, c_t, R)$ no pior caso. Para todos os valores n e m onde $n > 0$ e $m > 0$, vale a recorrência $T(n, m) = m + \max_{j \in [m]}\{T(\lfloor \frac{n}{2} \rfloor, j) + T(\lceil \frac{n}{2} \rceil - 1, m - j + 1)\}$. E no caso onde $n = 0$, para todo $m > 0$, temos que $T(1, m) = 1$.

Vamos provar que, para todo $n > 0$ e $m > 0$, vale que $T(n, m) \leq (m + n)\lg(n + 1)$. Utilizaremos indução em n. No caso base, quando $n = 0$, vale que $T(n, m) = 1 \leq (m + n)\lg(n + 1)$. Quando $n > 0$, assumimos que a tese é verdadeira com $n' \in [0\,..\,n-1]$. Existe algum j tal que $T(n, m) = m + T(\lfloor \frac{n}{2} \rfloor, j) + T(\lceil \frac{n}{2} \rceil + 1, m - j + 1)$. Por hipótese de indução, já que $\lfloor \frac{n}{2} \rfloor$ e $\lceil \frac{n}{2} \rceil + 1$ pertencem a $[0\,..\,n-1]$, vale que $T(n, m) \leq m + (j + \lfloor \frac{n}{2} \rfloor)\lg(\lfloor \frac{n}{2} \rfloor + 1) + (m - j + \lceil \frac{n}{2} \rceil)\lg(\lceil \frac{n}{2} \rceil)$. Desenvolvendo o lado direito desta desigualdade, obtemos que $T(n, m) \leq n\lg(n + 1) + m(\lg(\lceil \frac{n}{2} \rceil) + 1)$. Já que $1 = \lg(2)$ e $2(\lceil \frac{n}{2} \rceil) \leq n + 1$, podemos concluir desta última desigualdade que vale a tese, isto é $T(n, m) \leq (m+n)\lg(n+1)$. Disso concluímos que o Algoritmo 3.1 consome tempo $\mathcal{O}((n+m)\lg(n))$.

3.2 Aplicação em programação dinâmica

Utilizaremos a técnica apresentada aqui para resolver uma adaptação do problema "Internet Trouble" da Final Brasileira da Maratona de Programação de 2016. Informações sobre a prova em questão podem ser encontradas no link http://maratona.ime.usp.br/resultados16.

Problema 3.3. *Considere a função de custo c definida como $c(v) = \sum_{i=1}^{|v|} \min\{i-1, m-i\}v_i$ para cada vetor $v \in \mathbb{Q}^m$ com $m \in [n]$. Sejam $v \in \mathbb{Z}_+^n$ um vetor e $k \in [n]$ um inteiro. Escolher $k+1$ posições $p_0 = 1 \leq p_1 \leq \cdots \leq p_k = n$ do vetor v de forma a minimizar $\sum_{i=1}^{k} c(v[p_{i-1}\,..\,p_i])$.*

Podemos dar ao problema acima a interpretação do problema "Internet Trouble" citado. Temos n cidades numeradas de 1 até n dispostas em uma linha de forma que, para todo $i, j \in [n]$, a distância entre as cidades i e j é $|i-j|$. Queremos escolher $k+1$ destas cidades para instalar torres de distribuição de energia de forma a minimizar o custo de alimentar todas as cidades com energia sabendo que o custo de transferir energia de uma torre qualquer para uma cidade com h habitantes a uma distância d da torre é hd e que uma torre pode alimentar quantas cidades quiser. Nesta adaptação, as cidades 1 e n são escolhas obrigatórias para posições de torres.

Queremos resolver o problema com programação dinâmica. Definimos, para todo $\ell \in [k]$ e $j \in [n]$, o subproblema de escolher $\ell+1$ posições $p_0 = 1 \leq p_1 \leq \cdots \leq p_\ell = j$ do vetor $v[1\,..\,j]$ de forma a minimizar $\sum_{i=1}^{\ell} c(v[p_{i-1}\,..\,p_i])$. Definimos a matriz E que guarda em cada entrada $E[\ell][j]$ a resposta ótima do subproblema de parâmetros ℓ e j. É fácil ver que $E[k][n]$ guarda a resposta ótima do problema original. O valor de qualquer entrada $E[1][j]$ do vetor é $c(v[1\,..\,j])$ já que o único p possível a ser escolhido é tal que $p_0 = 1$ e $p_1 = j$. O valor de qualquer entrada $E[\ell][j]$ do vetor tal que $\ell \in [2\,..\,k]$ é igual a $\min\{E[\ell-1][i] + c(v[i\,..\,j]) \mid i \in [j]\}$. Esta definição de E indica uma solução com programação dinâmica que pode ser realizada em tempo $\mathcal{O}(kn^2)$ desde que a função c possa ser calculada para todo subvetor de v em tempo $\mathcal{O}(1)$.

Definimos as funções $s(i,j) = \sum_{x=i}^{j}(x-i)v_x$ e $t(i,j) = \sum_{x=i}^{j}(j-x)v_x$ para todo $i, j \in [n]$ onde $i \leq j$. Assim, é fácil ver que se $r = \lfloor \frac{i+j}{2} \rfloor$ ou $r = \lceil \frac{i+j}{2} \rceil - 1$ vale que $c(v[i\,..\,j]) = s(i,r) + t(r+1,j)$. Além disso, se pré-calcularmos, em tempo $\mathcal{O}(n)$ os vetores a, $b \in \mathbb{Z}^{[0\,..\,n]}$ tais que, para todo $j \in [n]$, vale que $a_j = \sum_{i \in [j]} v_i$ e $b_j = \sum_{i \in [j]} iv_i$, podemos escrever $s(i,j) = b_j - b_{i-1} - (a_j - a_{i-1})i$ e $t(i,j) = (a_j - a_{i-1})j - b_j + b_{i-1}$ para todo $i, j \in [n]$, o que nos dá uma forma de calcular estas funções em $\mathcal{O}(1)$ e, portanto, calcular c em $\mathcal{O}(1)$ para todo subvetor de v.

Vamos definir uma matriz $A \in \mathbb{Z}^{n \times n}$ tal que, para todo $\ell \in [2\,..\,k]$ e $j \in [n]$, valha que $E[\ell][j] = \min\{E[\ell-1][i] + A[i][j] \mid i \in [n]\}$. Basta escolher $A[i][j] = c(v[i\,..\,j])$ para todo $i \leq j$ e $A[i][j] = +\infty$ para todo $i > j$. Agora, podemos definir uma matriz B_ℓ para cada um destes ℓ de forma que $B_\ell[i][j] = E[\ell-1][i] + A[i][j]$ para todo $i, j \in [n]$. Desta forma, calcular um valor $E[\ell][j]$ com $\ell \in [2\,..\,k]$ é encontrar o mínimo da j-ésima coluna da matriz B_ℓ. Vamos mostrar que cada uma destas matrizes é monótona nos mínimos das colunas e, com isso, resolver este problema em tempo $\mathcal{O}(kn \lg(n))$ com a otimização da divisão e conquista.

Vamos provar que as matrizes B_ℓ são monótonas crescentes nos mínimos das colunas. Sejam $\ell \in [2\,..\,k]$, $j \in [n-1]$ um índice de coluna de B_ℓ e sejam $i', i \in [n]$ duas linhas de B_ℓ tais

que $i' < i \leq j$. Suponha que $B_\ell[i][j] < B_\ell[i'][j]$. Temos $E[\ell-1][i] + A[i][j] < E[\ell-1][i'] + A[i'][j]$. Tomamos as médias $r = \lfloor \frac{i+j}{2} \rfloor = \lceil \frac{i+j+1}{2} \rceil - 1$ e $r' = \lfloor \frac{i'+j+1}{2} \rfloor = \lceil \frac{i'+j+1}{2} \rceil - 1$. Vale a desigualdade $E[\ell-1][i] + s(i,r) + t(r+1,j) < E[\ell-1][i'] + s(i',r') + t(r'+1,j)$. Porém, já que $r' \leq r$, vale

$$\begin{aligned} t(r+1,j+1) - t(r+1,j) &= (j-r)v_{j+1} \\ &\leq (j-r')v_{j+1} \\ &= t(r'+1,j+1) - t(r'+1,j). \end{aligned}$$

Com este resultado concluímos que $t(r'+1,j) - t(r+1,j) \leq t(r'+1,j+1) - t(r+1,j+1)$ e, desta, temos que $E[\ell-1][i] + s(i,r) + t(r+1,j+1) < E[\ell-1][i'] + s(i',r') + t(r'+1,j+1)$, portanto, vale que $B_\ell[i][j+1] < B_\ell[i][j+1]$. Isso nos diz que se algum $i \leq j$ é o índice de mínimo da coluna j, então o índice de mínimo da coluna $j+1$ não pode ser menor do que i. Sabemos que se $i > j$ a entrada $B_\ell[i][j]$ não pode ser o mínimo da linha i pois tem valor $+\infty$. Com isso, concluímos que os índices de mínimos das colunas são crescentes e as matrizes B_ℓ são crescentes nos mínimos das colunas.

3.3 Interpretação em programação dinâmica

Muitas das técnicas apresentadas neste trabalho possuem formatos específicos de programas dinâmicos relacionados à técnica. Um bom exemplo é a otimização de Knuth-Yao do Capítulo 5, que é desenvolvida a partir da sua interpretação em programação dinâmica.

Em contextos voltados para competições de programação, é comum pensar e apresentar estas técnicas com uma visão fortemente voltada para programação dinâmica. Este olhar é bom e especialmente interessante para competições de programação por facilitar a identificação rápida da pertinência de problemas a esta técnica a partir da recorrência. Por este motivo, é interessante discutir esta interpretação também aqui.

Problema 3.4. *Dados inteiros n e k com $1 \leq k \leq n$ e uma função $f(i,j)$ definida para todo $i,j \in [n]$ onde $i \leq j$, calcular $E[k][n]$ onde, para todo $j \in [0\,..\,n]$ e $\ell \in k$ vale que*

$$E[\ell][j] = \begin{cases} A[1][j] & , \text{ se } \ell = 1 \text{ e} \\ \min\{E[\ell-1][i] + A[i][j] \mid i \in [j]\} & , \text{ caso contrário.} \end{cases}$$

O Problema 3.4 segue um formato clássico de programação dinâmica e indica uma solução clara que custa tempo $\mathcal{O}(kn^2)$. O Problema 3.3 cabe perfeitamente no formato descrito por este problema e foi otimizado utilizando a técnica da divisão e conquista. Para provar que a otimização era possível, precisamos construir uma série de matrizes e provar que estas eram monótonas e que o problema original se reduzia a encontrar os mínimos destas matrizes.

Definimos a matriz de cortes ótimos P da recorrência E. Para toda entrada ℓ, j de E onde $\ell > 1$ vale que $P[\ell][j]$ é o menor valor em $[j]$ que respeita $E[\ell][j] = A[1][P[\ell][j]] + A[P[\ell][j]][j]$. É claro da definição de E que esta matriz existe. É fácil calcular, junto com E, as entradas da matriz P. Esta matriz representa as escolhas que devem ser feitas em cada busca de mínimo da definição E para encontrar os valores verdadeiros das entradas da matriz.

Se as linhas da matriz P forem todas monótonas, isto é, se vale $P[\ell][i] \leq P[\ell][i']$ sempre que $i, i' \in [n]$ e $i < i'$, é possível utilizar divisão e conquista para resolver o problema. É fácil ver isso construindo matrizes B_ℓ iguais às construídas na solução do Problema 3.3 e verificando que $P[\ell][i]$ é o índice de mínimo da linha i da matriz B_ℓ.

Geralmente é uma boa prática, quando se está pensando em programação dinâmica, considerar a matriz P e pensar sobre suas propriedades. A observação de que as linhas desta matriz, quando monótonas, permitem a aplicação do método da divisão e conquista é uma boa maneira de perceber a pertinência desta otimização em problemas e configura uma habilidade especialmente útil em competições de programação.

Capítulo 4

SMAWK

Neste capítulo discutiremos o algoritmo SMAWK. Este algoritmo tem a mesma função da otimização divisão e conquista, porém, precisa que a matriz seja totalmente monótona, não apenas monótona, e tem uma complexidade assintótica melhor do que aquele. Ele é conhecido pela sua aplicação no problema de encontrar um vértice mais distante de cada vértice num polígono convexo em tempo linear [3], mas pode ser usado em vários outros problemas, assim como a otimização da divisão e conquista.

Dada uma matriz $A \in \mathbb{Q}^{n \times m}$, listamos os casos em que este algoritmo se aplica:

- Se A é totalmente monótona convexa ou côncava nas linhas, podemos encontrar os índices de mínimos e máximos das linhas em tempo $\mathcal{O}(n + m)$;
- 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. É fácil manipular o algoritmo para trabalhar com os outros casos.

4.1 Técnica primordial

Para facilitar a compreensão do algoritmo SMAWK, iremos apresentar uma técnica parecida com a otimização da divisão e conquista, apresentada no Capítulo 3, 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 índice de mínimo de cada uma das linhas de A. Se, para uma dada linha $i \in [2\,..\,n-1]$, conhecermos os índices ℓ e r de mínimos das linhas $i-1$ e $i+1$, sabemos, pela monotonicidade de A que o índice de mínimo da linha i está em $[\ell\,..\,r]$. Além disso, se $i = 1$ e $\ell = 1$ ou se $i = n$ e $r = m$, o resultado descrito continua valendo.

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. Graças a este fato, podemos remover todas as linhas pares da matriz, encontrar os mínimos recursivamente para todas as linhas restantes e depois utilizar as respostas conhecidas para calcular os índices de mínimos das linhas pares. Vamos mostrar

que esta última parte, encontrar os mínimos das linhas pares dados os mínimos das outras, custa tempo $\mathcal{O}(n+m)$.

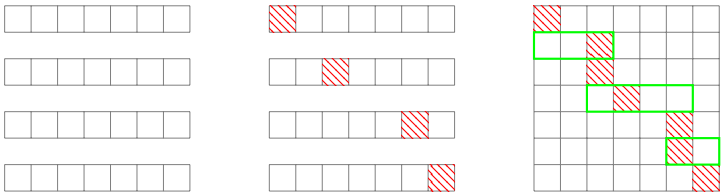

Figura 4.1: Progresso da técnica primordial. Primeiro, as linhas pares são removidas da matriz original. Depois os mínimos das linhas restantes, hachurados em vermelho, são definidos recursivamente. Finalmente, as linhas pares são restauradas e as posições circuladas em verde são percorridas em busca dos seus mínimos, em vermelho.

Definimos, para cada i ímpar, o valor t_i que representa o índice de mínimo da i-ésima linha, além dos valores $t_0 = 1$ e, caso n seja par, o valor $t_{n+1} = m$. Para cada linha i par, o parágrafo anterior sugere buscar o mínimo desta linha entre as posições t_{i-1} e t_{i+1}. Podemos escrever o tempo gasto para encontrar o mínimo de cada uma das linhas pares como $\sum_{i \in [n] \text{ par}} (t_{i+1} - t_{i-1})$ e concluir, desta expressão, que o tempo gasto ao todo é $\mathcal{O}(n+m)$.

Considerando que remover todas as linhas pares de uma matriz pode ser realizado em tempo $\mathcal{O}(1)$, é fácil ver que o tempo de execução total deste algoritmo é $\mathcal{O}((n+m)\lg(n)$, já que só é possível remover todas as linhas pares $\mathcal{O}(\lg(n))$ vezes.

4.2 Reduce

Chamamos de ótimas as células de uma matriz que são o mínimo de alguma linha e as colunas que contém pelo menos uma célula ótima. Lembre que a definição de mínimo permite apenas um mínimo por linha, portanto, uma matriz contém no máximo n colunas ótimas.

Queremos agilizar a técnica apresentada acima. Para isso, vamos adicionar a nova hipótese de que a matriz A é quadrada, ou seja, $n = m$. A cada passo, removemos as $\lfloor n/2 \rfloor$ linhas pares da matriz, gerando uma nova matriz A', resolvemos o problema recursivamente em A' e usamos a solução de A' para encontrar os mínimos das linhas restantes de A. 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, $m \geq n$. Queremos remover colunas não ótimas dessa matriz com mais colunas do que linhas fazendo com que a matriz resultante se torne quadrada.

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

1. $k \in [1\,..\,n]$,
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 seu índice de coluna não é ótima.

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.

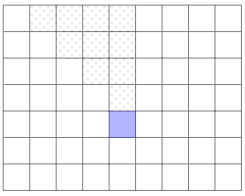

Figura 4.2: Invariante 3 do REDUCE. A célula (k, k) está preenchida em azul. As células que, segundo a Invariante 3, não são ótimas estão pontilhadas em preto.

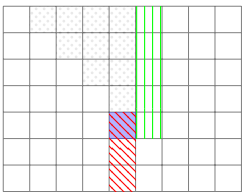

Figura 4.3: Casos do REDUCE. As células pontilhadas em preto não são ótimas por consequência da Invariante 3. Caso valha que $A[k][k] > A[k][k+1]$, as hachuradas em vermelho não são ótimas. No caso contrário, as com linhas verticais verdes não são ótimas.

Primeiro, se $A[k][k] > A[k][k+1]$, vamos mostrar que as entradas com índice de linha maior ou igual a k na coluna k não são ótimas. A célula (k, k) não é ótima como consequência direta da desigualdade. Tome uma linha $i > k$ qualquer, pela total monotonicidade de A vale que $A[i][k] > A[i][k+1]$, logo, a célula (i, k) não é ótima.

No outro caso, em que $A[k][k] \leq A[k][k+1]$, vamos mostrar que as células da coluna $k+1$ com índices de linha menores ou iguais a k não são ótimas. A célula $(k, k+1)$ não é ótima. Se $i < k$ é uma linha qualquer de A, pela total monotonicidade de A, vale $A[i][k] \leq A[i][k+1]$, logo, a célula $(i, k+1)$ não é ótima.

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

Algoritmo 4.4 Algoritmo REDUCE

```
1:  função REDUCE(A)
2:      k ← 1
3:      enquanto A tem mais colunas do que linhas
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
```

Mostraremos que as invariantes são válidas neste algoritmo. Olhamos para o primeiro passo, onde $k = 1$. 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 início do enquanto da linha 3 a célula $A[k][k+1]$ existe, uma vez que $k \leq n$ e $n < m$. Consideramos o caso onde $A[k][k] > A[k][k+1]$. A Invariante (1) sempre se mantém trivialmente. Já provamos que a coluna k não é ótima neste caso, o que faz com que a Invariante (2) se mantenha mesmo após a remoção da coluna k. Agora, se $k = 1$, vale (3) por vacuidade e, caso contrário, já que k decresce, a Invariante (3) também se mantém.

No caso onde valem $A[k][k] \leq A[k][k+1]$ e $k = n$ foi provado que os elementos de linhas menores ou iguais a k na coluna coluna $k+1$ não são ótimos. Já que $k = n$, isto representa toda a coluna $k+1$, assim, removê-la mantém a Invariante (2). As outras duas invariantes se mantém trivialmente. Finalmente, resta apenas considerar o caso onde $A[k][k] \leq A[k][k+1]$ e $k < n$. Foi provado, novamente, que as células com índices menores do que $k+1$ na coluna $k+1$ não são ótimos, assim, incrementar o k mantém a Invariante (3). As outras duas invariantes também se mantém trivialmente neste caso.

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 $n \leq m$.

4.3 SMAWK

Recebemos uma matriz A totalmente monótona convexa por linhas. Primeiramente, vamos fazer com que a matriz se torne quadrada mantendo os índices de mínimos das linhas. Se A tem mais colunas do que linhas, basta aplicar o REDUCE. Se A tem mais linhas do que colunas, podemos adicionar uma cópia da última coluna ao fim da matriz, mantendo a total monotonicidade da mesma, até que A se torne quadrada.

Agora estamos prontos para descrever e aplicar o algoritmo SMAWK. Vamos usar a ideia da técnica primordial da Seção 4.1, porém, vamos aplicar o algoritmo REDUCE a cada passo para melhorar a complexidade da solução. Em cada chamada de SMAWK, recebemos uma matriz A quadrada, resolvemos o problema recursivamente para a submatriz A' que consiste das linhas ímpares de A tomando o cuidado de aplicar o REDUCE nesta antes de realizar a chamada recursiva, e, depois, utilizamos estes resultados para calcular os mínimos das linhas restantes. O Algoritmo 4.5 descreve este processo.

4.4 Análise

Seja $T(n)$ o tempo gasto pelo algoritmo ao receber uma matriz quadrada de lado n onde $n \geq 1$. Sabemos que $T(1) = \mathcal{O}(1)$. Assumimos que, dada uma matriz A, sabemos gerar uma matriz A' que consiste apenas das linhas ímpares de A em tempo $\mathcal{O}(1)$. Se $n > 1$, geramos esta matriz, que tem $\lfloor \frac{n}{2} \rfloor$ linhas, e aplicamos o REDUCE nela, gastando tempo $\mathcal{O}(n)$. Após este procedimento, determinamos os mínimos das linhas pares de A na forma descrita na Seção 4.1, o que custa tempo $\mathcal{O}(n)$. Assim,

Algoritmo 4.5 Algoritmo SMAWK

```
1: função SMAWK(A)
2:     se A tem exatamente uma linha então
3:         A é uma matriz 1 × 1 e a resposta é trivial
4:     senão
5:         A′ é a matriz A sem as linhas pares
6:         SMAWK(REDUCE(A′))
7:         para i linha ímpar de A faça
8:             ℓ ← 1 e r ← m
9:             se i > 1 então
10:                ℓ ← índice de mínimo da linha i − 1
11:            se i < n então
12:                r ← índice de mínimo da linha i + 1
13:            Busca o índice de mínimo da linha i entre ℓ e r, inclusive
```

para todo $n > 1$, vale que $T(n) = \mathcal{O}(n) + T(\lfloor \frac{n}{2} \rfloor)$, o que nos leva a $T(n) = \mathcal{O}(n)$.

Se a matriz recebida tiver menos colunas do que linhas, a transformação inicial custa tempo $\mathcal{O}(1)$. No outro caso, quando a matriz tem mais colunas do que linhas, é necessário realizar uma chamada ao REDUCE, o que custa tempo $\mathcal{O}(n + m)$, onde n é a quantidade de linhas m é a quantidade de colunas da matriz original. Podemos escrever a complexidade no caso geral como $\mathcal{O}(n + m)$.

4.5 Implementação

Queremos encontrar uma maneira eficiente de remover as linhas pares da matriz, mas não podemos gerar explicitamente uma nova matriz. Queremos representar, a cada chamada, todas as linhas que podem ser visitadas. Inicialmente, todas as linhas visitáveis da matriz são aquelas da forma $1 + k$ onde k é um inteiro não negativo. Após remover as linhas pares, as visitáveis são da forma $1 + 2k$, depois $1 + 4k$ e assim por diante. Após t remoções, todas elas são representadas por $1 + 2^t k$ para algum k inteiro não negativo. Assim, basta manter o parâmetro t para descobrir as linhas visitáveis e incrementá-lo quando for necessário remover as linhas pares da matriz.

Precisamos representar as colunas visitáveis em A. Já que não há uma regra simples como a das linhas para a remoção de colunas, precisamos de alguma estrutura de dados que nos permita iterar pelos seus valores em ordem, removendo itens visitados quando necessário, eficientemente. Guardaremos, em uma lista duplamente ligada, todos os índices de colunas válidos. Já que, durante uma chamada de SMAWK, iteramos pelas colunas e todas as remoções realizadas são na coluna atual ou alguma coluna adjacente à atual, cada remoção é realizada em $\mathcal{O}(1)$. Após resolver o problema recursivamente, precisamos recuperar a lista ligada do início da chamada para podermos descobrir os valores de mínimo nas linhas ímpares daquela matriz. Para isso, basta, ao começo de cada chamada, criar uma cópia da lista ligada original, o que é feito em $\mathcal{O}(n)$ e, portanto, não afeta a análise do tempo do algoritmo. Note que a criação desta estrutura de dados e as cópias dela em cada chamada recursiva custam espaço e tempo $\mathcal{O}(m)$ ao todo, onde m é a quantidade de colunas da matriz original.

Como observado anteriormente, antes de chamar o SMAWK pela primeira vez, é necessário garantir que a matriz A seja quadrada. Se a quantidade de linhas é maior do que a quantidade

de colunas, precisamos adicionar várias cópias da última coluna ao final da matriz em tempo $\mathcal{O}(1)$. Lembre, do Capítulo 1, que as matrizes são sempre representadas por (f, n, m) onde f é uma função, n é a quantidade de linhas de uma matriz e m é a quantidade de colunas da matriz. Se recebermos uma entrada tal que $n > m$, basta gerar uma nova função h tal que, para todo $i \in [n]$ e $j \in [m]$, vale que $h(i, j) = f(i, j)$ e, para todo i, j diferente destes, vale que $h(i, j) = f(i, m)$. Podemos passar a matriz (h, n, n) adiante em vez da original. Após realizar este tratamento, basta inicializar a lista ligada referente às colunas da matriz em tempo $\mathcal{O}(m)$ e aplicar o REDUCE nesta, o que trata o caso onde $n < m$ e não faz nada em qualquer outro caso.

A implementação em C++ do algoritmo apresentado, levando em conta as considerações acima, leva o nome `SMAWK.cpp`.

4.6 Aplicações

Aggarwal, Klawe, Moran, Shor e Wilber [3] mostraram a aplicação do SMAWK na solução de vários problemas de geometria computacional. Um destes problemas, como mencionado no início do capítulo, é o Problema 4.6.

Problema 4.6 (Todos os pares mais distantes em um polígono convexo)**.** *Dado um polígono convexo p com n vértices indexados em sentido horário, encontrar, para cada vértice p_i, um outro vértice p_j que maximize $d(p_i, p_j)$, onde $d(p_i, p_j)$ é o quadrado da distância euclideana entre os pontos p_i e p_j.*

Consideramos convexo um polígono simples tal que nenhum vértice pode ser escrito como combinação convexa dos demais vértices do mesmo polígono. O quadrado distância euclideana entre dois pontos $a = (x_a, y_a)$ e $b = (x_b, y_b)$ é dado por $d(a, b) = (x_a - x_b)^2 + (y_a - y_b)^2$. Note que maximizar o quadrado da distância euclideana é equivalente a maximizar a distância euclideana. Por conveniência, se $j \in [n + 1 \,..\, 2n]$, definimos $p_j = p_{j-n}$.

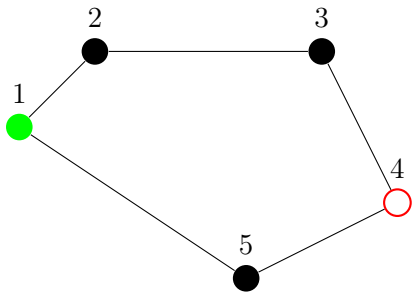

Figura 4.7: Exemplo de polígono convexo com os vértices indexados em sentido horário. O vértice vermelho sem preenchimento é o mais distante do vértice verde.

Vamos construir uma matriz A Monge com n linhas e $2n - 1$ colunas tal que, para cada $i \in [n]$, se j é uma coluna que atinge o máximo na linha i, então o vértice p_j maximiza $d(p_i, p_j)$. Para todo índice $i \in [n]$ de linha de A e $j \in [2n - 1]$ de coluna de A vale que

$$A[i][j] = \begin{cases} d(p_i, p_j) & \text{, se } i < j < i + n, \\ -C|j - i| & \text{, se } j \leq i \text{ e} \\ -C|j - n - i| & \text{, se } i + n \leq j \end{cases}$$

onde C é uma constante suficientemente grande. Tomamos $C = \max_{i \in [n]}\{d(p_i, p_{i+1})\}$, isto é, o comprimento da maior aresta do polígono, o que pode ser calculado em tempo $\mathcal{O}(n)$ trivialmente.

Primeiro, já que em cada linha i existe uma entrada $d(p_i, p_j)$ para cada $j \in [n]$ onde $i \neq j$ e estas são as únicas entradas positivas da linha, é verdade que, se uma coluna j alcança um máximo nesta linha, ele maximiza a distância $d(p_i, p_j)$. Se provarmos que a matriz A descrita é totalmente monótona por linhas, mostramos que o Problema 4.6 pode ser resolvido em tempo $\mathcal{O}(n)$ com o algoritmo SMAWK.

0	2	17	26	13	0	-13	-26	-39
-13	0	9	20	13	2	0	-13	-26
-26	-13	0	5	10	17	9	0	-13
-39	-26	-13	0	5	26	20	5	0
-52	-39	-26	-13	0	13	13	10	5

Figura 4.8: Matriz A referente ao polígono da Figura 4.7 com o quadrado da distância euclideana. Os valores em vermelho são os da forma $-C|x|$ para algum inteiro x.

Proposição 4.9. *A matriz A descrita é totalmente monótona côncava por linhas.*

Demonstração. Assuma que $n > 2$. Os dois casos restantes são triviais. Vamos mostrar que para qualquer escolha de índices $i \in [n-1]$ e $j \in [2n-2]$ vale a desigualdade $A[i][j] + A[i+1][j+1] \geq A[i][j+1] + A[i+1][j]$. Se $i > j$, então a desigualdade equivale a $-C(i-j) - C(i+1-j-1) \geq -C(i-j-1) - C(i+1-j)$. O caso onde $j > i+n$ se desenvolve de maneira similar.

Considere agora o caso onde $i = j$ ou $i = j + n$. Vale que $A[i][j] = A[i+1][j+1] = 0$, portanto, o lado esquerdo da desigualdade soma 0. Devemos provar que $0 \geq A[i+1][j] + A[i][j-1]$. Nas duas situações, o lado direito desta desigualdade é o comprimento da uma aresta do polígono polígono e $-C$. Pela escolha de C, a desigualdade vale. Nos casos onde $i + 1 = j$ ou $i + n - 1 = j$, é possível escolher vértices a, b e c do polígono de forma que a desigualdade proposta seja equivalente a $d(a, b) + d(b, c) \geq d(a, c)$, que é a desigualdade triangular. Se $i+1 = j$, escolhemos $a = p_i$, $b = p_{i+1}$ e $c = p_{i+2}$. Se $i + n - 1 = j$, escolhemos $a = p_{i+1}$, $b = p_{i+n-1}$ e $c = p_i$.

Resta provar o caso onde $i+1 < j < i+n-1$. Neste caso os vértices p_i, p_{i+1}, p_j e p_{j+1} são todos distintos e estão listados, nesta sequência, em sentido horário. Isso faz com que os segmentos p_ip_j e $p_{i+1}p_{j+1}$ se intersectem. Tome o ponto t de intersecção. É fácil perceber que o lado esquerdo da desigualdade equivale a $d(p_i, t) + d(t, p_j) + d(p_{i+1}, t) + d(t, p_{j+1})$. Pela desigualdade triangular, isto é maior ou igual a $d(p_i, p_{j+1}) + d(p_{i+1}, p_j)$, o lado direito da desigualdade.

Pelo Teorema 2.12, provamos que A é Monge côncava. Pelo Teorema 2.11, provamos que A é totalmente monótona nas linhas. □

É interessante observar que, na construção realizada acima, a matriz A é Monge côncava e possui valores potencialmente grandes em módulo, já que precisamos realizar a escolha do valor C de forma que ele supere todos os outros valores da matriz. Aggarwal, Klawe, Moran, Shor e Wilber [3] mostram uma forma de construir uma matriz parecida com a A que é apenas totalmente monótona (não Monge) e também pode ser utilizada para resolver o problema proposto na mesma complexidade. A ausência de valores desnecessariamente grandes como os que ocorrem na matriz modelada aqui pode ajudar a evitar complicações numéricas na aplicação do algoritmo.

Além de aplicações em geometria, o algoritmo SMAWK, bem como a otimização da divisão e conquista, é útil na agilização de problemas de programação dinâmica [10, 11]. No Capítulo 2 foi apresentado o Problema 2.13, formulado com programação dinâmica e resolvido com o algoritmo SMAWK, por exemplo.

Capítulo 5

Otimização de Knuth-Yao

O problema da árvore de busca binária ótima [7] é um exemplo clássico de aplicação de programação dinâmica que é facilmente resolvido em tempo $\mathcal{O}(n^3)$, onde n é o número de chaves da árvore. Esta solução associa o custo de cada subárvore a uma entrada de uma matriz $n \times n$ e relaciona tais entradas por meio de uma recorrência, reduzindo o problema original a resolver a recorrência. Aproveitando algumas propriedades não óbvias desta recorrência, Knuth [14] apresentou uma forma de resolvê-la em tempo $\mathcal{O}(n^2)$, o que agiliza a solução do problema original.

Mais tarde, a solução de Knuth foi estudada por Yao [17, 18], que mostrou que as propriedades observadas por Knuth eram consequência do fato de que a matriz em questão era Monge convexa. Desta maneira, foi possível perceber que a otimização de Knuth poderia ser útil em vários outros problemas de programação dinâmica.

Bein, Golin, Larmore e Zhang [4] buscaram enfraquecer a condição encontrada por Yao e mostraram que as matrizes derivadas dos problemas agilizados com a otimização de Knuth-Yao podem ser decompostas de três maneiras diferentes em matrizes totalmente monótonas. Com estas decomposições, é possível resolver ainda mais problemas. As descobertas do artigo citado neste parágrafo não serão discutidas aqui. Esta introdução foi baseada em tal artigo.

Vamos discutir os resultados observados por Yao, descrever a técnica desenvolvida por Knuth e aplicar este conhecimento para resolver um problema de programação dinâmica de forma mais eficiente do que a trivial.

5.1 Definições básicas

Vamos apresentar a otimização de Knuth-Yao para problemas de minimização, o que nos leva a trabalhar com convexidade. É fácil adaptar o conhecimento discutido neste capítulo para problemas de maximização, porém, em vez da convexidade, a concavidade deve ser usada para provar os resultados e modelar os problemas de interesse.

Definição 5.1 (Recorrência de intervalos)**.** *Uma matriz $A \in \mathbb{Q}^{n\times n}$ triangular inferior em 0 é considerada uma recorrência de intervalos se existe uma matriz $C \in \mathbb{Q}^{n\times n}$ tal que, para todo $i, j \in [n]$,*

$$A[i][j] = \begin{cases} C[i][j] & \text{se } i = j, \\ C[i][j] + \min\{A[i][k-1] + A[k][j] \mid i < k \leq j\} & \text{se } i < j. \end{cases}$$

A matriz C é chamada de matriz de custos de A.

É fácil resolver uma recorrência desta forma em tempo $\mathcal{O}(n^3)$. Existe uma interpretação interessante deste tipo de recorrência que corresponde a vários problemas cuja a solução é baseada em programação dinâmica. Inicialmente, existe um vetor $v \in \mathbb{Q}^n$. É possível dividir este vetor em duas partes, isto é, escolher uma posição $k \in [n-1]$ e substituir o vetor original pelos seus dois subvetores $v[1\,..\,k]$ e $v[k+1\,..\,n]$. Cada subvetor gerado pode ser dividido novamente. O custo de dividir um subvetor $v[i\,..\,j]$ qualquer de v é dado pela matriz $C[i][j]$. O problema consiste em dividir um vetor original sucessivamente até transformá-lo em n vetores de tamanho unitário minimizando o custo de realizar todas as divisões.

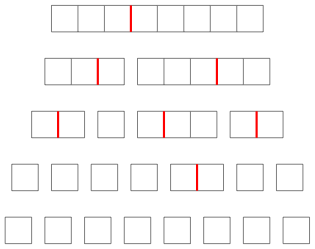

Figura 5.2: Interpretação da recorrência de intervalos como o problema de dividir um vetor. O vetor inicial aparece no primeiro nível e suas sucessivas divisões são mostradas nos níveis seguintes. Os cortes escolhidos em cada nível aparecem marcados em vermelho.

Esta interpretação da recorrência de intervalos nos leva naturalmente a considerar as posições onde o vetor deve ser quebrado a fim de obter uma resposta ótima. Será útil estudar as propriedades destes pontos de corte para agilizar a solução da recorrência. A Definição 5.3 formaliza a noção de posições ótimas de corte.

Definição 5.3 (Matriz de cortes ótimos)**.** *Se $A \in \mathbb{Q}^{n\times n}$ é uma recorrência de intervalos com matriz de custos C, definimos a matriz de cortes ótimos P de A. Para todo $i \in [n]$, $P[i][i] = i$ e, para todo $j \in [n]$ com $i < j$,*

$$P[i][j] = \min\{k \mid i < k \leq j \text{ e } A[i][j] = C[i][j] + A[i][k-1] + A[k][j]\}.$$

Assim, a matriz P guarda, para cada $i < j$, o menor argumento para o qual a função de mínimo na definição de $A[i][j]$ atinge seu valor ótimo. Note que, enquanto descobrimos os valores da matriz A em tempo $\mathcal{O}(n^3)$, descobrimos também os valores de P.

Definição 5.4 (Knuth-Yao otimizável)**.** *Se $A \in \mathbb{Q}^{n\times n}$ é uma recorrência de intervalos e P é sua matriz de cortes ótimos. Dizemos que A é Knuth-Yao otimizável se, para todo $i, j \in [n]$ com $i < j$, vale que $P[i][j-1] \leq P[i][j] \leq P[i+1][j]$.*

Vamos mostrar que se $A \in \mathbb{Q}^{n\times n}$ é Knuth-Yao otimizável, tanto A quanto sua matriz de cortes ótimos P podem ser calculadas em $\mathcal{O}(n^2)$.

5.2 Técnica

Seja $A \in \mathbb{Q}^{n\times n}$ uma matriz Knuth-Yao otimizável. Vamos calcular as entradas $A[i][j]$ onde $i \leq j$ em ordem crescente de $j - i$, ou seja, as entradas $A[i][i]$ serão calculadas para todo i, seguidas

Figura 5.5: Uma recorrência Knuth-Yao otimizável de acordo com a interpretação da Figura 5.2. A primeira linha mostra um subvetor $v[1\,..\,n-1]$ com seu ponto de corte ótimo $P[1][n-1]$ marcado em vermelho. A segunda, mostra o subvetor $v[2\,..\,n]$ também com seu ponto de corte ótimo $P[2][n]$ em vermelho. A terceira linha mostra o vetor v completo com as possíveis posições para o ponto de corte ótimo $P[1][n]$ circuladas em verde.

Algoritmo 5.6 Otimização de Knuth-Yao

```
1:  função KnuthYao(C, n)
2:      para i de 1 até n faça
3:          A[i][i] ← C[i][i]
4:          P[i][i] ← i
5:      para d de 1 até n − 1 faça
6:          para i de 1 até n − d faça
7:              j ← i + d
8:              A[i][j] ← +∞
9:              para k de max(i + 1, P[i][j − 1]) até P[i + 1][j] faça
10:                 v ← C[i][j] + A[i][k − 1] + A[k][j]
11:                 se v < A[i][j] então
12:                     A[i][j] ← v
13:                     P[i][j] ← k
14:     devolve (A, P)
```

das $A[i][i+1]$, $A[i][i+2]$ e assim por diante. É possível calcular as entradas nesta ordem pois ela respeita as relações de dependência da matriz A, isto é, ao calcular uma entrada $A[i][j]$ qualquer, todas as entradas $A[i][k-1]$ e $A[k][j]$ com $i < k \leq j$ já estarão disponíveis e, portanto, será possível descobrir o valor de $A[i][j]$ das entradas previamente calculadas.

Se calcularmos também as entradas da matriz P enquanto calculamos as da A, poderemos aproveitar o fato de que A é Knuth-Yao otimizável para buscar o valor de uma entrada de A em um intervalo menor do que o trivial. Formalmente, para todo $i, j \in [n]$ tal que $i < j$, vale a igualdade

$$A[i][j] = C[i][j] + \min\{A[i][k-1] + A[k][j] \mid i < k \leq j \text{ e } P[i][j-1] \leq k \leq P[i+1][j]\}. \tag{5.7}$$

Esta observação induz o Algoritmo 5.6 para calcular as entradas das matrizes A e P. Na linha 9 a variável k termina em $P[i+1][j]$ e não em $\min(j, P[i+1][j])$, como indicado pelo igualdade (5.7). Perceba que $P[i+1][j] \leq j$, portanto, $\min(j, P[i+1][j]) = P[i+1][j]$. A implementação em `C++` deste algoritmo como descrito acima pode ser encontrado na pasta de implementações com o nome `KnuthYao.cpp`.

5.3 Análise

Vamos analisar a complexidade do Algoritmo 5.6. Podemos escrever a quantidade de iterações do laço da linha 9 como

$$\sum_{d=1}^{n-1}\sum_{i=1}^{n-d}\sum_{k=P[i][i+d-1]}^{P[i+1][i+d]} 1 = \sum_{d=1}^{n-1}\sum_{i=1}^{n-d}(P[i+1][i+d] - P[i][i+d-1] + 1), \tag{5.8}$$

com um d fixo, a soma $\sum_{i=1}^{n-d}(P[i+1][i+d] - P[i][i+d-1])$ é uma soma telescópica e tem valor igual a $P[n-d+1][n] - P[1][1+d] = \mathcal{O}(n)$. Com isso, escrevemos (5.8) como $\sum_{d=1}^{n-1}\mathcal{O}(n) + n - 1 = \mathcal{O}(n^2)$.

5.4 Quebrando strings

Para exemplificar a otimização de Knuth e apresentar a relação das matrizes Monge com as definições da Seção 5.1, iremos resolver um outro problema clássico de programação dinâmica [7, Exercício 15-9] disponível no juíz online SPOJ em `http://www.spoj.com/problems/BRKSTRNG/`.

Considere uma linguagem de processamento de strings que consegue quebrar uma string s de tamanho $m > 1$ em qualquer posição $t \in [m-1]$, ou seja, gerar duas strings $s[1\,..\,t]$ e $s[t+1\,..\,m]$. Um programador quer usar esta linguagem para separar uma string n vezes, nas posições $p_1 < p_2 < \cdots < p_n$, porém, para quebrar uma string de tamanho m em qualquer posição, a linguagem gasta tempo m. Queremos descobrir qual é a melhor ordem para realizar estes cortes.

Suponha, por exemplo, que estamos interessados em quebrar uma string `stringdeexemplo` de tamanho 15 nas posições 6 e 8 para gerar as strings `string`, `de` e `exemplo`. Isso pode ser realizado de duas maneiras. Uma maneira é quebrar primeiro na posição 8, gerando as strings `stringde` e `exemplo` e depois na posição 6, gerando as três strings desejadas. A outra maneira é quebrar primeiro na posição 6, gerando `string` e `deexemplo`, e depois na posição 8. A primeira opção tem custo $15 + 8$, enquanto a segunda tem custo $15 + 9$, o que faz a resposta ótima ser a primeira alternativa.

Dados os valores n, m e os pontos $p_1 < p_2 < \cdots < p_n$ dos cortes desejados, chamamos de s a string que desejamos separar e definimos, por conveniência, $p_0 = 0$ e $p_{n+1} = m$. Assim, se $A \in \mathbb{Q}^{n\times n}$ guarda em toda posição $A[i][j]$ com $i \leq j$, a melhor solução para o subproblema que recebe a string $s[p_{i-1}+1\,..\,p_{j+1}]$ e as posições de corte $p_i, p_{i+1}, \ldots, p_j$ como entrada, podemos concluir facilmente que A é uma matriz de recorrência de intervalos com matriz de custo C onde $C[i][j] = p_{j+1} - p_{i-1}$. O valor de $A[1][n]$ nos dará o tempo mínimo de concluir a tarefa desejada e uma ordem ótima das quebras pode ser reconstruída através da matriz de cortes ótimos de A.

Como observado anteriormente, se A é uma recorrência de intervalos, então A pode ser calculada em tempo $\mathcal{O}(n^3)$. Pretendemos aproveitar propriedades da matriz C e alguns resultados provados por Yao [17] para concluir que A é Knuth-Yao otimizável e aplicar o Algoritmo 5.6 para calcular A em tempo $\mathcal{O}(n^2)$. Vamos começar provando a seguinte proposição.

Proposição 5.9. *C é uma matriz Monge convexa.*

Demonstração. Sejam $i, j \in [n-1]$ quaisquer. Temos

$$\begin{aligned} C[i][j] + C[i+1][j+1] &= p_{j+1} - p_{i-1} + p_{j+2} - p_i \\ &= p_{j+1} - p_i + p_{j+2} - p_{i-1} \\ &= C[i+1][j] + C[i][j+1]. \end{aligned}$$

Com isso, vale que $C[i][j] + C[i+1][j+1] \leq C[i+1][j] + C[i][j+1]$ e usamos o Teorema 2.12 para concluir que C é Monge convexa. □

Definição 5.10 (Monótona nos intervalos)**.** *Uma matriz $C \in \mathbb{Q}^{n \times n}$ é monótona nos intervalos se para todo $i, i', j, j' \in [n]$ onde $i \leq i' \leq j \leq j'$, vale que*

$$C[i'][j] \leq C[i][j'].$$

A Definição 5.10 relaciona a distância entre os índices de linha e coluna da matriz com o valor da matriz. Aplicando esse conceito ao problema discutido nesta subseção, dizer que a matriz C é monótona nos intervalos é equivalente a dizer que, quanto maior a string que está sendo cortada, mais caro é o corte. Vamos provar que esta propriedade vale. Sejam $i, i', j, j' \in [n]$ tais que $i \leq i' \leq j \leq j'$. Vale que $C[i'][j] = p_{i'+1} - p_{j-1}$, por definição. Já que $p_{i'+1} \geq p_{i+1}$ e $-p_{j-1} \geq -p_{j'-1}$, temos $p_{i'+1} - p_{j-1} \geq p_{i+1} - p_{j'-1} = C[i][j']$.

Lema 5.11. *Se $A \in \mathbb{Q}^{n \times n}$ é uma recorrência de intervalos com matriz de custos C Monge convexa e monótona nos intervalos, então A é Monge convexa.*

Demonstração. Sejam A e C matrizes que respeitam as condições do enunciado e P a matriz de cortes ótimos de A. Sejam ainda i, i', j e $j' \in [n]$ onde $i \leq i' \leq j \leq j'$. Queremos mostrar que sempre vale a desigualdade de Monge, isto é, $A[i][j] + A[i'][j'] \leq A[i][j'] + A[i'][j]$. Seja $l = j' - i$. Usaremos indução em l. Se $l = 0$, vale que $i = i' = j = j'$ e a desigualdade vale trivialmente.

Fixamos $l > 0$, assumindo que a desigualdade vale nos casos onde $j' - i < l$. Se $i = i'$ ou $j = j'$, a desigualdade vale trivialmente. Se $i < i' = j < j'$, tomamos $x = P[i][j']$ o ponto de corte ótimo do estado $A[i][j']$, ou seja, $A[i][j'] = C[i'][j] + A[i'][x-1] + A[x][j]$. Assumimos que $x \leq j$ e temos que

$$\begin{aligned} A[i][j] + A[i'][j'] - A[i'][j] &= A[i][j] + A[j][j'] - A[j][j] \\ &\leq C[i][j] + A[i][x-1] + A[x][j] + A[j][j'] - A[j][j] && (5.12) \\ &\leq C[i][j] + A[i][x-1] + A[x][j'] && (5.13) \\ &\leq C[i][j'] + A[i][x-1] + A[x][j'] && (5.14) \\ &= A[i][j'], \end{aligned}$$

A desigualdade (5.12) vale pois $A[i][j] = C[i][j] + \min\{A[i][k-1] + A[k][j]) \mid k \in [i+1\,..\,j]\}$ e $i < x \leq j$. A (5.13) vale pois $x \leq j \leq j'$ e $j' - x < l$, portanto, pela hipótese de indução, vale que $A[x][j] + A[j][j'] \leq A[x][j'] + A[j][j]$. Finalmente, vale (5.14) pois C é monótona nos intervalos. Com isso, provamos que $A[i][j] + A[i'][j'] \leq A[i'][j] + A[i][j']$. No caso em que $x > j$, basta utilizar $A[j][j'] \leq C[j][j'] + A[j][x-1] + A[x][j']$ em vez do que foi utilizado em (5.12) e adaptar o passo (5.13) para aplicar a hipótese de indução em $i \leq j \leq x - 1$.

Se $i' < j$, definimos os pontos ótimos de corte $x = P[i][j']$ e $y = P[i'][j]$ e seguimos um raciocínio parecido com o caso anterior. Assumindo que $x \leq y$, temos

$$\begin{aligned} A[i][j] + A[i'][j'] &\leq C[i][j] + C[i'][j'] + A[i][x-1] + A[x][j] + A[i'][y-1] + A[y][j'] && (5.15)\\ &\leq C[i][j] + C[i'][j'] + A[x][j'] + A[y][j] + A[i][x-1] + A[i'][y-1] && (5.16)\\ &\leq C[i][j'] + A[i][x-1] + A[x][j'] + C[i'][j] + A[i'][y-1] + A[y][j] && (5.17)\\ &= A[i][j'] + A[i'][j], \end{aligned}$$

onde (5.15) se justifica pois $i < x \leq j$ e $i' < y \leq j'$. A hipótese de indução é aplicada em (5.16) ,pois $x \leq y \leq j \leq j'$ e $j' - x < l$. Por fim, (5.17) vale pelo fato de que C é Monge convexa. O caso em que $x > y$ é similar. Primeiro, basta observar que $i' < x \leq j'$ e $i < y \leq j$, o que implica que $A[i][j] + A[i'][j'] \leq C[i][j] + C[i'][j'] + A[i][y-1] + A[y][j] + A[i'][x-1] + A[x][j']$. Depois, é necessário substituir o passo (5.15) por essa desigualdade adaptando o passo (5.16) de acordo, usando indução em $i \leq i' \leq y - 1 \leq x - 1$. □

Com o Lema 5.11, mostramos que a matriz A que representa o nosso problema atual é Monge convexa.

Teorema 5.18. *Uma recorrência de intervalos A Monge convexa é Knuth-Yao otimizável.*

Demonstração. Seja $A \in \mathbb{Q}^{n\times n}$ uma matriz Monge convexa que é uma recorrência de intervalos com matriz de cortes ótimos P. Queremos mostrar que, para todo $i, j \in [n]$ com $i < j$, vale que $P[i][j-1] \leq P[i][j] \leq P[i+1][j]$. Observe que o caso em que $i+1 = j$ é trivial e assuma que $i + 1 < j$.

Da definição de recorrência de intervalos, temos

$$A[i][j] = \min\{A[i][k-1] + A[k][j] + C[i][j] \mid k \in [i\,..\,j-1]\} \text{ e}$$

$$A[i][j-1] = \min\{A[i][k-1] + A[k][j-1] \mid k \in [i\,..\,j-2]\}$$

para alguma matriz C. Tome candidatos x e y a valor k destes mínimos, índices $x, y \in [i\,..\,j-2]$. Assuma, sem perda de generalidade, que $x < y$. Vamos mostrar que a escolha de k como x atinge um valor menor do que a escolha de k como y em $A[i][j]$, isso também ocorre em $A[i][j-1]$. Formalmente, vamos mostrar que a desigualdade $A[i][x-1] + A[x][j] \leq A[i][y-1] + A[y][j]$ implica que $A[i][x-1] + A[x][j-1] \leq A[i][y-1] + A[y][j-1]$.

Assumimos, portanto, que $A[i][x-1]+A[x][j] \leq A[i][y-1]+A[y][j]$. Da condição de Monge aplicada às linhas $x < y$ e colunas $j - 1 < j$ de A, temos que $A[x][j-1] + A[y][j] \leq A[x][j] + A[y][j-1]$. Manipulando e combinando estas duas desigualdades é possível obter

$$A[i][x-1] - A[i][y-1] \leq A[y][j] - A[x][j] \leq A[y][j-1] - A[x][j-1]$$

que equivale a $A[i][x-1] + A[x][j-1] \leq A[i][y-1] + A[y][j-1]$ e demonstra a implicação proposta. Desta implicação concluímos que se $x = P[i][j]$ então $P[i][j-1] \neq y$ para todo $y > x$. Em outras palavras, para todo $i, j \in [n]$ tal que $i < j$ vale que $P[i][j-1] \leq P[i][j]$.

De forma simétrica, para mostrar que $P[i][j] \leq P[i+1][j]$, basta perceber que se $x, y \in [n]$ e $x > y$, então será verdade que a desigualdade $A[i][x-1] + A[x][j] < A[i][y-1] + A[y][j]$ implica que $A[i+1][x-1] + A[x][j] < A[i+1][y-1] + A[y][j]$, o que contradiz $P[i][j] > P[i+1][j]$, provando que $P[i][j] \leq P[i+1][j]$. □

Com o Teorema 5.18, vale que A é Knuth-Yao otimizável e podemos aplicar a otimização de Knuth-Yao para resolver o problema em tempo $\mathcal{O}(n^2)$.

Capítulo 6

Estruturas de dados em programação dinâmica

Em certas ocasiões, ao aplicar programação dinâmica no cálculo de alguma recorrência, podemos manter algumas das entradas já calculadas em uma estrutura de dados que pode nos ajudar a calcular com mais eficiência as entradas seguintes. Neste capítulo, exemplificaremos esta possibilidade.

No Capítulo 7 apresentaremos um exemplo no qual a convexidade (ou concavidade) da recorrência calculada nos permite usar uma estrutura de dados de forma parecida com a apresentada aqui. Este capítulo, portanto, é uma introdução à utilização de estruturas de dados em programação dinâmica que deve facilitar a compreensão do Capítulo 7.

6.1 O problema da subsequência crescente de peso máximo

O problema discutido nesta seção é equivalente ao problema "Fundraising" da Final Brasileira da Maratona de Programação de 2017. Informações sobre esta prova podem ser encontradas em http://maratona.ime.usp.br/resultados17/.

Problema 6.1 (Subsequência crescente de peso máximo)**.** *Dados um natural n e dois vetores $v, w \in \mathbb{Q}^n$, descobrir uma subsequência s de $[1\,..\,n]$ crescente em v e de soma máxima em w. Formalmente, a subsequência s deve respeitar $v_{s_i} < v_{s_j}$ para todo par de índices $i, j \in [1\,..\,|s|]$ tal que $i < j$ e deve maximizar $\sum_{i \in s} w_i$.*

Podemos resolver este problema com tempo $\mathcal{O}(n^2)$ utilizando programação dinâmica na recorrência

$$E(i) = \max(\{E(j) \mid j \in [i-1] \text{ e } v_j < v_i\} \cup \{0\}) + w_j, \tag{6.2}$$

onde, para todo $i \in [1\,..\,n]$, o valor $f(i)$ é o peso da melhor sequência válida que contém i como último elemento.

Um caso especial deste problema onde o vetor w é um vetor unitário é o problema clássico da maior subsequência crescente. É conhecido o fato de que este problema pode ser resolvido em tempo $O(n \lg(n))$ [9].

6.2 Fronteira de Pareto

Apresentaremos uma estrutura de dados que vai nos ajudar a resolver o Problema 6.1 em tempo $\mathcal{O}(n \lg(n))$, para isso, vamos utilizar algumas definições e resultados.

Definição 6.3 (Relação de dominância)**.** *Sejam $a = (a_1, a_2)$ e $b = (b_1, b_2)$ elementos de $\mathbb{Q}^2$. Se $a_1 \geq b_1$ e $a_2 \geq b_2$ escrevemos $a \succeq b$ (a domina b). Além disso, se $S \subseteq \mathbb{Q}^2$, um ponto $a \in \mathbb{Q}$ é dominado em S se existe um outro ponto $b \in S$ que domina a.*

Proposição 6.4. *A relação de dominância é uma ordem parcial [7, Apêndice B] em $\mathbb{Q}^2$.*

Demonstração. Se $a = (a_1, a_2) \in \mathbb{Q}^2$, vale $a_1 \geq a_1$ e $a_2 \geq a_2$, portanto $a \succeq a$ e $\succeq$ é reflexiva. Se $a = (a_1, a_2)$ e $b = (b_1, b_2) \in \mathbb{Q}^2$ são tais que $a \succeq b$ e $b \succeq a$, então $a_1 = b_1$ e $a_2 = b_2$, portanto $a = b$ e $\succeq$ é antissimétrica. Se $a = (a_1, a_2), b = (b_1, b_2)$ e $c = (c_1, c_2) \in \mathbb{Q}^2$ são tais que $a \succeq b$ e $b \succeq c$, então $a_1 \geq c_1$ e $a_2 \geq c_2$, logo $a \succeq c$ e $\succeq$ é transitiva. □

Como dito, desenvolveremos uma estrutura de dados que irá agilizar o cálculo de uma recorrência. Tal estrutura será capaz de calcular, para todo $c \in \mathbb{Q}$, o máximo atingido pela segunda coordenada de algum elemento que tem valor pelo menos c na primeira coordenada e já foi adicionado à estrutura. A função $M(S, c)$ da Definição 6.5 abaixo ajuda a descrever este valor, já que, se S é o conjunto dos elementos adicionados na estrutura, $M(S, c)$ define exatamente o valor calculado pela estrutura.

Definição 6.5. *Dados $S \subset \mathbb{Q}^2$ finito e $c \in \mathbb{Q}$, definimos $M(S, c) = \max\{a_2 \mid (a_1, a_2) \in S \text{ e } a_1 \geq c\}$.*

Com estas definições estamos prontos para apresentar a estrutura de dados que vai agilizar a solução do Problema 6.1.

Definição 6.6 (Fronteira de Pareto)**.** *A fronteira de Pareto $\mathcal{P}(S)$ de um conjunto finito $S \subset \mathbb{Q}^2$ é uma estrutura de dados que contém todos os elementos não dominados de S, isto é, $x \in \mathcal{P}(S)$ se e somente se $x \in S$ e x não é dominado em S. Podemos realizar as seguintes operações sobre $\mathcal{P}(S)$:*

Insere$(\mathcal{P}(S), x)$ *Inserir um novo elemento $x \in \mathbb{Q}^2$ em S em tempo $\mathcal{O}(\lg(|S|))$ amortizado.*

CalculaM$(\mathcal{P}(S), c)$ *Descobrir, dado um $c \in \mathbb{Q}$, o valor $M(S, c)$ em tempo $\mathcal{O}(\lg(|S|))$.*

Para implementar esta estrutura de dados, vamos manter os elementos de $\mathcal{P}(S)$ em uma árvore de busca binária balanceada ordenada lexicograficamente, isto é, crescentemente pela primeira coordenada e, em caso de empate, crescentemente pela segunda. Se dois elementos $a, b \in \mathbb{Q}^2$ são tais que b é maior lexicograficamente do que a, denotamos $b \geq a$.

Definimos as operações **Sucessor**$(\mathcal{P}(S), a)$ e **Predecessor**$(\mathcal{P}(S), a)$ para cada elemento $a \in \mathbb{Q}^2$. A primeira retorna o menor elemento lexicograficamente maior ou igual a a e a segunda, similarmente, retorna o maior elemento lexicograficamente menor ou igual a a. Em ambas as operações, se o elemento desejado não existe, o valor NULL é retornado. Já que a árvore de busca binária é balanceada e está ordenada lexicograficamente, é possível realizar estas operações em tempo $\mathcal{O}(\lg(|\mathcal{P}(S)|))$. Permitimos a realização de chamadas onde a segunda coordenada de a é $-\infty$ ou ∞. É possível implementar este tipo de operação como um caso especial da primeira,

porém, também é possível definir, para cada aplicação, um valor suficientemente pequeno para $-\infty$ e um outro suficientemente grande para ∞ que torne esta chamada conveniente.

Guardar os elementos da fronteira nesta ordem mantém os elementos ordenados crescentemente pela primeira coordenada, o que garante uma propriedade interessante, descrita pela Proposição 6.7 a seguir.

Proposição 6.7. *Se $S \subset \mathbb{Q}^2$ é finito e os elementos de $\mathcal{P}(S)$ estão em ordem crescente na primeira coordenada, então eles estão em ordem decrescente na segunda coordenada.*

Demonstração. Suponha que existam $x = (x_1, x_2)$ e $y = (y_1, y_2) \in \mathcal{P}(S)$ tais que $x_1 \leq x_2$ e $y_1 \leq y_2$. Assim, $y \succeq x$ e $x \notin \mathcal{P}(S)$. □

A Proposição 6.7 mostra que $M(\mathcal{P}(S), c) = \textsc{Sucessor}(\mathcal{P}(S), (c, -\infty)$ para todo $c \in \mathbb{Q}$.

Proposição 6.8. *Se S é um conjunto finito e um ponto $x \in \mathbb{Q}^2$ é dominado em S, então, x é dominado em $\mathcal{P}(S)$.*

Demonstração. Sejam S e x dados como no enunciado. Se x é dominado em S existe pelo menos um elemento em S que domina x, dentre estes, tome por y algum que é maximal em relação a $\succeq$, isto é, que não seja dominado em S. Isto é possível pois $\succeq$ é uma ordem parcial e S é finito. Vale que $y \in \mathcal{P}(S)$, portanto, x é dominado em $\mathcal{P}(S)$. □

A Proposição 6.9 abaixo garante que $M(\mathcal{P}(S), c) = M(S, c)$ para todo $c \in \mathbb{Q}$. Já que $\lg(|\mathcal{P}(S)|) = \mathcal{O}(\lg(|S|))$ e calcular $M(\mathcal{P}(S), c)$ a partir de uma chamada a Sucessor custa tempo $\mathcal{O}(\lg(|\mathcal{P}(S)|))$, o cálculo da operação CalculaM é realizado em tempo $\mathcal{O}(\lg(|\mathcal{S}|))$.

Proposição 6.9. *Para todo $S \subset \mathbb{Q}^2$ finito e $c \in \mathbb{Q}$ vale $M(S, c) = M(\mathcal{P}(S), c)$.*

Demonstração. Já que $\mathcal{P}(S) \subseteq S$ segue que $M(S, c) \geq M(\mathcal{P}(S)$. Vamos provar o outro sentido da igualdade. Suponha que $M(S, c) > M(\mathcal{P}(S), c)$. Existe, portanto, algum elemento $a = (a_1, a_2) \in S$ tal que $a \notin \mathcal{P}(S)$, $a_2 \geq c$ e $a_1 > M(\mathcal{P}(S), c)$. Porém, já que $a \notin \mathcal{P}(S)$, vale que a é dominado em S e, pela Proposição 6.8, dominado em $\mathcal{P}(S)$, logo, existe $b \in \mathcal{P}(S)$ tal que $b \succeq a$. Concluímos que $b_2 \geq a_2 \geq c$ e $M(\mathcal{P}(S), c) \geq b_1 \geq a_1 = M(S, c)$, um absurdo. □

Para um $a \in \mathbb{Q}^2$ qualquer, podemos realizar a operação $\textsc{Insere}(\mathcal{P}(S), a)$ eficientemente. Primeiramente, devemos verificar se a é dominado em $\mathcal{P}(S)$. Para tanto, basta verificar se $\textsc{Sucessor}(\mathcal{P}(S), a)$ domina a. Se esta chamada retornar NULL, todo elemento com primeira coordenada maior ou igual à primeira coordenada de a, tem a segunda coordenada menor do que a de a, portanto, o elemento a não é dominado. Se a chamada retornar um elemento b que não domina a, este deve ter a segunda coordenada menor do que a segunda coordenada de a. Com isso, sabemos que todo elemento lexicograficamente maior do que b não domina a, graças à Proposição 6.7 e, portanto, não existe outro elemento em $\mathcal{P}(S)$ que domina a.

Além de descobrir se a é dominado, precisamos remover os elementos dominados por a de $\mathcal{P}(S)$. Para tanto, perceba que se algum elemento de $\mathcal{P}(S)$ for dominado por a, o elemento $\textsc{Predecessor}(\mathcal{P}(S), a)$ será, também, dominado por a. Graças a este fato, podemos buscar este elemento e removê-lo, caso seja dominado, sucessivamente, até que $\textsc{Predecessor}(\mathcal{P}(S), a)$ seja NULL ou não seja dominado por a.

Note que todo elemento pode ser removido da estrutura apenas uma vez. Concluímos, levando em conta os custos das chamadas a PREDECESSOR e SUCESSOR, que a operação INSERE é realizada em tempo $\mathcal{O}(\lg(|S|))$ amortizado. O Algoritmo 6.10 implementa as operações descritas aqui.

Algoritmo 6.10 Operações sobre a fronteira de Pareto de um conjunto

```
1:  função INSERE(P(S), x = (x_1, x_2))
2:      y ← SUCESSOR(P(S), x)
3:      se y ≠ NULL e y ⪰ x então
4:          Interrompe a execução da função
5:      y ← PREDECESSOR(P(S), x)
6:      enquanto y ≠ NULL e x ⪰ y
7:          Remove y de P(S)
8:          y ← PREDECESSOR(P(S), x)
9:      Insere x na árvore de P(S)
10: função CALCULAM(P(S), c)
11:     y = (y_1, y_2) ← SUCESSOR(P(S), (c, −∞))
12:     se y ≠ NULL então
13:         devolve y_2
14:     senão
15:         devolve −∞
```

Uma implementação de fronteira de Pareto como descrita pelo algoritmo acima em `C++` pode ser encontrada na pasta de implementações com o nome `Pareto.cpp`.

6.3 Otimização para a subsequência crescente de peso máximo

Com a possibilidade de manter a fronteira de Pareto de um subconjunto finito de $\mathbb{Q}^2$, vamos calcular cada $f(i)$ definido na solução original do Problema 6.1 em tempo $\mathcal{O}(\lg(n))$ onde n é o tamanho dos vetores v e w dados pelo problema. Para isso basta notar que podemos reescrever a recorrência (6.2) como $f(i) = M(S_i, -v_i)$ onde $S_i = \{(-v_j, f(j)) \mid j < i\} \cup \{(0, \infty)\}$. Aproveitando a estrutura da fronteira de Pareto, escrevemos o Algoritmo 6.11 que calcula todos os valores de f em tempo $\mathcal{O}(n \lg(n))$ e retorna o máximo entre 0 e todos estes valores calculados, já que esta é a resposta para o problema proposto.

Algoritmo 6.11 Solução do Problema 6.1

```
1: função SUBSEQUÊNCIACRESCENTEPESOMÁXIMO(n, v, w)
2:     Inicializa a fronteira de Pareto P com o elemento (0, ∞)
3:     r ← 0
4:     para i de 1 até n faça
5:         f(i) = CALCULAM(P, −v_i)
6:         INSERE(P, (−v_i, f(i)))
7:         r ← max(r, f(i))
8:     devolve r
```

Capítulo 7

Busca em matrizes online

Neste capítulo estudamos formas de encontrar índices ótimos em matrizes onde certas entradas são desconhecidas a priori. Inicialmente o formato das matrizes abordadas será explicitado juntamente com um exemplo de um caso específico interessante desta técnica apresentado por Galil e Park [11]. Galil e Giancarlo [10] mostraram que este problema pode ser resolvido em tempo $\mathcal{O}(n \lg(n))$ para matrizes $n \times n$ convexas e côncavas. Apresentaremos estas soluções aqui. O caso convexo é resolvido na Seção 7.1. Na Seção 7.2 mostramos como adaptar o que foi discutido para o caso côncavo. Em certas matrizes, é possível melhorar o tempo destes algoritmos de $\mathcal{O}(n \lg(n))$ para $\mathcal{O}(n)$. Esta possibilidade é explicada na Seção 7.3.

Para conseguir lidar com matrizes que não são inteiramente conhecidas à priori, é necessário especificar um formato que deve ser respeitado, definindo quais dependências podem aparecer nestas matrizes. O formato descrito pela Definição 7.1 abaixo aparece naturalmente em recorrências de programação dinâmica, portanto, é útil para esta aplicação. Desde já, discutiremos os algoritmos apenas em termos de buscas de índices de mínimos de linhas em matrizes triangulares superiores em 0. É fácil adaptar os conhecimentos e definições para os casos de busca de índices de máximos, bem como para matrizes triangulares superiores ou inferiores em c, desde que $c \geq 0$. Também é possível trabalhar com buscas de índices ótimos de colunas.

Definição 7.1 (Matriz triangular online)**.** *Dizemos que a matriz $A \in \mathbb{Q}^{n \times n}$ triangular superior em 0 com índices de mínimos de linhas R é online nas linhas se as entradas de uma coluna j da matriz podem depender de qualquer $R[j']$ onde $j' \geq j$, isto é, o índice de mínimo das linhas de índice maior ou igual a j.*

Dizemos que uma entrada $A[i][j]$ da matriz está disponível se for possível calcular imediatamente seu valor, isto é, se os índices de mínimos das linhas de índice maior ou igual j já foram calculados. Se todas as entradas de uma linha (coluna) estão disponíveis, dizemos que tal linha (coluna) está disponível. Inicialmente, apenas a coluna n está disponível, já que $R[n]$ é indefinido em toda matriz triangular superior em 0. Este fato faz com que, inicialmente, apenas a linha $n-1$ esteja disponível e, portanto, seja a única para a qual podemos calcular o índice de mínimo. Assim que calculamos a resposta da linha $n-1$, a coluna $n-1$ e, consequentemente, a linha $n-2$ se tornam disponíveis. Calculamos a resposta para a linha $n-2$, seguida da $n-3$ e assim por diante. Isso nos diz que é necessário calcular o vetor R em ordem decrescente de seus índices. Esta restrição inviabiliza os

algoritmos já discutidos nas Seções 3 e 4 para encontrar índices ótimos, já que estes exigem que o calculo seja realizado em ordens diferentes desta.

A Figura 7.2 ilustra as relações de dependência nas matrizes tratadas nesta seção. As entradas da coluna com listras verticais em verde dependem dos valores das entradas das linhas correspondentes hachuradas diagonalmente em vermelho. O Problema 7.3 exemplifica a utilidade de trabalhar com matrizes deste tipo.

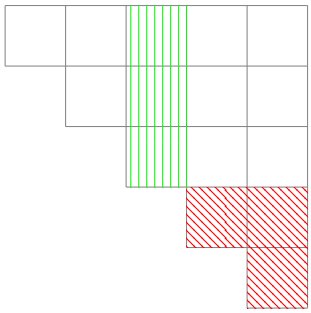

Figura 7.2: Dependências de uma matriz triangular superior online nas linhas.

Problema 7.3. *Dada uma matriz de custos* $C \in \mathbb{Q}^{n\times n}$*, computar o vetor* E *tal que* $E[n] = 0$ *e* $E[i] = \min_{j>i}\{C[i][j] + E[j]\}$ *para todo* $i \in [n-1]$.

Com uma matriz C e um vetor E definidos como no Problema 7.3, criamos a matriz $B \in \mathbb{Q}^{n\times n}$ triangular superior em 0 de forma que, para todo $i, j \in [n]$ onde $i < j$, vale que $B[i][j] = C[i][j]+E[j]$. Se R é o vetor de índices de mínimos das linhas de B, temos que $E[i] = B[i][R[i]]$ para toda linha $i \in [n-1]$ de B. Além disso, a matriz B é online nas linhas. Resolver o problema dado é equivalente a encontrar os índices de mínimos das linhas de B. O Lema 2.14 garante que se C é Monge, então B é Monge no mesmo sentido e, portanto, totalmente monótona nas linhas no mesmo sentido.

Lembramos que, já que estamos trabalhando com minimização de linhas, dizemos que uma coluna a de A é melhor do que uma outra coluna b de A em uma linha i de A se tem valor menor ou se tem valor igual e índice menor, isto é, se $A[i][a] < A[i][b]$ ou se $A[i][a] = A[i][b]$ e $a < b$. Dizemos também que a é pior do que b quando b é melhor do que a. Assumimos que, quando disponíveis, as entradas de uma matriz online podem ser calculadas em tempo $\mathcal{O}(1)$.

7.1 Caso convexo

Voltamos nosso foco ao problema de encontrar o vetor R de índices de mínimos das linhas de uma matriz A qualquer triangular superior em 0 online nas linhas, totalmente monótona convexa por linhas. Como já observado, devemos encontrar estes valores um a um da última até a primeira linha, necessariamente nesta ordem.

É fácil deduzir um algoritmo $\mathcal{O}(n^2)$ para encontrar o vetor R. Basta percorrer as linhas na ordem já dita e, para cada uma delas, buscar o mínimo olhando para todas as entradas da linha. Em vez de realizar esta busca ingênua, vamos fazer como no Capítulo 6 e manter uma estrutura de dados com informações úteis nesta busca.

Definição 7.4 (Envelope)**.** *Um envelope $\mathcal{E}$ é uma estrutura de dados que mantém uma matriz $A \in \mathbb{Q}^{n \times m}$, o inteiro n que representa a quantidade de linhas de A e um conjunto de colunas de A e dá suporte às seguintes operações:*

Constrói(A,n,m) *Devolve um envelope sobre a matriz A de n linhas e m colunas guardando um conjunto vazio de colunas;*

Insere($\mathcal{E}$,j) *Insere uma coluna j de A no envelope $\mathcal{E}$. A coluna j não pode possuir entradas indefinidas, deve estar disponível e deve ter índice menor do que todas as outras colunas já em $\mathcal{E}$;*

Remove($\mathcal{E}$) *Remove a última linha da matriz A e decresce o valor n de acordo;*

Calcula($\mathcal{E}$) *Devolve a melhor coluna de $\mathcal{E}$ na linha n. Formalmente, esta operação devolve o valor* $\min\{j \in \mathcal{E} \mid A[n][j] \leq A[n][j']$ *para todo* $j' \in \mathcal{E}\}$.

Com esta definição, estamos prontos para descrever o Algoritmo 7.5 que recebe a matriz A e devolve o vetor R. Faltará apenas descrever uma implementação eficiente da estrutura de envelope no caso onde A é uma matriz totalmente monótona convexa.

Algoritmo 7.5 Mínimos de linhas online

```
1: função ONLINE(A, n)
2:     E ← CONSTRÓI(A, n, n)
3:     para i de n − 1 até 1 faça
4:         REMOVE(E)
5:         INSERE(E, i + 1)
6:         R[i] = CALCULA(E)
7:     devolve R
```

Ao calcular o valor $R[i]$ para uma linha i qualquer, todas as colunas de índice maior do que i estão presentes no envelope e a matriz guardada pelo envelope tem i como sua última linha. Isso faz com que a função Calcula retorne o índice de mínimo da linha i e prova que o Algoritmo 7.5 funciona corretamente.

A Figura 7.6 ilustra uma possível progressão deste algoritmo e da estrutura de envelope em uma matriz com lado $n = 8$. A primeira imagem (da esquerda para a direita) ilustra o estado do envelope durante a execução da linha 6 quando $i = n - 1$. A três imagens seguintes ilustram o mesmo momento quando i é $n - 2$, $n - 3$ e $n - 4$, respectivamente. Em cada uma das imagens, a matriz A guardada pela estrutura é representada. A linha i fica circulada em vermelho. Em cada coluna disponível estão marcadas, com listras verticais em verde, as linhas para as quais esta coluna é melhor do que todas as outras disponíveis.

Se o algoritmo for executado como indicado pela Figura 7.6, teremos, nas quatro últimas posições, 7, 6, 5 e 4 do vetor R retornado, os valores, 8, 8, 8 e 5. Dizemos que uma coluna j é ótima para $\mathcal{E}$ em uma linha qualquer de A se ela é melhor do que todas as outras colunas de $\mathcal{E}$ naquela linha. Note que, na representação acima, em cada iteração, cada coluna é ótima para $\mathcal{E}$ apenas em um intervalo (potencialmente vazio) das linhas e que o índice de início deste intervalo é crescente no índice da coluna. Veremos adiante que estas propriedades são sempre respeitadas e são convenientes para a implementação do envelope. O Lema 7.7 prova a primeira delas.

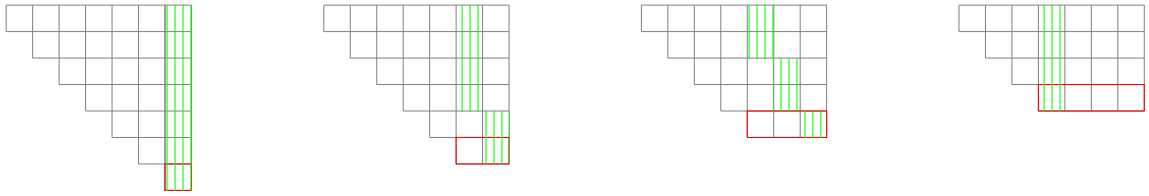

Figura 7.6: Progressão do Algoritmo 7.5.

Lema 7.7. *Se j é uma coluna de $\mathcal{E}$ ótima para $\mathcal{E}$ em duas linhas a e b tais que $a \leq b$, então ela é ótima para $\mathcal{E}$ em qualquer linha $c \in [a\,..\,b]$.*

Demonstração. Suponha que j é uma coluna de $\mathcal{E}$ ótima para $\mathcal{E}$ nas linhas a e b. Seja c uma linha de A em $[a\,..\,b]$ e j' uma coluna de $\mathcal{E}$ diferente de j e melhor do que j em c. Se $c = a$ ou $c = b$, não há nada a provar. Assumimos que $c \in [a+1\,..\,b-1]$. Se $j' < j$, temos que $A[b][j'] \leq A[b][j]$. Pela total monotonicidade de A, vale que $A[a][j'] \leq A[a][j]$ e j' é melhor do que j em A, um absurdo. Se $j < j'$, de maneira similar, vale que $A[c][j] \leq A[c][j']$. Pela total monotonicidade de A, vale que j é melhor do que j' em b. □

Considere um envelope $\mathcal{E}$ que guarda uma matriz A. Se existe uma coluna j de $\mathcal{E}$ que é pior do que alguma outra coluna do envelope em cada linha da matriz A, a coluna j pode ser ignorada pelo envelope. Dizemos que uma coluna deste tipo é inválida em $\mathcal{E}$ e que colunas que são ótimas para $\mathcal{E}$ em pelo menos uma linha são válidas em $\mathcal{E}$. Durante a implementação do envelope, vamos remover todas as colunas inválidas da estrutura.

Definição 7.8 (Coluna válida)**.** *Uma coluna $j \in J$ é inválida em um envelope $\mathcal{E}$ com uma matriz A se, para toda linha i de A, existe uma outra coluna j' de A melhor do que j em i.*

Além de remover as colunas inválidas, manteremos as colunas ordenadas crescentemente no envelope. Assim, podemos denotar por $\mathcal{E}_k$ a coluna válida de k-ésimo menor índice em $\mathcal{E}$. Com isso temos, por exemplo, que $\mathcal{E}_1$ é a menor coluna válida de $\mathcal{E}$. Por simplicidade, denotamos $\mathcal{E}_{-k} = \mathcal{E}_{|\mathcal{E}|-k}$ para todo k inteiro onde $-|\mathcal{E}| \leq k < 0$. Assim, por exemplo, $\mathcal{E}_{-1}$ é a coluna de $\mathcal{E}$ válida de maior índice.

Definimos o valor $s_{\mathcal{E}}(j)$ para toda coluna j válida de $\mathcal{E}$ que nos dá a primeira linha para a qual esta é ótima em $\mathcal{E}$. Pela definição de válida, se não existe tal linha, a coluna j é inválida.

Lema 7.9. *Se j e j' são duas colunas válidas de um envelope $\mathcal{E}$ onde $j < j'$, vale que $s_{\mathcal{E}}(j) < s_{\mathcal{E}}(j')$.*

Demonstração. Tome duas colunas válidas j e j' de $\mathcal{E}$ tais que $j < j'$. Suponha, por absurdo, que $b = s_{\mathcal{E}}(j) \geq s_{\mathcal{E}}(j') = a$. Vale que $A[b][j] \leq A[b][j']$ já que j é ótima para $\mathcal{E}$ em b. Pela total, monotonicidade de A, vale que $A[a][j] \leq A[a][j']$, uma contradição, já que j' é ótima para $\mathcal{E}$ em a. □

Definir s é muito útil para descobrir qual é o intervalo de linhas para o qual uma coluna de $\mathcal{E}$ é ótima. Tome um inteiro $k \in [1\,..\,|\mathcal{E}|-1]$ e considere a coluna $\mathcal{E}_k$, isto é, uma coluna válida do envelope diferente da última. Sabemos que esta coluna é ótima no intervalo $[s_{\mathcal{E}}(\mathcal{E}_k)\,..\,s_{\mathcal{E}}(\mathcal{E}_{k+1})-1]$. Além disso, a coluna $\mathcal{E}_{-1}$ é ótima no intervalo $[s_{\mathcal{E}}(\mathcal{E}_{-1})\,..\,n]$ onde n é a última linha da matriz guardada pelo envelope. De forma similar a s, definimos, para cada j válido de $\mathcal{E}$, o valor $t_{\mathcal{E}}(j)$, que nos dá o índice da última linha para a qual j é ótimo em $\mathcal{E}$.

Será interessante calcular o valor de s para colunas válidas do envelope. Para isso, definimos uma outra operação importante. Se a e b são duas colunas do envelope onde $a < b$, vale que INTERSECTA$(\mathcal{E}, a, b)$ é a primeira linha do envelope onde b é melhor do que a. Para calcular este valor, podemos realizar uma busca binária nas linhas da matriz guardada por $\mathcal{E}$. Se m é uma linha onde a é melhor do que b, pela total monotonicidade, b só é melhor do que a a partir de alguma linha maior do que m. Caso contrário, também pela total monotonicidade, b supera a no máximo na linha m. O Algoritmo 7.16 mostra como aplicar esta ideia.

Algoritmo 7.10 Intersecção de colunas no caso convexo

```
1:  função INTERSECTA(E, a, b)                          ▷ a < b
2:      ℓ ← 1
3:      r ← última linha da matriz de E
4:      enquanto ℓ < r
5:          m ← ⌊(ℓ + r)/2⌋
6:          se A[m][a] ≤ A[m][b] então                  ▷ a é melhor do que b em m
7:              ℓ ← m + 1
8:          senão
9:              r ← m
10:     devolve ℓ
```

Se $k \in [2\,..\,\mathcal{E}]$, vale que $\mathcal{E}_{k-1}$ é melhor do que $\mathcal{E}_k$ em toda linha antes de $s_{\mathcal{E}}(\mathcal{E}_k)$ e pior em todas as outras. Isso faz com que valha a igualdade $s_{\mathcal{E}}(\mathcal{E}_k) = \text{INTERSECTA}(\mathcal{E}, \mathcal{E}_{k-1}, \mathcal{E}_k)$. Além disso, vale que $s_{\mathcal{E}}(\mathcal{E}_1) = 1$. Isso nos dá uma forma de calcular $s_{\mathcal{E}}$ com facilidade a partir da função INTERSECTA.

Estamos prontos para descrever a implementação de um envelope no caso convexo. O envelope deve guardar uma lista $\mathcal{E}$ das colunas válidas adicionadas e o valor n que representa o índice da última linha da matriz. Cada operação sobre $\mathcal{E}$ deve manter duas invariantes: a lista $\mathcal{E}$ está ordenada crescentemente e guarda uma coluna que foi adicionada se e somente se ela é válida em $\mathcal{E}$. Note que estas duas invariantes são suficientes para mostrar que s pode ser calculada da maneira descrita no parágrafo anterior.

Algoritmo 7.11 Implementação de envelope no caso convexo

```
1:  função CALCULA(E)
2:      devolve E_{-1}
3:  função REMOVE(E)
4:      se existe E_{-1} e s_E(E_{-1}) = n então
5:          Tira E_{-1} de E
6:      n ← n − 1
7:  função INSERE(E, j)                                 ▷ j é menor do que E_1
8:      se não existe E_1 ou A[1][E_1] ≥ A[1][j] então  ▷ E_1 é pior do que j em 1
9:          enquanto existe E_1 e A[t_E(E_1)][E_1] ≥ A[t_E(E_1)][E_1]   ▷ E_1 é pior do que j em t_E(E_1)
10:             Tira E_1 de E
11:         j será o primeiro elemento de E
```

A operação CALCULA deve retornar a coluna ótima em n. Já que $\mathcal{E}_{-1}$ é a coluna válida de maior índice, ela é tal coluna. A operação REMOVE deve remover a última linha da matriz. Se houver alguma coluna que é ótima apenas nesta última linha, para que as invariantes sejam mantidas, a operação deve remover esta coluna de $\mathcal{E}$. Finalmente, a operação INSERE deve considerar dois casos.

Pode ser que a coluna j adicionada seja inválida. Pela Proposição 7.12, isso ocorre se e somente se existe pelo menos uma coluna em $\mathcal{E}$ e $\mathcal{E}_1$ é melhor do que j na linha 1. Se este for o caso, ignoramos a operação de inserção. Além disso, é possível que j invalide alguma coluna já presente em $\mathcal{E}$. Se isso ocorre, então, pela Proposição 7.13, a coluna $\mathcal{E}_1$ é invalidada. Assim, para remover todas as colunas invalidadas por j, basta checar se $\mathcal{E}_1$ é uma dessas. Caso não seja, o trabalho está feito. Caso seja, removemos esta e repetimos o procedimento até que não haja mais colunas invalidadas por j. Além disso, é fácil se convencer de que a adição de novas colunas em $\mathcal{E}$ não torna válida uma coluna que já estava em $\mathcal{E}$ e era inválida. Isso mostra que as invariantes propostas são mantidas pelas operações descritas.

Proposição 7.12. *Uma coluna j menor do que todas as colunas em $\mathcal{E}$ é inválida para $\mathcal{E}$ quando adicionada em $\mathcal{E}$ se e somente se existe $\mathcal{E}_1$ e j é pior do que $\mathcal{E}_1$ na linha 1.*

Demonstração. Se $\mathcal{E}$ é vazio, j será ótima para $\mathcal{E}$ em todas as linhas após adicionada. Se j for pior do que $\mathcal{E}_1$ na linha 1, pela total monotonicidade, também é pior do que $\mathcal{E}_1$ em toda linha maior do que 1, portanto, seria inválida se adicionada em $\mathcal{E}$. Se j for melhor do que $\mathcal{E}_1$ na linha 1, é melhor do que todas as outras de $\mathcal{E}$ também na linha 1, já que $\mathcal{E}_1$ é ótima para $\mathcal{E}$ na linha 1, portanto, seria válida se adicionada. □

Proposição 7.13. *Se alguma coluna de $\mathcal{E}$ é invalidada por uma coluna j de índice menor do que todas as colunas de $\mathcal{E}$, então $\mathcal{E}_1$ é invalidada por j.*

Demonstração. Suponha que j invalida uma coluna j' de $\mathcal{E}$ válida tal que $j' > \mathcal{E}_1$. Sabemos que $\mathcal{E}_1$ só pode ser ótima para $\mathcal{E}$ em linhas de índices menores do que $s_{\mathcal{E}}(j')$ pelos Lemas 7.7 e 7.9. Já que j é melhor do que j' em $s_{\mathcal{E}}(j')$, pela total monotonicidade, é melhor do que $\mathcal{E}_1$ em todas as linhas de índice menor do que $s_{\mathcal{E}}(j')$. Provamos que j invalida $\mathcal{E}_1$. □

Vamos analisar a complexidade do Algoritmo 7.5 assumindo a nossa implementação de envelope descrita pelo Algoritmo 7.11. As verificações das linhas 4 e 9 do Algoritmo 7.11 custam tempo $\mathcal{O}(\lg(n))$ já que envolvem chamadas à função INTERSECTA. Estas verificações são realizadas uma vez a cada chamada de REMOVE e, em INSERE, podem ser realizadas uma vez caso a coluna inserida seja válida e uma outra vez para cada coluna invalidada em $\mathcal{E}$ por j. Já que cada coluna só pode ser removida uma vez da estrutura e apenas $\mathcal{O}(n)$ colunas são adicionadas pelo Algoritmo 7.5, estas chamadas ocorrem no máximo $\mathcal{O}(n)$ vezes. Remover uma linha de A, bem como inserir ou remover colunas do início ou final da lista que mantém as colunas de $\mathcal{E}$ e devolver $\mathcal{E}_{-1}$, são operações realizadas em $\mathcal{O}(1)$. Esta análise mostra que o Algoritmo 7.11 leva tempo $\mathcal{O}(n \lg(n))$.

Uma implementação da estrutura de envelope em `C++` de acordo com o Algoritmo 7.11 pode ser encontrada na pasta de implementações com o nome `EnvelopeConvexo.cpp`. A estrutura `std::deque` permite que se insira e remova elementos do início ou fim da lista $\mathcal{E}$ em tempo constante, como necessário para a análise feita no parágrafo anterior.

7.2 Caso côncavo

Vamos resolver o problema de encontrar o vetor R de índices de mínimos das linhas de uma matriz A qualquer triangular superior em 0 online nas linhas, totalmente monótona côncava por linhas. Este

caso se desenvolve de maneira extremamente similar ao caso convexo. Basta apenas adaptar a estrutura de envelope já descrita. Com uma implementação adaptada de envelope, podemos utilizar o Algoritmo 7.5 exatamente da mesma forma como anteriormente. A definição de envelope também não muda, apenas a forma de implementar as operações.

A Figura 7.15 ilustra uma possível progressão dos quatro primeiros passos do caso côncavo para uma matriz de lado $n = 8$, como fizemos no caso convexo.

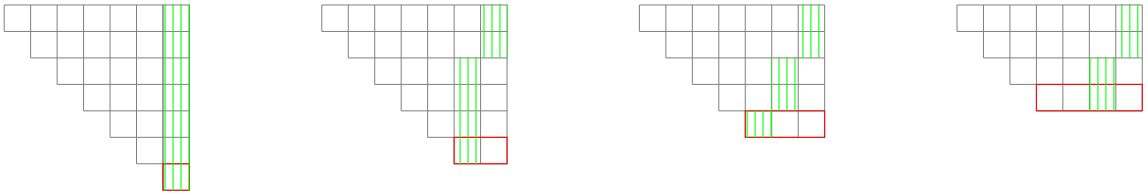

Figura 7.15: Progressão do Algoritmo 7.5.

Se o algoritmo for executado como indicado pela Figura 7.15, teremos, nas quatro últimas posições, 7, 6, 5 e 4 do vetor R retornado, os valores 8, 7, 6 e 7. Perceba que estes valores não formam um vetor de índices de mínimos das linhas crescente, o que pode ocorrer já que A é triangular. No caso convexo, pode ser provado que isso nunca ocorre, mas não usamos este fato lá. Além disso, é interessante notar que a coluna 5 se torna disponível no último passo, mas não é o mínimo de nenhuma linha da matriz.

Comparar as Figuras 7.6 e 7.15 explicita a diferença entre os casos convexo e côncavo. No caso convexo, por exemplo, os valores $s_{\mathcal{E}}(j)$ são crescentes no índice da coluna j. No caso côncavo, estes são decrescentes. A prova deste fato é análoga à do caso convexo, apresentada no Lema 7.9.

A operação INTERSECTA implementada no Algoritmo 7.16 é facilmente adaptável. Aqui, se a e b são colunas tais que $a < b$, então b pode ser melhor do que a apenas em um prefixo das linhas e a pode ser melhor do que b apenas em um sufixo das linhas de A. Por esse motivo, uma chamada de INTERSECTA$(\mathcal{E}, a, b)$ só pode ser realizada se $a > b$. Além desta diferença, graças à nossa definição de melhor, na linha 6, em vez de uma comparação com $\leq$, uma comparação estrita deve ser realizada. As implementações das três funções CALCULA, REMOVE e INSERE são diferentes do caso convexo. O Algoritmo 7.14 implementa esta versão adaptada.

Os argumentos para a validade da implementação do envelope convexo são análogos aos do caso

Algoritmo 7.14 Implementação de envelope no caso côncavo

```
1:  função CALCULA(E)
2:      devolve E_1
3:  função REMOVE(E)
4:      se existe E_1 e s_E(E_1) = n então
5:          Tira E_1 de E
6:      n ← n − 1
7:  função INSERE(E, j)                          ▷ j é menor do que E_1
8:      se não existe E_1 ou A[n][E_1] ≥ A[n][j] então   ▷ E_1 é pior do que j em n
9:          enquanto existe E_1 e A[s_E(E_1)][E_1] ≥ A[s_E(E_1)][j]   ▷ E_1 é pior do que j em s_E(E_1)
10:             Tira E_1 de E
11:         Insere j no início de E
```

Algoritmo 7.16 Intersecção de colunas no caso côncavo

```
1: função INTERSECTA(E, a, b)                                  ▷ a > b
2:     ℓ ← 1
3:     r ← última linha da matriz de E
4:     enquanto ℓ < r
5:         m ← ⌊(ℓ + r)/2⌋
6:         se A[m][a] < A[m][b] então                          ▷ a é melhor do que b em m
7:             ℓ ← m + 1
8:         senão
9:             r ← m
10:    devolve ℓ
```

côncavo, bem como a análise de complexidade. Uma implementação do envelope côncavo em `C++` pode ser encontrada em `EnvelopeConcavo.cpp`.

7.3 Envelope linear

Um caso especial interessante do problema resolvido nesta seção é aquele onde a matriz A é definida em termos de retas com coeficientes angulares ordenados. Formalmente, existem vetores $\alpha, \beta \in \mathbb{Q}^n$ tais que α é monótono e $A[i][j] = \alpha_j i + \beta_j$ para toda entrada definida da matriz A. É fácil provar que se α é decrescente, A é totalmente monótona convexa nas linhas e, se α é crescente, A é totalmente monótona côncava nas linhas.

Este é um bom exemplo de caso para o qual a função INTERSECTA pode ser calculada em tempo $\mathcal{O}(1)$. O Algoritmo 7.17 implementa este cálculo para esta situação no caso convexo. Para adaptar esta implementação para o caso côncavo, basta trocar o teto tomado na linha 2 pelo sucessor do chão da divisão, isto é, $\lfloor (\beta_a - \beta_b)/\alpha_b - \alpha_a) \rfloor + 1$. Se substituirmos esta função pela antiga INTERSECTA, tanto no caso convexo quanto a sua adaptação no côncavo, fazemos com que o Algoritmo 7.5 consuma tempo $\mathcal{O}(n)$ em vez de $\mathcal{O}(n \lg(n))$.

Algoritmo 7.17 Intersecção de colunas dadas por retas no caso convexo.

```
1: função INTERSECTA(E, a, b)
2:     devolve ⌈(β_a − β_b)/(α_b − α_a)⌉
```

É importante observar que, na prática, se os valores dos vetores α e β estiverem representados com ponto flutuante, não é trivial calcular, com exatidão, o valor $\lfloor (\beta_a - \beta_b)/(\alpha_b - \alpha_a) \rfloor$. Já que análise numérica não é o foco deste trabalho, a implementação desta técnica será realizada em `C++` com vetores α e β do tipo `long long`. Esta implementação para o caso convexo pode ser encontrada na pasta de implementações com o nome `EnvelopeConvexoLinear.cpp`.

Capítulo 8

Exemplos implementados

Neste capítulo descrevemos os exemplos implementados disponíveis em `implementacao/exemplos`. O objetivo de implementar exemplos é mostrar possibilidades de aplicações simples dos algoritmos discutidos e mostrar maneiras facilmente adaptáveis de utilizar as implementações desenvolvidas neste trabalho.

Cada exemplo pode ser implementado por mais de um dos algoritmos estudados. Uma implementação "trivial" também é fornecida para cada exemplo, demonstrando uma forma simples e pouco eficiente de resolver o problema proposto que não conta com os algoritmos discutidos. Aqui serão descritos os problemas resolvidos em cada exemplo, o formato de entrada e saída e serão listadas as soluções implementadas por cada um deles. Um eventual comentário sobre a modelagem feita pode ser realizado aqui.

Cada implementação de um exemplo leva um nome no formato `NomeDoExemplo.Algoritmo.cpp` e é compilado, a partir do utilitário `make` com comportamente descrito pelo `Makefile` presente em `implementacao`, para o arquivo `NomeDoExemplo.Algoritmo.out` onde `NomeDoExemplo` é especificado nas descrições abaixo.

8.1 Exemplo Monge

Este é um exemplo simples da aplicação dos algoritmos em matrizes Monge. Ele resolve o Problema 2.13 dado o vetor $w \in \mathbb{Q}_+^n$ e um inteiro k. A implementação leva o nome `ExemploMonge`.

O exemplo foi implementado com a otimização da Divisão e Conquista (`DivConq`) consumindo tempo $\mathcal{O}(kn \lg(n))$, com o algoritmo SMAWK (`SMAWK`) consumindo tempo $\mathcal{O}(kn)$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(kn^2)$.

Descrição da entrada

Na primeira linha da entrada são dados dois inteiros k e n separados por um espaço. Na segunda linha são dados os valores do vetor w, separados por espaços.

Descrição da saída

Na única linha da saída, deve ser impresso o valor mínimo da função otimizada pelo problema proposto.

Exemplo

Entrada:

```
3 7
13 7 4 9 14 2 5
```

Saída:

```
1010
```

8.2 Monge simples

Este exemplo demonstra a aplicação direta dos algoritmos estudados neste trabalho. Recebe uma matriz $A \in \mathbb{Q}^{n \times m}$ Monge convexa e retorna os valores dos mínimos de cada uma das linhas desta matriz. A implementação leva o nome `SimpleMonge`.

O exemplo foi implementado com a otimização da Divisão e Conquista (`DivConq`) consumindo tempo $\mathcal{O}((n+m)\lg(n))$, com o algoritmo SMAWK (`SMAWK`) consumindo tempo $\mathcal{O}(n+m)$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}((n+m)^2)$.

Descrição da entrada

Na primeira linha da entrada são dadas as dimensões n e m da matriz A. Cada uma das próximas n linhas contém m valores separados por espaços. O j-ésimo valor da i-ésima linha é a entrada $A[i][j]$ da matriz.

Descrição da saída

O programa deve imprimir n valores, um em cada linha. A i-ésima linha da saída deve conter o valor de mínimo da i-ésima linha de A.

Exemplo

Entrada:

```
3 4
3 7 10 5
2 5 7 2
9 5 6 0
```

Saída:

```
3
2
0
```

8.3 Internet Trouble simplificado

Este é um exemplo simples da aplicação dos algoritmos em matrizes Monge. Ele resolve o Problema 3.3, dado o vetor $v \in \mathbb{Q}_+^n$ e um inteiro k. A implementação leva o nome `InternetTroubleSimples`.

O exemplo foi implementado com a otimização da Divisão e Conquista (`DivConq`) consumindo tempo $\mathcal{O}(kn \lg(n))$, com o algoritmo SMAWK (`SMAWK`) consumindo tempo $\mathcal{O}(kn)$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(kn^2)$.

Descrição da entrada

Na primeira linha da entrada são dados os valores k e n separados por um espaço. Na próxima linha são dados n valores separados por espaços, que são as entradas do vetor v.

Descrição da saída

O programa deve imprimir um único valor na saída, a resposta do problema proposto.

Exemplo

Entrada:

```
3 7
13 7 4 9 14 2 5
```

Saída:

```
13
```

8.4 Problema online convexo

Resolve o Problema 7.3 no caso convexo, dado um vetor $w \in \mathbb{Q}_+^n$ onde a matriz $C \in \mathbb{Q}^{n \times n}$ é tal que, para todo $i, j \in [n]$, vale que $C[i][j] = (\sum_{k=1}^{j} w_k - \sum_{k=1}^{i-1} w_i)^2$. A implementação leva o nome `ProblemaOnlineConvexo`.

O exemplo foi implementado com o envelope convexo (`EnvelopeConvexo`) consumindo tempo $\mathcal{O}(n \lg(n))$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(n^2)$.

Descrição da entrada

Na primeira linha da entrada é dado o inteiro n. Na próxima linha são dados n valores separados por espaços. O i-ésimo deles (a partir do 1) é a entrada w_i.

Descrição da saída

A saída consiste de n valores em uma linha, o i-ésimo valor (à partir do 1) deve ser o valor correto de $E[i]$, conforme a descrição do problema.

Exemplo

Entrada:

```
7
13 7 4 9 14 2 5
```

Saída:

```
1315.000000 1315.000000 915.000000 794.000000 625.000000 256.000000 0.000000
```

8.5 Problema online côncavo

Resolve o Problema 7.3 no caso côncavo, dado um vetor $w \in \mathbb{Q}_+^n$ onde a matriz $C \in \mathbb{Q}^{n \times n}$ é tal que, para todo $i, j \in [n]$, vale que $C[i][j] = -(\sum_{k=1}^{j} w_k - \sum_{k=1}^{i-1} w_i)^2$. A implementação leva o nome `ProblemaOnlineConcavo`.

O exemplo foi implementado com o envelope côncavo (`EnvelopeConcavo`) consumindo tempo $\mathcal{O}(n \lg(n))$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(n^2)$.

Descrição da entrada

Na primeira linha da entrada é dado o inteiro n. Na próxima linha são dados n valores separados por espaços. O i-ésimo deles (a partir do 1) é a entrada w_i.

Descrição da saída

A saída consiste de n valores em uma linha, o i-ésimo valor (à partir do 1) deve ser o valor correto de $E[i]$, conforme a descrição do problema.

Exemplo

Entrada:

```
7
13 7 4 9 14 2 5
```

Saída:

```
-2634.000000 -2465.000000 -1412.000000 -985.000000 -785.000000 -256.000000 0.000000
```

8.6 Vértices mais distantes num polígono convexo

Resolve o Problema 4.6, dado um inteiro n e um polígono convexo p em sentido horário. A implementação leva o nome `VerticesMaisDistantes`.

O exemplo foi implementado com o algoritmo SMAWK (`SMAWK`) consumindo tempo $\mathcal{O}(n)$, com a otimização da divisão e conquista (`DivConq`) consumindo tempo $\mathcal{O}(n \lg(n))$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(n^2)$.

Descrição da entrada

Na primeira linha da entrada é dado o inteiro n. Nas n linhas seguintes é dado o polígono p. Na i-ésima destas linhas são dadas as coordenadas x e y, separadas por um espaço, do vértices p_i. Os vértices são dados em sentido horário.

Descrição da saída

A saída consiste de n valores em uma linha, o i-ésimo valor (à partir do 1) deve o quadrado da distância euclideana entre o vértice p_i e o vértice de p mais distante deste.

Exemplo

Entrada:

```
5
0 0
1 1
4 1
5 -1
3 -2
```

Saída:

```
26
20
17
26
13
```

8.7 NKLEAVES

Resolve o problema NKLEAVES disponível em http://www.spoj.com/problems/NKLEAVES/.

O exemplo foi implementado com o envelope convexo (`EnvelopeConvexo`) consumindo tempo $\mathcal{O}(kn \lg(n))$, com o envelope convexo linear (`EnvelopeConvexoLinear`) consumindo tempo $\mathcal{O}(kn)$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(kn^2)$. É importante observar que as implementações de envelope utilizam o formato `double` que não tem precisão suficiente para este problema. Para testar as implementações do exemplo nos casos disponíveis online é necessário adaptar as estruturas para usar um formato mais apropriado, como o `long long int`.

8.8 BRKSTRNG

Resolve o problema BRKSTRNG disponível em http://www.spoj.com/problems/BRKSTRNG/ e discutido na Seção 5.4.

O exemplo foi implementado com a otimização de Knuth-Yao (`KnuthYao`) consumindo tempo $\mathcal{O}(n^2)$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(n^3)$.

8.9 Fundraising

Resolve o problema "Fundraising" da Final Brasileira da Maratona de Programação de 2017. Informações sobre a prova podem ser encontradas em http://maratona.ime.usp.br/resultados17.

O exemplo foi implementado com a Fronteira de Pareto (`Pareto`) consumindo tempo $\mathcal{O}(n \lg(n))$ e da maneira trivial (`Trivial`) consumindo tempo $\mathcal{O}(n^2)$. É importante observar que a implementação da Fronteira de Pareto usa o formato `double` que não tem precisão suficiente para este problema. Para testar a implementação do exemplo nos casos utilizados oficialmente é necessário adaptar a estrutura para usar um formato mais apropriado, como o `long long int`.

Capítulo 9

Parte subjetiva

Estou feliz com o resultado deste trabalho. Trabalhei bastante durante todo o ano e acredito que, principalmente por isso, aprendi muito com ele. Aprendi muito sobre os algoritmos e objetos estudados. Aprendi também sobre como implementar, testar e explicar os tópicos com o máximo de cuidado que consegui.

Eu conhecia, antes de começar o trabalho, alguns dos algoritmos sobre os quais eu escrevi. Deixar eles no papel de uma forma que conversasse com o restante do trabalho, colocando a minha visão sobre eles no texto, foi extremamente trabalhoso. A toda leitura de cada parte do texto, surgiam novas formas de explicar ou novas observações sobre as ideias apresentadas. Aprendi que, em algum momento, é necessário aceitar que não tem como ficar perfeito. Isso é um pouco frustrante, mas me mostra o quanto eu realmente aprendi até mesmo sobre os algoritmos que eu já conhecia e já era capaz de implementar.

Cada texto da literatura estudada apresenta um algoritmo de uma certa maneira, eu decidi apresentar todos os algoritmos em termos de encontrar mínimos das linhas das matrizes. Isso me forçou a repensar inteiramente o algoritmo e o texto, verificando todos os passos e criando testes para ter certeza de que minhas adaptações estavam corretas. Este esforço me fez entender muito bem os tópicos, de forma que esse entendimento me ajuda a visualizar e aplicar os conhecimentos obtidos aqui em problemas que encontro durante competições de programação.

Eu tenho que agradecer à Maratona de Programação. Com isso eu me refiro à comunidade de programação competitiva que é forte tanto no Brasil quanto fora. Esta comunidade faz com que a importância das competições se reafirme, me forçando a continuar crescendo e estudando a fim de conseguir resultados cada vez melhores. Agradeço em especial ao MaratonIME, grupo do qual eu tenho muito orgulho de fazer parte e que espero que continue forte como hoje.

Agradeço também aos meus professores do Instituto de Matemática e Estatística, em especial aos ótimos professores de teoria do departamento de computação. A grande preocupação que estes tem com seus alunos me influenciaram a me interessar tanto por teoria da computação, o que é essencial para que eu me mantenha motivado a aprender e estudar. Além disso, é claro, agradeço ao apoio incessante dos meus amigos, em especial da minha família e da Olivia, que sempre se mostraram orgulhosos e me inspiraram a fazer o esforço que venho fazendo.

Referências Bibliográficas

[1] Dynamic programming optimizations. http://codeforces.com/blog/entry/8219, Maio de 2017.

[2] What is divide and conquer optimization in dynamic programming? https://www.quora.com/What-is-divide-and-conquer-optimization-in-dynamic-programming, Maio de 2017.

[3] Alok Aggarwal, Maria M Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2(1-4):195–208, 1987.

[4] Wolfgang Bein, Mordecai J. Golin, Lawrence L. Larmore, and Yan Zhang. The Knuth-Yao quadrangle-inequality speedup is a consequence of total monotonicity. *ACM Transactions on Algorithms (TALG)*, 6(1):17, 2009.

[5] Peter Brucker. Efficient algorithms for some path partitioning problems. *Discrete Applied Mathematics*, 62(1-3):77–85, 1995.

[6] Rainer E. Burkard, Bettina Klinz, and Rüdiger Rudolf. Perspectives of Monge properties in optimization. *Discrete Applied Mathematics*, 70(2):95–161, 1996.

[7] Thomas H. Cormen. *Introduction to Algorithms*. MIT press, 3rd edition, 2009.

[8] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. *Computational Geometry: Introduction*. Springer, 2008.

[9] Michael L. Fredman. On computing the length of longest increasing subsequences. *Discrete Mathematics*, 11(1):29–35, 1975.

[10] Zvi Galil and Raffaele Giancarlo. Speeding up dynamic programming with applications to molecular biology. *Theoretical Computer Science*, 64(1):107–118, 1989.

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

[12] Alan J. Hoffman. On simple linear programming problems. pages 317–327, 2003.

[13] Maria M. Klawe and Daniel J. Kleitman. An almost linear time algorithm for generalized matrix searching. *SIAM Journal on Discrete Mathematics*, 3(1):81–97, 1990.

[14] Donald E. Knuth. Optimum binary search trees. *Acta Informatica*, 1(1):14–25, 1971.

[15] Gaspard Monge. Mémoire sur la théorie des déblais et des remblais. *Histoire de l'Académie Royale des Sciences de Paris*, 1781.

[16] James Kimbrough Park. *The Monge array–an abstraction and its applications.* PhD thesis, Massachusetts Institute of Technology, 1991.

[17] F. Frances Yao. Efficient dynamic programming using quadrangle inequalities. In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*, pages 429–435. ACM, 1980.

[18] F. Frances Yao. Speed-up in dynamic programming. *SIAM Journal on Algebraic Discrete Methods*, 3(4):532–540, 1982.

[19] Étienne Ghys. Gaspard Monge. *Images des Mathématiques*, 2012.