Apostila Introdutória de Algoritmos

Celina M. H. de Figueiredo
Guilherme D. da Fonseca

Projeto financiado em parte pela FAPERJ em 2003

Conteúdo

Capítulo 1. Introdução 3
1.1. Os Problemas 3
1.2. Algoritmos e Paradigmas 4
1.3. Provas de Corretude 6
1.4. Complexidade de Tempo 8
1.5. Complexidade de Tempo de Pior Caso 9
1.6. Complexidade Assintótica 10
1.7. Análise de Complexidade 11
1.8. Resumo e Observações Finais 13
Exercícios 13

Capítulo 2. Estruturas de Dados 15
2.1. Estruturas Elementares 15
2.2. Grafos e Árvores 16
2.3. Subdivisões do Plano e Poliedros 18
2.4. Lista de Prioridades - Heap Binário 19
2.5. Árvores Binárias de Busca 23
2.6. Resumo e Observações Finais 26
Exercícios 26

Capítulo 3. Busca Binária 28
3.1. Busca em vetor 28
3.2. Busca em vetor ciclicamente ordenado 29
3.3. Ponto extremo de polígono convexo 30
3.4. Função de vetor 32
3.5. Resumo e Observações Finais 34
Exercícios 34

Capítulo 4. Método Guloso 37
4.1. Fecho convexo: Algoritmo de Jarvis 37
4.2. Árvore geradora mínima: Algoritmo de Prim 38
4.3. Compactação de dados: Árvores de Huffman 41
4.4. Compactação de dados: LZSS 45
4.5. Resumo e Observações Finais 47
Exercícios 48

Capítulo 5. Divisão e Conquista 50
5.1. Envelope Superior 50
5.2. Par de Pontos Mais Próximos 52
5.3. Conjunto Independente de Peso Máximo em Árvores 54
5.4. Multiplicação de Matrizes: Algoritmo de Strassen 55
5.5. Resumo e Observações Finais 57
Exercícios 58

Capítulo 6. Programação Dinâmica 60
6.1. Ordem de Multiplicação de Matrizes 60

6.2. Todos os caminhos mais curtos 61
6.3. Resumo e Observações Finais 63
Exercícios 63

Capítulo 7. Simplificação 65
7.1. Centro de Árvore 65
7.2. Seleção do k-ésimo 66
7.3. Ponte do Fecho Convexo 69
7.4. Resumo e Observações Finais 70
Exercícios 71

Capítulo 8. Construção Incremental 73
8.1. Arranjo de Retas 73
8.2. Fecho Convexo: Algoritmo de Graham 75
8.3. Programação Linear com Duas Variáveis 77
8.4. Resumo e Observações Finais 80
Exercícios 80

Capítulo 9. Refinamento de Solução 83
9.1. Fluxo em Redes 83
9.2. Resumo e Observações Finais 87
Exercícios 87

Capítulo 10. Problemas NP-Completos 89
10.1. Tempo Polinomial no Tamanho da Entrada 89
10.2. Problemas de Decisão e Reduções 90
10.3. Certificados Polinomiais e a Classe NP 91
10.4. Os Problemas NP-Completos 92
10.5. Satisfabilidade 94
10.6. Clique e Conjunto Independente 95
10.7. Resumo e Observações Finais 97
Exercícios 97

Índice 99

CAPíTULO 1

Introdução

Segundo o dicionário Aurélio, um algoritmo é um "*processo de cálculo, ou de resolução de um grupo de problemas semelhantes, em que se estipulam, com generalidade e sem restrições, regras formais para obtenção do resultado ou da solução do problema*". Embora os algoritmos não sejam necessariamente executados por computadores, este é o tipo de algoritmo que trataremos neste livro. O propósito deste livro é que o leitor não só conheça e entenda diversos algoritmos para problemas variados, como também que seja capaz de desenvolver por si próprio algoritmos eficientes.

As sessões deste livro, em sua maioria, explicam cinco itens:

- Problema: a explicação de que problema está sendo resolvido na sessão.
- Algoritmo: o método computacional para a resolução do problema.
- Prova de corretude: a argumentação de que o algoritmo apresentado resolve corretamente o problema.
- Complexidade: o tempo que o algoritmo leva para resolver o problema.
- Análise de complexidade: o cálculo deste tempo.

Não necessariamente os itens são explicados nesta ordem, ou de modo completamente separado. Muitas vezes, a prova de corretude é apresentada junto com a explicação do algoritmo, justificando o modo como ele é desenvolvido e facilitando seu entendimento.

Nesta introdução, falamos destes cinco itens, fornecendo a base necessária para o entendimento dos demais capítulos do livro.

1.1. Os Problemas

Problemas precisam ser resolvidos constantemente, em todas as áreas do conhecimento humano. Muitos problemas, principalmente de áreas sociais, humanas ou artísticas, não podem ser resolvidos por um computador. Porém, a maioria dos problemas das áreas chamadas de ciências exatas podem ser resolvidos de modo mais eficaz com o auxílio dos computadores. Este livro visa fornecer conhecimentos necessários para programar um computador de modo a resolver problemas não triviais eficientemente. Antes disso, devemos formalizar o que é um problema.

Todo o problema tem uma entrada, também chamada de instância. Nos problemas que estudamos, existem infinitas entradas possíveis. A entrada pode ser bastante simples como no problema cuja entrada é um número inteiro e desejamos descobrir se ele é primo. Em outros problemas, a entrada pode ser bastante complexa, tendo vários elementos relacionados, como grafos, vértices especiais dos grafos, particionamentos dos vértices etc.

Além da entrada, todo problema tem uma saída correspondente, que é a resposta do problema. Os algoritmos devem ser capazes de manipular a entrada para obter a saída.

O tipo de problema mais elementar é o chamado problema de decisão. Neste tipo de problema, formula-se uma pergunta cuja resposta é sim ou não. Vejamos alguns exemplos de problemas de decisão:

- Dado um número inteiro, dizer se este número é primo.
- Dado um conjunto, dizer se um elemento x pertence a este conjunto.
- Dado um conjunto de segmentos no plano, dizer se dois segmentos se interceptam.
- Dado um grafo, dizer se o grafo possui ciclos.

Embora a resposta para um problema de decisão seja sim ou não, é natural formular a chamada versão de construção de alguns desses problemas. Em um problema de construção,

não se deseja apenas saber se uma estrutura existe ou não, mas construir a estrutura que satisfaça algumas propriedades. As versões de construção dos dois últimos problemas de decisão apresentados é:

- Dado um conjunto de segmentos no plano, encontrar dois segmentos que se interceptam, se existirem.
- Dado um grafo, exibir um ciclo deste grafo, se existir.

Em outros problemas de construção, não há uma versão de decisão relacionada. Nos exemplos abaixo, não há dúvida que a estrutura exista, a única dificuldade é exibí-la:

- Dados dois números inteiros, calcular seu produto.
- Dado um conjunto de números reais, ordenar seus elementos.
- Dado um conjunto de pontos não colineares no plano, encontrar 3 pontos que formem um triângulo sem nenhum outro ponto em seu interior.
- Dada uma árvore, encontrar seu centro.

Um tipo especial de problema de construção é chamado de problema de otimização. Nestes problemas, não queremos construir uma solução qualquer, mas sim aquela que maximize ou minimize algum parâmetro. Vejamos alguns exemplos:

- Dados dois números inteiros, calcular seu maior divisor comum.
- Dado um conjunto de números reais, encontrar o menor.
- Dado um conjunto de pontos não colineares no plano, encontrar os 3 pontos que formem um triângulo sem nenhum outro ponto em seu interior que tenha perímetro mínimo.
- Dado um grafo, encontrar sua árvore geradora mínima.

A diferença entre esses problemas e os problemas de construção é sutil, e nem sempre precisamente definida. Por exemplo, o problema de construção onde se deseja encontrar o centro de uma árvore é um problema de otimização, pois o centro de uma árvore é o conjunto dos vértices cuja distância ao vértice mais distante é mínima. Ainda assim, é útil diferenciar estes tipos básicos de problemas, pois algumas técnicas que apresentaremos, se mostram especialmente eficientes para determinado tipo de problema.

Existem outros tipos de problemas que não resolveremos neste livro. Os problemas de enumeração são um exemplo. Nestes problemas deseja-se listar todas as estruturas que satisfazem uma propriedade. Associado a todo o problema de enumeração, existe um problema de contagem. No problema de contagem, não se está interessado em listar todas as soluções, mas apenas descobrir quantas soluções distintas existem. Alguns exemplos destes dois tipos de problema são:

- Dado um número inteiro, listar todos os seus fatores (primos ou não).
- Dado um conjunto, contar o número de sub-conjuntos com determinado número de elementos.
- Dado um conjunto de segmentos no plano, calcular o número de interseções entre os segmentos.
- Dado um grafo, exibir todos os seus ciclos.

1.2. Algoritmos e Paradigmas

Um algoritmo é uma maneira sistemática de resolver um problema. Algoritmos podem ser usados diretamente por seres humanos para diversas tarefas. Ao fazer uma conta de dividir sem usar calculadora, por exemplo, estamos executando um algoritmo. Porém, os algoritmos ganharam importância muito maior com os computadores. Vários problemas cuja solução era praticamente inviável sem um computador passaram a poder ser resolvidos em poucos segundos. Mas tudo depende de um bom algoritmo para resolver o problema.

Ao recebermos um problema, como fazemos para desenvolver um bom algoritmo para resolvê-lo? Não há resposta simples para esta pergunta. Todo este livro visa preparar o leitor para este desenvolvimento. Sem dúvida, conhecer bons algoritmos para muitos problemas ajuda bastante no desenvolvimento de novos algoritmos. Por isso, praticamente todos os livros sobre o assunto

apresentam vários problemas, junto com suas soluções algoritmicas. Geralmente, os problemas são organizados de acordo com a área do conhecimento a que pertencem (teoria dos grafos, geometria computacional, seqüências, álgebra...). Neste livro fazemos diferente.

Embora não exista uma receita de bolo para projetar um algoritmo, existem algumas técnicas que freqüentemente conduzem a "bons" algoritmos. Este livro está organizado segundo estas técnicas, chamadas de paradigmas. Vejamos, de modo simplificado, dois exemplos de paradigmas: "construção incremental" e "divisão e conquista".

- Construção incremental: Resolve-se o problema para uma entrada com apenas um elemento. A partir daí, acrescenta-se, um a um, novos elementos e atualiza-se a solução.
- Divisão e conquista: Quando a entrada tem apenas um elemento, resolve-se o problema diretamente. Quando é maior, divide-se a entrada em duas entradas de aproximadamente o mesmo tamanho, chamadas sub-problemas. Em seguida, resolvem-se os dois sub-problemas usando o mesmo método e combinam-se as duas soluções em uma solução para o problema maior.

Vamos exemplificar estes dois paradigmas no problema de ordenação:

Problema 1. *Dado um conjunto de números reais, ordene o conjunto do menor para o maior elemento.*

Neste problema, a entrada consiste de um conjunto de números reais e a saída é uma lista desses números, ordenada do menor para o maior. Nos dois paradigmas, precisamos saber resolver o caso em que a entrada possui apenas um elemento. Isto é fácil. Neste caso, a lista ordenada contém apenas o próprio elemento.

No paradigma de construção incremental, precisamos descobrir como acrescentar um novo elemento x em uma lista já ordenada. Para isto, podemos percorrer os elementos a partir do menor até encontrar um elemento que seja maior que x. Então, deslocamos todos os elementos maiores que x de uma posição, e colocamos o elemento x na posição que foi liberada. Este algoritmo é chamado de ordenação por inserção.

No paradigma de divisão e conquista, precisamos descobrir como combinar duas listas ordenadas L_1 e L_2 em uma única lista L. Podemos começar comparando o menor elemento de L_1 com o menor elemento de L_2. O menor elemento dentre esses dois é certamente o menor elemento de L. Colocamos então este elemento na lista L e removemos o elemento de sua lista de origem, L_1 ou L_2. Seguimos sempre comparando apenas o menor elemento de L_1 com o menor elemento de L_2 e colocando o menor elemento dentre esses dois no final da lista L, até que uma das listas L_1 ou L_2 se torne vazia. Quando uma das listas se tornar vazia, a outra lista é copiada integralmente para o final da lista L. Este algoritmo é chamado de *mergesort.*

Às vezes, explicar um algoritmo em parágrafos de texto pode ser confuso. Por isto, normalmente apresentamos também o chamado pseudo-código do algoritmo. Este pseudo-código é uma maneira estruturada de descrever o algoritmo e, de certa forma, se parece com sua implementação em uma linguagem de programação. O pseudo-código do algoritmo de ordenação por inserção está na figura 1.1. Há várias maneiras de escrever o pseudo-código para um mesmo algoritmo. Vejamos dois pseudo códigos diferentes para o algoritmo de divisão e conquista que acabamos de apresentar, escritos nas figuras 1.2 e 1.3.

O primeiro pseudo-código (figura 1.2) é mais curto e muito mais fácil de entender que o segundo (figura 1.3). Por outro lado, o segundo pseudo-código se parece mais com uma implementação real do algoritmo. Mas note que, mesmo o segundo pseudo-código ainda é bastante diferente de uma implementação real. Afinal, não nos preocupamos em definir os tipos de variáveis ou fazer as alocações de memória. Neste livro, quase sempre optaremos por um pseudo-código no estilo do primeiro, pois consideramos o entendimento do algoritmo mais importante que um pseudo-código "pronto para implementar". Embora a implementação do primeiro pseudo-código não seja imediata, qualquer bom programador deve ser capaz de compreendê-lo e implementá-lo em um tempo relativamente pequeno.

Entrada:
S: Conjunto de números reais a serem ordenados armazenado em um vetor.
Saída:
L: Conjunto S ordenado do menor para o maior.

```
Ordenar(S)
    Para i de 1 até |S|
        x ← S[i]
        j ← 1
        Enquanto j < i e L[j] < x
            j ← j + 1
        Para j de j até i
            Troque valores de L[j] e x
    Retorne L
```

FIGURA 1.1. Pseudo-código do algoritmo de ordenação por inserção.

Entrada:
S: Conjunto de números reais a serem ordenados armazenado em um vetor.
Saída:
L: Conjunto S ordenado do menor para o maior.

```
Ordenar(S)
    Se |S| = 1
        Retorne S[1]
    Divida S em S1 e S2 aproximadamente de mesmo tamanho
    L1 ← Ordenar(S1)
    L2 ← Ordenar(S2)
    Enquanto |L1| ≠ 0 e |L2| ≠ 0
        Se L1[1] ⩽ L2[1]
            Coloque L1[1] no final da lista L
            Remova L1[1] de L1
        Senão
            Coloque L2[1] no final da lista L
            Remova L2[1] de L2
    Se |L1| ≠ 0
        Coloque elementos de L1 no final de L, na mesma ordem
    Senão
        Coloque elementos de L2 no final de L, na mesma ordem
    Retorne L
```

FIGURA 1.2. Primeiro pseudo-código do algoritmo *mergesort*.

1.3. Provas de Corretude

Em alguns algoritmos, como os algoritmos de ordenação que acabamos de ver, é bastante claro que o algoritmo resolve corretamente o problema. Porém, em muitos outros, não é tão óbvio que a resposta encontrada realmente está correta. De fato, a diferença entre um algoritmo que funciona corretamente e outro que fornece respostas erradas pode ser bastante sutil. Por isso, é essencial provarmos que o algoritmo funciona corretamente, ou seja, faz aquilo que se propõe a fazer.

Entrada:
S: Conjunto de números reais a serem ordenados armazenado em um vetor.
n: Tamanho de S.
Saída::
L: Conjunto S ordenado do menor para o maior.

```
Ordenar(S, n)
    Se n = 1
        Retorne S
    Para i de 1 até ⌊n/2⌋
        S_1[i] ← S[i]
    Para i de ⌊n/2⌋ + 1 até n
        S_2[i − ⌊n/2⌋] ← S[i]
    L_1 ← Ordenar(S_1, ⌊n/2⌋)
    L_2 ← Ordenar(S_2, ⌈n/2⌉)
    i ← i_1 ← i_2 ← 1
    Enquanto i_1 ⩽ ⌊n/2⌋ e i_2 ⩽ ⌈n/2⌉
        Se L_1[i_1] ⩽ L_2[i_2]
            L[i] ← L_1[i_1]
            i_1 ← i_1 + 1
        Senão
            L[i] ← L_2[i_2]
            i_2 ← i_2 + 1
        i ← i + 1
    Se i_1 ≠ ⌊n/2⌋
        Para i de i até n
            L[i] ← L_1[i_1]
            i_1 ← i_1 + 1
    Senão
        Para i de i até n
            L[i] ← L_2[i_2]
            i_2 ← i_2 + 1
    Retorne L
```

FIGURA 1.3. Segundo pseudo-código do algoritmo *mergesort*.

Um exemplo que demonstra como a diferença entre um algoritmo funcionar e não funcionar pode ser sutil é o problema do troco. Neste problema, deseja-se formar uma quantia x em dinheiro, usando o mínimo de moedas possível. Provar que um algoritmo para este problema está correto significa provar que a quantia fornecida pelo algoritmo é x e que o número de moedas usado é realmente mínimo.

O nosso algoritmo procede da seguinte maneira. Para formarmos a quantia x, pegamos a moeda de valor m máximo dentre as moedas com valores menores ou iguais a x. Esta moeda de valor m é fornecida como parte do troco. Para determinar o restante do troco, subtraimos m de x, e procedemos da mesma maneira.

Vamos examinar este mesmo algoritmo com dois conjuntos diferentes de valores de moedas disponíveis. Estes conjuntos não são considerados parte da entrada do problema, mas sim parte de sua definição. A entrada do problema consiste do valor que desejamos fornecer como troco. Vamos supor, para simplificar nossa argumentação, que existam quantidades ilimitadas de moedas de cada valor disponível.

Digamos que temos moedas com os valores 1, 10, 25 e 50 centavos, e desejamos fornecer um troco no valor de 30 centavos. O nosso algoritmo, fornecerá primeiro uma moeda de 25 centavos e, em seguida, 5 moedas de 1 centavo, totalizando 6 moedas. Claramente, podemos formar esta quantia, com apenas 3 moedas de 10 centavos. Portanto, o algoritmo não está correto para este problema.

Vamos considerar agora outro problema, em que temos apenas moedas de 1, 5, 10 e 50 centavos. Neste caso o algoritmo funciona? Sim. Vejamos a prova:

TEOREMA 1.1. *O algoritmo apresentado acima funciona corretamente.*

DEMONSTRAÇÃO. Claramente a quantia fornecida pelo algoritmo soma x. Precisamos provar que o número de moedas é mínimo. O algoritmo fornece as moedas do troco em ordem, da maior para a menor. Seja $S = (m_1, m_2, \ldots, m_n)$ a seqüência de valores das moedas fornecidas pelo algoritmo. Suponha, para obter um absurdo, que $S' = (m'_1, m'_2, \ldots, m'_{n'})$, com $n' < n$, seja uma seqüência de valores de moedas que some x, ordenada do maior para o menor, que use o mínimo possível de moedas. Seja i o menor valor tal que $m_i \neq m'_i$. Certamente, $m_i > m'_i$, pois m_i, a moeda escolhida pelo algoritmo, é a maior moeda que não excederia a quantia x. Como as seqüências estão ordenadas, vale que $m_i > m'_j$, para j de i até n'. Também é claramente verdade que a soma das moedas de m'_i até $m'_{n'}$ vale pelo menos m_i. Unindo estas informações ao fato de que todas as moedas disponíveis (1, 5, 10 e 50 centavos) são múltiplas das moedas menores, então há um subconjunto não unitário das moedas de m'_i até $m'_{n'}$ que soma exatamente m_i. É possível melhorar a solução S', substituindo este subconjunto por uma moeda de valor m_i, o que contradiz a otimalidade de S'. $\square$

Neste caso, foi possível provar que o algoritmo está correto, porque o valor de toda moeda é um múltiplo dos valores das moedas menores. Isto não acontecia antes, porque a moeda de 25 centavos não é múltipla da moeda de 10 centavos.

Caso tenhamos moedas de 1, 5, 10, 25 e 50 centavos, o algoritmo funciona? Não vale a propriedade que toda moeda é múltipla das menores, porém, ainda assim, o algoritmo funciona corretamente. A condição de toda a moeda ser múltipla das menores é suficiente para o algoritmo funcionar, mas não é necessária. A prova que o algoritmo funciona corretamente neste último caso é mais trabalhosa e fica como exercício.

1.4. Complexidade de Tempo

Como podemos calcular o tempo gasto por um algoritmo resolvendo um determinado problema? Este tempo depende de diversos fatores, como a entrada do problema, a máquina que está executando o programa e de como foi feita a implementação do algoritmo. Por isso, determinar exatamente o tempo gasto por um algoritmo é um processo intrinsecamente experimental. Implementa-se o algoritmo, define-se uma entrada ou conjunto de entradas e executa-se o algoritmo para estas entradas em uma máquina específica, medindo os tempos. Esta abordagem experimental tem vantagens e desvantagens com relação a abordagem teórica que estudamos neste livro. Vamos apresentar primeiro alguns pontos fracos da abordagem experimental.

- Dependência da entrada: O tempo gasto por um algoritmo pode ser extremamente dependente de alguns detalhes sutis da entrada. Há, por exemplo, algoritmos de ordenação bastante eficientes quando a entrada está bem embaralhada, mas que são muito lentos quando a entrada já está quase completamente ordenada. Por outro lado, há algoritmos que são muito rápidos quando a entrada já está quase completamente ordenada, mas que são extremamente ineficientes na maioria dos casos. Muitas vezes, é difícil saber se as entradas escolhidas para o experimento representam bem as entradas com que o algoritmo será de fato usado.
- Dependência da máquina: Este caso é bem menos crítico que o anterior. De um modo geral, se um algoritmo a foi mais rápido que um algoritmo b em uma determinada máquina, o algoritmo a também será mais rápido que o algoritmo b em qualquer outra máquina. Mas há exceções. Por exemplo, uma máquina com operações de ponto

flutuante extremamente rápidas pode se beneficiar de algoritmos que usem fortemente ponto flutuante, enquanto outra máquina pode se beneficiar de algoritmos que façam menos operações de ponto flutuante. Em máquinas com um *cache* de memória pequeno, um algoritmo que acesse os dados com maior localidade pode ser preferível, enquanto em máquinas com um *cache* maior, ou sem nenhum *cache*, outro algoritmo pode ser preferível.

- Dependência da implementação: Digamos que você crie um algoritmo a e resolva escrever um artigo argumentando que seu algoritmo é mais rápido que o algoritmo b. Como criador do algoritmo a, você provavelmente conhece muito bem este algoritmo e é capaz de implementá-lo de modo extremamente eficiente. A sua implementação do algoritmo a será provavelmente muito melhor que a sua implementação do algoritmo b. Deste modo, a comparação é bastante injusta.
- Incomparabilidade: Digamos que alguém apresente o tempo que uma implementação de um determinado algoritmo levou em uma determinada máquina com uma entrada específica e outra pessoa apresente o tempo que outro algoritmo para o mesmo problema levou com outra entrada em outra máquina. É completamente impossível comparar estes dois resultados para determinar qual algoritmo será mais rápido no seu caso.
- Alto custo: Devido a impossibilidade de comparar execuções dos algoritmos com entradas diferentes ou em máquinas diferentes, é necessário implementar e testar diversos algoritmos para determinar qual é mais rápido no seu caso específico. O tempo e o custo dessas tarefas podem ser bastante elevados.

A seguir, vamos introduzir a complexidade de tempo assintótica de pior caso, que usamos para avaliar a eficiência dos algoritmos. Esta análise tem se mostrado extremamente útil por fornecer uma expressão simples que permite comparar facilmente dois algoritmos diferentes para o mesmo problema, independente da máquina, implementação ou da entrada.

1.5. Complexidade de Tempo de Pior Caso

Primeiro vamos explicar como fazemos a análise independer da entrada. Para isto, consideramos sempre a pior entrada possível, ou seja, a que leva mais tempo para ser processada. Como estamos lidando com entradas ilimitadamente grandes, precisamos fixar o tamanho da entrada, ou alguma outra propriedade dela. Por enquanto, não vamos considerar a dependência da máquina ou da implementação. Vamos considerar que estamos falando sempre de uma máquina previamente definida e de uma implementação específica.

Podemos falar, no problema de ordenação, da lista de n elementos que leva mais tempo para ser ordenada por um determinado algoritmo (com relação a todas as listas com n elementos). No problema de, dado um conjunto de n pontos no plano, determinar o par de pontos mais próximos, podemos expressar a complexidade de tempo em função do número n de pontos da entrada. No problema de, dado um conjunto de polígonos, dizer se dois polígonos se interceptam, *não* é razoável expressar a complexidade de tempo em função do número de polígonos da entrada. Afinal, um polígono pode ter qualquer número de vértices. Uma entrada com apenas 2 polígonos pode ser extremamente complexa se estes polígonos tiverem muitos vértices. Já uma entrada com vários triângulos pode ser bem mais simples. Por isso, neste problema, é razoável expressar a complexidade de tempo em função do número total de vértices dos polígonos.

Em todos estes casos, queremos definir uma função $T(n)$ que representa o tempo máximo que o algoritmo pode levar em uma entrada com n elementos. Às vezes, podemos expressar o tempo em função de vários parâmetros da entrada, simultaneamente. Quando a entrada é um grafo, por exemplo, podemos expressar a complexidade de tempo em função do número n de vértices e do número m de arestas do grafo. Assim, desejamos obter uma função $T(n, m)$. Por enquanto, porém, vamos desconsiderar este caso de várias variáveis.

Há outras alternativas para a complexidade de pior caso, mas, na maioria das situações, a complexidade de pior caso é considerada a melhor opção. Uma alternativa é a chamada complexidade de caso médio. Esta opção é motivada pela idéia que, se um algoritmo é rápido

para a esmagadora maioria das entradas, então pode ser aceitável que este algoritmo seja lento para algumas poucas entradas. Há algumas desvantagens da complexidade de caso médio. A primeira delas é que, na complexidade de caso médio, é necessário ter uma distribuição de probabilidade para as entradas. Outra desvantagem é que o cálculo da complexidade de caso médio pode ser extremamente complicado. Não adianta ter uma medida de complexidade que ninguém consegue calcular.

1.6. Complexidade Assintótica

Neste ponto, já definimos que a nossa função $T(n)$ corresponde ao tempo que uma determinada implementação do algoritmo leva em uma determinada máquina para a entrada de tamanho n mais demorada. Vamos agora nos livrar da dependência da máquina específica e dos detalhes de implementação. Para isto, lançamos mão da hierarquia assintótica, que explicamos nos próximos parágrafos.

Dizemos que $f(n) \preccurlyeq g(n)$ se existem constantes positivas c e n_0 tais que $f(n) \leqslant cg(n)$, para todo $n > n_0$. Analogamente, dizemos que $f(n) \succcurlyeq g(n)$ se existem constantes positivas c e n_0 tais que $f(n) \geqslant cg(n)$, para todo $n > n_0$.

Se $f(n) \preccurlyeq g(n)$ e $f(n) \succcurlyeq g(n)$, dizemos que $f(n) \asymp g(n)$. Se $f(n) \preccurlyeq g(n)$, mas não é verdade que $f(n) \asymp g(n)$, então dizemos que $f(n) \prec g(n)$. Analogamente, se $f(n) \succcurlyeq g(n)$, mas não é verdade que $f(n) \asymp g(n)$, então dizemos que $f(n) \succ g(n)$.

Vejamos alguns exemplos com polinômios:

$$
\begin{aligned}
3n^2 + 2n + 5 &\preccurlyeq n^2 \\
3n^2 + 2n + 5 &\asymp n^2 \\
3n^2 + 2n + 5 &\prec n^3 \\
1 \prec n \prec n^2 &\prec n^3 \prec \cdots
\end{aligned}
$$

Com algumas funções mais complexas, podemos escrever, por exemplo:

$$1 \prec \lg\lg n \prec \lg n \prec \lg^2 n \prec n^{1/3} \prec \sqrt{n} \prec n/\lg n \prec n$$

$$n \prec n \lg n \prec n^2 \prec n^3 \prec 2^n \prec e^n \prec n! \prec n^n$$

Esta notação assintótica que acabamos de apresentar, embora correta, é raramente utilizada em computação. No seu lugar, utiliza-se a comumente chamada notação O. Denota-se por $O(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \preccurlyeq g(n)$. Denota-se por $\Omega(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \succcurlyeq g(n)$. Denota-se por $\Theta(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \asymp g(n)$. Denota-se por $o(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \prec g(n)$. Denota-se por $\omega(g(n))$ uma função $f(n)$ qualquer que satisfaça $f(n) \succ g(n)$. Esta equivalência está resumida a seguir:

$$
\begin{array}{rcl}
f(n) = O(g(n)) & \equiv & f(n) \preccurlyeq g(n) \\
f(n) = \Omega(g(n)) & \equiv & f(n) \succcurlyeq g(n) \\
f(n) = \Theta(g(n)) & \equiv & f(n) \asymp g(n) \\
f(n) = o(g(n)) & \equiv & f(n) \prec g(n) \\
f(n) = \omega(g(n)) & \equiv & f(n) \succ g(n)
\end{array}
$$

Esta notação tem alguns aspectos extremamente práticos e outros extremamente confusos. Um ponto forte da notação O é que ela pode ser usada diretamente dentro de equações. Podemos dizer, por exemplo que $2n^4 + 3n^3 + 4n^2 + 5n + 6 = 2n^4 + 3n^3 + O(n^2)$. Um ponto negativo é que a notação O anula a reflexividade da igualdade. Podemos dizer que $n^2 = O(n^3)$, mas *não* podemos dizer que $n^3 = O(n^2)$.

Uma propriedade importante da notação O é que ela despreza constantes aditivas e multiplicativas. Sejam c_1 e c_2 constantes, então $c_1 f(n) + c_2 = \Theta(f(n))$. Desta propriedade seguem algumas simplificações como $\lg n^k = \Theta(\lg n)$ e $\log_k n = \Theta(\lg n)$, para qualquer constante k. Sempre que usamos um logaritmo dentro da notação O, optamos pela função $\lg n$, o logaritmo

de n na base 2. Afinal, como $\log_k n = \Theta(\lg n)$, qualquer logaritmo é equivalente nesse caso e o logaritmo na base 2 é o mais natural em computação.

Agora podemos terminar de definir o método que usamos para medir o tempo gasto por um algoritmo, independente da máquina. Certamente, uma máquina mais rápida está limitada a executar qualquer programa um número de vezes mais rápido que outra máquina. Assim, se expressarmos a função $T(n)$ usando notação O, não é necessário depender de uma máquina específica. Com isto, também não dependemos de muitos detalhes de implementação, embora alguns detalhes de implementação possam alterar a complexidade assintótica. Esta avaliação do algoritmo é chamada de complexidade de tempo assintótica de pior caso, mas muitas vezes nos referimos a ela apenas como complexidade de tempo, ou mesmo complexidade.

Como o próprio nome diz, a complexidade de tempo assintótica avalia o tempo gasto pelo algoritmo para entradas cujo tamanho tende a infinito. Se um algoritmo a tem complexidade de tempo $O(f(n))$ e outro algoritmo b tem complexidade de tempo $O(g(n))$, com $f(n) \prec g(n)$, então, certamente, a partir de algum valor de n o algoritmo a se torna mais rápido que o algoritmo b. Porém, pode ser verdade que o algoritmo a seja mais lento que o algoritmo b para entradas "pequenas".

1.7. Análise de Complexidade

Vamos agora mostrar algumas técnicas usadas para analisar a complexidade de um algoritmo através de dois exemplos simples: os dois algoritmos de ordenação vistos anteriormente. Primeiro vamos analisar a ordenação por inserção, cujo pseudo-código está na figura 1.1.

Temos 3 *loops* neste algoritmo. O *loop* mais externo é repetido exatamente n vezes, onde n é o número de elementos da entrada. O número exato de repetições dos loops mais internos depende da entrada, porém é possivel notar que o primeiro *loop* realiza no máximo $i-1$ repetições e o segundo *loop* realiza no máximo i repetições. De fato, o número de repetições dos dois loops internos somados é exatamente i, mas não precisamos entrar nesse nível de detalhes para obtermos um limite superior para a complexidade. O que importa é que os *loops* internos realizam $O(i)$ repetições e, dentro deles, só há operações cujo tempo independe do valor de n. Assim, a complexidade de tempo do algoritmo é

$$\sum_{i=1}^{n} O(i) = \sum_{i=1}^{n} O(n) = nO(n) = O(n^2).$$

Neste cálculo, substituimos $O(i)$ por $O(n)$, pois $i \leqslant n$. Claro que poderíamos estar perdendo precisão nesta substituição. Se quisermos fazer os cálculos justos, não podemos usar este truque e também precisamos garantir que há caso em que os *loops* internos realizam $\Omega(i)$ repetições, o que é verdade já que os dois loops somados realizam exatamente i repetições para qualquer entrada. Como $1 + 2 + \ldots + n = n(n-1)/2 = \Theta(n^2)$, temos

$$\sum_{i=1}^{n} \Theta(i) = \Theta(n^2).$$

Deste modo, finalizamos a análise do algoritmo de ordenação por inserção. Outra análise que podemos fazer é a chamada complexidade de espaço, ou seja, a quantidade de memória necessária para a execução do algoritmo. No caso da ordenação por inserção, a complexidade de memória é claramente $\Theta(n)$, pois só temos 2 vetores com n elementos, além de um número constante de variáveis cujo tamanho independe de n.

A análise do algoritmo de ordenação por divisão e conquista é mais complicada. Este algoritmo divide a entrada em duas partes aproximadamente iguais, executa-se recursivamente para essas duas partes e depois combina as duas soluções. A fase de combinação das duas soluções leva tempo linear no tamanho da entrada. Com isso, podemos dizer que

$$T(n) = \begin{cases} 2T(n/2) + \Theta(n) & \text{para } n > 1 \\ O(1) & \text{para } n \leqslant 1 \end{cases}$$

Esta é uma relação de recorrência, pois $T(n)$ está expresso em função da própria função $T(\cdot)$. Usamos freqüentemente relações de recorrência para analisar a complexidade de tempo de algoritmos. Quando usamos relações de recorrência para este fim, podemos fazer algumas simplificações. A primeira delas é omitirmos o caso base (no caso, $n = 1$). Para qualquer algoritmo, o tempo que o algoritmo leva para entradas de tamanho constante é constante. Assim, usando notação assintótica, $T(k) = \Theta(1)$ para qualquer *constante* k. Por isso, o caso base $T(k) = \Theta(1)$ é sempre satisfeito e, para simplificarmos, podemos escrever a recorrência acima como:

$$T(n) = 2T(n/2) + \Theta(n).$$

Além disso, como estamos interessados apenas na complexidade assintótica de $T(n)$, podemos alterar livremente as constantes multiplicativas de funções não recorrentes de n, ou seja, podemos substituir, por exemplo, $\Theta(1)$ por 1, ou $n(n-1)/2$ por n^2. Assim, podemos reescrever nossa recorrência como:

$$T(n) = 2T(n/2) + n.$$

Resolver relações de recorrência não é uma tarefa simples, de modo geral. Porém, se temos um chute da resposta, podemos prová-lo ou derrubá-lo usando indução. Para obtermos este chute, vamos imaginar a execução do algoritmo como uma árvore como na figura 1.4. Cada vértice representa uma execução do procedimento e o número indicado nele representa o número de elementos na entrada correspondente. Os dois filhos de um vértice correspondem as duas chamadas recursivas feitas a partir do vértice pai. O tempo gasto pelo algoritmo, conforme a relação de recorrência, é o número de elementos da entrada mais o tempo gasto em duas execuções recorrentes com metade dos elementos. Assim, desejamos obter a soma dos valores representados nos vértices da árvore. A soma dos vértices no último nível da árvore vale $\Theta(n)$, ou seja, o tempo gasto em todas as execuções com um elemento na entrada é $\Theta(n)$. O mesmo é válido para todas as execuções com 2 (ou 4, ou 8...) elementos na entrada, que correspondem a cada um dos níveis da árvore. Como a altura da árvore é $\Theta(\lg n)$, a soma das complexidades de tempo vale $\Theta(n \lg n)$.

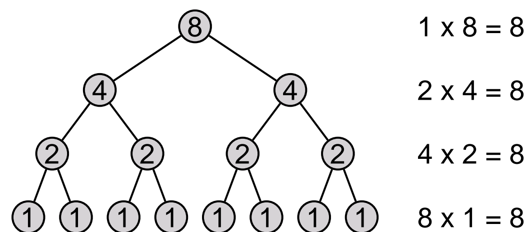


FIGURA 1.4. Árvore correspondente a execução do algoritmo de divisão e conquista em entrada de tamanho inicial 8.

Para provarmos que $T(n) \leqslant cn \lg n$ para alguma constante c, usando indução, fazemos:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leqslant 2cn/2 \lg(n/2) + n \\ &= cn \lg(n/2) + n \\ &= cn \lg n - cn + n \\ &\leqslant cn \lg n. \end{aligned}$$

1.8. Resumo e Observações Finais

Apresentamos três tipos de problemas que estudaremos nesse livro: problemas de decisão, problemas de construção e problemas de otimização. Todo problema possui uma entrada, ou instância, e uma saída desejada para cada entrada.

Um algoritmo é um método computacional para a solução do problema. Um paradigma é uma técnica usada para desenvolver algoritmos.

Quando desenvolvemos um algoritmo, precisamos provar que o algoritmo funciona, isto é, fornece a solução correta para o problema. Isto é chamado de prova de corretude. Algumas provas de corretude são bastante simples, enquanto outras são bastante complicadas.

Para compararmos a eficiencia de algoritmos, precisamos definir o que chamamos de complexidade de tempo, pois uma medição de tempo na prática apresenta várias deficiências. A medida que mais usamos é chamada de complexidade de tempo assintótica de pior caso. O termo pior caso é usado porque sempre nos preocupamos com a entrada de um tamanho definido para a qual o algoritmo leva mais tempo. O termo assintótica é usado porque avaliamos quanto tempo o algoritmo leva para entradas grandes, com tamanho tendendo a infinito. Para expressarmos grandezas assintóticas, definimos a notação O.

Analisar a complexidade de tempo de um algoritmo nem sempre é uma tarefa simples. Muitas vezes, usamos relações de recorrência ou somatórios para esta tarefa.

Exercícios

1.1) Liste três problemas de cada um dos seguintes tipos: decisão, construção e otimização.

1.2) Descreva com pseudo-códigos os algoritmos usados normalmente para fazer adição e multiplicação de inteiros “na mão”. Analise a complexidade de tempo assintótica desses algoritmos, no pior caso, em função do número de algarismos dos dois operandos.

1.3) Realize as seguintes tarefas práticas com os dois algoritmos de ordenação descritos neste capítulo:
   (a) Implemente corretamente os dois algoritmos da maneira mais eficiente que conseguir.
   (b) Compare o tempo que cada um dos algoritmos gasta para ordenar listas aleatoriamente embaralhadas com tamanhos variados.
   (c) Determine o tamanho k de lista para o qual o algoritmo de ordenação por inserção leva o mesmo tempo que o algoritmo de divisão e conquista.
   (d) Modifique o algoritmo de divisão e conquista para, quando a lista possuir tamanho menor ou igual ao valor de k determinado no item anterior, executar o algoritmo de ordenação por inserção.
   (e) Compare o tempo que esse novo algoritmo gasta para entradas de tamanhos variados.

1.4) Preencha a tabela abaixo com os valores de cada função. Em seguida, escreva cada função na forma mais simples usando notação Θ. Finalmente, coloque estas funções em ordem crescente segundo a hierarquia assintótica.

	2	3	5	10	30	100
$7n + \sqrt{n}$						
$2^n/100$						
$n/\lg n$						
$\lg n^3$						
$2n^2$						
$n! - n^3$						
$(\lg \lg n)^2$						
$\lg n + \sqrt{n}$						
$\lg(n!)$						

1.5) Considere a recorrência

$$T(n) = T(n/2) + 1.$$

A solução correta desta recorrência satisfaz $T(n) = \Theta(\lg n)$. Ache o erro na demostração abaixo, que prova que $T(n) = O(\lg \lg n)$:

Vamos supor, para obter uma prova por indução, que $T(i) = O(\lg \lg i)$ para $i \leqslant n$. Vamos calcular $T(n+1)$. Temos: $T(n+1) = T(n/2) + 1 = O(\lg \lg(n/2)) + 1$. Como $\lg \lg(n/2) = O(\lg \lg(n+1))$ temos $T(n+1) = O(\lg \lg(n+1)) + 1 = O(\lg \lg(n+1))$, finalizando a indução.

1.6) Prove que a recorrência $T(n) = T(n/2) + 1$ satisfaz $T(n) = O(\lg n)$.

*1.7) Prove que a recorrência abaixo satisfaz $f(n) = n$, considerando o caso base $f(1) = 1$:

$$f(n) = \sum_{i=0}^{n-2} \binom{n-2}{i} \frac{1}{2^{n-3}} f(i+1).$$

CAPíTULO 2

Estruturas de Dados

Este capítulo não visa introduzir o leitor ao tópico de estruturas de dados, mas apenas revisar este tópico, estabelecer a notação usada nos demais capítulos e servir como referência sucinta. Recomendamos a quem não tiver estudado o assunto que consulte um livro específico. Uma estrutura de dados é normalmente vista como uma caixa preta capaz de realizar um conjunto de operações, que incluem o armazenamento de dados. Neste capítulo, examinamos o que acontece dentro dessas caixas pretas, analisando a complexidade de tempo das operações.

2.1. Estruturas Elementares

A estrutura de dados mais elementar é uma variável. Variáveis podem ser de diversos tipos básicos, como:

- *booleana ou binária*: Armazena apenas dois valores, como 0 ou 1, ou possivelmente *verdadeiro* ou *falso*.
- *caractere*: Armazena uma letra ou símbolo.
- *inteira*: Armazena um número inteiro.
- *real*: Armazena um número real.
- *ponteiro*: Aponta para uma posição da memória da máquina.

Há outros tipos básicos de variáveis como, por exemplo, uma variável que só armazene inteiros positivos. Além disso, em uma máquina real, uma variável inteira está limitada a um intervalo dos números inteiros, possuindo valores minímo e máximo armazenáveis. Geralmente, ao longo deste livro, consideramos a capacidade de armazenamento de variáveis inteiras ilimitada. Também consideramos que variáveis reais realmente armazenam um número real, e não um arredondamento com ponto flutuante como acontece na prática.

A combinação de um conjunto de variáveis é chamada de *estrutura*. Uma estrutura para pontos no plano pode conter duas variáveis reais, uma para armazenar a coordenada x e outra para armazenar a coordenada y do ponto. Nos referimos a estes atributos de um ponto p como $p.x$ e $p.y$, respectivamente.

Uma seqüência de variáveis de um mesmo tipo, ocupando posições sucessivas da memória, é chamada de *vetor*. Os elementos de um vetor são referenciados através de um índice inteiro entre colchetes. O primeiro elemento de um vetor v é referenciado como $v[1]$, e assim por diante. Um vetor possui uma capacidade associada a ele, que representa o número máximo de elementos que o vetor pode armazenar, ou seja, o maior valor de n para o qual $v[n]$ é uma posição válida.

Freqüentemente, falamos em vetores cíclicos. Em um vetor cíclico com capacidade n, quando ocorre um acesso a posição $v[i]$ com $i < 1$ ou $i > n$, este acesso é convertido a um acesso no intervalo válido por meio de adições ou subtrações do valor n. Por exemplo, em um vetor com capacidade 5, é equivalente falarmos em $v[2]$, $v[7]$, $v[22]$ ou $v[-3]$. Vetores cíclicos podem ser implementados usando a operação de resto da divisão, por isso, são também chamados de vetores com índice módulo n.

A utilização mais freqüente de vetores é para armazenar listas. Uma lista é um conjunto de elementos listados em determinada ordem. Embora os elementos de uma lista, sempre possuam uma ordem associada a eles, não necessariamente esta ordem possui um significado. Por exemplo, o vetor $v = (5, 1, 3, 9, 7)$ é uma representação válida para o conjunto dos 5 primeiros números ímpares. Também é possível forçarmos os elementos do vetor a estar armazenados segundo uma ordem definida. O vetor ordenado crescentemente que armazena os 5 primeiros números ímpares é $v = (1, 3, 5, 7, 9)$.

Quando vetores são usados como listas, nos referimos ao número de elementos armazenados no vetor v como $|v|$. O parâmetro $|v|$ pode ser armazenado pelo programa como uma variável inteira separada ou ser definido implicitamente através de um símbolo especial para marcar o final do vetor. Nos parágrafos a seguir, nos concentraremos na primeira alternativa.

Vejamos a complexidade de tempo de algumas operações com listas armazenadas em vetor. Para inserirmos um elemento no final da lista, basta fazermos $|v| \leftarrow |v|+1$ e $v[|v|] \leftarrow x$, onde x é o novo elemento. Portanto, essa operação leva tempo $\Theta(1)$. Para removermos o último elemento da lista, basta fazermos $|v| \leftarrow |v|-1$, também levando tempo $\Theta(1)$. Para buscarmos um elemento podemos precisar percorrer a lista inteira, portanto a busca de um elemento leva no pior caso tempo $\Theta(|v|)$. Para removermos um elemento qualquer da lista, é necessário deslocarmos todos os elementos seguintes, levando tempo $\Theta(|v|)$. Para inserirmos um elemento em uma posição específica da lista, a situação é equivalente, levando tempo $\Theta(|v|)$.

Existem dois tipos especiais de listas, que são freqüentemente armazenados em vetores: pilhas e filas. Pilhas e filas possuem apenas duas operações básicas, inserir e remover. A operação de remoção, além de remover o elemento, retorna seu valor. Uma pilha é uma lista onde os elementos são sempre inseridos e removidos no final da lista, chamado de topo da pilha. Uma fila é uma lista onde os elementos são inseridos no final da lista, chamado de fim da fila, e removidos do início da lista, chamado de início da fila.

Em uma pilha armazenada em um vetor v, inserir(v, x) corresponde a $|v| \leftarrow |v| + 1$ e $v[|v|] \leftarrow x$. A função remover(v) corresponde a $|v| \leftarrow |v| - 1$ e retorne $v[|v| + 1]$.

Para armazenarmos uma fila em um vetor precisamos utilizar um vetor cíclico. Guardamos dois índices módulo n, um para indicar o início e outro para indicar o final da fila. Para inserir um elemento na fila, coloca-se este elemento no final, incrementando o índice correspondente. Para remover um elemento, basta incrementar o índice correspondente ao início da fila.

Outra maneira de armazenar listas é usando listas encadeadas. Em uma lista encadeada, cada elemento aponta para o elemento seguinte na lista. Deste modo, é possível realizar operações de inserir e remover em qualquer posição da lista em tempo $\Theta(1)$. Outra vantagem das listas encadeadas é que não é necessário definir previamente uma capacidade para a lista, como acontecia no vetor. Porém, as listas encadeadas possuem algumas desvantagens. Uma delas é que as constantes multiplicativas da complexidade de tempo ocultas pela notação O são maiores que nos vetores. Outra desvantagem é que não é possível acessar em tempo $\Theta(1)$ qualquer elemento da lista, como acontecia no vetor. Com isto, não é possível realizar os métodos de busca binária que serão vistos no capítulo 3.

2.2. Grafos e Árvores

Um grafo é uma estrutura combinatória extremamente útil para a modelagem de diversos problemas. Um grafo G é definido como dois conjuntos, $V(G)$ e $E(G)$. Os elementos do conjunto $V(G)$ são chamados de vértices do grafo. Os elementos do conjunto $E(G)$ são pares não ordenados de vértices de $V(G)$, sendo chamados de arestas. Grafos são muito mais fáceis de visualisar quando representados graficamente. Por exemplo, o grafo com $V(G) = \{a, b, c, d, e\}$ e $E(G) = \{(a, b), (a, c), (a, e), (b, d), (c, e), (d, e)\}$ está representado na figura 2.1(a). Há outras maneiras de representar este mesmo grafo, como mostra a figura 2.1(b).

Outra estrutura útil é chamada de grafo direcionado, ou digrafo (pronuncia-se di-GRA-fo, pois não há acento como na palavra dígrafo). Em um grafo direcionado, o conjunto de arestas é formado por pares ordenados. Deste modo, as arestas possuem direção. Quando representamos um digrafo graficamente, desenhamos as arestas como setas, como mostra a figura 2.1(c).

Há duas maneiras muito usadas para representar um grafo ou digrafo no computador. A primeira delas é chamada de matriz de adjacências. A matriz de adjacências de um grafo G com n vértices é uma matriz M binária $n \times n$ onde $m_{i,j} = 1$ se $(v_i, v_j) \in E(G)$ e $m_{i,j} = 0$ caso contrário. A matriz de adjacências dos grafo G com $V(G) = \{a, b, c, d, e\}$ e $E(G) = \{(a, b), (a, c), (a, e), (b, d), (c, e), (d, e)\}$ é:

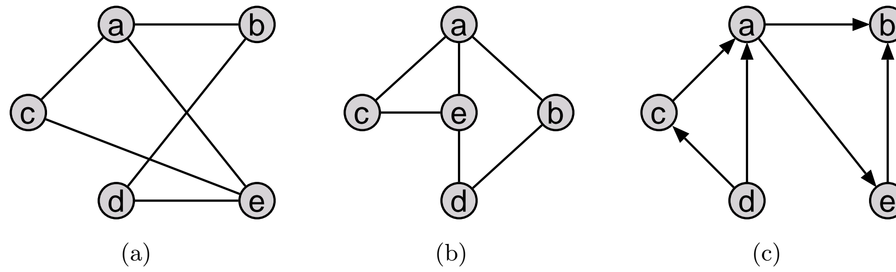


FIGURA 2.1. (a) Grafo G com $V(G) = \{a, b, c, d, e\}$ e $E(G) = \{(a, b), (a, c), (a, e), (b, d), (c, e), (d, e)\}$. (b) Outra representação para o grafo da figura anterior. (c) Grafo direcionado G com $V(G) = \{a, b, c, d, e\}$ e $E(G) = \{(a, b), (a, e), (c, a), (d, c), (d, a), (e, b)\}$

	a	b	c	d	e
a	0	1	1	0	1
b	1	0	0	1	0
c	1	0	0	0	1
d	0	1	0	0	1
e	1	0	1	1	0

Outra alternativa é armazenarmos, para cada vértice, uma lista contendo os vértices adjacentes a ele, chamada lista de adjacências. Esta alternativa apresenta algumas vantagens em relação a matriz de adjacências. A primeira delas é que a complexidade de espaço para o armazenamento de um grafo com n vértices e m arestas é $\Theta(n^2)$ na matriz de adjacências contra $\Theta(n + m)$ nas listas de adjacências. Outra vantagem é que, para listarmos todos os vizinhos de um vértice usando matriz de adjacências, levamos tempo $\Theta(n)$, enquanto usando listas de adjacências, levamos tempo proporcional ao número de vizinhos. A matriz de adjacências também possui vantagens. Podemos verificar se uma aresta pertence ao grafo em tempo $\Theta(1)$ usando matriz de adjacências contra $\Theta(n)$, no pior caso, usando listas de adjacências. Às vezes, pode ser útil manter simultaneamente as duas representações do mesmo grafo.

Chamamos de caminho em um grafo G uma seqüência de vértices distintos $(v_1, v_2, \ldots, v_k)$ tal que $(v_i, v_{i+1}) \in E(G)$, para $1 \leqslant i < k$. Em grafos direcionados, podemos falar em caminhos direcionados e caminhos não direcionados. Um caminho direcionado em um digrafo G é uma seqüência de vértices distintos $(v_1, v_2, \ldots, v_k)$ tal que $(v_i, v_{i+1}) \in E(G)$, para $1 \leqslant i < k$. Um caminho *não* direcionado em um digrafo G é uma seqüência de vértices distintos $(v_1, v_2, \ldots, v_k)$ tal que ou $(v_i, v_{i+1}) \in E(G)$, ou $(v_{i+1}, v_i) \in E(G)$, para $1 \leqslant i < k$.

O comprimento de um caminho é o número de arestas na seqüência correspondente ao caminho, ou seja o número de vértices da seqüência menos uma unidade. A distância entre dois vértices u e v é o comprimento do caminho de menor comprimento iniciado em u e terminado em v.

Um grafo que possui caminho entre qualquer par de vértices é chamado de conexo. Na maioria dos casos, tratamos apenas de grafos conexos. Em um grafo conexo, $m \geqslant n - 1$, portanto $O(n) = O(m)$. Um digrafo que possui caminhos direcionados entre todo par de vértices é chamado de fortemente conexo, enquanto um digrafo que possui caminhos não direcionados entre todo par de vértices é chamado de fracamente conexo.

Chamamos de ciclo em um grafo ou digrafo G uma seqüência de vértices distintos $(v_1, v_2, \ldots, v_k)$ tal que $(v_i, v_{i+1}) \in E(G)$, para $1 \leqslant i < k$ e $(v_k, v_1) \in E(G)$, ou seja, um caminho fechado.

Um grafo conexo que não possui ciclos é chamado de árvore, ou árvore livre. Um grafo não necessariamente conexo que não possui ciclos é chamado de floresta.

Em uma árvore com n vértices, o número de arestas $m = n - 1$. Em uma floresta, o número de arestas $m \leqslant n - 1$.

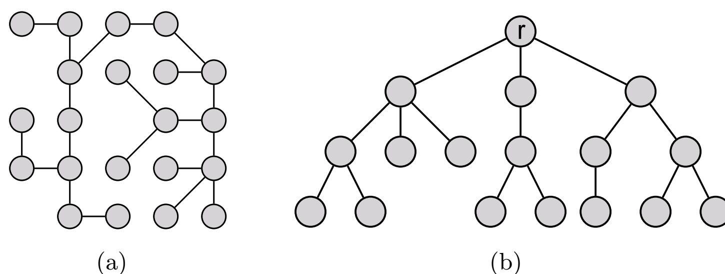


FIGURA 2.2. (a) Árvore livre. (b) Árvore enraizada.

Esta definição de árvore é um pouco diferente do que se define como árvore fora de teoria dos grafos. Normalmente, árvores são usadas para representar uma hierarquia. Este tipo de árvore é chamado, em teoria dos grafos, de árvore enraizada. Uma árvore enraizada T é um conjunto de vértices $V(T)$, com um vértice especial r, chamado de raiz da árvore. Cada vértice possui um conjunto de filhos, sendo que todo vértice com excessão da raiz aparece no conjunto de filhos de exatamente um vértice. A raiz não aparece no conjunto de filhos de nenhum vértice.

Os vértices pertencentes ao conjunto de filhos de um vértice v são chamados de filhos de v e o vértice que possui v como filho é chamado de pai de v. Os vértices que não possuem filhos são chamados de folhas da árvore. Os vértices que não são folhas são chamados de nós internos. Os ancestrais de um vértice v são os vértices que estão no caminho do vértice v até a raiz. Se o vértice u é ancestral de v, diz-se que o vértice v é descendente de u.

O nível de um vértice em uma árvore enraizada é sua distância até a raiz da árvore. Também pode-se falar no nível k de uma árvore como o conjunto dos vértices de nível k. A altura de um vértice é sua distância ao seu descendente mais distante. A altura da árvore é o nível do vértice de maior nível, ou seja, a altura de sua raiz.

Uma subárvore é uma árvore formada por um subconjunto dos vértices de outra árvore, juntamente com as arestas entre esses vértices. Chamamos de subárvore de T enraizada em um vértice v, a subárvore de T que tem como vértices v e todos os seus descendentes em T.

Uma árvore k-ária é uma árvore enraizada onde cada vértice possui no máximo k filhos. O caso mais comum é o das árvores binárias ($k = 2$). Uma árvore estritamente binária é uma árvore onde cada vértice, com exceção das folhas, possui exatamente 2 filhos. Geralmente, os filhos de uma árvore binária possuem dois nomes especiais: direito e esquerdo. Estes dois filhos podem possuir significados distintos, não podendo ser trocados.

O número de arestas de uma árvore é igual ao número de vértices menos 1. Por causa disso, a forma mais usada para representar árvores é com listas de adjacências. Em árvores binárias, isto é ainda mais simples, pois cada vértice tem que armazenar apenas ponteiros para o filho esquerdo, o filho direito e o pai. Caso a árvore seja percorrida apenas da raiz para as folhas, não é necessário armazenar um ponteiro para o pai. Em algumas situações, árvores podem ser armazenadas eficientemente em vetores, como acontece no heap binário (sessão 2.4).

2.3. Subdivisões do Plano e Poliedros

É natural particionarmos os pontos do plano em regiões fechadas usando segmentos de reta (ou, possivelmente, segmentos curvos). Isto é feito, por exemplo, na divisão política de um mapa. Este tipo de divisão possui três elementos: *vértices, arestas e faces.* Uma aresta é um segmento de reta. Um vértice é o ponto de encontro de duas ou mais arestas. Uma face é uma região fechada ou aberta delimitada por arestas.

Desejamos que a estrutura permita que alterações como a inserção de novas arestas, vértices e faces sejam realizadas eficientemente. Também desejamos que relações de adjacência sejam listadas rapidamente, como por exemplo, determinar as arestas incidentes a um vértice, as duas faces adjacentes a uma aresta, as arestas adjacentes a uma face ou as arestas adjacentes a uma aresta.

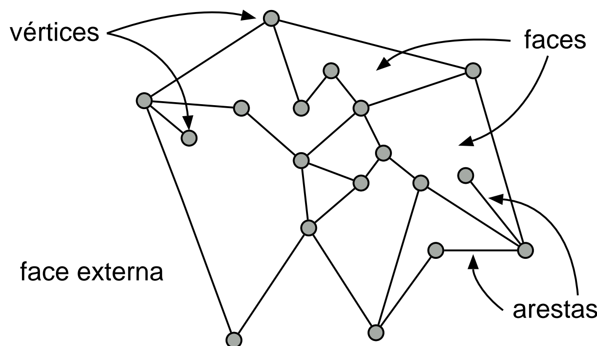


FIGURA 2.3. Divisão do plano e seus elementos.

Consideramos apenas divisões do plano sem buracos, ou seja, subdiviões do plano em que se pode chegar de qualquer vértice a qualquer vértice caminhando apenas pelas arestas. Não é difícil tratar o caso com buracos, bastando armazenar os buracos em estruturas separadas, ligadas as faces onde os buracos ocorrem.

Existem várias estruturas eficientes para armazenar subdivisões do plano. A estrutura que apresentamos aqui chama-se DCEL (doubly connected edge list - lista de arestas duplamente encadeada). O elemento principal da DCEL são as arestas, mais precisamente as semi-arestas. Um vértice tem como atributos um par de coordenadas x, y e um ponteiro para apenas uma semi-aresta que parte dele. Uma face contém apenas um ponteiro para uma semi-aresta adjacente a ela. Uma semi-aresta, por sua vez, possui diversos atributos: seu vértice de origem, sua semi-aresta gêmea, a face adjacente a ela, e duas outras semi-arestas, chamadas de próxima e anterior. As semi-arestas sempre percorrem as faces internas no sentido anti-horário e semi-arestas gêmeas sempre possuem sentidos opostos, comportando-se ao contrário da direção dos carros em vias de mão dupla. Deste modo, a face adjacente a uma semi-aresta está sempre à sua esquerda. A próxima semi-aresta de uma aresta e é a semi-aresta mais a esquerda (com relação a e) dentre as semi-arestas que têm como origem o vértice destino de e. Devido a natureza extremamente geométrica da estrutura DCEL, é mais fácil compreendê-la examinando o exemplo da figura 2.4.

Os algoritmos para implementar operações básicas nessa estrutura são relativamente simples. É um excelente exercício escrever o pseudo-código de alguns destes algoritmos. Apresentamos aqui apenas o pseudo-código da operação que listas todos os vértices adjacentes a um vértice v, no sentido horário, na figura 2.5.

Uma estrutura DCEL também pode ser usada para representar o contorno de poliedros no espaço tridimensional.

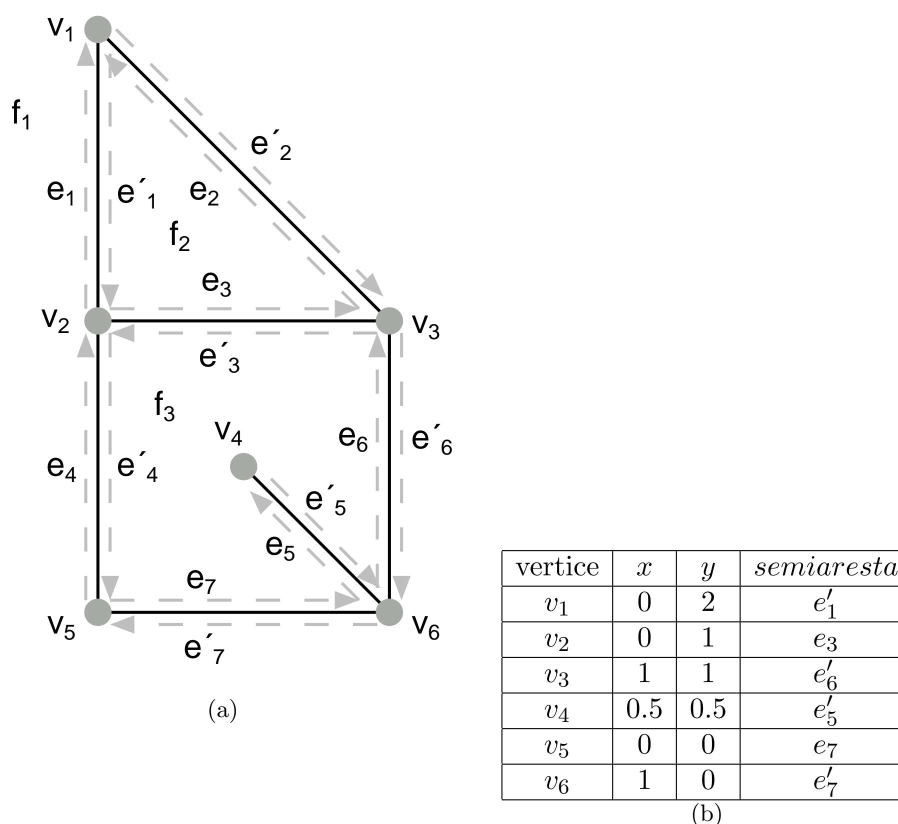
2.4. Lista de Prioridades - Heap Binário

Listas de prioridades são estruturas de dados bastante usadas em vários algoritmos. As principais operações suportadas por uma lista de prioridades são as seguintes:

- Criar(S): retorna uma lista de prioridades contendo os elementos do conjunto S.
- Inserir(H, e): insere elemento e, com prioridade $e.prioridade$, em H.
- Máximo(H): retorna o elemento de maior prioridade de H.
- ExtrairMáximo(H): retorna o elemento de maior prioridade de H, removendo-o de H.

Também são permitidas operações para alterar a prioridade de um elemento, ou remover um elemento da lista. Porém, para usar essas operações é importante armazenar um ponteiro para o elemento dentro da lista de prioridades, pois a estrutura não permite que a busca de um elemento na lista seja realizada eficientemente.

Alternativamente, uma lista de prioridades pode retornar o elemento mínimo e não o elemento máximo. Nesta sessão, trataremos de uma lista de prioridades que retorna o elemento máximo, mas o outro caso é análogo.



vertice	x	y	*semiaresta*
v_1	0	2	e'_1
v_2	0	1	e_3
v_3	1	1	e'_6
v_4	0.5	0.5	e'_5
v_5	0	0	e_7
v_6	1	0	e'_7

(b)

face	*semiaresta*
f_1	e_1
f_2	e_3
f_3	e'_3

(c)

semiaresta	*origem*	*gemea*	*face*	*proxima*	*anterior*
e_1	v_2	e'_1	f_1	e'_2	e_4
e'_1	v_1	e_1	f_2	e_3	e_2
e_2	v_3	e'_2	f_2	e'_1	e_3
e'_2	v_1	e_2	f_1	e'_6	e_1
e_3	v_2	e'_3	f_2	e_2	e'_1
e'_3	v_3	e_3	f_3	e'_4	e_6
e_4	v_5	e'_4	f_1	e_1	e'_7
e'_4	v_2	e_4	f_3	e_7	e'_3
e_5	v_6	e'_5	f_3	e'_5	e_7
e'_5	v_4	e_5	f_3	e_6	e_5
e_6	v_6	e'_6	f_3	e'_3	e'_5
e'_6	v_3	e_6	f_1	e'_7	e'_2

(d)

FIGURA 2.4. (a) Divisão do plano. (b) Estruturas dos vértices correspondentes. (c) Estruturas das faces correspondentes. (d) Estruturas das semi-arestas correspondentes.

Para construirmos uma lista de prioridades, usamos uma árvore binária chamada heap. Cada vértice da árvore é associado a um elemento armazenado. Esta árvore deve satisfazer as seguintes propriedades:

Ordenação de heap: A prioridade de todo vértice é maior que a prioridade de seus filhos.

Balanceamento: Todos os vértices que não possuem exatamente 2 filhos estão nos dois últimos níveis da árvore.

Um exemplo de heap está representado na figura 2.6(a). A propriedade de ordenação de heap serve para que o elemento máximo possa ser encontrado rapidamente. Em uma árvore com

VertAdjVertHor(vertice v)
    $e \leftarrow inicio \leftarrow v.semiaresta$
    Repita
        Listar $e.gemea.origem$
        $e \leftarrow e.gemea.proxima$
    Enquanto $e \neq inicio$

FIGURA 2.5. Algoritmo que lista todos os vértices adjacentes a um vértice v, no sentido horário.

ordenação de heap, o elemento máximo está sempre na raiz. A propriedade de balanceamento serve para garantir que a altura da árvore seja logaritmica, de modo que inserções e remoções sejam realizadas eficientemente, como veremos a seguir.

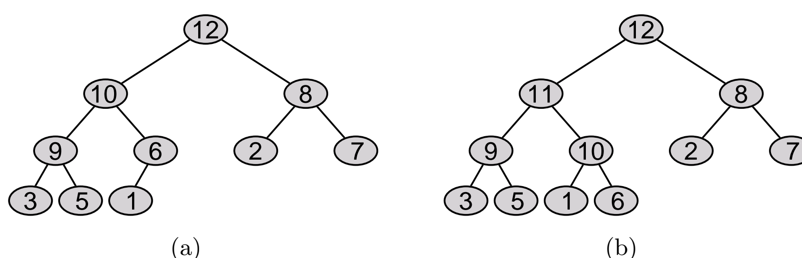


FIGURA 2.6. (a) Exemplo de heap binário. (b) Inserção do elemento 11 no heap da figura (a).

A primeira operação que apresentamos é alterar a prioridade de um elemento do heap. Em seguida, usamos esta operação para construir as demais. Vamos dividir a operação de alterar prioridade em duas operações: aumentar prioridade e reduzir prioridade.

Para aumentar a prioridade de um elemento, primeiro trocamos o valor desta prioridade, possivelmente violando a ordenação de heap. Em seguida, seguimos trocando a posição do elemento que teve a prioridade aumentada com seu pai, até que a ordenação de heap seja reestabelecida, como ilustra a figura 2.7.

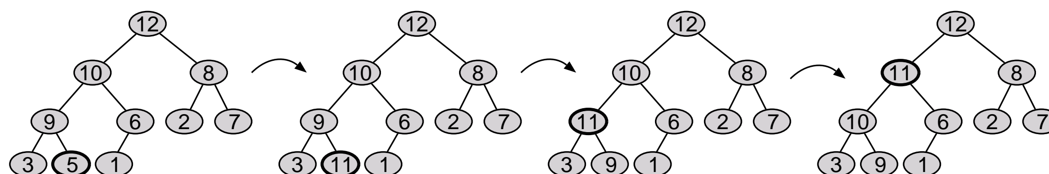


FIGURA 2.7. Aumento da prioridade de um elemento de 5 para 11.

Para reduzir a prioridade de um elemento, primeiro trocamos o valor desta prioridade, possivelmente violando a ordenação de heap. Em seguida, seguimos trocando a posição do elemento que teve a prioridade reduzida com seu filho de maior prioridade, até que a ordenação de heap seja reestabelecida, como ilustra a figura 2.8.

A complexidade de tempo dessas operações é proporcional à altura da árvore, sendo, portanto, $\Theta(\lg n)$, onde n é o número de elementos armazenados no heap.

Para inserirmos um elemento, colocamos uma nova folha na árvore, filha do elemento de nível mais alto que ainda não possuir dois filhos. Esta folha tem, inicialmente, prioridade $-\infty$. Então, aumentamos a prioridade desta folha para o valor desejado, com o procedimento descrito anteriormente.

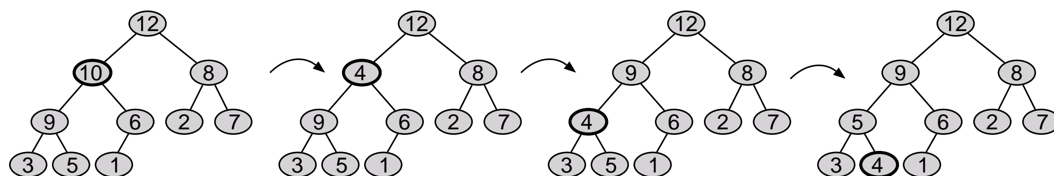


FIGURA 2.8. Redução da prioridade de um elemento de 10 para 4.

Para removermos um elemento e, primeiro escolhemos uma folha f qualquer no último nível da árvore e removemos esta folha. A remoção direta desta folha não altera nem o balanceamento da árvore nem a ordenação de heap, porém, este não é o elemento que desejávamos remover. Então, colocamos o elemento f na posição do elemento e, alterando a prioridade com o método que já descrevemos. Note que podemos estar aumentando ou diminuindo a prioridade. Resumindo, para removermos um elemento e, escolhemos uma elemento f que podemos remover facilmente e movemos este elemento para o lugar de e, em seguida restaurando a ordenação de heap como na alteração de prioridade.

Deste modo, os procedimentos de inserção e remoção levam tempo $\Theta(\lg n)$, onde n é o número de elementos armazenados no heap. Pode-se criar um heap inicial com n elementos fazendo n inserções sucessivas. Porém, este procedimento leva tempo $\Theta(n \lg n)$. É possível criar um heap inicial com n elementos em tempo $O(n)$, usando o método descrito abaixo.

Começamos distribuindo os elementos arbitrariamente na árvore, satisfazendo a propriedade de balanceamento, mas sem nos preocuparmos com a ordenação de heap. Então, restauramos a ordenação de heap de cima para baixo. Chamamos de subárvores do nível l as subárvores enraizadas nos vértices que ocupam o nível l da árvore. Todas as subárvores do último nível já satisfazem a ordenação de heap trivialmente, por conter apenas um vértice. Para que todas as subárvores do penúltimo nível satisfaçam a ordenação de heap, usamos o procedimento de reduzir prioridade em suas raízes. Repetimos esse procedimento para todos os vértices, partindo do nível mais baixo para o mais alto, até chegar na raiz, quando todo o heap passa a satisfazer a ordenação de heap. Este procedimento está ilustrado na figura 2.9.

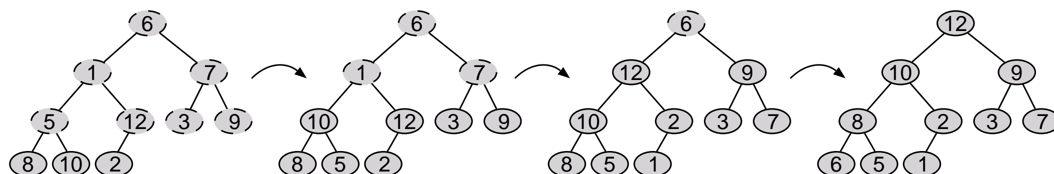


FIGURA 2.9. Criação de um heap inicial em tempo linear.

Em princípio, pode não ser claro que este método é mais eficiente que inserir os elementos um a um. A complexidade de tempo deste procedimento é proporcional a soma das alturas dos vértices. Em um heap com n elementos, o número de vértices de altura h é no máximo $n/2^{h+1}$. Consequentemente, a complexidade do tempo da criação de um heap com n elementos é, no máximo:

$$T(n) = \sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} h = n \sum_{h=0}^{\lg n} \frac{h}{2^{h+1}}.$$

Claramente,

$$T(n) \leqslant n \sum_{h=0}^{\infty} \frac{h}{2^{h+1}}.$$

Para calcularmos este somatório, podêmos fazer uma decomposição em várias progressões geométricas de razão $1/2$:

$$\sum_{h=1}^{\infty} \frac{h}{2^{h+1}} = \left.\begin{array}{ccccccccc} 1/4 & + & 1/8 & + & 1/16 & + & \ldots & = & 1/2 \\ & + & 1/8 & + & 1/16 & + & \ldots & = & 1/4 \\ & & & + & 1/16 & + & \ldots & = & 1/8 \\ & & & & & + & \ldots & = & \ldots \end{array}\right\} = 1$$

Com isso, provamos que a complexidade de tempo do procedimento de criação de um heap binário com n elementos é $O(n)$.

Normalmente, heaps binários são armazenados em vetor, e não usando listas de adjacências. A raiz do heap é a primeira posição do vetor, seu filho esquerdo a segunda, seu filho direito a terceira e assim por diante, como ilustra a figura 2.10.

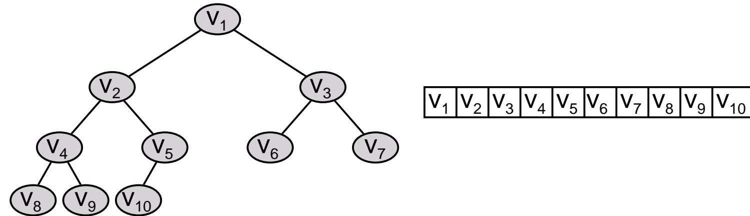


FIGURA 2.10. Heap armazenado em vetor.

Note que, armazenando o heap em um vetor $v = (v_1, \ldots, v_n)$, a ordenação de heap pode ser escrita como

$$v_i \geqslant v_{2i} \text{ e } v_i \geqslant v_{2i+1},$$

ou ainda como

$$v_i \leqslant v_{\lfloor i/2 \rfloor}.$$

Os pseudo-códigos das diversas operações para um heap armazenado em vetor estão na figura 2.11.

Uma outra estrutura para listas de prioridades é chamada de heap de Fibonacci. Esta estrutura tem complexidades de tempo melhores que o heap binário, permitindo que n inserções sejam realizadas em tempo $O(n)$. Dizemos que cada inserção tem complexidade de tempo amortizada $O(1)$, pois, embora uma inserção possa demorar mais, em média as inserções sempre levam tempo $O(1)$. A operação de redução de prioridade também tem tempo amortizado de $O(1)$. As demais operações que estudamos aqui levam o mesmo tempo que no heap binário. Na prática, porém, o heap binário é mais eficiente que o heap de Fibonacci porque as constantes multiplicativas ocultas na notação O são muito grandes, fazendo com que o heap de Fibonacci só seja mais rápido para quantidades de elementos maiores que valores processados na prática. Ainda assim, essa estrutura é de grande interesse teórico e obter uma lista de prioridades com as mesmas complexidades assintóticas do heap de Fibonacci, porém eficiente na prática, é um problema em aberto bastante estudado.

2.5. Árvores Binárias de Busca

A estrutura de dados estudada nessa sessão possui o seguinte conjunto de operações:

- Inserir(T, e): insere elemento e, com chave $e.chave$, em T.
- Remover(T, x): remove o elemento que possui chave x.
- Buscar(T, x): retorna o elemento e tal que $e.chave = x$, se existir.

Normalmente, o campo chave pertence a um conjunto ordenado, ou seja, dadas duas chaves distintas x_1 e x_2, ou $x_1 < x_2$, ou $x_2 < x_1$. Neste caso, podemos usar árvores binárias de busca para construir esta estutura de dados.

Uma árvore binária de busca é uma árvore em que cada vértice está associado a um elemento e, para todo o elemento e, vale que: as chaves dos elementos na subárvore esquerda de e são

Observações:
Neste pseudo-código, consideramos que os elementos são apenas prioridades, sem possuir outros atributos.
h: Vetor que armazena o heap.
n: Número de elementos de h.
p: Prioridade de um elemento.
i: Posição de um elemento de h.
S: Vetor com n elementos.

```
AlterarPrioridade(h,n,i,p)
    Se p > h[i]
        AumentarPrioridade(h,i,p)
    Senão
        ReduzirPrioridade(h,n,i,p)

AumentarPrioridade(h,i,p)
    h[i] ← p
    Enquanto i > 1 e h[⌊i/2⌋] < h[i]
        Troca h[i] e h[⌊i/2⌋]

ReduzirPrioridade(h,n,i,p)
    h[i] ← p
    Enquanto 2i ⩽ n
        Se (h[2i + 1] > n ou h[2i] > h[2i + 1]) e h[2i] > h[i]
            Troca h[i] e h[2i]
        Senão se h[2i + 1] ⩽ n e h[2i + 1] > h[i]
            Troca h[i] e h[2i + 1]

Criar(S,n)
    h ← S
    Para i de n até 1
        ReduzirPrioridade(h,n,i,h[i])
    Retorne h

Inserir(h,n,p)
    n ← n + 1
    AumentarPrioridade(h,n,p)

Remover(h,n,i)
    n ← n − 1
    AlterarPrioridade(h,n,i,h[n + 1])
```

FIGURA 2.11. Pseudo-código das operações de um heap binário em vetor.

menores que $e.chave$ e as chaves de todos os elementos na subárvore direita de e são maiores que $e.chave$. Dois exemplos de árvores binárias de busca estão representados na figura 2.12.

Para buscar uma chave x em uma árvore binária de busca, começamos comparando x com a chave da raiz r. Se $x.chave = r.chave$, já encontramos o elemento desejado e podemos parar a busca. Caso $x.chave < r.chave$, sabemos que, se existir elemento com chave x, este elemento está na subárvore esquerda de r. Nesse caso, chamamos o procedimento recursivamente para buscar x na subárvore esquerda de r. O caso $x.chave > r.chave$ é análogo. No lugar de fazermos a busca recursivamente na subárvore esquerda de r, o fazemos na subárvore direita de r. Este procedimento segue até encontrarmos o elemento ou tentarmos fazer a busca em uma árvore vazia. Neste último caso, constatamos que a chave procurada não está armazenada na árvore. Este procedimento está exemplificado na figura 2.13(a).

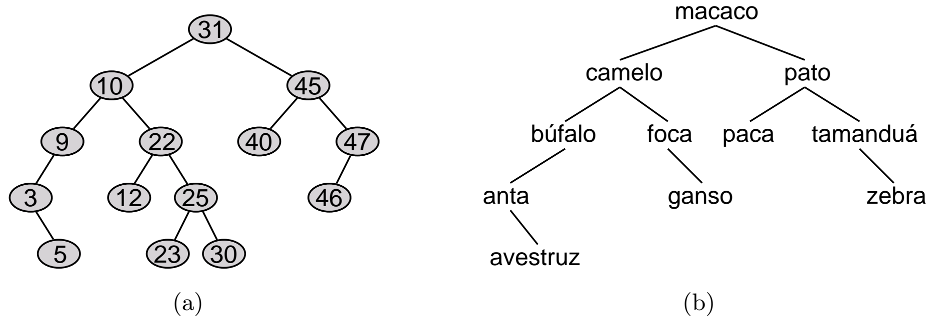


Figura 2.12. (a) Árvore binária de busca com chaves inteiras. (b) Árvore binária de busca com chaves de cadeias de caracteres.

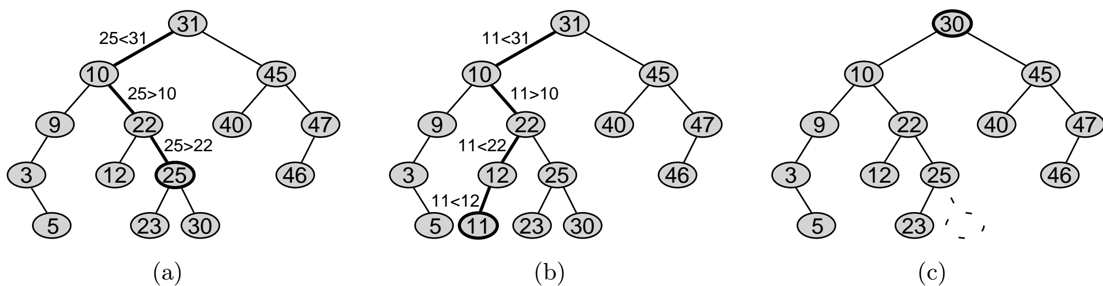


Figura 2.13. (a) Busca de elemento com chave 25 na árvore da figura 2.12(a). (b) Inserção de elemento com chave 11 na árvore da figura 2.12(a). (c) Remoção do elemento de chave 31 na árvore da figura 2.12(a).

Para inserirmos um elemento, começamos fazendo uma busca de sua chave. Caso a chave já esteja na árvore, não devemos inserí-la, pois todos os elementos devem ter chaves distintas. Caso não esteja, nossa busca terminará em uma subárvore vazia. Podemos colocar o elemento que desejamos inserir nesta posição. Este procedimento está exemplificado na figura 2.13(b).

Para removermos um elemento, também começamos fazendo uma busca de sua chave. Caso o elemento seja uma folha, basta removê-lo. Caso tenha apenas um filho, também pode-se remover o elemento desejado diretamente, subindo o filho em um nível. Caso tenha dois filhos, uma alternativa simples é buscar o maior elemento de sua subárvore esquerda, que ou é uma folha, ou possui apenas um filho. Remove-se diretamente este elemento, movendo-o para a posição do elemento que realmente desejamos remover. Este procedimento está exemplificado na figura 2.13(c).

A complexidade de tempo dessas operações depende da altura da árvore. Infelizmente, estes métodos de inserção e remoção não garantem que a altura da árvore seja logarítmica. Para um pior caso, imagine que os elementos são inseridos em ordem crescente. Nesse caso, a árvore obtida não passa de uma lista encadeada, pois nenhum elemento possui filho esquerdo.

Existem várias maneiras de garantir que a altura de uma árvore binária de busca com n elementos seja $O(\lg n)$. Todas elas se baseiam no conceito de rotações, normalmente realizadas nas operações de inserção e remoção. Uma rotação é uma alteração local na topologia da árvore que preserva a propriedade de árvore binária de busca. As duas rotações principais estão apresentadas graficamente na figura 2.14. Alguns exemplos de árvores binárias de busca balanceadas, ou seja, com altura $O(\lg n)$ são árvores rubro-negras, árvores AVL, árvores de difusão (splay trees, complexidade amortizada) e treaps (estrutura randomizada). Nenhuma destas estruturas é muito simples e todas estão bem documentadas em livros de estruturas de dados, por isso não entramos em detalhes aqui.

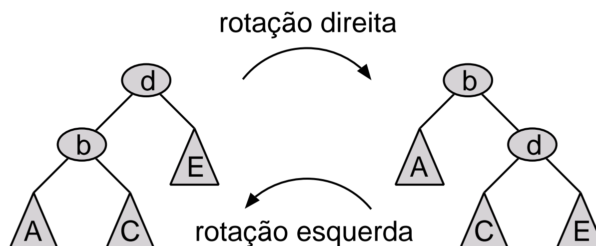


FIGURA 2.14. Rotações direita e esquerda em árvores binárias de busca.

2.6. Resumo e Observações Finais

Neste capítulo, fizemos um resumo de diversas estruturas de dados. Partimos das estruturas elementares, chamadas variáveis. Agrupamentos de variáveis são chamados de estruturas. Vetores são uma seqüência de variáveis do mesmo tipo.

Uma lista armazena uma seqüência de elementos. Vetores servem para armazenar listas, que também podem ser armazenadas através de listas encadeadas. Dois tipos especiais de listas são chamados de filas e pilhas. Em uma fila, os elementos são sempre inseridos em um extremo e removidos do extremo oposto da lista. Em uma pilha, os elementos são sempre inseridos e removidos no mesmo extremo.

Grafos são uma estrutura combinatória muito estudada e com diversas aplicações. Um grafo consiste em um conjunto de vértices e um conjunto de arestas, que são pares de vértices. Grafos podem ser armazenados como matrizes de adjacência ou listas de adjcências, sendo que a última é normalmente preferível para grafos com poucas arestas.

Uma árvore é um tipo especial de grafo que não possui ciclos. Uma árvore enraizada é uma árvore como um vértice especial chamado de raiz, e serve para representar hierarquias. Uma árvore binária é uma árvore enraizada em que cada vértice possui dois filhos diferentes, chamados de filho direito e filho esquerdo.

Uma subdivisão do plano por segmentos pode ser representada eficientemente com uma estrutura DCEL. Esta estrutura tem como elemento principal as semi-arestas.

Listas de prioridades são estruturas de dados não triviais extremamente úteis para o desenvolvimento de algoritmos eficientes. Uma lista de prioridades armazena um conjunto de elementos, sujeito a inserções e remoções, permitindo que o elemento máximo seja determinado rapidamente. A estrutura mais usada para armazenar listas de prioridades é o heap binário, que é uma árvore balanceada onde todo vértice é maior que seus filhos.

Uma árvore binária de busca permite que elementos sejam inseridos, removidos, ou encontrados a partir de uma chave. Para garantir que as operações sejam realizadas eficientemente, entretanto, é preciso usar árvores binárias de busca especiais. Estas árvores, como AVL, rubro-negra etc, não são apresentadas aqui e usam rotações para garantir que a altura da árvore seja logaritmica.

Exercícios

2.1) Compare vantagens e desvantagens em armazenar uma lista em vetor ou como lista encadeada.

2.2) Seja h_n a menor altura possível para uma árvore binária com n vértices. Prove que $h_n = \Theta(\lg n)$.

2.3) Escreva o pseudo-códigos que lista todos os vértices de uma face, armazenada em estrutura DCEL, no sentido horário.

2.4) Explique porque o método descrito a seguir não deve ser usado para remover um elemento de um heap binário. Inicia-se o procedimento, esvaziando-se o vértice correspondente ao elemento que desejamos remover. Em seguida, determina-se seu maior filho, e move-se

o elemento correspondente a ele para o vértice pai, esvaziando o filho. Repete-se este procedimento até chegar em uma folha.

2.5) Escreva os pseudo-códigos das operações de busca, inserção e remoção em árvores binárias de busca.

CAPíTULO 3

Busca Binária

A técnica de busca binária consiste em examinar um número pequeno de elementos da entrada (normalmente apenas um) e, com isso, descartar imediatamente uma fração constante dos elementos da entrada (normalmente metade). Procede-se desta maneira até que o conjunto de elementos candidatos a serem a solução do problema seja suficientemente pequeno.

3.1. Busca em vetor

Um vetor $v = (v_1, \ldots, v_n)$ contém n números reais e desejamos saber se um número x está ou não no vetor. Um algoritmo trivial é percorrer este vetor do primeiro ao último elemento, comparando-os com x. Ao encontrarmos um elemento com valor x, podemos parar. Mas, se nenhum elemento tiver este valor, somos obrigados a ler o vetor inteiro. Claramente não podemos fazer melhor que isso no pior caso, pois qualquer posição é candidata a ter o valor x e não inspecionar esta posição nos levaria a uma resposta errada.

Vamos mudar um pouco o problema. Agora sabemos que o vetor $v = (v_1, \ldots, v_n)$ está ordenado, mais especificamente, para i de 1 até $n-1$ temos $v_i \leqslant v_{i+1}$.

Problema 2. *Dados um vetor $v = (v_1, \ldots, v_n)$ ordenado, contendo elementos reais e um número real x, determinar se existe uma posição i tal que $v_i = x$.*

O algoritmo anterior também funciona para este problema, mas sua complexidade de tempo de pior caso é $O(n)$. Será que podemos fazer melhor? A resposta é sim. Usando uma técnica chamada busca binária, podemos melhorar a complexidade para $O(\lg n)$. De fato, em um vetor com 1000 elementos, o número de comparações no pior caso reduz de 1000 para 10.

A técnica se torna mais intuitiva se apresentada como um jogo. Um jogador João pensa em um número de 1 a 1000 e uma jogadora Maria deve adivinhar este número. Quando Maria chuta um número, João responde se ela acertou ou, caso contrário, se o número em que ele pensou é maior ou menor do que o que ela chutou. A melhor estratégia para Maria é sempre dividir o intervalo em duas partes iguais. Começa chutando 500 (poderia ser 501 também). Em seguida chuta 250 ou 750, de acordo com a resposta de João.

Retornando ao problema de encontrar um elemento de valor x em um vetor ordenado, primeiro examinamos o elemento $v_{\lfloor (n+1)/2 \rfloor}$. Se $x > v_{\lfloor (n+1)/2 \rfloor}$, então sabemos que só as posições de $\lfloor (n+1)/2 \rfloor + 1$ a n são candidatas a ter valor x. Analogamente, se $x < v_{\lfloor (n+1)/2 \rfloor}$, então sabemos que só as posições de 1 a $\lfloor (n+1)/2 \rfloor - 1$ são candidatas a ter valor x. Claro que, se $x = v_{\lfloor (n+1)/2 \rfloor}$, o problema já está resolvido. Repetimos este processo até encontrarmos um elemento de valor x ou o intervalo ter apenas um elemento ou nenhum elemento. O pseudo-código deste algoritmo pode ser encontrado na figura 3.1.

É trivial provar que este algoritmo funciona, isto é, resolve o problema 2. Ainda assim vamos fazer a prova formalmente.

Teorema 3.1. *O algoritmo que acabamos de descrever resolve corretamente o problema 2.*

Demonstração. Ao examinarmos um elemento v_i do vetor $v = (v_1, \ldots, v_n)$, procurando um elemento x temos três opções: $x < v_i$, $x = v_i$ e $x > v_i$. Caso $x = v_i$ o algoritmo retorna v_i, funcionando corretamente. Caso $x < v_i$, como o vetor está ordenado, somente os elementos de v_1 a v_{i-1} são candidatos a ter valor x e o algoritmo resolve este problema recursivamente. O caso $x > v_i$ é análogo.

Entrada:
v: Vetor de reais em ordem crescente.
$inicio$: Primeiro elemento da partição do vetor. Inicialmente 1.
fim: Último elemento da partição do vetor. Inicialmente o tamanho do vetor.
x: Valor que está sendo procurado.
Saída:
Índice i tal que $v[i] = x$, se existir.

```
BuscaBinária(v, inicio, fim, x)
    Se inicio < fim
        Retorne "x ∉ v"
    Se inicio = fim
        Se v[inicio] = x
            Retorne inicio
        Senão
            Retorne "x ∉ v"
    meio ← ⌊(inicio + fim)/2⌋
    Se v[meio] > x
        Retorne BuscaBinária(v, inicio, meio − 1, x)
    Se v[meio] < x
        Retorne BuscaBinária(v, meio + 1, fim, x)
    Retorne meio
```

FIGURA 3.1. Solução do Problema 2

O caso base é quando o vetor tem apenas 1 elemento ou nenhum elemento. Caso o vetor não tenha nenhum elemento, claramente não tem elemento com valor x. Caso tenha apenas 1 elemento o algoritmo resolve o problema comparando este elemento com x. □

Resta agora analisarmos a complexidade de tempo do algoritmo. Faremos uma prova geral que servirá de base para todos os algoritmos baseados em busca binária. A idéia é que, como a cada passo descartamos uma fração constante dos elementos, a complexidade de tempo é logarítmica. Vamos chamar de $T(n)$ o tempo gasto pelo algoritmo para um vetor de tamanho n. Em um tempo constante, o algoritmo descarta uma fração $\alpha < 1$ constante (normalmente $\alpha = 1/2$) dos elementos. Temos então

$$T(n) = T(\alpha n) + 1.$$

Podemos assumir que o tempo constante de cada passo seja 1, pois a notação O ignora constantes multiplicativas.

Vamos provar que $T(n) = \Theta(\lg n)$, supondo que $T(\alpha n) = \Theta(\lg n)$. Usando indução temos

$$T(n) = T(\alpha n) + 1 = c \lg(\alpha n) + 1 = c \lg n + c \lg \alpha + 1.$$

Se fizermos $c = -1/\lg \alpha$ temos $T(n) = c \lg n$ e finalizamos a indução.

Com isto temos:

TEOREMA 3.2. *O algoritmo que descrevemos tem complexidade de tempo* $\Theta(\lg n)$*, onde* n *é o número de elementos do vetor.*

3.2. Busca em vetor ciclicamente ordenado

Muitas vezes, falaremos de índices de vetores módulo n. Com isto queremos dizer que, se $v = (v_1, \ldots, v_n)$ e nos referimos a um elemento v_i fora do intervalo, ou seja, $i < 1$ ou $i > n$, então estamos nos referindo ao elemento do intervalo obtido somando ou subtraindo n a i quantas

vezes for necessário. Por exemplo, em um vetor $v = (v_1, \ldots, v_5)$, quando dizemos v_{-5}, v_0 ou v_{10} estamos nos referindo ao elemento v_5.

Seja $v = (v_1, \ldots, v_n)$ um vetor de reais com índices módulo n. Dizemos que v está ciclicamente ordenado se o número de elementos v_i tais que $v_i \leqslant v_{i+1}$ para i de 1 a n é igual a $n - 1$. Por exemplo, o vetor $(5, 8, 9, 10, 1, 3)$ está ciclicamente ordenado.

PROBLEMA 3. *Dados um vetor v ciclicamente ordenado, contendo elementos reais e um número real x, determinar a posição i tal que $v[i] = x$, se existir.*

Para resolvermos este problema devemos examinar duas posições ao invés de uma. É útil pensarmos no vetor como um círculo. Examinamos os elementos v_i e v_j com $i < j$ de modo que o número de elementos entre v_i e v_j pelos dois lados do círculo seja aproximadamente igual. Caso $v_i \leqslant v_j$, sabemos que se $v_i \leqslant x < v_j$ então x só pode estar nas posições de i até $j - 1$ e se $x < v_i$ ou $x \geqslant v_j$ então x só pode estar nas posições menores que i ou maiores ou iguais a j. Caso $v_i > v_j$, sabemos que se $x \geqslant v_i$ ou $x < v_j$ então x está nas posições de i até $j - 1$ e se $v_j \leqslant x < v_i$ então x está nas posições menores ou iguais a j ou maiores que i.

TEOREMA 3.3. *O algoritmo que acabamos de descrever resolve corretamente o problema 3.*

DEMONSTRAÇÃO. Buscando um elemento com valor x, examinamos dois elementos v_i e v_j do vetor $v = (v_1, \ldots, v_n)$, com $i < j$. Caso $v_i \leqslant v_j$ o vetor formado pelos elementos de v_i à v_j está ordenado e x é candidato a estar nas posições de índice i até $j - 1$ se e só se $v_i \leqslant x < v_j$. O procedimento é chamado recursivamente para a partição do vetor candidata a conter elemento de valor x. Caso $v_i > v_j$ o vetor formado pelos elementos após v_j e anteriores a v_i está ordenado e o argumento é análogo.

O caso base é quando o vetor tem apenas 1 elemento ou nenhum elemento. Caso o vetor não tenha nenhum elemento, claramente não tem elemento com valor x. Caso tenha apenas 1 elemento o algoritmo resolve o problema comparando este elemento com x. □

Para facilitar a implementação podemos sempre pegar como p_i o ponto com o menor índice i dentro do intervalo, como está ilustrado na figura 3.2. Assim evitamos que a partição do vetor seja descontínua na memória.

A complexidade de tempo deste algoritmo é $\Theta(\lg n)$, pelo mesmo princípio do algoritmo da sessão 3.1.

3.3. Ponto extremo de polígono convexo

A técnica de busca binária tem várias aplicações em geometria computacional, especialmente quando a entrada é um polígono convexo.

Um ponto no plano é representado por um par de coordenadas reais. Representamos um polígono de n vértices como um vetor $v = (v_1, \ldots, v_n)$ contendo n pontos no plano. A posição v_1 contém um dos vértices (qualquer um), v_2 o próximo vértice no sentido anti-horário e assim por diante. Denotamos por $\circlearrowleft(p_1, p_2, p_3)$ o ângulo positivo $\widehat{p_1 p_2 p_3}$ medido no sentido anti-horário. Devido a natureza cíclica dos polígonos, trabalharemos com índices módulo n, ou seja, se o índice do vetor for maior do que n ou menor do que 1, devemos somar ou subtrair n até que o índice esteja neste intervalo. Um polígono é convexo se, para i de 1 à n, o ângulo $\circlearrowleft(v_{i-1}, v_i, v_{i+1})$ for maior que 180° (figura 3.3(a)). Note que quando $i = 1$, ao dizermos $i - 1$ estamos nos referindo a posição n. Quando $i = n$, ao dizermos $i + 1$ estamos nos referindo a posição 1.

Existem várias definições equivalentes para polígono convexo. A maioria caracteriza a interseção do polígono com uma reta. Uma definição deste tipo é: um polígono é convexo se sua interseção com uma reta ou é nula ou é um ponto ou um segmento de reta. Esta definição considera o polígono cheio, ou seja, o interior do polígono também é considerado parte do polígono. Esta última definição não nos fornece diretamente nenhum algoritmo para verificar se, dado um polígono, ele é convexo. Já a definição do parágrafo anterior nos fornece um algoritmo linear para verificar convexidade. Basta examinarmos todos os ângulos.

Dizemos que um vértice v_i de um polígono $P = (v_1, \ldots, v_n)$ é extremo na direção de um vetor d se $d \cdot v_i \geqslant d \cdot v_j$ para todo $j \neq i$. Denotamos por $u \cdot v$ o produto escalar $u_x v_x + u_y v_y$.

Entrada:
v: Vetor de reais ciclicamente ordenado.
$inicio$: Primeiro elemento da partição do vetor. Inicialmente 1.
fim: Último elemento da partição do vetor. Inicialmente o tamanho do vetor.
x: Valor que está sendo procurado.
Saída:
Índice i tal que $v[i] = x$, se existir.

```
BuscaBináriaCíclica(v, inicio, fim, x)
    Se inicio < fim
        Retorne "x ∉ v"
    Se inicio = fim
        Se v[inicio] = x
            Retorne inicio
        Senão
            Retorne "x ∉ v"
    meio ← ⌊(inicio + fim + 1)/2⌋
    Se v[inicio] ⩽ v[meio]
        Se x ⩾ v[inicio] e x < v[meio]
            Retorne BuscaBináriaCíclica(v, inicio, meio − 1, x)
        Senão
            Retorne BuscaBináriaCíclica(v, meio, fim, x)
    Senão
        Se x ⩾ v[meio] e x < v[inicio]
            Retorne BuscaBináriaCíclica(v, meio, fim, x)
        Senão
            Retorne BuscaBináriaCíclica(v, inicio, meio − 1, x)
```

FIGURA 3.2. Solução do Problema 3

Uma outra definição mais geométrica é que v_i é extremo na direção d se a reta perpendicular a d que passa por v_i divide o plano em dois semiplanos tais que todos os pontos do polígono que não estão sobre a reta estão em um mesmo semiplano e o ponto $v_i + d$ está no outro semiplano (figura 3.3(b)).

Agora podemos definir o problema:

PROBLEMA 4. *Dados um polígono convexo P e um vetor d determinar o vértice de P extremo na direção d.*

Vamos começar pegando dois vértices quaisquer v_i e v_j do polígono $P = (v_1, \ldots, v_n)$, com $i < j$. Podemos usar este par de vértices para decompor P em dois polígonos convexos $P_1 = (v_i, v_{i+1}, \ldots, v_j)$ e $P_2 = (v_1, v_2, \ldots, v_i, v_j, v_{j+1}, \ldots, v_n)$. Para usarmos o princípio de busca binária precisamos descobrir qual desses dois polígonos contém o ponto extremo. Primeiro comparamos $d \cdot v_i$ com $d \cdot v_j$. Vamos considerar inicialmente que $d \cdot v_i > d \cdot v_j$ e depois trataremos do outro caso. Comparamos então $d \cdot v_i$ com $d \cdot v_{i+1}$. Caso $d \cdot v_i > d \cdot v_{i+1}$ o polígono que contém o ponto extremo é $P_1 = (v_i, v_{i+1}, \ldots, v_j)$. Para provarmos este fato vamos considerar a reta r perpendicular a d que passa por v_i e os dois semiplanos S e $\bar{S}$ definidos por ela. Chamamos de S o semiplano que contém v_{i+1}. Os pontos que estão em S não são candidatos a serem extremos, pois o produto escalar de qualquer um desses pontos com d é menor que $d \cdot v_i$. Todos os pontos de P_2 estão em S, pois caso contrário r interceptaria o interior de P_2 e também tangenciaria P_2 no vértice v_i. Caso $d \cdot v_i < d \cdot v_{i+1}$ o polígono que contém o ponto extremo é P_2, usando o mesmo argumento. Caso $d \cdot v_i < d \cdot v_j$, devemos comparar $d \cdot v_j$ com $d \cdot v_{j+1}$. Se $d \cdot v_j > d \cdot v_{j+1}$,

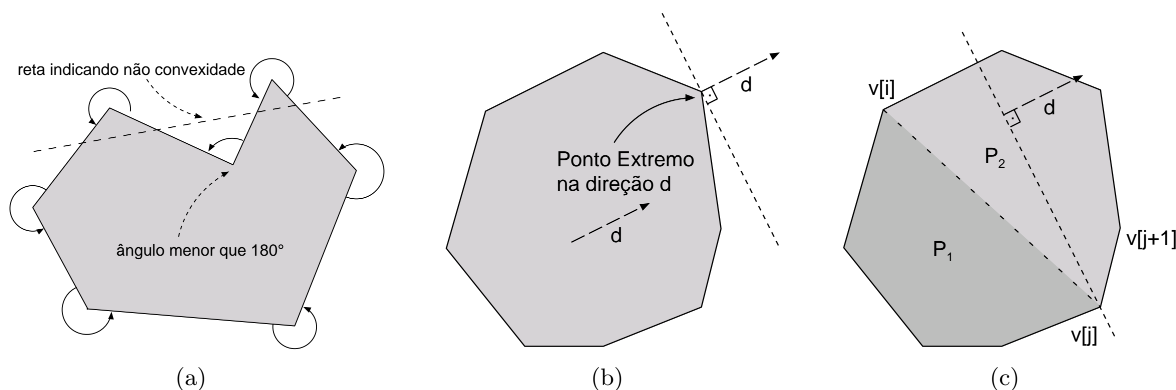


FIGURA 3.3. (a) Polígono não convexo. (b) Ponto extremo na direção d. (c) Algoritmo examinando v_i, v_j e v_{j+1} com $d \cdot v_i < d \cdot v_j$ e $d \cdot v_j < d \cdot v_{j+1}$

então P_1 contém o ponto extremo. Caso contrário P_2 contém o ponto extremo. Um exemplo está ilustrado na figura 3.3(c).

Frequentemente, ao projetarmos um algoritmo para resolver um determinado problema, acabamos descobrindo que nosso algoritmo é ainda mais poderoso do que o planejado, podendo resolver problemas aparentemente mais difíceis. Para provarmos que nosso último algoritmo funciona, usamos o fato do polígono ser convexo para garantir que se r intercepta um polígono em um vértice, então r não pode interceptar o polígono em uma região que não tem extremo neste vértice. Mas r não é uma reta qualquer, e sim uma reta perpendicular a d. Podemos definir um polígono d-monótono como: um polígono é d-monótono se sua interseção com uma reta perpendicular a d ou é nula ou é um ponto ou um segmento de reta. Nosso algoritmo não funciona apenas com polígonos convexos, mas também com polígonos d-monótonos quaisquer!

TEOREMA 3.4. *O algoritmo que acabamos de descrever resolve corretamente o problema 4.*

Para obtermos complexidade $O(\lg n)$, devemos escolher v_i e v_j de modo que o número de vértices entre v_i e v_j nos dois sentidos seja aproximadamente igual. O pseudo-código da figura 3.4 nos leva a uma implementação bastante simples e eficiente deste algoritmo, embora alguns detalhes sejam diferentes de nossa descrição.

Uma das técnicas usadas neste pseudo-código é usar uma função PontoExtremoLin que examina os vértices um a um e encontra o vértice extremo (esta função simples não está descrita no pseudo-código). Existem duas vantagens em chamar esta função quando o polígono é muito pequeno. Uma é que para polígonos pequenos é mais rápido examinar todos os vértices do que fazer uma busca binária. A outra vantagem é que assim não precisamos nos preocupar com casos pequenos, que teriam de ser tratados de forma especial.

A complexidade de tempo deste algoritmo é $\Theta(\lg n)$, pelo mesmo princípio do algoritmo da sessão 3.1.

3.4. Função de vetor

Vamos finalizar o capítulo com um exemplo de algoritmo que, embora seja baseado em busca binária, não tem complexidade logaritmica. O nosso problema é o seguinte:

PROBLEMA 5. *Dados um vetor $v = (v_1, \ldots, v_n)$ ordenado, contendo elementos reais e uma função $f : \mathbb{R} \to \mathbb{R}$, construir um vetor $v' = (v'_1, \ldots, v'_n)$ tal que $v'_i = j$ se existir j com $f(v_i) = v_j$ e $v'_i = \varepsilon$ caso contrário.*

Claramente não podemos esperar resolver este problema em tempo $O(\lg n)$ porque temos que construir um vetor com n elementos. Ainda assim a técnica de busca binária resolve diretamente o problema, como está ilustrado no pseudo-código da figura 3.5.

A prova do teorema abaixo é trivial e fica como exercício.

Entrada:

v: Vetor com os vértices do polígono.
$inicio$: Primeiro elemento da partição do vetor. Inicialmente 1.
fim: Último elemento da partição do vetor. Inicialmente o tamanho do vetor.
d: Direção em que se deseja maximizar.

Saída:

Vértice extremo na direção d.

Observações:

PontoExtremoLin é uma função idêntica em funcionalidade à PontoExtremo, que examina os vértices um a um, escolhendo o extremo.

```
PontoExtremo(v, inicio, fim, d)
    Se fim − inicio ⩽ 8
        Retorne PontoExtremoLin(v, inicio, fim, d) //Função que examina os pontos um a um
    meio ← ⌊(inicio + fim + 1)/2⌋
    Se d.x ∗ v[inicio].x + d.y ∗ v[inicio].y < d.x ∗ v[meio].x + d.y ∗ v[meio].y
        Se d.x ∗ v[meio].x + d.y ∗ v[meio].y < d.x ∗ v[meio + 1].x + d.y ∗ v[meio + 1].y
            Retorne PontoExtremo(v, meio + 1, fim, d)
        Senão
            Retorne PontoExtremo(v, inicio + 1, meio, d)
    Senão
        Se d.x ∗ v[inicio].x + d.y ∗ v[inicio].y < d.x ∗ v[fim].x + d.y ∗ v[fim].y
            Retorne PontoExtremo(v, meio + 1, fim, d)
        Senão
            Retorne PontoExtremo(v, inicio, meio − 1, d)
```

FIGURA 3.4. Solução do Problema 4

Entrada:

v: Vetor de números reais em ordem crescente.
f: Função $f : \mathbb{R} \to \mathbb{R}$

Saída:

Vetor v' tal que $v'[i] = j$ se existir j com $f(v[i]) = v[j]$ e $v'[i] = \varepsilon$ caso contrário.

```
FunçãoDeVetor(v, f)
    Alocar vetor v′ com n posições inteiras
    Para i de 1 até n
        j ← BuscaBinária(v, 1, n, f(v[i]))
        Se j = "x ∉ v"
            v[i] ← ε
        Senão
            v[i] ← j
    Retorne v′
```

FIGURA 3.5. Solução do Problema 5

TEOREMA 3.5. *O algoritmo da figura 3.5 resolve corretamente o problema 5.*

Vamos analisar a complexidade de tempo deste algoritmo. Temos um *loop* com n repetições. Cada repetição chama BuscaBinária com um vetor com n elementos. Como a complexidade

de tempo de pior caso da função BuscaBinária é $\Theta(\lg n)$, a complexidade total é $O(n \lg n)$. Podemos dizer que a complexidade de tempo é $\Theta(n \lg n)$ pois a busca binária pode levar o tempo de pior caso em todas as chamadas. Em toda esta análise consideramos que a função f pode ser computada em tempo $O(1)$. Com isto temos:

TEOREMA 3.6. *O algoritmo da figura 3.5 tem complexidade de tempo de pior caso* $\Theta(n \lg n)$.

Note que, caso o vetor de entrada v não estivesse ordenado, poderíamos ordená-lo em tempo $\Theta(n \lg n)$ antes de iniciarmos as buscas binárias. Esta ordenação não alteraria a complexidade assintótica do procedimento completo, que permaneceria $\Theta(n \lg n)$. Em muitos problemas em que a entrada não está ordenada da maneira que desejamos, vale a pena iniciarmos ordenando a entrada convenientemente.

3.5. Resumo e Observações Finais

A técnica de busca binária fornece algoritmos extremamente eficientes para diversos problemas. A idéia central é, a cada passo, descartarmos metade (ou alguma outra fração constante) dos elementos da entrada, examinando apenas um número constante de elementos. Isto é possível em casos onde a entrada é fornecida segundo alguma ordenação conveniente. Para buscarmos um elemento em um vetor ordenado, examinamos o elemento central do vetor, e assim podemos determinar qual a metade candidata a conter o elemento procurado. Em vetores ciclicamente ordenados, podemos proceder de forma semelhante, dividindo o vetor. No caso de polígonos convexos, usamos sempre o fato de que, ao unirmos dois vértices quaisquer de um polígono convexo, os dois novos polígonos obtidos também são convexos. Graças a isso, podemos usar a técnica de busca binária para resolver problemas como o ponto extremo de um polígono convexo em uma dada direção.

Em muitos problemas, a entrada não é fornecida ordenada. Pode valer a pena ordená-la, para que se possa usar a técnica de busca binária. Devido a alta performance prática dos algoritmos de ordenação e sua complexidade de $\Theta(n \lg n)$, pode-se pensar em um vetor ordenado como uma estrutura de dados extremamente simples e eficiente.

Um caso que não foi estudado aqui, é quando o número de elementos que podem conter a solução é desconhecido ou muito maior que a posição onde se espera encontrar a solução. Uma alternativa eficiente nesses casos é examinar os elementos segundo uma progressão geométrica. Chamamos este procedimento de busca ilimitada. Por exemplo, examinamos inicialmente o elemento v_4, em seguida v_8, v_{16} e assim por diante, até descobrirmos que o elemento que procuramos tem índice menor do que o examinado. Procedemos então com a busca binária tradicional. Nesse caso, o algoritmo é sensível a saída, tendo complexidade de tempo em função do índice do elemento procurado no vetor. Pode-se usar esta técnica, por exemplo, para encontrar o máximo de uma função.

Uma alternativa a busca binária é usarmos interpolação. Imagine que desejamos encontrar a palavra "bola" em um dicionário de 1000 páginas. Certamente não vamos começar examinando a página 500, mas sim uma página próxima do início, como 100. A busca por interpolação pode ser extremamente eficiente quando o vetor em que a busca é realizada tem estrutura previsível, porém, há casos em que a busca por interpolação tem complexidade de tempo linear, e não logarítmica como a busca binária, sendo muito ineficiente.

Uma aplicação geral de busca binária é quando desejamos encontrar o maior valor para o qual uma determinada propriedade é válida. Muitas vezes, é mais simples escrever um algoritmo que teste se a propriedade vale para um valor dado. Podemos então fazer uma busca ilimitada para encontrar o menor valor para o qual a propriedade falha.

Exercícios

3.1) Escreva o pseudo-código do algoritmo de busca binária em vetor sem usar recursão.

3.2) Ache o erro na demostração abaixo, que prova que a complexidade da busca binária é $O(\lg\lg n)$:

Seja $T(n)$ o tempo do algoritmo de busca binária em um vetor com n elementos, no pior caso. Temos:

$$T(n) = T(\lceil (n-1)/2 \rceil) + O(1)$$

$$T(n) = O(1), n \leqslant 1$$

Vamos supor, para obter uma prova por indução, que $T(i) = O(\lg\lg n)$ para $i \leqslant n$. Vamos calcular $T(n+1)$. Temos: $T(n+1) = T(\lceil n/2 \rceil) + O(1) = O(\lg\lg(\lceil n/2 \rceil)) + O(1)$. Como $\lg\lg(\lceil n/2 \rceil) \preceq \lg\lg(n+1)$ temos $T(n+1) = O(\lg\lg(n+1)) + O(1) = O(\lg\lg(n+1))$, finalizando a indução.

3.3) Escreva um algoritmo eficiente que receba como entrada um vetor $v = (v_1, \ldots, v_n)$ de números inteiros e responda se existe $v_i = i$. Analise a complexidade de tempo do seu algoritmo e prove que ele funciona.

3.4) Projete um algoritmo que recebe como entrada um polígono convexo P armazenado em um vetor e dois pontos u e v. O algoritmo deve retornar o vértice p_i de P que minimiza $\circlearrowleft (u, v, p_i)$. A complexidade de tempo deve ser $O(\lg |P|)$. Prove que o seu algoritmo funciona.

3.5) Projete um algoritmo que recebe como entrada um polígono convexo P armazenado em um vetor e um ponto u. O algoritmo deve responder se u está ou não no interior de P em tempo $O(\lg |P|)$.

3.6) Dados dois vetores de números reais em ordem crescente, escreva dois algoritmos, um deles baseado em busca binária e o outro não, para dizer se os dois vetores possuem algum elemento em comum. Analise a complexidade dos algoritmos em função de m e n, os tamanhos dos dois vetores. Quando é mais vantajoso usar cada um dos algoritmos?

3.7) Dados uma função real $f(x)$ e um valor α, o problema de achar uma raiz da função consiste em encontrar um valor de x tal que $|f(x)| < \alpha$. Escreva um algoritmo que resolve o problema usando busca binária caso $f(0) < 0$ e $f(1) > 0$. Escreva um algoritmo mais completo que resolva o problema para qualquer função que possua somente uma raiz, ou seja, existe apenas um valor de x tal que $f(x) = 0$. Nesse último caso, deve-se usar busca binária ilimitada em duas direções simultaneamente, e a complexidade de tempo deve depender do módulo do valor da raiz.

3.8) Neste exercício, o algoritmo que você deve projetar não é para ser usado por um computador. Embora a técnica de busca binária não apareça neste problema, o exercício trabalha análise de complexidade e conceitos de busca ilimitada. Imagine que você foi colocado em um corredor com infinitas portas para ambos os lados (pelo menos você não avista final). Você sabe que existe uma porta que leva a saída, mas não parece fácil encontrá-la, pois todas as portas que você abriu até então são fraudes, levando a uma parede de tijolos. Escreva um algoritmo que defina como você deve caminhar para examinar as portas, de modo a andar, no total, apenas $O(d)$ metros, onde d é a distância entre sua posição inicial e a porta que leva a saída.

*3.9) Projete um algoritmo com complexidade de tempo sub-linear ($o(n)$) ou prove que isto é impossível. A entrada é um polígono convexo P com n vértices armazenado em um vetor e um ponto u.

(a) O algoritmo deve retornar o vértice de P mais próximo de u.

(b) O algoritmo deve retornar o ponto do interior ou bordo de P mais próximo de u.

*3.10) Modifique o procedimento BuscaBinária substituindo a linha "$meio \leftarrow \lfloor (inicio + fim)/2 \rfloor$" por "$meio \leftarrow$ variável aleatória inteira com distribuição uniforme de $inicio$ a fim". O algoritmo continua retornando sempre a resposta certa? Calcule a complexidade assintótica de $E[T(n)]$, o valor esperado do tempo do algoritmo para uma entrada de tamanho n.

CAPíTULO 4

Método Guloso

O método guloso consiste em construir a solução aos poucos, mas sem nunca voltar atrás. Uma vez que o método guloso define que um elemento está na solução do problema, este elemento não será retirado jamais. O método procede acrescentando novos elementos, um de cada vez.

4.1. Fecho convexo: Algoritmo de Jarvis

Já vimos como é conveniente trabalhar com polígonos convexos. Muitas vezes, não temos um polígono convexo, ou sequer um polígono, mas apenas um conjunto S de pontos no plano. Digamos, por exemplo, que desajamos responder uma série de consultas de pontos extremos de S para várias direções d. Neste caso, vale a pena descobrir qual o menor polígono convexo P que envolve os pontos de S. Este polígono P é chamado de fecho convexo de S e está ilustrado na figura 4.1(a). Este é o nosso problema:

PROBLEMA 6. *Dado um conjunto S de pontos no plano, determinar P o fecho convexo de S.*

É geometricamente intuitivo que todos os vértices de P são pontos de S, mas nem todos os pontos de S precisam ser vértices de P. Este fato pode ser provado usando combinação linear. O fecho convexo pode ser também definido como o polígono convexo que tem todos os seus vértices em S e contém todos os pontos de S em seu interior.

A idéia do método guloso é adicionar um elemento de cada vez na solução e, jamais, remover algum elemento dela. Neste problema, vamos começar descobrindo um primeiro vértice de P. Isto é fácil, o ponto extremo em qualquer direção é um vértice de P. Podemos pegar o ponto de S mais a direita (de maior coordenada x) e chamar este ponto de v_1. Não podemos usar o algoritmo da sessão 3.3 para encontrar um vértice extremo porque ainda não temos um polígono convexo. Ainda assim, podemos resolver o problema em tempo linear inspecionando todos os vértices e retornando o de maior coordenada x.

Já temos um vértice. Como fazemos para achar outro? Geometricamente, podemos inclinar aos poucos a reta vertical que passa por v_1 até que ela toque outro vértice $v_2 \in S$, ou seja, pegamos o ponto $v_2 \in S$ que minimiza a função $\circlearrowleft(v_1 - (0,1), v_1, v_2)$ (a função $\circlearrowleft(p_1, p_2, p_3)$ está definida na sessão 3.3 e retorna o ângulo $\widehat{p_1p_2p_3}$ medido no sentido anti-horário). Para acharmos o próximo vértice, pegamos o ponto $v_3 \in S$ que minimiza $\circlearrowleft(v_1, v_2, v_3)$ e assim por diante, até retornarmos ao ponto v_1 (figura 4.1(b)). Caso tenhamos mais de um ponto $v \in S$ com o mesmo valor de $\circlearrowleft(v_i, v_{i+1}, v)$, devemos escolher, dentre esses, o mais distante de v_{i+1}. Este algoritmo se chama algoritmo de Jarvis ou embrulho para presente.

Vamos provar que o algoritmo de Jarvis funciona, isto é, encontra um polígono convexo cujos vértices pertencem a S e todos os pontos de S estão no interior deste polígono. Para isto, usaremos a definição de que um polígono $v = (v_1, \ldots, v_n)$ é convexo se $\circlearrowleft(v_{i-1}, v_i, v_{i+1}) < 180°$ para i de 1 a n com os índices módulo n (ver página 30 para explicação de índices módulo n).

TEOREMA 4.1. *O polígono $P = (v_1, \ldots, v_{|P|})$ gerado pelo algoritmo de Jarvis tendo como entrada um conjunto S de pontos no plano é realmente o fecho convexo de S.*

DEMONSTRAÇÃO. Claramente, $P \subseteq S$. Vamos chamar de $P' = (v'1, \ldots, v'_{|P'|})$ o fecho convexo de S. Primeiro assumimos que $v'_1 = v_1$, pois como v_1 é um ponto extremo de S, então qualquer polígono com vértices em S que contenha v_1 em seu interior tem que ter v_1 como vértice. Sem perda de generalidade podemos dizer que $v'_1 = v_1$.

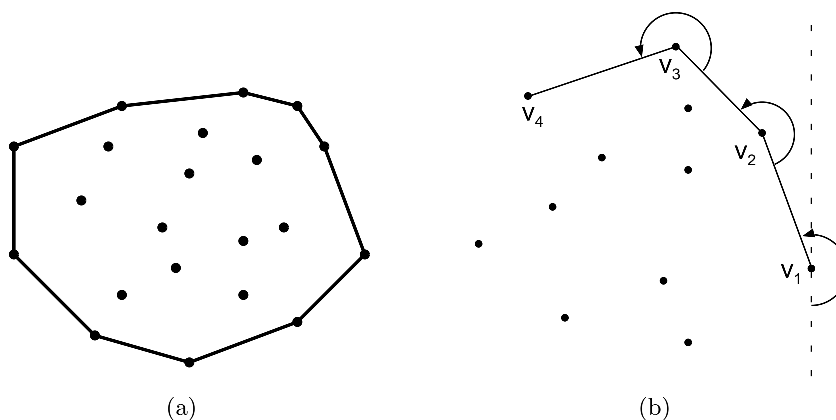


FIGURA 4.1. (a) Fecho convexo de um conjunto de pontos no plano. (b) Algoritmo de Jarvis fazendo o "embrulho para presente".

Vamos supor que v'_i seja o primeiro vértice de P' tal que $v'_i \neq v_i$. Se o ângulo $\circlearrowleft(v_{i-2}, v_{i-1}, v_i) > \circlearrowleft (v_{i-2}, v_{i-1}, v'_i)$ temos um absurdo, pois o vértice escolhido teria sido v'_i e não v_i. Se o ângulo $\circlearrowleft(v_{i-2}, v_{i-1}, v_i) < \circlearrowleft (v_{i-2}, v_{i-1}, v'_i)$ então podemos traçar a reta r que passa por $v_{i-1} = v'_{i-1}$ e por v'_i. Como esta reta r contém uma aresta de P' e separa os pontos v_{i-2} e v_i, então ou P' não é convexo, ou P' não contém v_i em seu interior. Ambas as possibilidades são absurdas, pois P' é o fecho convexo convexo de S. Nesta argumentação consideramos $i > 2$. Caso $i = 2$ devemos considerar um ponto auxiliar $v_0 = v'_0 = v_1 - (0, 1)$. Também consideramos que não há empate na avaliação da função $\circlearrowleft(p_1, p_2, p_3)$. Podemos colocar como critério de desempate a distância de p_2 a p_3. □

Qual a complexidade de tempo do algoritmo de Jarvis? Cada passo leva tempo $\Theta(|S|)$, tanto no melhor quanto no pior caso, e, no pior caso, todos os pontos de S são vértices de P. Então a complexidade de tempo no pior caso é $\Theta(|S|^2)$. Normalmente, porém, $|P|$ é muito menor que $|S|$. Por isso é útil dizer que a complexidade de tempo deste algoritmo é $\Theta(|S||P|)$. Dizemos que este algoritmo é sensível a saída, pois sua complexidade é medida não só em função do tamanho da entrada, mas também em função do tamanho da saída.

TEOREMA 4.2. *A complexidade de tempo do pior caso do algoritmo de Jarvis é $\Theta(nh)$, onde n é o número de pontos de entrada e h é o número de vértices do fecho convexo.*

4.2. Árvore geradora mínima: Algoritmo de Prim

Um problema de grafos com diversas aplicações em telecomunicações, redes de computadores, confecção de placas de circuitos etc é o da árvore geradora mínima. Alguns termos de grafos serão necessários agora. Se você não está familiar com eles a notação básica para grafos está na sessão 2.2.

PROBLEMA 7. *Seja G um grafo com pesos reais nas arestas, obter a árvore geradora mínima de G.*

Seja G um grafo com pesos reais nas arestas. Chamamos o peso $c(e)$ de uma aresta e de custo de e. Uma árvore geradora de G é uma árvore T com $V(T) = V(G)$ e $E(T) \subseteq E(G)$ (conseqüentemente $|E(T)| = |V(G) - 1|$). O custo desta árvore $c(T)$ é definido como:

$$c(T) = \sum_{e \in E(T)} c(e).$$

Uma árvore geradora mínima de G é uma árvore geradora T de G que minimiza $c(T)$. Note que esta árvore não precisa ser única. Vamos construir esta árvore usando um algoritmo guloso. Neste caso, não é muito intuitivo que o algoritmo funcione. Precisamos provar com cuidado este

fato, ou seja, que o algoritmo realmente encontra uma árvore geradora (fácil) e que esta árvore geradora é mínima (bem mais difícil).

Começamos pegando uma aresta que sabemos pertencer a uma árvore geradora mínima de G. Que aresta poderíamos escolher? Uma opção é a aresta de custo mínimo, mas escolheremos uma outra opção. Vamos escolher um vértice v qualquer e pegar a aresta e_1 incidente a v que tenha custo mínimo. Será que e_1 realmente pertence a alguma árvore geradora mínima de G? Vamos chamar de T' uma árvore geradora mínima de G que não contém e_1 e usar esta árvore para construir uma árvore geradora mínima de G que contém e_1. Adicionando e_1 a T' obtemos um grafo com um único ciclo que, abusando da notação, chamamos de $T' \cup \{e_1\}$. Qualquer aresta removida deste ciclo faz com que obtenhamos uma árvore geradora, pois quebraremos o único ciclo do grafo. Se removemos uma aresta e' incidente a v deste ciclo obtemos uma árvore com custo $c(T' \cup \{e_1\} - \{e'\}) = c(T') + c(e_1) - c(e')$. Como e' também é incidente a v, temos $c(e_1) \leqslant c(e')$ e $c(T' \cup \{e_1\} - \{e'\}) \leqslant c(T')$. Portanto existe árvore geradora $T' \cup \{e_1\} - \{e'\}$ de custo mínimo que contém e_1.

Como podemos obter a próxima aresta? Também existem várias respostas que funcionam para esta questão (ver exercício 4.3, para um outro exemplo). Vamos pegar uma aresta que seja adjacente em exatamente um dos extremos às arestas já escolhidas para a árvore geradora mínima e que tenha custo mínimo (figuras 4.3(b) e 4.3(c)). Este é o algoritmo de Prim (pseudo-código na figura 4.2), que funciona devido ao teorema que veremos a seguir.

Entrada:
  G: Grafo conexo com custos associados às arestas.
Saída:
  T: Árvore geradora mínima de G.

```
Prim(G)
    V(T) ← um vértice qualquer de V(G)
    Enquanto |V(T)| ≠ |V(G)|
        Encontre aresta (u, v) com u ∈ V(T) e v ∉ V(T) de custo mínimo
        Adicione v à V(T) e (u, v) à E(T)
    Retorne T
```

FIGURA 4.2. Solução do Problema 7

TEOREMA 4.3. *Seja T' uma árvore geradora mínima de G e T uma árvore tal que $E(T) \subset E(T')$. Seja $e = (v_1, v_2)$ uma aresta de G que tenha custo mínimo dentre as arestas (v_1, v_2) tais que $v_1 \in V(T)$ e $v_2 \notin V(T)$. Então existe uma árvore geradora mínima de G que contém $E(T) \cup \{e\}$.*

DEMONSTRAÇÃO. Suponha que T' seja uma árvore geradora mínima de G que não contém e mas contém todas as arestas de T. Caso não exista T', o teorema já está provado. Se adicionamos e a T' obtemos o grafo $T' \cup \{e\}$ que contém um único ciclo. Como $v_1 \in V(T)$, $v_2 \notin V(T)$ e T é uma árvore, então existe uma aresta $e' = (v_1', v_2')$ neste ciclo tal que $v_1' \in V(T)$ e $v_2' \notin V(T)$, como ilustra a figura 4.3(a). O custo da árvore obtida a partir de T' pela remoção de e' e adição de e é $c(T' \cup \{e\} - \{e'\}) = c(T') + c(e) - c(e')$. Como $c(e) \leqslant c(e')$ então $c(T' \cup \{e\} - \{e'\}) \leqslant c(T')$. Portanto existe árvore geradora $T' \cup \{e\} - \{e'\}$ de custo mínimo que contém $E(T) \cup \{e\}$. □

Existem várias maneiras de implementar este algoritmo, usando diferentes estruturas de dados. Cada implementação tem uma complexidade de tempo diferente. A alternativa mais simples não usa nenhuma estrutura de dados sofisticada, usando apenas vetores, como mostra a figura 4.4.

Basta olharmos para os *loops* para vermos que o algoritmo tem complexidade de tempo $\Theta(n^2)$, onde n é o número de vértices do grafo. Considerando apenas n, isto é o melhor que

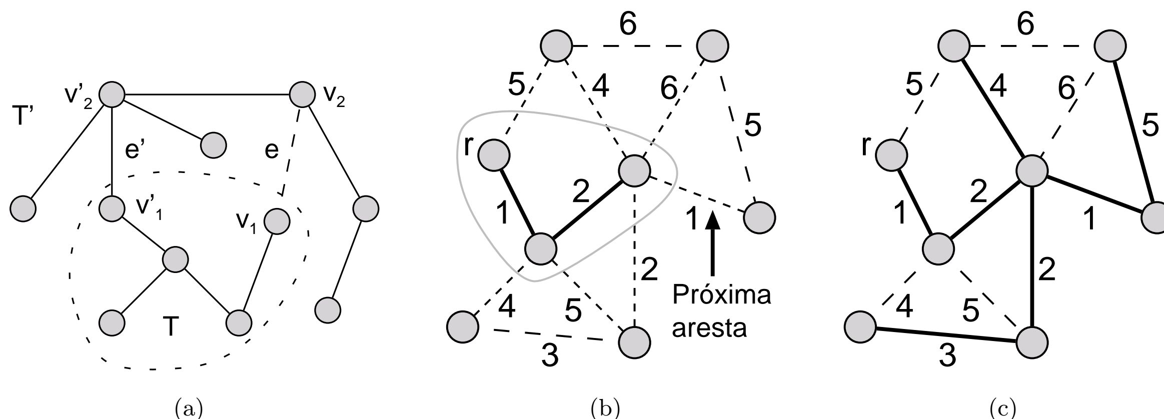


FIGURA 4.3. (a) Ilustração referente a prova do teorema 4.3. (b) Execução do algoritmo de Prim na 3ª iteração. (c) Ávore gerada pelo algoritmo de Prim.

Entrada:

G: Grafo conexo com custo $custo(u, v)$ associado a toda aresta (u, v). Se $(u, v) \notin E(G)$, então $custo(u, v) = \infty$.

Saída:

T: Árvore geradora mínima de G.

```
PrimVetor(G)
    V(T) ← v ← um vértice qualquer de V(G)
    Marcar v
    Para todo vértice u ≠ v
        u.custo ← G.custo(u, v)
        u.vizinho ← v
    Enquanto |V(T)| ≠ |V(G)|
        v ← vértice não marcado com menor custo
        Adicione v à V(T) e (u, v) à E(T)
        Marcar v
        Para todo vértice u não marcado
            Se custo(u, v) < u.custo
                u.custo ← G.custo(u, v)
                u.vizinho ← v
    Retorne T
```

FIGURA 4.4. Solução do Problema 7 usando apenas vetores.

---

podemos fazer, pois o número de arestas do grafo pode ser $\Theta(n^2)$ e todas precisam ser examinadas. Na prática, grafos esparços (poucas arestas) são muito mais comuns do que grafos densos. Por isso, seria bom que conseguíssemos expressar a complexidade de tempo não só em função de n, mas também em função do número de arestas m. O algoritmo de Prim pode ser facilmente modificado para ter complexidade de tempo $O(m \lg n)$, usando um *heap* binário. O novo pseudo-código está na figura 4.5.

São executadas $O(n)$ inserções, $O(n)$ extrações de mínimo e $O(m)$ reduções de custo no *heap*. Em um *heap binário*, todas estas operações levam tempo $O(\lg n)$, totalizando tempo $O(m \lg n)$, pois como G é conexo $m \geqslant n - 1$. Na teoria, podemos usar um *heap de Fibonacci*, onde o tempo amortizado da operação de redução de custo é $O(1)$. Neste caso, a complexidade de tempo do

Entrada:
G: Grafo conexo com custo $custo(u, v)$ associado a toda aresta (u, v). Se $(u, v) \notin E(G)$, então $custo(u, v) = \infty$.
Saída:
T: Árvore geradora mínima de G.
Observações:
H: Heap mínimo que armazena vértices usando como chave o campo $custo$.

```
PrimHeap(G)
    V(T) ← v ← um vértice qualquer de V(G)
    Marcar v
    Para todo vértice u ≠ v
        u.custo ← G.custo(u, v)
        u.vizinho ← v
        Inserir(H,u)
    Enquanto |V(T)| ≠ |V(G)|
        v ← ExtrairMínimo(H)
        Adicione v à V(T) e (u, v) à E(T)
        Marcar v
        Para todo vértice u não marcado e adjacente à v
            Se custo(u, v) < u.custo
                ReduzirCusto(H, u, G.custo(u, v))
                u.vizinho ← v
    Retorne T
```

FIGURA 4.5. Solução do Problema 7 usando um heap.

algoritmo fica $O(n \lg n + m)$. Na prática, porém, as costantes multiplicativas no tempo do *heap de Fibonacci* tornam-o mais lento do que o *heap binário* para qualquer grafo tratável.

4.3. Compactação de dados: Árvores de Huffman

Vamos estudar agora um problema de compactação de dados. Nós vamos nos concentrar em arquivos de texto, para simplificar os exemplos, mas as técnicas estudadas aqui independem do tipo de arquivo. Para armazenar um arquivo de texto em um computador cada caractere é armazenado em um *byte* (8 *bits*). Certamente, nem todos os $2^8 = 256$ caracteres possíveis são usados.

Uma alternativa fácil para reduzir o tamanho do arquivo é verificar se menos de 2^k caracteres são usados, usando apenas k *bits* neste caso, mas isto não compactará praticamente nada. Uma técnica mais inteligente é considerar as freqüências dos caracteres e codificar cada caractere com um número diferente de *bits*. Vejamos um exemplo.

Queremos compactar a palavra *"cabana"*. As freqüências dos caracteres nesta palavra são $f(c) = f(b) = f(n) = 1/6$, $f(a) = 1/2$. Temos 4 caracteres, então poderíamos usar 2 *bits* por caractere, totalizando 12 *bits*. Outra opção é usar os seguintes códigos: c : 000, b : 001, n : 01, a : 1. Com isto obtemos a palavra compactada 00010011011 com 11 bits. Não parece que ganhamos muito, mas este exemplo é pequeno e contém pouca redundância (tente algo semelhante com a palavra *"aaabaaacaaad"*). De fato, a compactação de Huffman sozinha não é muito eficiente, mas ela está presente como parte importante de praticamente todos os melhores compactadores usados atualmente.

Um ponto importante é como podemos decodificar a mensagem. Primeiro, precisamos saber o código de cada caractere. Depois veremos como fornecer esta informação. Além disso, precisamos ter uma maneira de descobrir quando acaba um caractere e começa outro. Vejamos o

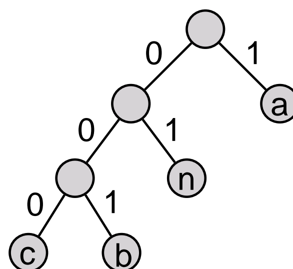


FIGURA 4.6. Árvore de Huffman da palavra *"cabana"*.

exemplo do parágrafo anterior. Começamos lendo 0. Os caracteres iniciados por 0 são c, b e n. Lemos então outro 0. Agora sabemos que se trata ou de um c ou um b. Ao lermos 0 novamente, temos certeza que é um c. Isto é possível porque geramos um código de prefixo. Um código de prefixo é uma atribuição de uma seqüência distinta de *bits* para cada caractere de modo que uma seqüência não seja um prefixo da outra. Um código ser de prefixo é uma condição suficiente para permitir que qualquer mensagem escrita com ele possa ser decodificada de modo único, mas não é uma condição necessária para isto. Não provaremos aqui, mas, se estamos limitados a atribuir uma seqüência de um número inteiros de *bits* para cada caractere, sempre existe código de prefixo que usa o mínimo de *bits* para codificar a mensagem dentre todos os códigos que permitiriam que qualquer mensagem escrita com ele fosse decodificada de modo único. Assim, não estamos perdendo nada por nos limitarmos a códigos de prefixo.

Existe uma relação direta entre árvores binárias e códigos de prefixo. A cada folha podemos associar um caractere. Cada *bit* indica se devemos seguir para a direita ou esquerda na árvore. A árvore correspondente ao código para a palavra *cabana* está na figura 4.6.

Vamos chamar de $C = \{c_1, \ldots, c_n\}$ o nosso conjunto de caracteres e de $f(c_i)$ a freqüência do caractere c_i. Queremos construir uma árvore binária T que tenha como conjunto de folhas o conjunto C e que minimize

$$c(T) = \sum_{i=1}^{n} f(c_i) l(c_i),$$

onde $l(c_i)$ é o nível da folha c_i, definido como sua distância até a raiz, ou seja, $l(c_i)$ é o número de *bits* usados para codificar c_i. Esta árvore é chamada de árvore de Huffman de C. Chamamos $c(T)$ de custo da árvore T. Note que as freqüências dos caracteres podem ser multiplicadas livremente por qualquer constante, por isso podemos usar freqüências que somem 1 ou o número de aparições de cada caractere livremente.

PROBLEMA 8. *Dado um conjunto de caracteres C, com suas freqüências, construir uma árvore de Huffman de C.*

O algoritmo que resolve este problema é bem diferente dos outros algoritmos gulosos que vimos. No problema de fecho convexo, queríamos encontrar um conjunto de vértices (embora ordenado) que satisfizesse certa propriedade. No problema de árvore geradora mínima, queríamos encontrar um conjunto de arestas que satisfizesse uma propriedade. Agora, a nossa solução não é mais um conjunto. O algoritmo que apresentaremos não é nem um pouco intuitivo, tanto que, por muitos anos, foi utilizado para resolver este problema um algoritmo que encontrava árvores sub-ótimas.

Começamos criando um nó para cada caractere. Estas serão as folhas da árvore e têm como freqüência a própria freqüência do caractere. A cada passo do algoritmo, os dois nós de menor freqüencia são colocados como filhos de um novo nó. A freqüência do novo nó é a soma das freqüências de seus filhos. Este procedimento está ilustrado no pseudo-código da figura 4.7 e exemplificado na figura 4.8.

Entrada:
C: Vetor de caracteres a serem codificados.
f: Vetor com as freqüencias dos caracteres.
Saída:
Vértice raiz da árvore de Huffman de C.
Observações:
H: Heap mínimo que armazena vértices usando como chave o campo $freq$.

Huffman(C, f)
Para i de 1 até $|C|$
$v \leftarrow$CriarVértice()
$v.caractere \leftarrow C[i]$
$v.freq \leftarrow f[i]$
Inserir(H, v)
Para i de 1 até $|C| - 1$
$v \leftarrow$CriarVértice()
$v.dir \leftarrow$ ExtrairMínimo(H)
$v.esq \leftarrow$ ExtrairMínimo(H)
$v.freq \leftarrow v.dir.freq + v.esq.freq$
Inserir(H, v)
Retorne ExtrairMínimo(H)

FIGURA 4.7. Solução do Problema 8

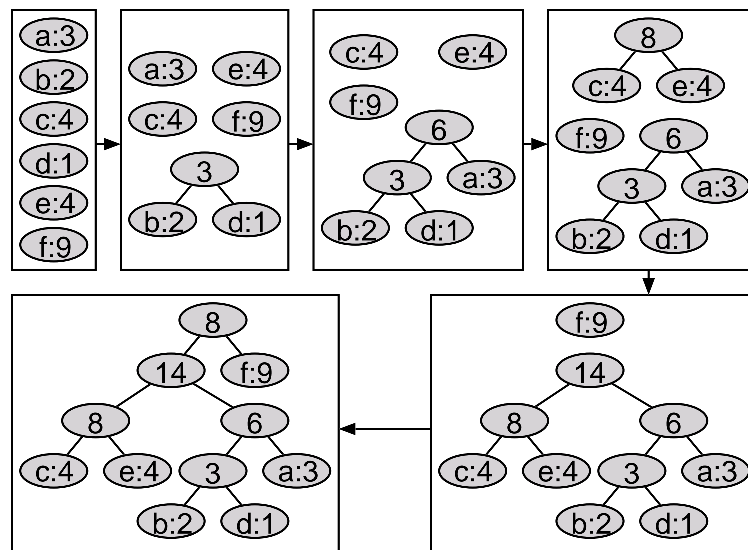


FIGURA 4.8. Algoritmo de Huffman passo a passo, para uma entrada $C = (\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}, \mathsf{f})$ e $f = (3, 2, 4, 1, 4, 9)$.

Para fazer a análise de complexidade notamos que o *loop* é repetido n vezes em um alfabeto com n caracteres e a cada repetição as operações feitas no *heap* levam tempo $O(\lg n)$. No total temos a complexidade de tempo de $O(n \lg n)$.

A prova de que o algoritmo funciona, ou seja, de fato gera uma árvore de Huffman, é bem mais complicada. Esta prova será feita por indução. Primeiro, provaremos que podemos colocar os dois vértices de menor freqüência como folhas irmãs na árvore. Em seguida, provaremos que podemos considerar cada árvore de Huffman já construída pelo algoritmo como um único vértice

cuja freqüência é a soma das freqüências dos dois filhos, na criação de uma árvore maior. Antes disso, provaremos que toda árvore de Huffman é estritamente binária.

LEMA 4.4. *Toda árvore de Huffman é estritamente binária.*

DEMONSTRAÇÃO. Por definição a árvore de Huffman é binária. Suponha que um vértice v tenha apenas um filho. Neste caso podemos remover v da árvore, fazendo que o filho de v seja filho do pai de v. A árvore obtida tem custo menor do que uma árvore de Huffman, o que é absurdo. □

LEMA 4.5. *Seja $C = \{c_1, \ldots, c_n\}$ um alfabeto onde todo caractere c_i tem freqüência $f(c_i)$ e $f(c_i) \leqslant f(c_{i+1})$. Neste caso, existe árvore de Huffman onde c_1 e c_2 são folhas irmãs.*

DEMONSTRAÇÃO. Usaremos uma árvore de Huffman T' onde c_1 e c_2 não são folhas irmãs para construir uma árvore de Huffman T onde c_1 e c_2 são folhas irmãs. Como T' é estritamente binária, existem pelo menos duas folhas irmãs no último nível de T'. Como c_1 e c_2 são os dois caracteres menos freqüêntes, se colocarmos c_1 e c_2 nestas duas folhas, e colocarmos os caracteres que estavam nelas nas posições de c_1 e c_2, obtemos uma árvore T com $c(T) \leqslant c(T')$. □

LEMA 4.6. *Se T_C é uma árvore de Huffman para o alfabeto $C = \{c_1, \ldots, c_n\}$, então $T_{C'}$, obtida acrescentando dois filhos c'_1 e c'_2 a uma folha c_k de T_C onde $f(c_k) = f(c'_1) + f(c'_2)$ e $f(c'_1), f(c'_2) \leqslant f(c_i)$ para $1 \leqslant i \leqslant n$, é uma árvore de Huffman para o alfabeto $C - \{c_k\} \cup \{c'_1, c'_2\}$.*

DEMONSTRAÇÃO. O custo de $T_{C'}$ é $c(T_{C'}) = c(T_C) + f(c'_1) + f(c'_2)$, então $c(T_C) = c(T_{C'}) - f(c'_1) - f(c'_2)$. Para obter um absurdo, suponha que $T'_{C'}$ seja uma árvore de Huffman de C' com $c(T'_{C'}) < c(T_{C'})$. Pelo lema 4.5 podemos considerar que c'_1 e c'_2 são folhas irmãs em $T'_{C'}$. Removendo estas duas folhas c'_1 e c'_2 e atribuindo o caractere c_k ao pai delas obtemos uma árvore T'_C com $c(T'_C) = c(T'_{C'}) - f(c'_1) - f(c'_2) < c(T_C)$, o que contradiz o fato de $c(T_C)$ ser uma árvore de Huffman para C. □

TEOREMA 4.7. *O algoritmo gera uma árvore de Huffman para C.*

DEMONSTRAÇÃO. O lema 4.5 é a base da indução. O lema 4.6 fornece o passo indutivo. Note que provamos que a árvore é uma árvore de Huffman na ordem contrária a que ela é construída. Começamos provando que a raiz com seus dois filhos é uma árvore de Huffman e vamos descendo na árvore, como mostra a figura 4.8. □

Para que o descompactador decodifique um arquivo compactado com o código de Huffman ele precisa ter conhecimento da árvore. Analisaremos 4 alternativas para este problema:

1) Usar uma árvore pré-estabelecida, baseada em freqüências médias de cada caractere. Esta técnica só é viável em arquivos de texto. Ainda assim, de um idioma para outro a freqüência de cada caractere pode variar bastante.
2) Fornecer a árvore de Huffman, direta ou indiretamente, no início do arquivo. A árvore de Huffman para 256 caracteres pode ser descrita com 256 caracteres mais 511 *bits* usando percurso em árvore. Outra opção mais simples é informar a frequência de cada caractere e deixar que o descompactador construa a árvore. É necessário cuidado para garantir que a árvore do compactador e do descompactador sejam idênticas.
3) Fornecer a árvore de Huffman, direta ou indiretamente, para cada bloco do arquivo. Esta técnica divide o arquivo em blocos de um tamanho fixo e constrói árvores separadas para cada bloco. A vantagem é que se as freqüências dos caracteres são diferentes ao longo do arquivo, pode-se obter maior compactação. A desvantagem é que várias árvores tem que ser fornecidas, gastando espaço e tempo de processamento.
4) Usar um código adaptativo. Inicia-se com uma árvore em que todo caractere tem a mesma freqüência e, a cada caractere lido, incrementa-se a freqüência deste caractere, atualizando a árvore. Neste caso não é necessário enviar nenhuma árvore, mas não há compactação significativa no início do arquivo. Não apresentamos aqui algoritmo para fazer esta atualização na árvore eficientemente.

4.4. Compactação de dados: LZSS

Uma técnica simples que produz bons resultados de compactação é o método chamado LZSS. Este método, completamente diferente do método de Huffman, se baseia no fato de algumas seqüências de caracteres se repetirem ao longo do arquivo. A idéia é, ao invés de escrevermos todos os caracteres do arquivo explicitamente, fazermos referências a seqüências anteriores. Modelaremos formalmente esta idéia a seguir. Os primeiros modelos não fornecem um compactador eficiente, mas ajudam a entender as idéias centrais da técnica.

Temos como entrada uma sequência de caracteres (o arquivo descompactado) e queremos gerar uma seqüência de simbolos correspondente ao arquivo (o arquivo compactado). Um símbolo ou é um caracter ou um par (p, l). Temos que incluir no arquivo uma maneira de distinguir entre estes dois tipos de símbolo, mas só discutiremos este detalhe bem mais tarde. O significado de um par (p, l) é uma referência a posições anteriores do arquivo: os l caracteres iniciados a partir de p caracteres anteriores no arquivo descompactado. Vejamos alguns exemplos:

Descompactado1: métodogulosogulosométodo

Compactado1: métodoguloso(6,6)(18,6)

Descompactado2: bananadadebanana

Compactado2: ban(2,3)dade(10,6)

Note que em um símbolo (p, l), p nunca pode referenciar um caractere que ainda não foi codificado, portanto $p \geqslant 1$. Podemos também forçar $l \geqslant 2$, pois preferimos escrever um único caractere explicitamente a referenciá-lo. O descompactador para este arquivo é bem simples e seu pseudo código está na figura 4.9. Neste exemplo trabalhamos com vetores e não arquivos.

<u>Entrada</u>:
  C: Vetor compactado.
  D: Vetor onde será escrito o arquivo descompactado.
<u>Saída</u>:
  Retorna o número de caracteres do arquivo descompactado.

```
DescompactadorLZSS1(C, D)
    Di ← 1
    Para Ci de 1 até |C|
        Se C[Ci] é um caractere
            D[Di] ← C[Ci]
            Di ← Di + 1
        Senão
            Para i de 0 até C[Ci].l − 1
            D[Di] ← D[C[Ci].l + i]
            Di ← Di + 1
    Retorne Di − 1
```

FIGURA 4.9. Descompactador LZSS em vetor.

Temos alguns problemas no método de compactação que definimos. Um deles é como representarmos no arquivo um par (p, l) já que tanto p quanto l podem ser tão grandes quanto o tamanho do arquivo descompactado. Outro problema é que o descompactador teria que voltar no arquivo várias vezes para encontrar as referências, o que tornaria a descompactação lenta. A solução para estes dois problemas é limitarmos o valor de p e de l e usarmos um *buffer* circular em memória. Assim nos limitamos a armazenar em memória as últimas posições escritas no arquivo descompactado. Limitaremos p ao valor p^* $(1 \leqslant p \leqslant p^*)$ e l ao valor l^* $(2 \leqslant l \leqslant l^*)$. O nosso *buffer* precisa armazenar apenas p^* caracteres. O descompactador em arquivo usando um *buffer* circular está ilustrado no pseudo-código da figura 4.10.

Entrada:
C: Arquivo compactado.
D: Arquivo onde será escrito o arquivo descompactado.
p^*: Valor máximo de p em uma símbolo (p, l).
Observações:
A operação $i \mod n$ é definida como: $i \mod n$ é o resto da divisão de i por n se i não é divisível por n e $i \mod n = n$ se i é divisível por n.

```
DescompactadorLZSS2(C, D, p*)
    B ← AlocarVetor(p*)
    Bi ← 1
    Enquanto o arquivo C não tiver chegado ao fim
        c ← LerSímbolo(C)
        Se c é um caractere
            Escrever(D,c)
            B[Bi] ← c
            Bi ← Bi + 1  mod p*
        Senão
            Para i de 1 até c.l
                Escrever(D,B[Bi − c.p]  mod p*)
                B[Bi] ← c
                Bi ← Bi + 1  mod p*
```

FIGURA 4.10. Descompactador LZSS em arquivo.

Vários arquivos compactados diferentes podem corresponder a um mesmo arquivo compactado. Podemos por exemplo listar todos os caracteres explicitamente, não produzindo qualquer compactação. Queremos compactar o máximo possível. Vamos definir o tamanho do arquivo compactado como o número de símbolos que ele contém. Embora esta medida não seja totalmente fiel a realidade, ela é necessária para nossos resultados teóricos e nos leva a bons resultados práticos (para obtermos realmente o menor arquivo possível, teríamos que minimizar a soma dos *bits* gastos, que dependem do símbolo ser um caracter ou um par de valores, mas este problema é bem mais difícil).

PROBLEMA 9. *Dada uma seqüência de caracteres D e dois valores p^* e l^*, encontrar a seqüência de símbolos C correspondente a D que contém o menor número de símbolos com a restrição de que todo símbolo (p, l) satisfaz $1 \leqslant p \leqslant p^*$ e $1 \leqslant l \leqslant l^*$*

O nosso algoritmo guloso é bastante simples. Sempre tentamos gerar o par (p, l) com o maior valor possível de l. Se este valor for 0 ou 1, geramos o caractere explicitamente. Será óbvio que este algoritmo gera o mínimo de símbolos? Vejamos um exemplo de variação do problema onde o método guloso não funciona bem.

Uma variação é quando temos um dicionário definido e ou fornecemos o caractere explicitamente ou uma referência a palavra no dicionário. Por exemplo se o dicionário contém as palavras: $p_1 = ab$, $p_2 = de$ e $p_3 = bcdef$, então podemos codificar $abcdef$ como $a\ p_3$, mas o método guloso codificaria como $p_1\ c\ p_2\ f$.

Visto isso, parece bem razoável que devemos provar que o método guloso gera o mínimo de símbolos no caso do nosso problema.

TEOREMA 4.8. *O algoritmo guloso gera uma seqüência de simbolos correspondente a entrada com o número mínimo de símbolos possível.*

DEMONSTRAÇÃO. Seja $C_1, \ldots, C_n$ uma seqüência de símbolos gerada pelo algoritmo guloso. Suponha, para obter um absurdo, que $C'_1, \ldots, C'_{n'}$ seja uma outra seqüência correspondente a

entrada com $n' < n$. Definimos o tamanho de um símbolo $|C_i|$ como $|C_i| = 1$ se C_i é um caractere e $|C_i| = l$ se $C_i = (p, l)$. Vamos definir

$$T(k) = \sum_{i=1}^{k} |C_i| \text{ e } T'(k) = \sum_{i=1}^{k} |C'_i|.$$

Como $n' < n$, todo símbolo tem tamanho positivo e $T(n) = T'(n')$, então existe um menor inteiro k tal que $T'(k) > T(k)$, com $1 \leqslant k \leqslant n'$. Claramente $C'_k = (p, l)$. Neste caso, o algoritmo guloso teria escolhido o par $(p, l - T(k-1) + T'(k-1))$ ou algum de tamanho maior. Assim $T(k) = T(k-1) + l - T(k-1) + T'(k-1) = T'(k-1) + l = T'(k)$. Vale notar que como $T'(k-1) \leqslant T(k-1)$ e $l \leqslant l^*$ então $l - T(k-1) + T'(k-1) \leqslant l^*$. □

Vários detalhes práticos foram omitidos na apresentação do problema. Veremos alguns deles, sem nos aprofundarmos.

Para distinguirmos um par (p, l) de um caractere, podemos colocar um *bit* antes de cada símbolo que indica se o símbolo seguinte é um caractere ou um par (p, l). Mais prático, porém, é agruparmos esses *bits* de 8 em 8. Assim temos um *byte* que indica a natureza dos próximos 8 símbolos do arquivo.

Uma escolha comum de p^* e l^* é $p^* = 4096$ (12 *bits*) e deixarmos 4 *bits* para l. É razoável forçarmos $l > 2$, pois, quando $l \leqslant 2$, não ganhamos muito escrevendo um par (p, l). Então podemos fazer $3 \leqslant l \leqslant 18$.

O compactador precisará manter um *buffer* de história com p^* caracteres para encontrar as referências e um *buffer* com os próximos l^* caracteres a serem lidos. Estes dois *buffers* devem ser de fato um único *buffer* circular.

Se fizermos a busca do maior par (p, l) examinando os *buffers* de maneira trivial a complexidade de tempo do compactador será $O(np^*)$, onde n é o número de caracteres do arquivo descompactado.

Para que a busca do maior par (p, l) seja feita eficientemente uma alternativa é usar uma árvore binária balanceada (AVL, rubro-negra, *splay*, *treap*...). Esta árvore deve conter as p^* cadeias de comprimento l^* que podem ser referenciadas. A cada caractere avançado do arquivo de entrada, deve-se atualizar a árvore. Neste caso, a complexidade de tempo do compactador será $O(nl^* \lg p^*)$

4.5. Resumo e Observações Finais

A idéia central do método guloso é construir aos poucos a solução, sem nunca voltar atrás. Muitas vezes, desejamos encontrar um subconjunto da entrada que satisfaz uma propriedade específica, e procedemos incluindo um elemento em cada passo. Normalmente os algoritmos contruídos usando o método guloso são extremamente simples, porém muitas vezes provar que eles funcionam corretamente é uma tarefa delicada, que merece atenção especial.

O algoritmo de Jarvis encontra o fecho convexo de um conjunto de pontos no plano, seguindo sucessivamente pelos pontos do fecho, como um embrulho para presente. Este algoritmo é sensível a saída, ou seja, sua complexidade de tempo não depende somente do tamanho da entrada, mas também do tamanho da saída. No caso de n pontos na entrada e h pontos na saída, a complexidade de tempo do algoritmo de Jarvis é $\theta(nh)$. Os melhores algoritmos conhecidos para este problema, tem complexidade de tempo $\theta(n \lg h)$. É possível provar que não é possível reduzir ainda mais esta complexidade, portanto esses algoritmos de complexidade de tempo $theta(n \lg h)$ são ótimos.

O algoritmo de Prim constrói a árvore geradora de custo mínimo de um grafo. Este algoritmo segue acrescentando sempre a aresta de menor custo que tem um extremo na parte já construída da árvore e outro extremo fora dela. Duas implementações diferentes são estudadas, uma usando heap e outra não, com diferentes complexidades de tempo. A escolha por uma ou outra implementação depende de quão esparso é o grafo, ou seja, quantas arestas tem nele. Uma terceira alternativa usa um heap de Fibonnaci, tendo excelente complexidade de tempo,

porém não sendo muito útil na prática. Este é um exemplo em que a complexidade de tempo assintótica deve ser vista com cautela, pois pode não refletir diretamente a performance prática.

A árvore de Huffman tem muitas aplicações em compactação de dados. O algoritmo que estudamos constrói essa árvore de forma bastante engenhosa, unindo sempre os dois elementos menos freqüentes em um novo elemento cuja freqüência é a soma das freqüências dos filhos.

Outro algoritmo importante em compactação de dados é o LZSS. Estudamos como o método guloso constrói uma seqüência eficiente de referências a palavras de um *buffer*.

Não vimos aqui uma variação do método guloso onde, ao invés de construirmos a solução incrementalmente, construimos a solução decrementalmente. Em outras palavras, no lugar de acrescentarmos aos poucos elementos a solução, vamos descartando elementos que sabemos não pertencer a solução, um de cada vez.

Exercícios

4.1) Escreva o pseudo-código da função $\circlearrowleft(p_1, p_2, p_3)$.

4.2) O envelope superior de um conjunto de retas S no plano cartesiano é a seqüência de segmentos de retas de S com valor y máximo para x variando de $-\infty$ à $+\infty$ (figura 4.11). Escreva um algoritmo guloso que determina o envelope superior de um conjunto de retas no plano. Calcule a complexidade de tempo deste algoritmo em função do tamanho da entrada e da saída.

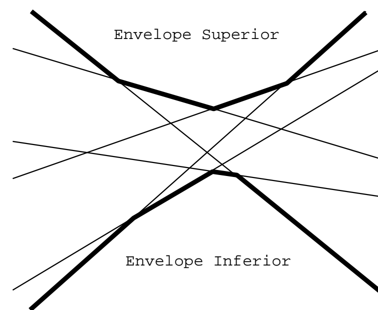


FIGURA 4.11. Envelope superior e envelope inferior de um conjunto de retas.

4.3) O algoritmo de Kruskal resolve o problema da árvore geradora mínima escolhendo a cada passo a aresta de menor custo que não adiciona ciclos ao grafo. Ao longo do algoritmo não temos uma árvore crescendo, mas várias árvores surgindo e crescendo até se unirem em uma única árvore. Prove que este algoritmo resolve corretamente o problema.

4.4) Forneça um único conjunto de caracteres, todos com freqüências distintas, para o qual existem duas árvores de Huffman não isomorfas.

4.5) Dado um grafo direcionado G, escreva um algoritmo para colorir suas arestas de modo que não haja duas arestas com a mesma cor entrando em um vértice nem duas arestas com a mesma cor saindo de um vértice. A solução obtida deve minimizar o número de cores diferentes usadas. Prove que seu algoritmo está correto. A mesma abordagem funciona para o caso do grafo não direcionado?

4.6) Digrafos acíclicos têm diversas aplicações. Eles podem ser usados, por exemplo, para representar o sistema de matérias com pré-requisitos de uma faculdade. Uma informação extremamente útil sobre os digrafos acíclicos é sua ordenação topológica. Dado um digrafo acíclico, uma ordenação topológica dele é uma ordenação de seus vértices onde todas as arestas partam de um vértice anterior para um vértice posterior na ordenação. Escreva um algoritmo guloso que encontre a ordenação topológica de digrafos acíclicos, prove que seu algoritmo está correto e analise sua complexidade de tempo.

4.7) Um problema natural em grafos é encontrar o caminho mais curto entre pares de vértices. Na versão com pesos nas arestas, o comprimento de um caminho é a soma dos pesos das arestas pertencentes a ele. Escreva um algoritmo guloso que, dados um grafo com pesos nas arestas e um vértice v deste grafo, encontre a distância de v a todos os demais vértices do grafo. Prove que seu algoritmo está correto. Sugestão: seu algoritmo deve ser bastante semelhante ao algoritmo de Prim para árvore geradora mínima, mas deve construir a árvore formada pela união dos caminhos mais curtos que partem de v. Este algoritmo é chamado de algoritmo de Dijkstra.

4.8) Um país possue moedas de 1, 5, 10, 25 e 50 centavos. Você deve programar uma máquina capaz de dar troco com essas moedas de modo a fornecer sempre o número mínimo de moedas, qualquer que seja a quantia. Considere que a máquina possui quantidades ilimitadas de todas essas moedas. Prove que seu algoritmo funciona. O algoritmo continuaria funcionando se a máquina tivesse apenas moedas de 1, 10 e 25 centavos?

4.9) Escreva um algoritmo guloso onde: A entrada é um conjunto de palavras (cadeias de caracteres quaisquer) que formam um dicionário D e uma frase f (outra cadeia de caracteres), onde todo caractere de f está em D. A saída é uma seqüência de segmentos de palavras que concatenados formam f. Um segmento de palavra é representado por uma tripla (p, ini, fim) onde p é o número da palavra no dicionário D, ini é o número do primeiro caractere do segmento e fim é o número do último caractere do segmento. Por exemplo: $D = (camelo, aguia, sapo)$, $f = guloso$; saída:$(2, 2, 3), (1, 5, 6), (3, 1, 1), (1, 6, 6)$. Claro que o seu algoritmo deve gerar o mínimo de triplas possível. Prove que o seu algoritmo resolve o problema com este número mínimo de triplas.

4.10) Deseja-se realizar o máximo possível de tarefas de um conjunto de tarefas onde cada tarefa tem um horário de início e um horário de término. Duas tarefas não podem estar sendo realizadas simultaneamente. Toda tarefa que for realizada deverá iniciar exatamente no seu horário de início e terminar exatamente no seu horário de término. Escreva um algoritmo guloso que fornece o maior número possível de tarefas que podem ser realizadas. Prove que seu algoritmo funciona.

*4.11) O fecho convexo de um conjunto de pontos no espaço é o menor poliedro convexo que contém todos estes pontos. Dado um conjunto de n pontos no espaço tridimensional, escreva um algoritmo para determinar seu fecho convexo em tempo $O(nh)$, onde h é o número de vértices do poliedro do fecho convexo.

CAPíTULO 5

Divisão e Conquista

Resolver problemas pequenos é quase sempre mais simples que resolver problemas maiores. É natural dividir um problema grande em sub-problemas menores e resolver cada um dos sub-problemas separadamente. Feito isto, temos que combinar as soluções dos problemas menores para obtermos a solução do problema total. Os algoritmos de divisão e conquista têm, então, três fases: dividir, conquistar e combinar.

Na primeira fase, a divisão, o problema é decomposto em dois (ou mais) sub-problemas. Em alguns algoritmos, esta divisão é bastante simples, enquanto em outros, é a parte mais delicada do algoritmo.

Na segunda fase, a conquista, resolvemos os sub-problemas. A beleza da técnica reside no fato de que os problemas menores podem ser resolvidos recursivamente, usando o mesmo procedimento de divisão e conquista, até que o tamanho do problema seja tão pequeno que sua solução seja trivial ou possa ser feita mais rapidamente usando algoritmos mais simples.

Na terceira fase, a combinação das soluções, temos que unir as soluções dos problemas menores para obtermos uma solução unificada. Este procedimento nem sempre é trivial, e muitas vezes pode ser simplificado se a divisão (primeira fase) for feita de modo inteligente.

5.1. Envelope Superior

O envelope superior de um conjunto de retas S no plano cartesiano é a seqüência de segmentos de retas de S com valor y máximo para x variando de $-\infty$ à $+\infty$ (figura 5.1). O nosso problema é:

Problema 10. *Dado um conjunto S de retas no plano, construa o envelope superior de S.*

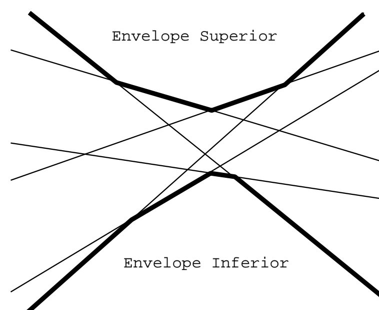


Figura 5.1. Envelope superior e envelope inferior de um conjunto de retas.

Nesta sessão, o nosso algoritmo fará a divisão de modo bastante simples, apenas dividimos S em S_1 e S_2 de mesmo tamanho (ou tamanhos diferindo de no máximo uma unidade, se $|S|$ for ímpar). A parte mais delicada do algoritmo consiste em combinar as duas soluções em uma solução unificada. Queremos resolver então o seguinte problema: dados dois envelopes superiores $U^1 = (U^1_1, \ldots, U^1_{|U^1|})$ e $U^2 = (U^2_1, \ldots, U^2_{|U^2|})$, obter o envelope superior $U = (U_1, \ldots, U_{|U|})$ das retas de $U^1 \cup U^2$. Para combinarmos os dois envelopes superiores, usaremos uma técnica chamada de linha de varredura. Nesta técnica, vamos resolvendo o problema da esquerda para a direita.

Iniciamos comparando os coeficientes angulares das retas U^1_1 e U^2_1, as retas que contém os segmentos mais a esquerda nos envelopes superiores U_1 e U_2. A reta de menor coeficiente angular

dentre U_1^1 e U_1^2 será colocada na posição U_1. Digamos que esta reta seja U_1^1. Seguimos então descobrindo qual a primeira reta que intercepta U_1, examinando apenas U_2^1 e U_1^2, e colocamos esta reta na posição U_2. Repetimos este procedimento até obtermos todo o envelope superior U. O pseudo-código do algoritmo está na figura 5.2.

Entrada:
 U^1: Vetor com retas formando um envelope superior, da esquerda para a direita.
 U^2: Idem, para outro conjunto de retas.
Saída:
 U: Envelope superiror de $U^1 \cup U^2$.
Observações:
 $\measuredangle(r)$: Coeficiente angular da reta r.
 No caso de acessos além do limite dos vetores de entrada, considere que qualquer reta intercepta uma dada reta antes de uma reta inexistente.

```
CombinaEnvelopes(U¹, U²)
    i ← i₁ ← i₂ ← 1
    Se ∡(U¹[1]) < ∡(U²[1])
        U[1] ← U¹[1]
        i₁ ← i₁ + 1
    Senão
        U[1] ← U²[1]
        i₂ ← i₂ + 1
    Enquanto i₁ ⩽ |U¹| e i₂ ⩽ |U²|
        Se U¹[i₁] intercepta U[i] antes de U²[i₂]
            i ← i + 1
            U[i] ← U¹[i₁]
            i₁ ← i₁ + 1
        Senão
            i ← i + 1
            U[i] ← U²[i₂]
            i₂ ← i₂ + 1
    Retorne U
```

FIGURA 5.2. Fase de combinação do problema 10

Com este algoritmo de combinação de dois envelopes superiores concluído, é bastante simples escrever um algoritmo para resolver o problema original. Na primeira fase, dividimos S em S_1 e S_2 de mesmo tamanho (ou tamanhos diferindo de no máximo uma unidade, se $|S|$ for ímpar). Na segunda fase, resolvemos recursivamente o problema para os dois subconjuntos, a não ser que um dos conjuntos tenha apenas uma reta, quando sabemos que o envelope superior é a própria reta. Na terceira fase, combinamos as soluções com o algoritmo que acabamos de ver.

Vamos agora analisar a complexidade de tempo de nosso algoritmo. A primeira fase leva tempo constante e a terceira fase leva tempo linear. A complexidade da segunda fase é colocada na forma de recorrência

$$T(n) = 2T(n/2) + n.$$

Para provarmos um limite superior para $T(n)$ por indução, precisamos ter uma estimativa de quanto vale $T(n)$. Vamos imaginar a execução do algoritmo como uma árvore como na figura 5.3. Cada vértice representa uma execução do procedimento e o número indicado nele representa o número de retas na entrada. Os dois filhos de um vértice correspondem às duas chamadas recursivas feitas a partir do vértice pai. O tempo gasto em todas as execuções com

uma reta na entrada, no total, é $O(n)$. O mesmo é válido para todas as execuções com 2 retas na entrada, e assim por diante. Como a altura da árvore é $O(\lg n)$, a soma das complexidades de tempo vale $O(n \lg n)$.

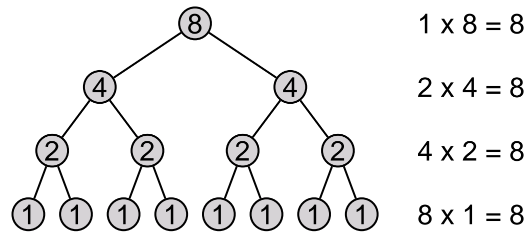


FIGURA 5.3. Árvore correspondente à execução do algoritmo de divisão e conquista em entrada de tamanho inicial 8.

Podemos então fazer provar o teorema abaixo usando indução:

TEOREMA 5.1. *A complexidade de tempo do algoritmo apresentado para determinar o envelope superior de um conjunto de n retas é $O(n \lg n)$.*

DEMONSTRAÇÃO. Como o algoritmo divide o problema em duas partes de aproximadamente o mesmo tamanho e as etapas de divisão e combinação levam tempo no máximo linear, temos a recorrência $T(n) = 2T(n/2) + n$. Por indução provamos que $T(n) \leqslant cn \lg n$:

$$\begin{aligned} T(n) &= 2T(n/2) + n \\ &\leqslant 2cn/2 \lg(n/2) + n \\ &= cn \lg(n/2) + n \\ &= cn \lg n - cn + n \\ &\leqslant cn \lg n. \end{aligned}$$

□

5.2. Par de Pontos Mais Próximos

PROBLEMA 11. *Dado um conjunto S de n pontos no plano, encontrar o par de pontos mais próximos.*

Um algoritmo trivial para este problema tem complexidade de tempo $\theta(n^2)$, simplesmente calculando a distância entre todos os pares de pontos e encontrando a distância mínima. Desejamos obter um algoritmo mais eficiente que este, para n grande. No problema anterior, dividimos os elementos da entrada em dois conjuntos quaisquer de tamanho aproximadamente iguais. Desta vez, seremos mais exigentes em nossa divisão. Vamos dividir S em dois conjuntos S_1 e S_2 por uma reta vertical r, de modo que S_1 e S_2 são aproximadamente do mesmo tamanho. Podemos fazer isso de modo simples se ordenarmos inicialmente os pontos de S segundo o eixo x. Note que esta ordenação só precisa ser feita uma vez no início do algoritmo.

Digamos que o par de pontos mais próximos de S_1 tenha distância d_1 e o par de pontos mais próximos de S_2 tenha distância d_2. Como podemos obter o par de pontos mais próximos de $S_1 \cup S_2$? Certamente o par de pontos mais próximos tem distância menor ou igual a $\min(d_1, d_2)$. Mas é possível que o par que estamos procurando tenha um ponto em S_1 e outro em S_2. Ainda assim, podemos descartar seguramente os pontos que distam mais que $\min(d_1, d_2)$ da reta vertical r que usamos para dividir os pontos (figura 5.4(a)). Vamos chamar de S' o conjunto de pontos que distam no máximo $\min(d_1, d_2)$ de r. Nosso algoritmo precisa apenas calcular o par de pontos mais próximo em S' e comparar sua distância com $\min(d_1, d_2)$, escolhendo a menor. Na maioria das situações, só isto já reduziria bastante o tempo de processamento, porém, no pior caso, pode ser que não descartemos nenhum ponto, portanto ainda teríamos de calcular $O(n^2)$ distâncias.

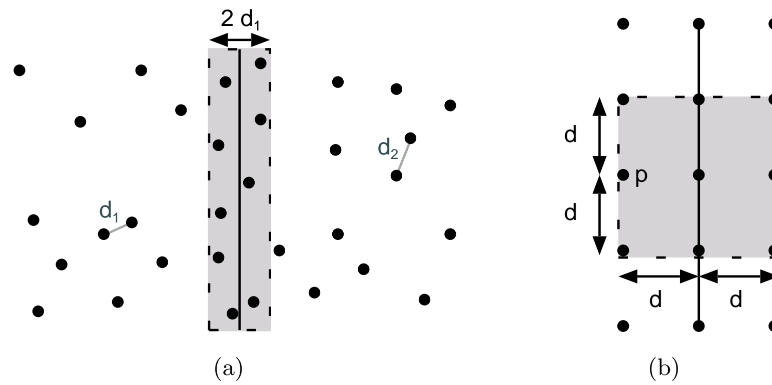


FIGURA 5.4. (a) Execução do algoritmo para encontrar o par de pontos mais próximos. (b) Número máximo de pontos que podem estar contidos no quadrado.

Para todo ponto p em S', podemos nos limitar a calcular a distância entre p e os pontos de S' cuja diferença de coordenada y seja menor que $\min(d_1, d_2)$. Como o número desses pontos é no máximo 8, garantimos que só precisamos fazer um número linear de comparações nessa fase. Para justificarmos que o número desses pontos é no máximo 8, note que, para cada ponto p, os pontos de S' que distam até $\min(d_1, d_2)$ de p, estão dentro de um quadrado de lado $2\min(d_1, d_2)$, e não há dois ponto na mesma metade desse quadrado que distem menos de $\min(d_1, d_2)$ (figura 5.4(b)). Porém, para fazermos selecionarmos estes pontos, devemos percorrer os pontos de S' de cima para baixo, sendo necessário ordenar os pontos segundo o eixo y.

Para analisarmos a complexidade de tempo deste algoritmo, construímos a recorrência abaixo, lembrando que a complexidade de tempo para ordenar n elementos é $\theta(n \lg n)$:

$$T(n) = 2T(n/2) + n \lg n.$$

É possível provar por indução que $T(n) = \theta(n \lg^2 n)$, porém não vamos fazer esta prova, já que reduziremos ainda mais a complexidade do nosso algoritmo. Para isto, basta fazermos a ordenação dos pontos segundo o eixo y apenas uma vez, e sempre passarmos o conjunto de pontos como parâmetro ordenado pelo eixo y. Note que, na fase de divisão, não precisamos trabalhar com os pontos ordenados segundo o eixo x, mas apenas ter acesso a esta ordenação de modo a encontrar rapidamente o elemento com coordenada x mediana (de fato, é possível encontrar a mediana em tempo linear sem usar ordenação, como veremos na sessão 7.2, mas neste caso é mais eficiente na prática ordenarmos os pontos).

Assim, o nosso algoritmo começa ordenando os pontos segundo o eixo x e segundo o eixo y separadamente. Dividimos então os pontos, ordenados pelo eixo y em dois conjuntos separados por uma reta vertical. Calculamos recursivamente o par de pontos mais próximos nestes dos sub-conjuntos, sem refazermos qualquer ordenação, pois os pontos já estão ordenados segundo o eixo y e temos acesso a sua ordem segundo o eixo x. Computamos então o conjunto de pontos S', próximos a reta vertical r, e examinamos as distâncias entre os pontos verticalmente próximos de S', de cima para baixo, bastando examinarmos distâncias entre pontos à esquerda de r e pontos à direita de r. Retornamos então a distância mínima, dentre as distâncias entre os pontos de S', d_1 e d_2. O pseudo-código deste algoritmo está na figura 5.5. Alguns detalhes do algoritmo estão um pouco diferentes no pseudo-código, de modo a melhorar ainda mais a performance.

A complexidade de tempo do nosso algoritmo é $O(n \lg n)$, pois as ordenações iniciais levam tempo $O(n \lg n)$ e o restante obedece a já conhecida recorrência

$$T(n) = 2T(n/2) + n.$$

```
Entrada:
    S: Conjunto de pontos no plano
Saída:
    (p, p'): Par de pontos mais próximos

PontosMaisPróximos(S)
    S_x ← ordenação de S segundo o eixo x
    S_y ← ordenação de S segundo o eixo y
    Retorne PMPOrdenado(S_x, 1, |S|, S_y)

PMPOrdenado(S_x, inicio, fim, S_y)
    Se fim − inicio ⩽ 4
        Retorne a solução do problema obtida comparando todas as distâncias
    meio ← ⌊(inicio + fim)/2⌋
    Para i de 1 até |S_y|
        Se S_y[i].x ⩽ S_x[meio], então acrescenta S_y[i] ao final de S_1
        Senão, acrescenta S_y[i] ao final de S_2
    (p_1, p'_1) ← PMPOrdenado(S_x, inicio, meio, S_1)
    (p_2, p'_2) ← PMPOrdenado(S_x, meio + 1, fim, S_2)
    Se |p_1 − p'_1| < |p_2 − p'_2|, então p ← p_1 e p' ← p'_1
    Senão, p ← p_2 e p' ← p'_2
    d ← |p − p'|
    Para i de 1 até |S_1|
        Se S_x[meio + 1].x − S_1[i].x < d
            Acrescenta S_1[i] ao final de S'_1
    Para i de 1 até |S_2|
        Se S_1[i].x − S_x[meio].x < d
            Acrescenta S_2[i] ao final de S'_2
    j_2 ← 1
    Para i_1 de 1 até |S'_1|
        Enquanto S'_1[i_1].y − S'_2[j_2].y > d
            j_2 ← j_2 + 1
        i_2 ← j_2
        Enquanto S'_2[i_2].y − S'_1[i_1].y < d
            Se |S'_2[i_2] − S'_1[i_1]| < d
                p ← S'_1[i_1] e p' ← S'_2[i_2]
                d ← |p − p'|
            i_2 ← i_2 + 1
            Se i_2 > |S'_2|, então sai do 'enquanto'
    Retorne (p, p')
```

FIGURA 5.5. Solução do Problema 11

5.3. Conjunto Independente de Peso Máximo em Árvores

Um conjunto independente em um grafo, é um subconjunto de seus vértices que não contém nenhum par de vértices que sejam adjacentes. Chama-se de conjunto independente máximo, o maior conjunto independente do grafo (figura 5.6(a)). Na versão com pesos nos vértices, deseja-se maximizar a soma dos pesos dos vértices do conjunto. Este problema é extremamente complexo de ser resolvido, estando na categoria de problemas *NP-difíceis*, como veremos no capítulo 10. Porém, se nos restringirmos a árvores (grafos sem ciclos), podemos resolver o problema eficientemente usando divisão e conquista.

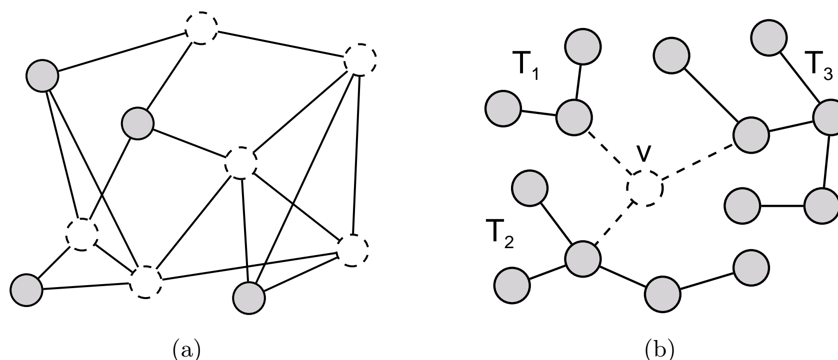


FIGURA 5.6. (a) Conjunto independente máximo de uma árvore sem pesos. (b) Árvore dividida em três sub-árvores pela remoção do vértice v.

PROBLEMA 12. *Dado uma árvore T, com pesos nos vértices, encontrar um conjunto independente de peso máximo de T.*

Nos outros problemas dessa sessão, nos preocupamos em fazer a divisão de modo balanceado, ou seja, queríamos obter sub-problemas de aproximadamente o mesmo tamanho. Neste caso, entretanto, veremos que isto não é necessário. Vamos começar escolhendo um vértice v da árvore T. Com este vértice, é natural dividir a árvore em algumas sub-árvores (figura 5.6(b), pois a remoção de v vai tornar a árvore desconexa (a não ser que v seja uma folha, mas, se for, também não há problema nenhum). Como podemos usar a solução dos problemas para as sub-árvores de modo a obter uma solução para o problema maior?

Pensando um pouco sobre isso, você vai notar que não há qualquer maneira óbvia de fazê-lo, pois caso algum vértice adjacente a v em T esteja no conjunto independente máximo de uma das sub-árvores, não será possível acrescentar v ao novo conjunto independente, aproveitando as soluções dos sub-problemas. Para resolver isto, vamos complicar um pouco nosso problema.

Nosso novo problema é: dados uma árvore T, com pesos nos vértices, e um vértice v, calcular: (i) um conjunto independente de maior peso dentre os conjuntos independentes que não contém v; (ii) um conjunto independente de peso máximo. Com isto, podemos descrever nosso algoritmo de divisão e conquista.

Na primeira iteração, como não é fornecido nenhum vértice v, iniciamos escolhendo um vértice v qualquer. Para cada sub-árvore T_i obtida pela remoção de v, chamamos de v_i o vértice de T_i adjacente à v. Calculamos recursivamente os conjuntos independentes máximos de cada sub-árvore T_i, podendo conter e sem poder conter o vértice v_i. O conjunto independente máximo C_1 de T, com a restrição de não conter v, é, claramente, a união dos conjuntos independentes máximos das sub-árvores obtidas pela remoção de v. Para calcularmos o conjunto independente máximo real de T, construímos um outro conjunto independente C_2. O conjunto independente C_2 é obtido pela união do vértice v aos conjuntos independentes máximos das sub-árvores T_i, que não contém v_i. O conjunto independente máximo de T é, então, o conjunto independente de maior peso dentre C_1 e C_2. O pseudo-código deste algoritmo está na figura 5.7.

Provar que a complexidade de tempo do algoritmo é linear no número de vértices é simples e fica como exercício.

5.4. Multiplicação de Matrizes: Algoritmo de Strassen

PROBLEMA 13. *Dadas duas matrizes $n \times n$, A e B, obter a matriz $C = A \cdot B$.*

Uma solução bastante simples é usar a definição de produto de matrizes, que é

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}.$$

Entrada:
T: Árvore com pesos nos vértices.
v: Vértice de T, inicialmente qualquer vértice.
Saída:
(C^1, C^2), onde:
C^1: Conjunto independente que não contém v de peso máximo
C^2: Conjunto independente de peso máximo de T

```
ConjuntoIndependente(T, v)
    Para cada sub-árvore T_i obtida pela remoção de v de T
        v_i ← vértice de T_i adjacente a v em T
        (C_i^1, C_i^2) ← ConjuntoIndependente(T_i,v_i)
        C^1 ← C^1 ∪ C_i^2
        C^2 ← C^1 ∪ C_i^1
    C^2 ← C^1 ∪ {v}
    Se peso(C^2)<peso(C^1)
        C^2 ← C^1
    Retorne (C^1,C^2)
```

FIGURA 5.7. Solução do problema 12

Este algoritmo tem complexidade de tempo $O(n^3)$, pois para calcularmos cada elemento na matriz C, fazemos um número linear de operações elementares. Como podemos usar divisão e conquista neste problema, ou seja, decompor o problema em sub-problemas menores? Primeiro vamos simplificar um pouco o problema, nos restringindo a matrizes onde n é uma potência de 2. Não perdemos muito com isto, pois caso a largura de nossa matriz não seja uma potência de 2, podemos completá-la com elementos nulos.

Sabemos que o produto de duas matrizes 2×2 é dado por

$$\begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \cdot \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}.$$

Podemos dividir cada uma das nossas matrizes $n \times n$, A e B, em quatro sub-matrizes $n/2 \times n/2$, pois consideramos que n é potência de 2. Usamos então a fórmula para multiplicação de matrizes 2×2. Assim, teremos que fazer 8 multiplicações de matrizes $n/2 \times n/2$. Estas multiplicações são resolvidas recursivamente. Note que este algoritmo é bem diferente dos outros algoritmos de divisão e conquista que vimos antes. Não estamos apenas dividindo a entrada em conjuntos disjuntos, e resolvendo recursivamente o problema nesses conjuntos. Agora, dividimos cada uma das matrizes da entrada em 4 partes e criamos 8 sub-problemas combinando estas partes. Deste modo, a fase de divisão, onde definimos os sub-problemas a serem resolvidos, tornou-se bem mais elaborada.

Para analisarmos a complexidade de tempo deste algoritmo, vamos apenas contar o número de multiplicações elementares realizadas, já que o número de adições e outras operações é uma constante vezes o número de multiplicações. Contamos exatamente este número com a recorrência

$$T(n) = \begin{cases} 8T(n/2) & \text{, se } n > 2 \\ 8 & \text{, se } n = 2 \end{cases}.$$

É fácil notar que $T(n) = n^3$, portanto não ganhamos absolutamente nada com nosso algoritmo de divisão e conquista. Porém, nosso algoritmo agora é fortemente baseado em uma operação bastante simples, a multiplicação de matrizes 2×2. Se conseguirmos descobrir uma maneira mais eficiente de multiplicarmos estas matrizes, podemos melhorar nosso algoritmo

imediatamente. Não é nem um pouco trivial, multiplicar duas matrizes 2×2 com menos de 8 multiplicações elementares, mas mostraremos aqui como fazê-lo. Considere as variáveis abaixo:

$$\begin{aligned}
m_1 =& (a_{21} + a_{22} - a_{11})(b_{22} - b_{12} + b_{11}),\\
m_2 =& a_{11}b_{11},\\
m_3 =& a_{12}b_{21},\\
m_4 =& (a_{11} - a_{21})(b_{22} - b_{12}),\\
m_5 =& (a_{21} + a_{22})(b_{12} - b_{11}),\\
m_6 =& (a_{12} - a_{21} + a_{11} - a_{22})b_{22},\\
m_7 =& a_{22}(b_{11} + b_{22} - b_{12} - b_{21}).
\end{aligned}$$

É possível, embora um pouco trabalhoso, verificar que

$$AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}.$$

De fato, multiplicar matrizes 2×2 usando este método é extremamente ineficiente, pois no lugar de 4 adições e 8 multiplicações, realizamos 24 adições ou subtrações e 7 multiplicações (é possível reduzir o número de adições e subtrações para 15 usando variáveis adicionais). Porém, se as adições puderem ser realizadas muito, muito, muito mais rápido que as multiplicações, pode valer a pena. Este é o caso das operações com matrizes grandes. Como este método não se baseia na comutatividade da multiplicação, podemos considerar que os elementos são matrizes, e não números reais. Assim, se usarmos este método para escolher que sub-problemas desejamos resolver, obtemos a recorrência

$$T(n) = 7T(n/2).$$

Para resolvermos esta recorrência, vamos considerar $T(1) = 1$. Assim, obtemos $T(2) = 7$, $T(4) = 7^2$ etc. É fácil perceber que $T(2^n) = 7^n$. Substituindo n por $\lg n$, temos

$$T(n) = 7^{\lg n} = 7^{\log_7 n / \log_7 2} = n^{1/\log_7 2} = n^{\lg 7} \approx n^{2,80735}.$$

Teorema 5.2. *O algoritmo de Strassen calcula o produto de duas matrizes $n \times n$ em tempo $O(n^2, 81)$.*

Desta forma, conseguimos reduzir a complexidade de tempo de $\theta(n^3)$ para $\theta(n^{2,81})$. Claro que, com isso, aumentamos a constante oculta pela notação O, portanto este algoritmo só é mais rápido na prática para multiplicar matrizes realmente muito grandes. Assim, quando n é menor que um certo valor (que pode ser determinado experimentalmente), é preferível chamar o algoritmo cúbico de multiplicação, e não continuar recursivamente.

5.5. Resumo e Observações Finais

Apresentamos neste capítulo a técnica de divisão e conquista, que se baseia em decompor um problema em sub-problemas menores, e resolvê-los recursivamente, combinando suas soluções depois. Apresentamos aqui quatro problemas: envelope superior de retas, par de pontos mais próximos, conjunto independente máximo em árvores e multiplicação de matrizes. Os algoritmos de divisão e conquista têm três fases: dividir, conquistar e combinar.

Na primeira fase, a divisão, o problema é decomposto em dois (ou mais) sub-problemas. No problema 'envelope superior', simplesmente dividimos as retas em dois conjuntos de mesmo tamanho. No problema 'par de pontos mais próximos', dividimos os pontos em dois conjuntos do mesmo tamanho usando uma reta vertical. No problema 'conjunto independente máximo', particionamos a árvore, removendo um vértice a cada iteração. No problema 'multiplicação de matrizes', criamos sete sub-problemas, combinando partições das duas matrizes originais e suas somas.

Na segunda fase, a conquista, resolvemos os sub-problemas, recursivamente, usando o mesmo procedimento de divisão e conquista, até que o tamanho do problema seja tão pequeno que sua solução seja trivial ou possa ser feita mais rapidamente usando algoritmos mais simples.

Na terceira fase, a combinação das soluções, também chamada de casamento, temos que unir as soluções dos problemas menores para obtermos uma solução unificada. No caso do 'envelope superior', usamos o paradigma de linha de varredura para computar esta combinação da esquerda para a direita. No problema 'par de pontos mais próximos', tivemos que usar uma técnica bem mais sofisticada, para calcular apenas um número linear de distâncias adicionais. No 'conjunto independente máximo', aumentamos o problema para retornar também o conjunto independente máximo sem um elemento, de modo que pudéssemos fazer a combinação. No problema 'multiplicação de matrizes', unimos as soluções usando equações nada triviais.

A técnica de busca binária, examinada no capítulo 3, é um caso particular do paradigma de divisão e conquista onde a divisão é feita examinando apenas um número constante de elementos e escolhendo um único sub-problema para resolver. A técnica de simplificação, que será estudada no capítulo 7, também é um caso particular da divisão e conquista, onde é resolvido apenas um sub-problema.

Uma variação do paradigma de divisão e conquista que não vimos aqui consiste em combinar antes de conquistar (ou casar antes de conquistar). Nesta variação, primeiro analisamos como as soluções serão combinadas, para depois seguirmos na conquista, nos beneficiando da informação obtida na combinação. Esta variação é útil no exercício 5.6.

Um artifício que também não comentamos neste capítulo, mas é essencial para a eficiência de alguns algoritmos de divisão e conquista, é chamado de memorização. Em alguns casos, o nosso algoritmo pode chamar duas vezes o procedimento para encontrar a solução exatamente do mesmo problema. Neste caso, devemos consultar uma tabela e recuperarmos a resposta que anotamos na tabela, sem perder tempo fazendo duas vezes o mesmo cálculo. O algoritmo do exercício 5.4 tem complexidade exponencial sem memorização, mas complexidade linear com memorização.

Uma alternativa à técnica de divisão e conquista e memorização é chamada de programação dinâmica (capítulo 6). Na técnica de programação dinâmica, no lugar de resolvermos um problema maior dividindo-o em problemas menores, resolvemos primeiro problemas menores e seguimos combinando suas soluções até chegar na solução do problema maior que desejamos resolver.

Exercícios

5.1) A recorrência $T(n) = T(\alpha n) + T((1-\alpha)n) + n$, com α constante entre 0 e 1, é obtida no cálculo de complexidade caso a divisão da entrada em duas partes não seja simétrica. Prove que esta recorrência também satisfaz $T(n) = O(n \lg n)$.

5.2) Escreva um algoritmo que calcule o elemento máximo e o elemento mínimo de um conjunto com n elementos, usando apenas $3\lceil n/2 \rceil - 2$ comparações.

5.3) Escreva um algoritmo para ordenar um vetor com n elementos em tempo $O(n \lg n)$ usando divisão e conquista.

5.4) Os números de Fibonacci F_i são definidos recursivamente pela recorrência $F_i = F_{i-1} + Fi - 2$, com $F_0 = 0$ e $F_1 = 1$. Escreva um algoritmo recursivo para calcular o F_i. Use o recurso de memorização para tornar a complexidade de seu algoritmo linear em i.

5.5) Dado um conjunto de n pontos no plano, escreva um algoritmo para determinar seu fecho convexo em tempo $O(n \lg n)$, usando divisão e conquista. Prove que o algoritmo está correto e analise sua complexidade de tempo.

5.6) Dado um conjunto de n pontos no plano, escreva um algoritmo para determinar seu fecho convexo em tempo $O(n \lg h)$, onde h é o número de vértices no fecho convexo. Seu algoritmo pode usar livremente uma função que, dado um conjunto de m pontos no plano e uma reta vertical r, retorna as arestas do fecho convexo que interceptam a reta r. Prove que o algoritmo está correto e analise sua complexidade de tempo.

5.7) Uma triangulação de um conjunto S de pontos no plano é uma subdivisão de seu fecho convexo em triângulos disjuntos (exceto em seus bordos), onde os vértices dos triângulos são exatamente os pontos de S (figura 5.8(a)). Escreva um algoritmo para computar uma triangulação de um conjunto de pontos no plano.

*5.8) Uma triangulação de Delaunay é uma triangulação que satisfaz a propriedade que os círculos circunscritos aos triângulos da triangulação não contém nenhum ponto em seus interiores (figura 5.8(b)). Outra definição é que uma aresta pertence a triangulação de Delaunay se e só se existe círculo com os dois pontos da aresta no seu bordo e nenhum ponto em seu interior. Escreva um algoritmo baseado em divisão e conquista que, dado um conjunto S de pontos no plano, compute sua triangulação de Delaunay em tempo $O(|S| \lg |S|)$.

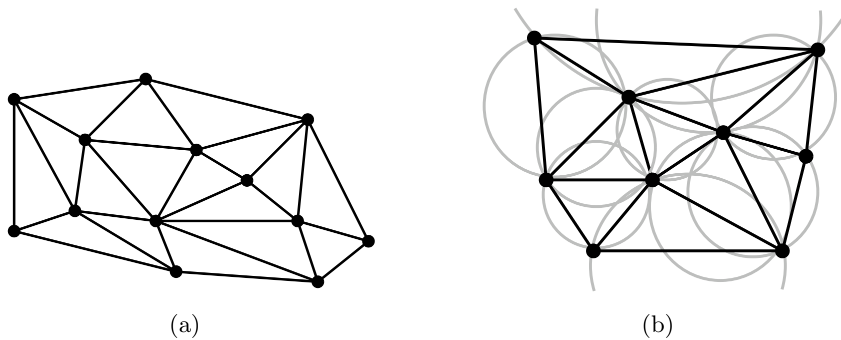


FIGURA 5.8. (a) Triangulação de um conjunto de pontos. (b) Triangulação de Delaunay de um conjunto de pontos.

*5.9) O fecho convexo de um conjunto de pontos no espaço é o menor poliedro convexo que contém todos estes pontos. Dado um conjunto de n pontos no espaço tridimensional, escreva um algoritmo para determinar seu fecho convexo em tempo $O(n \lg n)$. Prove que o algoritmo está correto e analise sua complexidade de tempo.

CAPíTULO 6

Programação Dinâmica

A técnica de programação dinâmica é uma técnica de decomposição que resolve um problema decompondo-o em subproblemas cujas soluções são armazenadas em uma tabela.

6.1. Ordem de Multiplicação de Matrizes

Imagine que você tem que multiplicar três matrizes A, B e C, na mão, usando apenas papel e lápis. Considere que A é uma matriz 10×2 (10 linhas e 2 colunas), B é uma matriz 2×20 e C é uma matriz 20×5. Imagine que o tempo está correndo e quanto mais rápido você resolver o problema, maior será sua nota. O que você faz? Se a sua resposta é: 'começo a multiplicar imediatamente', então você provavelmente fez a escolha errada. Vamos contar quantas multiplicações você terá que fazer.

Para multiplicar A por B, você fará $10 \cdot 2 \cdot 20 = 400$ multiplicações. Em seguida, para multiplicar (AB) por C, você fará $10 \cdot 20 \cdot 5 = 1000$ multiplicações. No total, fará $400 + 1000 = 1400$ multiplicações.

Porém, se você olhar para o problema com um pouco mais de cuidado, poderá notar que vale mais a pena começar multiplicando B por C, fazendo $2 \cdot 20 \cdot 5 = 200$ multiplicações. Em seguida, você multiplica A por (BC), fazendo mais $10 \cdot 2 \cdot 5 = 100$ multiplicações. No total, você faz $200 + 100 = 300$ multiplicações, enquanto quem começou multiplicando as matrizes na ordem fornecida fez 1100 multiplicações a mais!

Note que esta escolha da ordem da multiplicação é possível porque a multiplicação de matrizes, embora não seja comutativa, é associativa. Imagine que um computador tem que multiplicar uma seqüência de n matrizes. Não há dúvida que vale a pena, antes de iniciar a multiplicação, escolher a melhor ordem para fazê-lo. Isto é válido independente do algoritmo usado para fazer a multiplicação em si. Este é o problema estudado nesta sessão.

Problema 14. *Dada uma seqüência de n matrizes $A_1, \ldots, A_n$, escolher a ordem para multiplicá-las que minimiza o tempo total gasto.*

O primeiro passo para resolvermos o problema é nos familiarizarmos com ele. Não precisamos nos preocupar com o conteúdo das matrizes que desejamos multiplicar, apenas com suas dimensões. Como, para multiplicarmos a matriz A pela matriz B, a largura de A tem que ser igual a altura de B, podemos condensar as dimensões das n matrizes que desejamos multiplicar em um vetor v com $n+1$ posições, contendo as dimensões das matrizes, ou seja, M_i, a i-ésima matriz da multiplicação, tem dimensões $v_i \times v_{i+1}$. No nosso exemplo do início da sessão, o vetor seria $v = (10, 2, 20, 5)$. Nosso algoritmo não assumirá nada sobre o tempo gasto para multiplicar duas matrizes. Consideraremos que o tempo gasto para multiplicar uma matriz $a \times b$ por outra matriz $b \times c$ é $f(a, b, c)$. Esta função será considerada conhecida, e será avaliada pelo nosso algoritmo diversas vezes. Normalmente, porém, considerar $f(a, b, c) = abc$ é uma boa escolha, portanto, usaremos esta definição para os exemplos concretos.

Construiremos nossa solução de baixo para cima, ou seja, partiremos de problemas menores até chegarmos ao problema total que desejamos resolver. Vamos criar um vetor bidimensonal $T[1 \ldots n, 1 \ldots n]$ e preencheremos na posição $T[i, j]$ a melhor maneira de multiplicarmos as matrizes de A_i até A_j. Queremos, no final, obter $T[1, n]$, a solução para nosso problema. Para simplificarmos nossa explicação, computaremos apenas o tempo gasto na ordem ótima de multiplicação, e não a maneira explícita de fazê-lo. Porém, não é difícil usar o mesmo método para

	1	2	3	4	5
1	0	400	300	290	620
2		0	200	230	320
3			0	300	1200
4				0	225
5					0

TABELA 6.1. Tabela T tal que $T[i,j]$ é o custo de multiplicar as matrizes de M_i até M_j, onde M_k é uma matriz $v_k \times v_{k+1}$ segundo $v = (10, 2, 20, 5, 3, 15)$. Consideramos o custo de multiplicar uma matriz $a \times b$ por uma matriz $b \times c$ como sendo $f(a,b,c) = abc$.

obter realmente a maneira como as multiplicações devem ser realizadas. Vamos começar pelos casos triviais.

Quando temos apenas uma matriz, não há nada a fazer, portanto $T[i,i] = 0$, para $1 \leqslant i \leqslant n$. Quando temos apenas duas matrizes para multiplicar, há apenas uma maneira de fazê-lo, portanto $T[i,i+1] = f(v_i, v_{i+1}, v_{i+2})$, para $1 \leqslant i \leqslant n-1$. Quando temos três matrizes, A, B, C, podemos multiplicar primeiro AB ou BC. Assim, para minimizarmos o custo, fazemos $T[i,i+2] = \min(T[i,i+1] + f(v_i, v_{i+2}, v_{i+3}), T[i+1,i+2] + f(v_i, v_{i+1}, v_{i+3}))$. De um modo geral, para multiplicarmos as matrizes de M_i até M_j, podemos, para $i \leqslant k < j$, multiplicar primeiro as matrizes de M_i até M_k e também de M_{k+1} até M_j e depois multiplicarmos as duas matrizes obtidas. Temos então:

$$T[i,j] = \min_{k=i}^{j-1}(T[i,k] + T[k+1,j] + f(v_i, v_{k+1}, v_{j+1})).$$

Um exemplo da tabela T para a entrada $v = (10, 2, 20, 5, 3, 15)$ está na tabela 6.1. No exemplo, a função de custo usada foi $f(a,b,c) = abc$. Preenchemos as células $T[i,j]$ da tabela 6.1 em ordem não decrescente da diferença de subscrito, ou seja, primeiro preenchemos a diagonal principal com as células $T[i,i]$, em seguida a diagonal com as células $T[i,i+1]$, e assim por diante, até a última diagonal que consiste da célula $T[1,n]$. Note que para preencher uma célula da $T[i,j]$ tabela, basta consultar células $T[i,k]$ e $T[k,j]$ com k entre i e j. Com isto, é fácil escrever o pseudo-código da figura 6.1.

A complexidade de tempo do algoritmo é claramente $O(n^3)$, onde n é o número de matrizes a ser multiplicadas. Isto ocorre porque a tabela tem $O(n^2)$ posições e, para preenchermos uma posição, precisamos examinar outras $O(n)$ células da tabela.

Em muitos casos, uma complexidade de tempo cúbica no tamanho da entrada é inaceitável para propósitos práticos, porém, no caso da ordem de multiplicação de matrizes, esta complexidade é perfeitamente aceitável. Afinal, desejamos, após este pré-processamento, realmente multiplicar as matrizes e esta última fase do processo será provavelmente ainda mais demorada. Assim, não é provável que o número de matrizes seja grande a ponto de tornar a utilização de um algoritmo cúbico inviável.

6.2. Todos os caminhos mais curtos

Uma outra aplicação da técnica de programação dinâmica é o problema de todos os caminhos mais curtos num grafo direcionado.

PROBLEMA 15. *Dado um grafo direcionado com pesos positivos nas arestas, encontrar para cada par de vértices o caminho mais curto.*

Neste caso, é dado um grafo direcionado D, definido por dois conjuntos: o conjunto de vértices $V(D) = \{v_1, v_2, \ldots, v_n\}$ e o conjunto de arestas $E(D)$, pares ordenados de vértices em $V(D)$. Também é dada uma matrix W de pesos associados às arestas do grafo direcionado. A diagonal da matrix W é composta de zeros, enquanto que para $i \neq j$, $w(i,j)$ é o peso da aresta

Entrada:
n: Número de matrizes $M_1, \ldots, M_n$ a serem multiplicadas.
v: Vetor com $n+1$ posições onde a matriz M_i tem dimensões $v[i] \times v[i+1]$.
Saída:
Custo total mínimo de multiplicar as matrizes de M_1 até M_n.
Observações:
$f(a, b, c)$: Custo de multiplicar uma matriz $a \times b$ por uma matriz $b \times c$.

```
OrdemMultMatrizes(n, v)
    Para i de 1 até n
        T[i,i] ← 0
    Para Δ de 1 até n − 1
        Para i de 1 até n − Δ
            j ← i + Δ
            T[i,j] ← ∞
            Para k de i até j − 1
                Se T[i,j] > T[i,k] + T[k+1,j] + f(v_i, v_{k+1}, v_{j+1})
                    T[i,j] ← T[i,k] + T[k+1,j] + f(v_i, v_{k+1}, v_{j+1})
    Retorne T[1,n]
```

Figura 6.1. Solução do problema 14

(v_i, v_j), caso $(v_i, v_j) \in E(D)$, e caso contrário, quando $(v_i, v_j) \notin E(D)$, $w(i, j)$ é definido como ∞.

Um caminho no grafo direcionado D é uma seqüência $P = v_1, v_2, \ldots, v_k$ de vértices tal que vértices consecutivos na seqüência são adjacentes no grafo direcionado, ou seja, $(v_i, v_{i+1}) \in E(D)$. O comprimento de um caminho P é $w(P) = \sum_{i=1}^{k-1} w(v_i, v_{i+1})$. Um menor caminho do vértice u ao vértice v em D é um caminho de u até v em D de comprimento mínimo e a distância de u a v em D é o comprimento do menor caminho de u até v em D.

Considere um menor caminho $P = v_1, v_2, \ldots, v_k$ de v_1 até v_k. Chame de P_{ij} o subcaminho de P de v_i até v_j, definido pela seqüência de vértices $v_i, v_{i+1}, \ldots, v_j$. Claramente, P_{ij} é um menor caminho de v_i até v_j. Chamamos esta observação de princípio de otimalidade. O algoritmo que descrevemos a seguir é conseqüência deste princípio.

O algoritmo considera uma seqüência $W_0, W_1, \ldots, W_n$ de n matrizes $n \times n$. Definimos W_0 como a matriz de pesos W de entrada. A matriz W_1 contém como entrada $w^1(i, j)$ o comprimento de um menor caminho de v_i até v_j, sujeito à condição de que os vértices intermediários pertencem a $\{v_1\}$. A matriz W_2 contém como entrada $w^2(i, j)$ o comprimento de um menor caminho de v_i até v_j, sujeito à condição de que os vértices intermediários pertencem a $\{v_1, v_2\}$. E assim por diante, finalmente a última matriz W_n contém como entrada $w^n(i, j)$ o comprimento de um menor caminho de v_i até v_j, sujeito à condição de que os vértices intermediários pertencem a $\{v_1, v_2, \ldots, v_n\}$, ou seja, trata-se da matriz solução. Veja o pseudo-código correspondente na figura 6.2.

Podemos estabelecer a corretude deste algoritmo provando por indução que cada matriz W_k, onde $1 \leq k \leq n$, computada pelo algoritmo de fato contém na entrada $w^k(i, j)$ o comprimento de um menor caminho de v_i até v_j, sujeito à condição de que os vértices intermediários pertencem a $\{v_1, v_2, \ldots, v_k\}$. Para isto, basta aplicar o princípio de otimalidade.

Analisar a complexidade de tempo deste algoritmo é bastante simples. Basta notarmos que há três *loops* aninhados no algoritmo, cada um deles executando no máximo n repetições. Dentro desses *loops* todas as operações levam tempo constante. Assim, a complexidade de tempo do algoritmo é $\Theta(n^3)$.

Entrada:
  Grafo direcionado D, com n vértices.
  Matrix de pesos positivos nas arestas W.
Saída:
  Matriz de todos os caminhos mais curtos W_n.

```
TodosCaminhos(n, W)
    W_0 ← W
    Para k de 1 até n
        Para i de 1 até n
            Para j de i até n
                Se w^{k-1}(i,k) + w^{k-1}(k,j) < w^{k-1}(i,j)
                    w^k(i,j) ← w^{k-1}(i,k) + w^{k-1}(k,j)
    Retorne W_n
```

FIGURA 6.2. Solução do problema 15

6.3. Resumo e Observações Finais

A idéia central do método de programação dinâmica consiste em decompor o problema a ser resolvido em subproblemas e armazenar a solução dos subproblemas evitando que um mesmo subproblema seja calculado repetidamente.

O algoritmo para ordem de multiplicação de matrizes encontra a parentização ótima de modo a minimizar o custo de se multiplicar uma seqüência de matrizes.

O algoritmo para todos os caminhos mais curtos calcula a matriz distância que contém cada uma das distâncias entre os diferentes pares de vértices num grafo direcionado.

Exercícios

6.1) A subsequencia máxima comum (LCS) de duas sequências de caracteres T e P é a maior sequência L tal que L é sequência de T e de P. A supersequência mínima comum (SCS) de duas sequências T e P é a menor sequência L tal que T e P são sequências de L.

Descreva algoritmos que usam programação dinâmica para encontrar LCS e SCS de duas sequências dadas.

6.2) O fecho transitivo de um grafo direcionado $G(V, E_1)$ é um digrafo $T_G(V, E_2)$ tal que se existe caminho de u a v em G, então $uv \in E_2$. Claramente temos $E_1 \subseteq E_2$. Descreva um algoritmo que constroi a matriz de adjacências $A(T_G)$ de T_G dada a matriz de adjacências de G. Esta matriz é chamada de matriz de alcançabilidade de G.

O seu algoritmo usa programação dinâmica?

6.3) Dado o grafo direcionado D, onde $V(D) = \{v_1, v_2, v_3, v_4, v_5\}$ e $E(D) = \{(v_1v_2), (v_1v_3), (v_1v_5), (v_2v_4), (v_2v_5), (v_3v_2), (v_4v_1), (v_4v_3), (v_5v_4)\}$, com pesos nas arestas: $w(v_1v_2) = 9$, $w(v_1v_3) = 14$, $w(v_1v_5) = 2$, $w(v_2v_4) = 7$, $w(v_2v_5) = 13$, $w(v_3v_2) = 10$, $w(v_4v_1) = 8$, $w(v_4v_3) = 1$, $w(v_5v_4) = 12$. Pede-se a matriz M que tem como entradas $m_(i, j)$, a distância do vértice v_i ao vértice v_j em D.

6.4) Um ladrão encontra um cofre com N tipos de objetos, de tamanho e valor variados, vários objetos de cada tipo, e carrega uma mochila com capacidade M. O objetivo do ladrão é encontrar a combinação de objetos que cabe na mochila e maximiza o valor.

Descreva um algoritmo que usa programação dinâmica para resolver este problema da mochila.

6.5) São dadas n chaves $s_1, s_2, \ldots, s_n$ com correspondentes pesos $p_1, p_2, \ldots, p_n$. Queremos encontrar a árvore binária de busca que minimize a soma, sobre todas as chaves, dos

pesos vezes a distância da chave à raiz (o custo de acessar a chave associada àquele vértice da árvore).

Descreva um algoritmo que usa programação dinâmica para resolver este problema da árvore binária de busca ótima.

CAPíTULO 7

Simplificação

A técnica de simplificação é, de certa forma, um caso degenerado do paradigma de divisão e conquista. No método de divisão e conquista, quebrava-se um problema em sub-problemas menores e depois combinavam-se as soluções. No método de simplificação, o problema é reduzido a um único sub-problema menor que é resolvido pelo mesmo método, até que sucessivas simplificações levem a um problema trivial ou pequeno o suficiente para ser resolvido por força bruta. Este paradima também é chamado de *podar e buscar*. No caso onde o tamanho da entrada sempre diminui por uma fração constante, o método é chamado de *dizimar*.

7.1. Centro de Árvore

A palavra centro tem um significado geométrico muito forte, embora nem sempre preciso. A idéia de centro nos leva a elementos que estejam relativamente próximos de todos os outros elementos. Mesmo em objetos geométricos simples como triângulos, podemos falar em vários tipos de centros, como incentro, circuncentro, baricentro etc. Para definirmos o centro de um grafo, vamos primeiro definir a excentricidade de um vértice do grafo. Dado um grafo G, a excentricidade de um vértice v de G é a maior distância $d(v, v')$ de v a algum vértice v'. O centro de um grafo é o conjunto de vértices de excentricidade mínima, ou seja, o conjunto de vértices cuja distância ao vértice mais distante é mínima. No caso de árvores, o centro é sempre um único vértice ou um par de vértices adjacentes, como veremos. As excentricidades dos vértices de uma árvore estão ilustradas na figura 7.1(a).

Problema 16. *Dada uma árvore T, encontrar seu centro.*

Caso a árvore tenha apenas 1 ou 2 vértices, é claro que a árvore inteira é seu próprio centro. Estes são os casos triviais para nosso método de simplificação. Mas como podemos obter o centro de árvores maiores? As folhas não fazem parte do centro. Graças ao teorema abaixo, podemos descartá-las.

Teorema 7.1. *Seja T uma árvore com pelo menos 3 vértices e T' a árvore obtida pela remoção de todas as folhas de T. O centro de T é igual ao centro de T'.*

Demonstração. Para todo vértice v da árvore, os vértices mais distantes de v são folhas. Os vértices adjacentes aos vértices mais distantes de v tem distância de v igual a distância do vértice mais distante de v menos uma unidade. Portanto, se removermos todas as folhas da árvore, a excentricidade de todos os vértices diminuirá de uma unidade, não alterando o

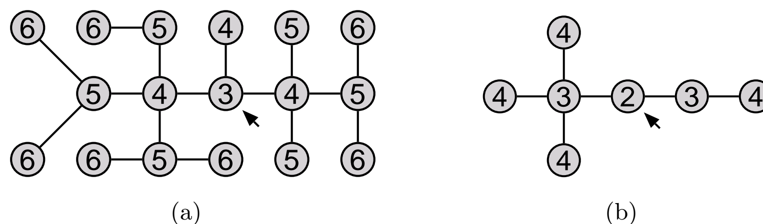


Figura 7.1. (a) Árvore com as excentricidades dos vértices escritas e o centro da árvore destacado. (b) Árvore da figura obtida após a remoção das folhas da árvore da figura (a).

conjunto de vértices cuja excentricidade é mínima. Como a árvore tem pelo menos 3 vértices, o centro não será removido nesse processo. □

Nosso algoritmo é, então, bastante simples. A cada iteração removemos todas as folhas da árvore. Quando sobrar apenas 1 ou 2 vértices, retornamos este(s) vértice(s) como o centro.

A complexidade de tempo do algoritmo é linear. Primeiro, construímos uma lista com todas as folhas. Em seguida, removemos todas as folhas, construindo uma lista das novas folhas criadas nesse processo. Ou seja, ao removermos uma folha f, verificamos se o vértice adjacente a f passa a ter grau 1. Em caso afirmativo, adicionamos o vértice adjacente a f na lista de folhas criadas. Repetimos este procedimento até restarem apenas 1 ou 2 folhas. Como a complexidade de tempo de cada etapa de remoção de folhas é linear no número de folhas removidas e nenhum vértice é removido mais de uma vez, a complexidade de tempo é linear no número de vértices.

7.2. Seleção do k-ésimo

Vários algoritmos se baseiam em dividir um conjunto S em dois conjuntos S_1 e S_2 de aproximadamente o mesmo tamanho. Muitas vezes, é útil adicionar a propriedade que os elementos de S_1 são menores que os elementos de S_2. Porém, para fazermos esta divisão, precisamos determinar o elemento mediano de S, ou seja, o elemento de posição $\lfloor |S|/2 \rfloor$ em S ordenada. Para encontrarmos o elemento mediano, uma alternativa é ordenarmos S e, em seguida, pegarmos o elemento de posição $\lfloor |S|/2 \rfloor$. Esta alternativa leva tempo $O(n \lg n)$. Será que podemos fazer melhor?

Para resolvermos este problema, vamos primeiro torná-lo um pouco mais geral. Trataremos, então, do problema de determinar o k-ésimo menor elemento de S. Assim, fazendo $k = \lfloor |S|/2 \rfloor$, obtemos o elemento mediano.

Problema 17. *Dados um conjunto S e um inteiro k, determinar o k-ésimo menor elemento de S.*

A solução deste problema usa a técnica de simplificação de modo bastante complexo, por isso, apresentaremos a solução em partes. Inicialmente, vamos supor que temos acesso a uma função pronta de mediana aproximada, com complexidade de tempo linear no tamanho de S. Esta função recebe como entrada um conjunto S e retorna um elemento $x \in S$ tal que pelo menos 30% dos elementos de S são menores ou iguais a x e pelo menos 30% dos elementos de S são maiores ou iguais a x. Estamos considerando que S representa um conjunto, portanto não tem elementos repetidos. É fácil adaptar os algoritmos para funcionarem no caso de elementos repetidos.

Podemos usar a mediana aproximada de um conjunto para dividir este conjunto em duas partes, S_1 e S_2, 'aproximadamente' de mesmo tamanho, com a propriedade que os elementos de S_1 são menores que os elementos de S_2. Se desejamos encontrar o k-ésimo menor elemento, sabemos que S_1 contém este elemento se e só se $|S_1| \geqslant k$. Caso $|S_1| < k$, temos que o k-ésimo menor elemento de S está em S_2. Não só isso, como podemos fazer afirmações ainda mais fortes. Caso $|S_1| \geqslant k$, então o k-ésimo menor elemento de S é o k-ésimo menor elmento de S_1. Caso $|S_1| < k$, então o k-ésimo menor elemento de S é o $(k - |S_1|)$-ésimo elemento de S_2.

Deste modo, temos o algoritmo da figura 7.2 que encontra o k-ésimo menor elemento de S. Para analirmos sua complexidade, escrevemos a recorrência

$$T(n) = T(7/10n) + n$$

Pode-se provar por indução que $T(n) = O(n)$. Para ganhar mais intuição sobre este limite, note que, na primeira iteração do algoritmo, são examinados n elementos. Na segunda iteração, são examinados no máximo $7/10n$ elementos. Na i-ésima iteração, são examinados no máximo $(7/10)^{i-1}n$ elementos. Esses valores formam uma progressão geométrica de termo inicial n e razão $7/10$, portanto, mesmo que somássemos infinitos termos, o que não é o caso, a soma não excederia $n/(1 - 7/10) = 10n/3 = O(n)$.

Entrada:
S: Conjunto de números reais.
k: Número inteiro tal que $1 \leqslant k \leqslant |S|$.
Saída:
O k-ésimo menor elemento de S.
Observações:
MedianaAproximada(S): Função que retorna um elemento $x \in S$ tal que pelo menos 30% dos elementos de S são menores ou iguais a x e pelo menos 30% dos elementos de S são maiores ou iguais a x.

```
Selecionar(S,k)
    Se |S| < 10
        Ordene S e retorne S[k]
    x ← MedianaAproximada(S)
    S1 ← elementos de S menores ou iguais a x
    S2 ← elementos de S maiores que x
    Se k ⩽ |S1|
        Retorne Selecionar(S1,k)
    Senão
        Retorne Selecionar(S2,k − |S1|)
```

FIGURA 7.2. Primeira parte da solução do problema 17

Como podemos construir a função que calcula a mediana aproximada? Imagine que agrupamos os elementos de S, arbitrariamente, em subconjuntos de 5 elementos. Vamos considerar que $|S|$ é múltiplo de 5 para simplificarmos nossa análise, porém caso $|S|$ não seja múltiplo de 5, não há problema em deixarmos um subconjunto com menos de 5 elementos. Podemos ordenar cada um destes sub-conjuntos de 5 elementos em tempo $O(1)$, assim ordenando todos os sub-conjuntos em tempo $O(|S|)$. Criamos, então, um conjunto M com as medianas de cada um dos sub-conjuntos de 5 elementos. Vale o seguinte teorema:

TEOREMA 7.2. *A mediana de M é maior ou igual a pelo menos* 30% *dos elementos de S e menor ou igual a pelo menos* 30% *dos elementos de S.*

DEMONSTRAÇÃO. Estamos considerando que $|S|$ é múltiplo de 5. Vamos chamar de x a mediana de M. Como x é a mediana de M, pelo menos metade dos elementos de M são menores ou iguais a x. Para cada $y \in M$, há pelo menos outros dois elementos de S que são menores que y, pois y é mediana de um conjunto de 5 elementos de S. O tamanho de M é $|S|/5$. Assim,

$$3 \cdot \frac{1}{2} \cdot \frac{|S|}{5}$$

elementos de S são menores ou iguais a x. O mesmo argumento prova que pelo menos 30% dos elementos de S são maiores ou iguais a x. A prova deste teorema está representada graficamente na figura 7.3. □

Com este teorema, temos um algoritmo para obter a mediana aproximada, ilustrado na figura 7.4. Porém, encontramos uma situação bastante atípica. Nosso algoritmo para selecionar o k-ésimo chama nosso algoritmo de mediana aproximada (além de chamar a si próprio recursivamente) e nosso algoritmo de mediana aproximada chama nosso algoritmo de encontrar o k-ésimo de modo a obter a mediana exata de um conjunto cinco vezes menor. Vamos agora provar algo surpreendente. A complexidade de tempo de nosso algoritmo de selecionar o k-ésimo menor elemento é linear! Vejamos a recorrência abaixo, que define sua complexidade de tempo:

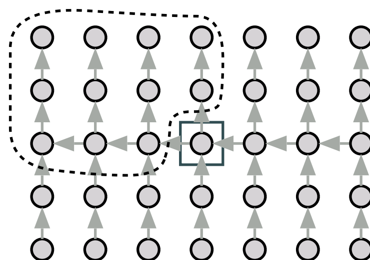


FIGURA 7.3. Conjunto de elementos com setas indicando relação 'maior que'.

Entrada:
S: Conjunto de números reais, tal que $|S|$ é múltiplo de 5.
Saída:
Elemento $x \in S$ tal que pelo menos 30% dos elementos de S são menores ou iguais a x e pelo menos 30% dos elementos de S são maiores ou iguais a x.

```
MedianaAproximada(S)
    Se |S| < 10
        Ordene S e retorne S[⌊|S|/2⌋]
    Particione S em sub-conjuntos com aproximadamente 5 elementos
    Ordene os sub-conjuntos
    Crie o conjunto M das medianas dos sub-conjuntos de S
    Retorne Selecionar(M,⌊|M|/2⌋)
```

FIGURA 7.4. Segunda parte da solução do problema 17

$$T(n) = n + T(7n/10) + T(n/5).$$

A recorrência da complexidade de tempo do algoritmo tem três termos. O primeiro corresponde a complexidade de $O(n)$ necessária para particionar o conjunto S em S_1 e S_2, assim como ordenar os sub-conjuntos de 5 elementos. O segundo corresponde a chamar o algoritmo recursivamente para S_1 ou S_2. O terceiro termo corresponde a encontrar a mediana das medianas dos conjuntos de 5 elementos.

Como sabemos estar interessados em obter um algoritmo de complexidade de tempo linear, podemos provar por indução que $T(n) = \Theta(n)$. Vamos supor $T(n) = cn$. Por indução,

$$\begin{aligned} T(n) &= n + T(7n/10) + T(n/5) \\ &= n + 7cn/10 + cn/5 \\ &= n(1 + 7c/10 + c/5) \\ &= n(1 + 9c/10). \end{aligned}$$

Portanto, $c = 10$ e $T(n) = 10n$.

Caso não tivéssemos idéia de um palpite para fazermos nossa prova por indução, poderíamos pensar que $T(n)$, não sendo uma função sub-linear, deve ser convexa, pelo menos para n suficientemente grande. Neste caso vale que $T(a + b) \geqslant T(a) + T(b)$. Com isto, temos:

$$T(n) \leqslant n + T(9n/10).$$

Esta recorrência se parece bastante com a vista no início da sessão e é claramente linear pelo argumento da soma dos termos da progressão geométrica (agora com razão 9/10).

Embora nosso algoritmo de mediana aproximada necessite de $|S|$ ser múltiplo de 5 para garantir a cota de 30%, caso $|S|$ não seja múltiplo de 5, a nossa cota será alterada apenas pela adição de uma constante, não alterando a complexidade de tempo do nosso algoritmo de seleção do k-ésimo menor elemento.

O algoritmo visto não é um exemplo típico de simplificação, pois resolve não um, mas dois sub-problemas em cada chamada. Porém, preferimos colocá-lo nesta sessão porque a motivação do projeto do algoritmo, como foi apresentado aqui, se baseia em uma idéia de simplificação.

7.3. Ponte do Fecho Convexo

No exercício 5.6, deve-se escrever um algoritmo que determina o fecho convexo de um conjunto S de n pontos no plano em tempo $O(n \lg h)$, onde h é o número de pontos do fecho convexo. Este algoritmo usa uma função que, dados um conjunto S de n pontos no plano e uma reta vertical r, obtem as arestas do fecho convexo de S que interceptam r (figura 7.5(a)), em tempo $O(n)$. Nesta sessão, descreveremos esta função.

Problema 18. *Dados um conjunto S de n pontos no plano e uma reta vertical r, encontre as arestas do fecho convexo de S que interceptam r.*

A reta r interceptará duas arestas do fecho convexo, sendo uma do fecho convexo superior e outra do fecho convexo inferior. Nos concentraremos em obter a aresta do fecho convexo superior que intercepta r, também chamada de “ponte”. A obtenção da aresta do fecho convexo inferior que intercepta r é análoga. Também consideraremos que r realmente intercepta o fecho convexo de S, pois isto só não acontece caso todos os pontos de S estejam do mesmo lado de r.

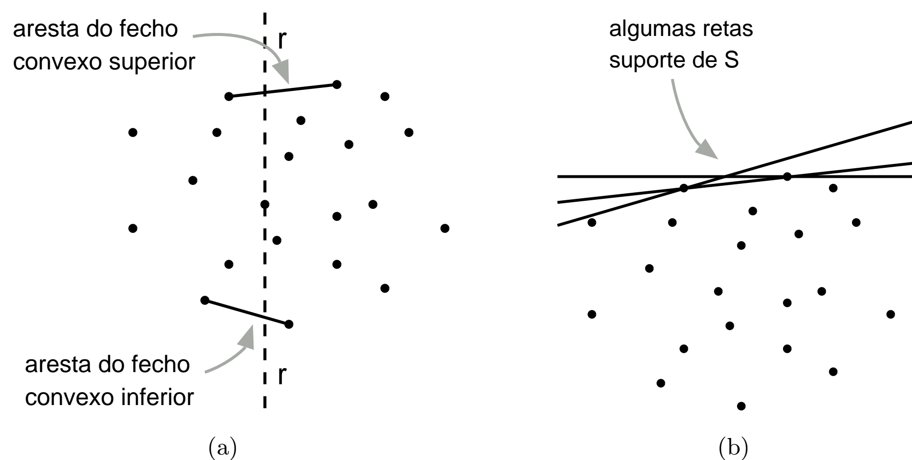


Figura 7.5. (a) Ponte do fecho convexo. (b) Algumas retas suporte de S.

Desejamos obter a ponte usando um algoritmo de simplificação. Nosso algoritmo procederá eliminando, a cada iteração, vértices que não são candidatos a serem um dos dois vértices da ponte. Começamos agrupando os pontos de S, arbitrariamente, em pares $(p_1, q_1), \ldots, (p_{\lfloor n/2 \rfloor}, q_{\lfloor n/2 \rfloor})$. Consideramos que, nos pares (p_i, q_i), p é o ponto mais à esquerda e q o mais à direita. Caso $|S|$ seja ímpar, um dos pontos fica sozinho e não concorre a ser descartado na iteração atual.

Como podemos fazer para descobrirmos pontos que, com certeza, não são candidatos a serem vértices da ponte? Definimos uma reta suporte de S como uma reta ρ que contém pelo menos um ponto de S e todos os demais pontos de S estão abaixo da reta ρ (figura 7.5(b)). Dada uma inclinação, é fácil determinar a única reta suporte com esta inclinação. Digamos que nos seja fornecida uma reta suporte qualquer. Caso a reta suporte contenha tanto pontos à esquerda da reta vertical r quanto à direita (provavelmente um ponto de cada lado), então a reta suporte contém a ponte e nosso problema está resolvido. Normalmente, porém, isto não acontecerá. Digamos que a reta suporte ρ contém apenas um ou mais pontos de S que estão a direita da reta vertical r.

Poderíamos rodar esta reta no sentido anti-horário, como um embrulho para presente, até encontrarmos a ponte. Não vamos fazer isto, porque o tempo gasto não seria linear. Mas segue desta observação um teorema importantíssimo para o nosso algoritmo:

TEOREMA 7.3. *Se ρ é uma reta suporte de S que contém apenas pontos à direita de r, então a ponte de S que intercepta r tem coeficiente angular maior que o de ρ. Analogamente, se ρ é uma reta suporte de S que contém apenas pontos à esquerda de r, então a ponte de S que intercepta r tem coeficiente angular menor que o de ρ.*

Vamos continuar supondo que ρ contém apenas pontos à direita de r. O outro caso é análogo. Digamos que um dos nossos pares de pontos (p_i, q_i) defina um segmento de coeficiente angular *menor* que o coeficiente angular de ρ. Neste caso, podemos dizer seguramente que q_i, o vértice da direita do par, não é um dos vértices da ponte. Vamos justificar com cuidado este fato, em princípio não muito óbvio. Suponha, por absurdo, que q_i seja um vértice da ponte. O coeficiente angular da ponte tem que ser menor ou igual ao coeficiente angular de (p_i, q_i), pois caso contrário p_i estaria acima da reta que contém a ponte. Isto é absurdo, pois sabemos que a ponte tem coeficiente angular maior que ρ, que por sua vez tem coeficiente angular maior que (p_i, q_i).

Deste modo, dada uma reta suporte ρ que contenha apenas pontos a *direita* de r, podemos descartar os vértices da direita de todos os pares (p_i, q_i) com coeficientes angulares menores que o de ρ. Analogamente, dada uma reta suporte ρ que contenha apenas pontos à *esquerda* de r, podemos descartar os vértices da esquerda de todos os pares (p_i, q_i) com coeficientes angulares maiores que o de ρ.

Não falamos até agora sobre como obter a inclinação conveniente para nossa reta suporte. Queremos que tanto o número de segmentos com coeficiente angulares maiores que o da reta suporte quanto com coeficientes angulares menores que o da reta suporte sejam grandes, pois não sabemos, em princípio, se nossa reta suporte conterá pontos à direita ou à esquerda de ρ. Usando o algoritmo da sessão anterior, podemos escolher a inclinação mediana dentre os segmentos (p_i, q_i). Deste modo, descartaremos um dos pontos de metade dos segmentos, assim descartando $1/4$ do total de pontos. Como, a cada iteração, uma fração constante dos pontos é descartada, pelo argumento já apresentado de progressão geométrica, a complexidade de tempo do algoritmo é linear no número de pontos da entrada. O pseudo-código deste algoritmo está na figura 7.7.

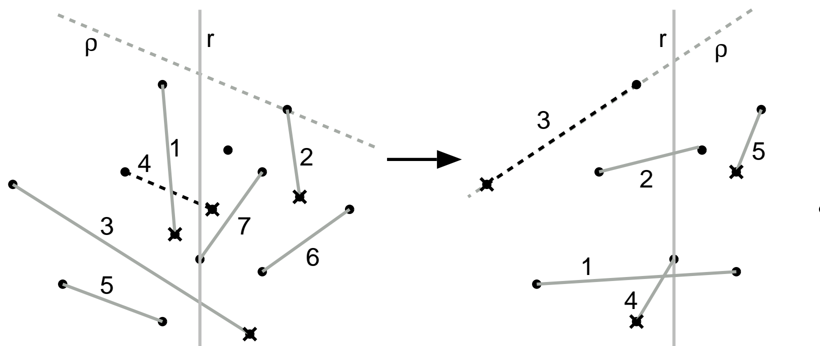


FIGURA 7.6. Duas iterações do algoritmo para encontrar a ponte. Segmentos numerados segundo os coeficientes angulares.

7.4. Resumo e Observações Finais

A técnica de simplificação consiste em reduzir um problema com uma entrada grande ao mesmo problema com uma entrada menor. A simplificação é um caso particular do paradigma de divisão e conquista, onde só é necessário resolver recursivamente um único problema menor. Quando o problema é pequeno o suficiente, podemos resolvê-lo diretamente.

Entrada:
  S: Conjunto de pontos no plano.
  r: Reta vertical que separa os pontos.
Saída:
  (p, q): Par de pontos da ponte.
Observações:
  $\measuredangle(p, q)$: Coeficiente angular do segmento (p, q).

```
Ponte(S,r)
    R ← Conjunto de ⌊n/2⌋ segmentos (p,q) ∈ S com p.x < q.x
    Cρ ← coeficiente angular mediano dentre os segmentos de R
    ρ ← reta suporte de coeficiente angular Cρ
    Se ρ contém pontos à direita e à esquerda de r
        p ← ponto de S mais à esquerda sobre ρ
        q ← ponto de S mais à direita sobre ρ
        Retorne (p,q)
    Se ρ contém somente pontos de S à direita de r
        Para todo (p,q) ∈ R
            Se ∡(p,q) ⩽ Cρ
                Remova de S o ponto q
        Retorne Ponte(S,r)
    Se ρ contém somente pontos de S à esquerda de r
        Para todo (p,q) ∈ R
            Se ∡(p,q) ⩾ Cρ
                Remova de S o ponto p
        Retorne Ponte(S,r)
```

FIGURA 7.7. Solução do problema 18

No primeiro problema estudado, desejamos obter o centro de uma árvore. Simplificamos o problema através da remoção de todas as folhas da árvore, o que não altera o centro. Paramos quando a árvore obtida possuir apenas 1 ou 2 vértices, que são seu próprio centro.

Em seguida, examinamos o algoritmo para determinar o k-ésimo menor elemento de um conjunto, que engloba o caso particular de determinar o elemento mediano. Neste problema, conseguimos descartar 20% dos elementos a cada iteração do algoritmo. Para fazermos isso, entretanto, precisamos chamar o próprio algoritmo de seleção da mediana recursivamente.

Uma ponte do fecho convexo é a aresta do fecho convexo superior que intercepta uma reta vertical r. Consideramos o problema de dados um conjunto de n pontos e uma reta vertical r obter a ponte. Uma maneira trivial de resolver este problema seria determinando o fecho convexo do conjunto de pontos, o que leva tempo $\Theta(n \lg n)$. Porém, podemos resolvê-lo diretamente, gastando tempo $O(n)$. Para isso, usamos o algoritmo de cálculo da mediana de modo que conseguimos descartar um quarto dos pontos a cada iteração.

Exercícios

7.1) O maior divisor comum (mdc) de um par de números inteiros é o maior número que divide, sem deixar resto, os dois números do par. O algoritmo de Euclides encontra mdc de dois números inteiros por simplificação. Dados dois inteiros a, b, com $a \geqslant b$, se b divide a, então $mdc(a, b) = b$. Caso contrário, seja r o resto da divisão de a por b, então $mdc(a, b) = mdc(b, r)$. Prove que este algoritmo funciona corretamente.

7.2) Escreva um algoritmo para particionar um conjunto S com n elementos em m conjuntos $S_1, \ldots, S_m$ com $\lfloor n/m \rfloor$ ou $\lceil n/m \rceil$ elementos de modo que os elementos de S_i são menores que os elementos de S_{i+1} para i de 1 até $m-1$. Uma solução trivial usa ordenação e tem complexidade de tempo $O(n \lg n)$. Porém, a sua solução deve ter complexidade de tempo $O(n \lg m)$.

7.3) Na sessão 5.3 vimos um algoritmo de divisão e conquista que encontra um conjunto independente máximo em uma árvore com pesos nos vértices. Resolva, usando simplificação, a versão mais simples do problema onde não há pesos nos vértices.

*7.4) Dados um conjunto de desigualdades lineares de duas variáveis, da forma $ax + by \leqslant c$, e uma outra função linear $f(x, y)$, escreva um algoritmo que encontre o valor de (x, y) que maximiza $f(x, y)$ e satisfaz todas as desigualdades. Seu algoritmo deve ter complexidade de tempo linear no número de desigualdades. Este problema é chamado de programação linear com duas variáveis.

*7.5) No algoritmo da figura 7.2, substitua a chamada a MedianaAproximada pela escolha de um elemento aleatório de S, com distribuição uniforme. Prove que o valor esperado da complexidade de tempo do algoritmo se mantém linear em $|S|$.

CAPíTULO 8

Construção Incremental

O método de construção incremental consiste em, inicialmente, resolver o problema para um sub-conjunto pequeno dos elementos da entrada e, então, adicionar os demais elementos um a um. Em muitos casos, se os elementos forem adicionados em uma ordem ruim, o algoritmo não será eficiente. Uma alternativa é escolher uma ordem conveniente para adicionar os elementos. Outra opção é permutar aleatoriamente os elementos e fazer a análise de complexidade como uma esperança em função desta permutação aleatória.

8.1. Arranjo de Retas

Um conjunto de retas no plano divide-o em várias regiões e as retas se interceptam em vários pontos. Esta estrutura geométrica fundamental é o que chamamos de arranjo de retas. Existem diversas aplicações desta estrutura para resolver vários problemas geométricos.

Dado um conjunto de retas no plano, podemos falar em vértices (pontos onde retas se interceptam), arestas (segmento de reta, possivelmente infinito, que não é interceptado por nenhuma outra reta) e faces (regiões parcialmente limitadas pelas arestas). Isto é ilustrado na figura 8.1(a). Para trabalharmos com este arranjo de retas, devemos representá-lo em uma estrutura de subdivisão do plano como a estrutura DCEL (sessão 2.3).

Problema 19. *Dado um conjunto S de retas no plano, construa seu arranjo em uma estrutura DCEL.*

Para que nosso algoritmo funcione corretamente, devemos criar 4 retas especiais que formam um grande retângulo contendo o arranjo (figura 8.1(b)). Todo par de retas de S deve se interceptar somente no interior deste retângulo. Podemos considerar os vértices deste retângulo como tendo coordenadas infinitas ou encontrar as interseções extremas (isto pode ser feito trivialmente em tempo $O(n^2)$).

Nosso algoritmo usa o paradigma de construção incremental. Vamos iniciá-lo com o arranjo que contém apenas as quatro arestas do retângulo envoltório. Como procedemos para inserir uma reta em um arranjo que já contém algumas retas? Primeiro temos que determinar qual a aresta do retângulo é interceptada pelo extremo esquerdo da reta. Isto pode ser verificado em tempo $O(1)$ examinando os lados esquerdo, de cima e de baixo do retângulo. Seguindo os segmentos da

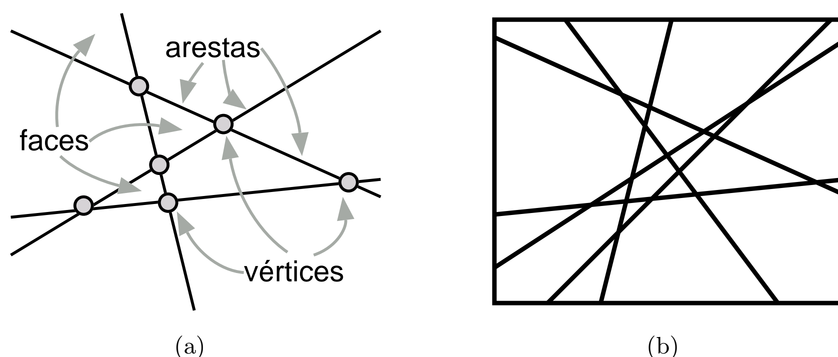
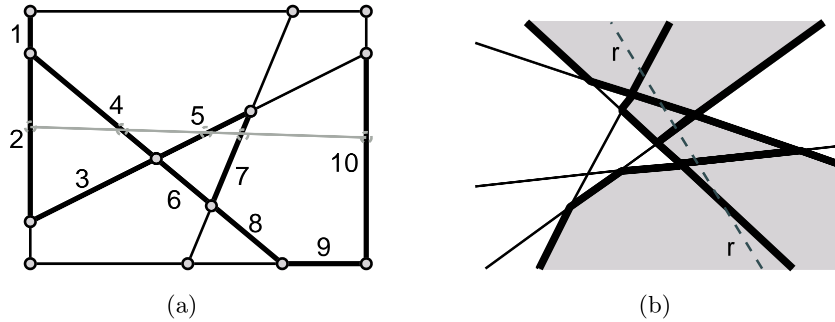


Figura 8.1. (a) Vértices, arestas e faces em um arranjo de retas. (b) Arranjo de retas colocado dentro de um retângulo.



(a) (b)

FIGURA 8.2. (a) Ordem em que as arestas são examinadas quando uma nova reta (em cinza) é inserida. (b) Vizinhança da reta r.

face externa do retangulo, achamos qual aresta do arranjo é interceptada. Então, dividimos esta aresta quebrando-a em um novo vértice. Para descobrirmos qual a próxima aresta do arranjo que deve ser quebrada pelo acréscimo de um novo vértice, percorremos sequencialmente as arestas da face interceptada pela reta. Percorreremos sempre estas arestas no sentido anti-horário, como ilustra a figura 8.2(a). Este procedimento se repete até chegarmos em outro lado do retângulo envoltório.

Deste modo, o algoritmo constrói o arranjo de retas usando sucessivas inserções, em qualquer ordem. A primeira vista, o algoritmo não parece muito eficiente. Uma análise superficial da complexidade de tempo do algoritmo indica que, a cada reta inserida, é necessário examinar $O(n^2)$ arestas. Deste modo, a complexidade de tempo do algoritmo é $O(n^3)$. Felizmente, podemos refinar nossa análise e provar que o algoritmo tem complexidade $\Theta(n^2)$. Para provarmos este fato, precisamos mostrar que o número de arestas examinadas ao inserir a n-ésima reta no arranjo é $O(n)$, e não apenas $O(n^2)$ como é fácil perceber.

Definimos a vizinhança de uma reta no arranjo como o conjunto de arestas que pertencem as faces interceptadas por esta reta (figura 8.2(b)). Claramente, só as arestas da vizinhança da reta inserida são candidatas a ser examinadas. O teorema abaixo é chamado de teorema da vizinhança.

TEOREMA 8.1. *O número de arestas na vizinhança de uma reta r em um arranjo de n retas tem no máximo $6n$ arestas.*

DEMONSTRAÇÃO. Nossa prova será por indução em n, mas, antes de começarmos a indução, vamos dividir as arestas da vizinhança em dois conjuntos: arestas esquerdas e arestas direitas . Uma aresta esquerda é aquela que limita o bordo esquerdo de uma face da vizinhança (figura 8.3(a)). Uma aresta direita é aquela que limita o bordo direito de uma face da vizinhança. Algumas arestas podem ser ao mesmo tempo esquerdas e direitas, por fazerem parte de duas células diferentes. Essas arestas serão contadas duas vezes. Provaremos que o número de arestas esquerdas na vizinhança não excede $3n$, deste modo provando o teorema.

Para tornarmos nossa explicação mais clara, vamos considerar que a reta r seja horizontal. Nosso argumento não fará a indução nas retas em qualquer ordem, mas sim da esquerda para a direita segundo as interseções com r. A escolha da ordem em que os elementos são adicionados pode simplificar extremamente uma prova por indução. No caso base com $n = 1$, temos apenas uma aresta esquerda na vizinhança de r, portanto a hipótese é válida para o caso base.

Suponha que um arranjo com $n - 1$ retas possui no máximo $3(n - 1)$ arestas na vizinhança esquerda. Provaremos que a inclusão de uma reta l_n que intercepta r a direita de todas as demais acrescenta no máximo 3 arestas a vizinhança de r, assim a hipótese vale para n.

A primeira aresta esquerda nova que notamos com a inclusão da reta l_n é formada pela própria reta l_n. Como l_n intercepta r a direita de todas as demais retas e estamos contando apenas as arestas esquerdas, esta é a única aresta nova sobre a própria reta l_n.

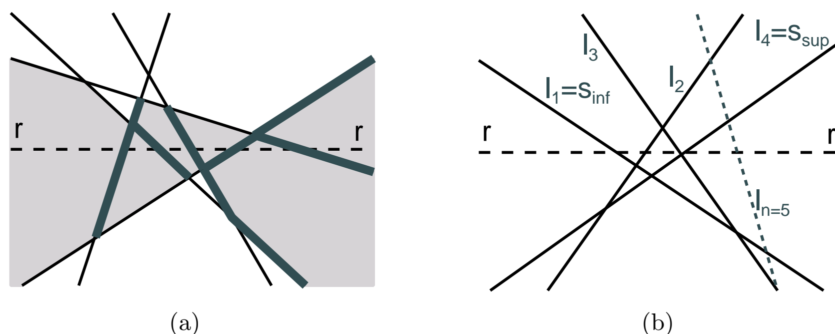


FIGURA 8.3. (a) Vizinhança com as arestas esquerdas da vizinhança destacadas. (b) Ilustração do argumento indutivo da prova do teorema da vizinhança.

Temos que contar mais duas arestas que podem criadas com a inclusão de l_n. Estas arestas são formadas devido a l_n poder cortar duas arestas esquerdas de vizinhança de r, uma acima e uma abaixo de r, na face extrema direita da vizinhança de r. Precisamos ainda garantir que nenhuma outra aresta esquerda da vizinhança de r é interceptada. Vamos chamar de s_{sup} e s_{inf} as retas que contém as arestas esquerdas interceptadas por l_n na face extrema esquerda da vizinhança de r. Estas retas ficam entre l_n e pontos de r a direita de l_n. Este parágrafo está exemplificado na figura 8.3(b).

Como tanto o número de arestas esquerdas da vizinhança quanto o número de arestas direitas da vizinhança (por argumento análogo) é no máximo $3n$, o total de arestas da vizinhança não excede $6n$. □

Como vimos, só as arestas da vizinhança da nova reta adicionada pelo algoritmo incremental são candidatas a serem percorridas nesta adição. Deste modo, a complexidade de tempo de adicionar uma reta em um arranjo com n retas é $O(n)$. Assim, para adicionarmos todas as n retas, a complexidade total de tempo é $O(n^2)$. Como o número máximo de vértices (assim como o número de faces e arestas) do arranjo de retas é $\Theta(n^2)$, então o nosso algoritmo é ótimo.

8.2. Fecho Convexo: Algoritmo de Graham

Como vimos na sessão 4.1, o fecho convexo de um conjunto de pontos no plano é o menor polígono convexo que envolve todos os pontos do conjunto. Na sessão 4.1, apresentamos um algoritmo de complexidade $O(nh)$, onde n é o número de pontos da entrada e h é o número de pontos da saída, ou seja, os vértices do fecho convexo. No exercício 5.5, pede-se que, usando o paradigma de divisão e conquista, se escreva um algoritmo que determine o fecho convexo em tempo $O(n \lg n)$. No exercício 5.6, pedimos um algoritmo de complexidade de tempo $O(n \lg h)$, usando a função que encontra uma ponte do fecho convexo em tempo linear vista na sessão 7.3. Usando árvores de decisão algébricas, foi provado que não é possível resolver o problema de fecho convexo em tempo menor que $O(n \lg h)$, em função dos parâmetros n e h.

Nesta sessão, fazemos o aparentemente impossível: apresentamos um algoritmo que constrói o fecho convexo de um conjunto de pontos no plano em tempo linear. Como fazemos esta mágica? Reposta: modificamos ligeiramente a entrada do nosso problema. A entrada do problema não é mais um conjunto de pontos do plano, mas sim, um conjunto de pontos do plano *ordenado segundo o eixo x*.

Como a complexidade de tempo da ordenação é $O(n \lg n)$, caso os pontos não estejam ordenados convenientemente, o nosso algoritmo não leva tempo $O(n)$, mas sim $O(n \lg n)$. Mesmo neste caso, o algoritmo que apresentamos é extremamente eficiente na prática, pois os algoritmos de ordenação são muito rápidos e não necessitam de fazer contas com as coordenadas dos pontos (geralmente números de ponto flutuante). Chamamos este algoritmo de algoritmo de Graham (embora o algoritmo originalmente proposto por Graham não use a ordenação segundo o eixo x, mas sim uma ordenação angular).

PROBLEMA 20. *Dado um conjunto S de n pontos do plano, ordenados segundo o eixo x, determinar seu fecho convexo.*

Na maioria dos algoritmos para resolver o problema de fecho convexo, a explicação torna-se mais simples quando o fecho convexo é dividido em duas partes: fecho convexo superior e fecho convexo inferior, como ilustra a figura 8.4. Nosso algoritmo determinará apenas o fecho convexo superior. A determinação do fecho convexo inferior é análoga e juntar os dois em um único polígono convexo é trivial. Para simplificarmos a explicação, também não consideraremos o caso em que dois pontos possuem a mesma coordenada x.

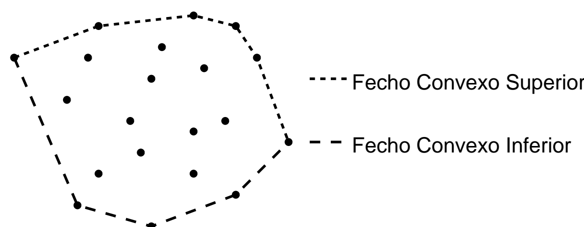


FIGURA 8.4. Fecho convexo superior e fecho convexo inferior.

O fecho convexo superior de um único ponto é o próprio ponto. O fecho convexo superior de um par de pontos é a aresta que une estes pontos. Dado o fecho convexo superior de um conjunto de pontos, como podemos adicionar mais um ponto no conjunto? Caso o ponto adicionado esteja sob o fecho convexo superior, não há nada a ser feito. Caso contrário, temos que descobrir como atualizar o fecho convexo superior. Fazer esta atualização pode não parecer muito simples. Porém, podemos modificar um pouco nosso algoritmo de modo a não precisarmos considerar a inserção de um ponto qualquer. Fazemos isso modificando a ordem com que os pontos são inseridos.

Na sessão anterior, o nosso algoritmo incremental acrescentava os elementos da entrada em qualquer ordem. Na grande maioria dos casos, este procedimento não conduz a algoritmos eficientes. Muitas vezes, é preciso descobrir uma ordem conveniente para adicionar os elementos em nossa construção incremental. Nesta sessão, acrescentaremos os pontos da esquerda para a direita. Deste modo, é necessário que a entrada esteja ordenada segundo o eixo x ou, caso contrário, que façamos esta ordenação.

Assim, a pergunta que precisamos responder é: Dado o fecho convexo superior de um conjunto de pontos, como podemos adicionar mais um ponto a direita dos demais pontos do conjunto? Certamente o novo ponto adicionado fará parte do fecho convexo superior. Precisamos descobrir a que outro vértice devemos conectá-lo, removendo os vértices intermediários do fecho convexo superior, como ilustra a figura 8.5(a). Para isto, basta percorrermos as arestas do fecho convexo da direita para a esquerda, examinando o ângulo entre o novo ponto e cada aresta. Caso o ângulo seja maior que 180°, seguimos para a próxima aresta, como ilustra a figura 8.5(b). Pela definição de polígono convexo como um polígono que tem todos os ângulos internos menores que 180°, o algoritmo funciona corretamente. O pseudo código deste algoritmo está na figura 8.6.

Uma análise superficial da complexidade de tempo do algoritmo, mostra que o algoritmo tem complexidade $O(n^2)$, pois a cada ponto inserido, podemos examinar no máximo um número linear de pontos. Porém, é possível refinar a análise e mostrar que a complexidade de tempo do algoritmo é bem menor, sendo $O(n)$. Para isto, argumentamos que, ao adicionarmos um ponto, todos os pontos examinados, com exceção do último ponto examinado, são eliminados do fecho convexo, não sendo candidatos a serem examinados novamente. Assim, embora a complexidade de tempo de uma única inserção de um novo ponto seja linear, a soma da complexidade de tempo de todas as inserções também é linear.

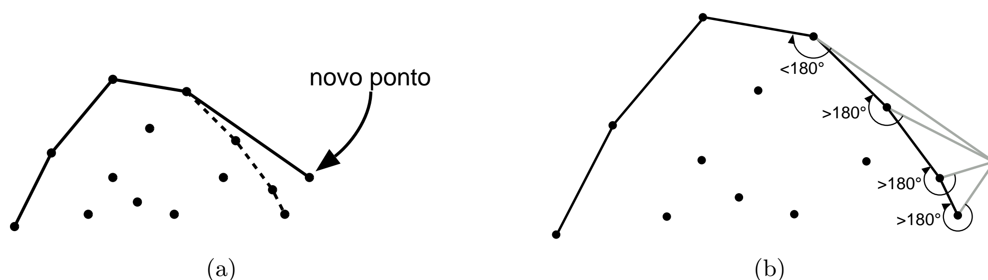


FIGURA 8.5. (a) Alteração do fecho convexo superior pela adição de um novo ponto a direita dos demais. (b) Critério de escolha dos vértices a serem removidos.

Entrada:
  S: Conjunto de no mínimo dois pontos no plano, ordenado segundo o eixo x.
Saída:
  O fecho convexo superior de S, da esquerda para a direita.
Observações:
  $\circlearrowright(p_1, p_2, p_3)$: Ângulo entre os pontos $p_1p_2p_3$, medido no sentido horário.

FechoConvexoSuperior(S)
  $FC[1] \leftarrow S[1]$
  $h \leftarrow 1$
  Para i de 2 até $|S|$
    Enquanto $h \geqslant 2$ e $\circlearrowright(S[i], FC[h], FC[h-1]) > 180°$
      $h \leftarrow h - 1$
    $h \leftarrow h + 1$
    $FC[h] \leftarrow S[i]$
  Retorne FC

FIGURA 8.6. Algoritmo de Graham determinando o fecho convexo superior.

8.3. Programação Linear com Duas Variáveis

Um problema de programação linear com d variáveis consiste em determinar o vetor d-dimensional X que maximiza a função linear $f = CX$, satisfazendo um sistema de desigualdades lineares $AX \leqslant B$, onde A é uma matriz $n \times d$. A função f a ser maximizada é chamada de função objetivo. Sem usar notação matricial temos:

$$\begin{array}{rl} \text{Maximizar:} & f = c_1x_1 + c_2x_2 + \ldots + c_dx_d \\ \text{Satisfazendo:} & a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,d}x_d \leqslant b_1 \\ & a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,d}x_d \leqslant b_2 \\ & \vdots \\ & a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,d}x_d \leqslant b_n \end{array}.$$

Geometricamente, podemos pensar nas desigualdades lineares como semi-espaços d-dimensionais. Maximizar uma função linear sujeita a estas desigualdades consiste em determinar o ponto extremo na direção C contido na interseção destes semi-espaços. No caso com apenas duas variáveis ($d = 2$), o problema pode ser formulado no plano como obter o ponto extremo em uma direção, na interseção de semi-planos (figura 8.7(a)), chamada de região viável. O problema com duas variáveis pode ser escrito como:

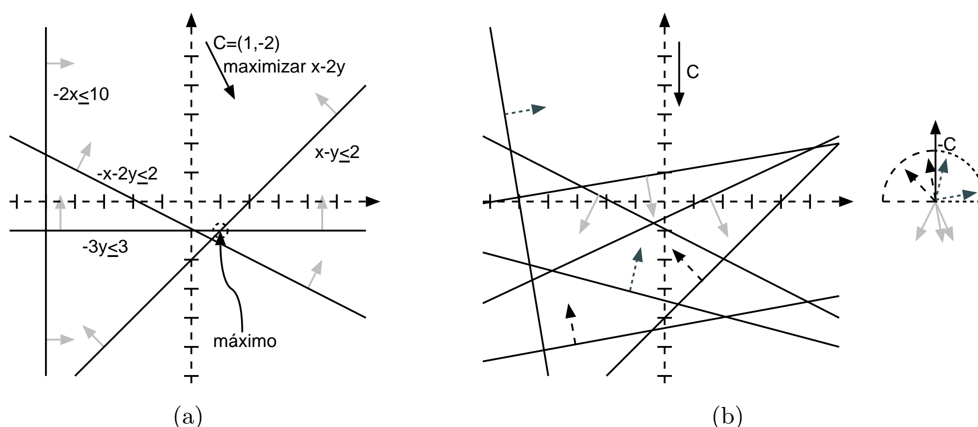


FIGURA 8.7. (a) Interpretação geométrica de um problema de programação linear com duas variáveis. (b) Método usado para obter duas restrições iniciais que limitam o problema, ou determinar que o problema é ilimitado.

$$\begin{array}{rl} \text{Maximizar:} & f = c_1 x + c_2 y \\ \text{Satisfazendo:} & a_{1,1}x + a_{1,2}y \leqslant b_1 \\ & a_{2,1}x + a_{2,2}y \leqslant b_2 \\ & \vdots \\ & a_{n,1}x + a_{n,2}y \leqslant b_n \end{array} .$$

PROBLEMA 21. *Dados um conjunto de desigualdades lineares em duas variáveis x, y e uma função linear $f(x, y)$, encontrar o valor de x, y que satisfaz as desigualdades e maximiza f.*

Há alguns casos especiais que precisam ser tratados com cuidado. Um caso é quando não é possível satisfazer simultaneamente todas as desigualdades. Neste caso não há solução e dizemos que o problema é inviável (figura 8.8(a)). Outro caso acontece quando podemos aumentar arbitrariamente o valor da função objetivo f, satisfazendo todas as desigualdades, ou seja, a região viável é aberta na direção C. Neste caso, dizemos que o problema é ilimitado (figura 8.8(b)) e também não retornamos uma solução.

O caso do problema poder ser ilimitado é o primeiro que trataremos. Desejamos agora obter um algoritmo que ou diga que o problema é ilimitado, ou encontre duas desigualdades que sozinhas limitem o problema. Para isto, basta olharmos para o vetor C correspondente a direção de maximização e os vetores normais as desigualdades que apontam para o lado onde a desigualdade é satisfeita (linhas de A com sinal invertido). Caso o problema seja limitado, existem duas desigualdades cujos vetores normais formam ângulo menor que 90° com o vetor $-C$, nos dois sentidos (figura 8.7(b)). Graficamente, desenhamos os vetores perpendiculares a todas as desigualdades em uma mesma origem, assim como o vetor simétrico a direção de maximização. Escolhemos então dois vetores com ângulo menor que 90°, para esquerda e direita, com o vetor simétrico a direção de maximização. Claramente, isto pode ser feito em tempo linear no número de desigualdadades.

Graças ao procedimento do parágrafo anterior, não só podemos nos concentrar apenas em problemas limitados, como também sabemos como obter duas restrições que limitam o problema, caso elas existam. Esta é a condição inicial do nosso algoritmo incremental. Acrescentaremos as restrições uma a uma, atualizando o nosso ponto de máximo satisfazendo as restrições.

Temos, então, um conjunto de restrições e um vértice v que satisfaz estas restrições e maximiza a função objetivo. Como fazemos para acrescentar uma nova restrição? Existem duas situações que devemos considerar. Em uma delas a nova restrição já era satisfeita pelo vértice v e não há nada a ser feito. Verificar se v satisfaz a nova restrição é uma questão simples de substituir as coordenadas de v na desigualdade e testá-la, podendo ser feito em tempo $O(1)$.

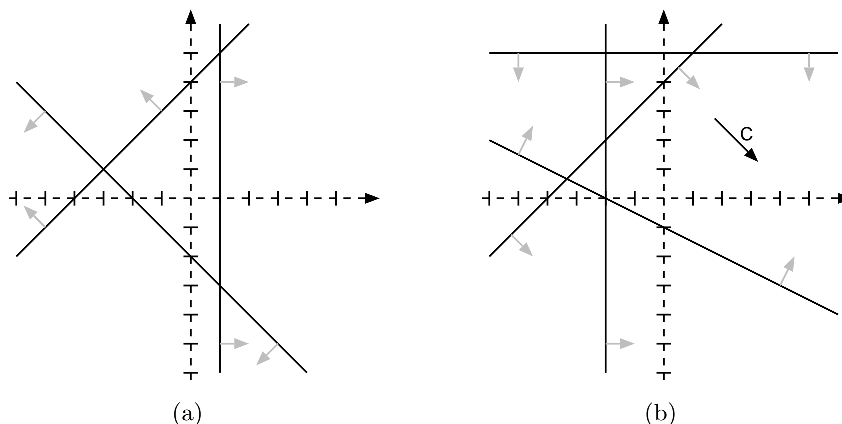


FIGURA 8.8. (a) Problema de programação linear inviável. (b) Problema de programação linear ilimitado.

A outra situação é quando o vértice de máximo v não satisfaz a nova desigualdade. Neste caso, precisamos encontrar o novo vértice ótimo. Para isto, note que certamente uma das duas desigualdades que definem este vértice tem que ser a desigualdade que acabamos de acrescentar. Com isto, podemos limitar nossa busca aos pontos sobre a reta que acabamos de acrescentar, ou seja, temos que resolver um problema de programação linear em apenas uma variável, como o ilustrado na figura 8.9. Este problema pode ser resolvido trivialmente em tempo linear no número de desigualdades. É possível também que, neste procedimento, não encontremos nenhum ponto viável. Neste caso, podemos afirmar que o problema é inviável, pois não há solução que satisfaz a nova restrição ao mesmo tempo que todas as anteriores.

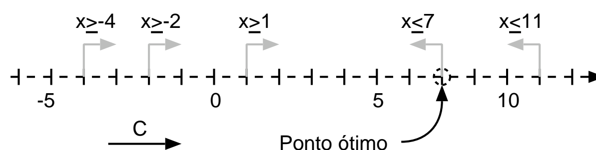


FIGURA 8.9. Problema de programação linear com apenas uma variável.

Assim, para acrescentarmos uma desigualdade em um problema com i desigualdades a complexidade de tempo é $O(i)$. Para acrescentarmos, uma a uma, as n desigualdades, começando com 2 desigualdades, a complexidade de tempo é:

$$\sum_{i=2}^{n-1} O(i) = O(n^2)$$

Desejamos melhorar esta complexidade de tempo para $O(n)$. Como podemos fazer isto? Uma idéia é escolhermos convenientemente a ordem em que as restrições são acrescentadas pelo método incremental. Isto é mais ou menos o que faremos. De fato, escolher esta ordem é um problema bastante complicado, mas podemos lançar mão da probabilidade. Escolhemos uma ordem aleatória e argumentamos que o valor esperado da complexidade de tempo do algoritmo é $O(n)$. Note que este valor esperado depende apenas da ordem aleatória com que acrescentamos as restrições, e não da entrada do problema em si. Esta ordem aleatória será uma distribuição uniforme das permutações das restrições (com exceção das duas restrições iniciais). Este tipo de permutação aleatória pode ser construída com um algorimo semelhante ao da figura 8.10, que gera uma permutação aleatória de um vetor. O pseudo-código do algoritmo completo está na figura 8.11.

Entrada:
v: Vetor a ser permutado.
n: Tamanho de v.
Saída:
O vetor v será permutado aleatoriamente.
Observações:
rand(n): número aleatório distribuido uniformemente de 1 até n

PermutaçãoAleatória(v, n)
Para i decrescendo de n até 2
Troca $v[i]$ com $v[\text{rand}(i)]$

FIGURA 8.10. Algoritmo que permuta aleatoriamente um vetor.

Precisamos calcular qual a probabilidade $p(i)$ da solução do problema com as primeiras i restrições (segundo nossa ordem aleatória) ser diferente da solução onde se acrescenta a $(i+1)$-ésima restrição. Estas soluções são diferentes se e só se a $(i+1)$-ésima restrição é uma das *duas* restrições que definem o vértice de máximo v. Como estamos falando de uma restrição aleatória em um universo com $i+1$ restrições, a probabilidade disto ocorrer é $2/(i+1)$. Assim, o valor esperado da complexidade de tempo do nosso algoritmo é

$$\sum_{i=2}^{n-1} O(1/i)O(i) = \sum_{i=2}^{n-1} O(1) = O(n)$$

Este algoritmo é bem simples de implementar e bastante eficiente na prática. Algoritmos randomizados como este, onde a complexidade de tempo é uma esperança que independe da entrada, são excelentes alternativas em várias situações.

8.4. Resumo e Observações Finais

Neste capítulo, examinamos um paradigma bastante natural para o desenvolvimento de algoritmos, chamado de construção incremental. Começamos resolvendo um problema trivialmente pequeno e, então, adicionamos os elementos da entrada um a um, atualizando a solução.

O primeiro problema estudado é armazenar um arranjo de retas em uma estrutura DCEL. Neste problema, não nos importamos com a ordem com que os elementos são inseridos. Qualquer que seja ela, a complexidade de tempo do algoritmo é $O(n^2)$, devido ao teorema da vizinhança.

No problema do fecho convexo, podemos tornar nosso algoritmo mais simples inserindo os pontos da esquerda para a direita. Deste modo, conseguimos um algoritmo que, uma vez tendo os pontos ordenados, determina seu fecho convexo em tempo linear no número de pontos.

No problema de programação linear com duas variáveis, ao invés de determinarmos uma boa ordem para inserir os elementos da entrada, preferimos inseri-los segundo uma ordem aleatória. Deste modo, conseguimos uma boa esperança da complexidade de tempo. Note que esta esperança independe da entrada, dependendo apenas da permutação aleatória usada pelo algoritmo. Assim, não há entradas ruins que podem fazer com que o algoritmo demore mais que o desejado.

Exercícios

8.1) Escreva um algoritmo incremental para determinar o maior elemento em um conjunto com n números reais em tempo $O(n)$.

8.2) Escreva um algoritmo para ordenar um conjunto de n números reais usando o paradigma de construção incremental. A complexidade de tempo do seu algoritmo deve ser $O(n^2)$. Qual seria a complexidade de tempo do seu algoritmo em uma máquina que pudesse

```
Entrada:
    n: Número de desigualdades.
    A: Matriz n × 2 de números reais.
    B: Vetor com n números reais.
    C: Vetor com 2 números reais.
Saída:
    X: Vetor com 2 elementos que maximiza CX satisfazendo AX ⩽ B.

ProgLin(n, A, B, C)
    // Determina duas retas que limitam o problema
    Para i de 1 até n
        Se (A[i][1], A[i][2]) está à esquerda de (C[1], C[2]), fazendo um ângulo de até 90°
            v[1] ← i
            Sai do loop
    Para i de 1 até n
        Se (A[i][1], A[i][2]) está à direita de (C[1], C[2]), fazendo um ângulo de menos de 90°
            v[2] ← i
            Sai do loop
    Se v[1] ou v[2] não foi definido
        Retorne "problema ilimitado"
    // Acrescenta índice das demais retas ao vetor
    j ← 3
    Para i de 1 até n
        Se i ≠ v[1] e i ≠ v[2]
            v[j] ← i
            j ← j + 1
    // Permuta aleatoriamente o vetor, exceto as 2 primeiras posições
    PermutaçãoAleatória(v + 2, n − 2)
    X ← interseção das retas correspondentes as linhas v[1] e v[2]
    // Início da construção incremental
    Para i de 3 até n
        j ← v[i]
        Se X viola restrição correspondente a linha j
            X ← vértice sobre reta da linha j que maximiza CX e satisfaz desigualdades
correspondente as linhas com índices de v[1] até v[i]
            Se X não existe
                Retorne "problema inviável"
    Retorne X
```

FIGURA 8.11. Solução do problema de programação linear (problema 21).

mover um segmento contínuo de dados de uma região da memória para outra em tempo constante, independente do tamanho do segmento?

8.3) Use uma estrutura de dados como, por exemplo, árvores rubro-negras ou AVL para melhorar a complexidade de tempo do algoritmo do exercício anterior para $O(n \lg n)$.

8.4) Escreva um algoritmo que, dado um conjunto de retas no plano, decida se 3 ou mais retas do conjunto se interceptam em um mesmo ponto. Sugestão: use o algoritmo para gerar a estrutura DCEL de um arranjo de retas.

8.5) Dado um conjunto S de pontos no plano cartesiano, um ponto $p \in S$ é considerado um ponto de máximo se, para todo $p' \in S$, ou a coordenada x de p é maior que a coordenada

x de p' ou a coordenada y de p é maior que a coordenada y de p'. Escreva um algoritmo para determinar todos os pontos de máximo de um conjunto S. Estabeleça uma relação entre os pontos de máximo e os vértices do fecho convexo.

8.6) Uma triangulação de um conjunto S de pontos no plano é uma subdivisão de seu fecho convexo em triângulos disjuntos (exceto em seus bordos), onde os vértices dos triângulos são exatamente os pontos de S (figura 5.8(a)). Escreva um algoritmo para computar uma triangulação de um conjunto de pontos no plano, previamente ordenados segundo o eixo x. A complexidade de tempo do seu algoritmo deve ser linear no número de pontos.

8.7) Generalise o algoritmo que resolve o problema de programação linear com duas variáveis para d variáveis. Mostre que a complexidade de tempo do algoritmo randomizado é $O(d!n)$.

*8.8) Dado um conjunto de n pontos no plano, escreva um algoritmo randomizado que, em tempo $O(n)$, determine o menor círculo que contém todos os pontos do conjunto.

CAPíTULO 9

Refinamento de Solução

A técnica de refinamento de solução consiste em partir de uma configuração inicial que não é uma solução correta para o problema e melhorar sucessivamente esta configuração até obter uma solução correta. Esta técnica é freqüentemente usada em problemas de otimização, onde se parte de uma configuração que satisfaz todas as restrições, mas não maximiza a propriedade desejada. Então, modifica-se sucessivamente a configuração, sem deixar de satisfazer as restrições, e sempre aumentando o valor da propriedade desejada. Para cada problema, é importante provar que o método usado para refinar a solução sempre leva a uma solução correta, lembrando que o problema pode possuir máximos locais que são diferentes do máximo global.

9.1. Fluxo em Redes

Grafos direcionados são um modelo natural para diversos problemas práticos. Um tipo de problema freqüentemente modelado com grafos direcionados é o de transportar um produto usando um conjunto de vias, como, por exemplo, levar água através de canos, energia elétrica através de fios, pacotes de dados em uma rede de computadores etc. Na prática, estas vias tem uma vazão limitada, como a corrente máxima suportada por um fio ou o número de bits por segundo que um canal é capaz de transmitir. Como devemos proceder para maximizarmos o transporte de uma origem até um destino determinados? Este é o problema que resolvemos nessa sessão.

Chamamos de rede um grafo direcionado com pesos nas arestas e dois vértices especiais s e t, tal que nenhuma aresta entra em s e nenhuma aresta sai de t. Em outras palavras, s é uma fonte e t é um sumidouro. O vértice s é chamado de origem e corresponde ao ponto de partida do nosso transporte. O vértice t é chamado de destino e corresponde ao ponto de chegada do nosso transporte. O peso $c(e)$ de uma aresta e é chamado de capacidade de e e define, por exemplo, a vazão do canal de comunicação correspondente ou a corrente máxima suportada no fio correspondente. Uma rede está representada graficamente na figura 9.1(a).

Chamamos de fluxo $f(e)$ de uma aresta e o quanto está sendo efetivamente transportado por aquela aresta. O fluxo em uma aresta deve ser um valor entre 0 e a capacidade da aresta, ou seja, $0 \leqslant f(e) \leqslant c(e)$. O fluxo de uma rede é uma atribuição de fluxos a suas arestas tal que a soma dos fluxos das arestas que entram em um vértice seja igual a soma dos fluxos das arestas que saem daquele vértice, com exceção da fonte e do sumidouro. Observe que a soma dos fluxos

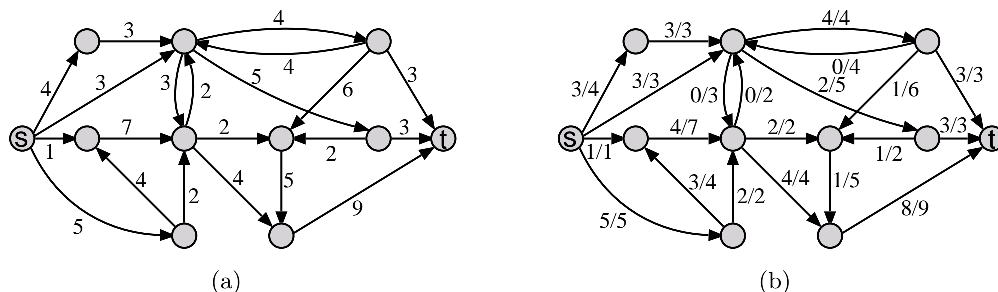


Figura 9.1. (a) Representação de uma rede. (b) Rede com o fluxo máximo onde está representado ao lado de cada aresta a capacidade / o transporte da aresta.

das arestas saindo da fonte é igual a soma dos fluxos das arestas que entram no sumidouro. Chamamos este número de valor do fluxo da rede e denotamos o valor do fluxo f por $|f|$. Um fluxo de uma rede é máximo quando seu valor é máximo dentre todos os fluxos da rede. Um fluxo máximo da rede da figura 9.1(a) está representado na figura 9.1(b).

PROBLEMA 22. *Dada uma rede, determinar seu fluxo máximo.*

A idéia do método de refinamento de solução é partir de um fluxo inicial e tentar aumentar este fluxo até obter um fluxo máximo. Como podemos fazer isso? Dizemos que uma aresta e está saturada quando $f(e) = c(e)$, ou seja, o fluxo que passa por esta aresta não pode ser aumentado. Se um fluxo em uma rede possui um caminho que leva da origem ao destino tal que nenhuma aresta do caminho esteja saturada, então certamente podemos aumentar este fluxo até saturarmos alguma das arestas desse caminho. Assim, o nosso algoritmo pode, a cada iteração, encontrar um caminho da origem ao destino sem arestas saturadas e aumentar o fluxo. O algoritmo termina quando não houver caminho da origem ao destino sem arestas saturadas. Será que este método leva necessariamente ao fluxo máximo? A resposta é não. Veja na figura 9.2(a) um exemplo de fluxo onde todo o caminho da origem ao destino possui arestas saturadas e na figura 9.2(b) outro fluxo de valor maior para a mesma rede.

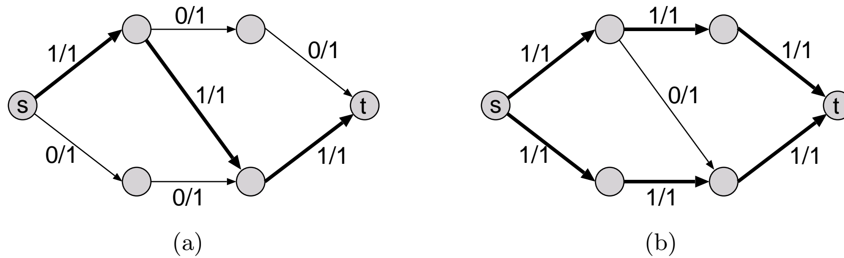


FIGURA 9.2. (a) Fluxo de valor 1 onde todo o caminho da origem ao destino possui aresta saturada. (b) Fluxo de valor 2 para a mesma rede.

A solução para este problema é, no lugar de procurarmos caminhos da origem ao destino na própria rede em questão, procurarmos este caminho na chamada rede residual. Dados uma rede D e um fluxo f, vamos definir a rede residual D_f. Os vértices, a origem e o destino de D e D_f são os mesmos. Para cada aresta direcionada $e = (v_1, v_2) \in E(D)$, criamos duas arestas $e' = (v_1, v_2)$ e $e'' = (v_2, v_1)$ na rede residual D_f. A capacidade da aresta $e' \in E(D_f)$ é $c(e') = c(e) - f(e)$. A capacidade da aresta $e'' \in E(D_f)$ é $c(e'') = f(e)$. Caso alguma dessas arestas tenha capacidade 0, devemos remover a aresta da rede residual. Caso tenhamos arestas direcionadas duplicadas, substituímos estas arestas por uma única aresta cuja capacidade é a soma das capacidades das arestas removidas. A rede residual do fluxo da figura 9.3(a) está na figura 9.3(b).

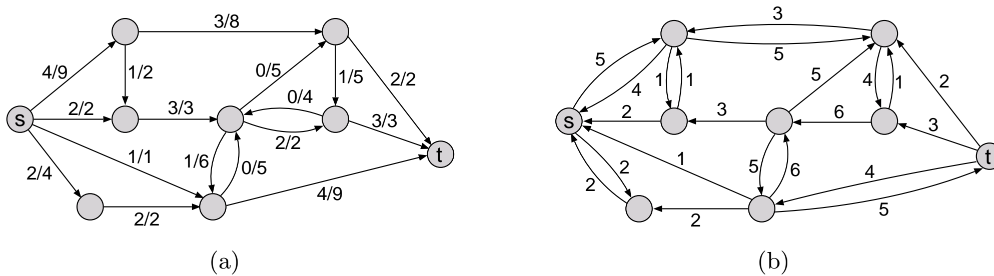


FIGURA 9.3. (a) Rede com um fluxo. (b) Rede residual do fluxo da figura (a).

Qual o significado da rede residual? A capacidade das arestas da rede residual D_f correspondem as variações que o fluxo f pode sofrer. Ao procurarmos um caminho da origem

ao destino na rede original, que não tivesse arestas saturadas, não nos permitíamos reduzir o fluxo por nenhuma aresta. Porém, usando a rede residual D_f, podemos colocar um fluxo em uma aresta e no sentido contrário ao fluxo $f(e)$ que passava originalmente por e, deste modo reduzindo o fluxo por esta aresta. Claramente, qualquer caminho da origem ao destino na rede residual D_f corresponde a um aumento no valor do fluxo f. Estes caminhos são chamados de caminhos aumentantes. O valor do novo fluxo será acrescido da capacidade da aresta de menor capacidade no caminho aumentante. Deste modo, o algoritmo procede encontrando caminhos na rede residual, aumentando o fluxo e construindo uma nova rede residual, até não existir mais caminho da origem ao destino na rede residual. Este algoritmo é chamado de algoritmo de Ford-Fulkerson. O pseudo-código do algoritmo encontra-se na figura 9.4. Será que este algoritmo realmente encontra o fluxo máximo? A resposta é sim, mas para provarmos este fato temos que definir alguns termos e provar um teorema importante.

Entrada:
D: Digrafo com capacidades associadas as arestas.
s: Vértice origem. Deve ser uma fonte em D.
t: Vértice destino. Deve ser um sumidouro em D.
Saída:
f: Fluxo máximo de s para t em D, onde $f[e]$ é o fluxo pela aresta e.

```
FluxoMáximo(D,s,t)
    f ← fluxo nulo em todas as arestas
    Enquanto existir caminho p de s para t em D_f
        min ← capacidade mínima dentre arestas de p
        Para toda aresta e de p
            f[e] = f[e] + min
    Retorne f
```

Figura 9.4. Pseudo-código do algoritmo de Ford-Fulkerson para fluxo máximo em redes.

Um corte (S, T) em uma rede D é uma partição dos vértices de D em dois conjuntos S e T tais que $s \in S$ e $t \in T$. O valor de um corte (S, T) é a soma das capacidades das arestas direcionadas (u, v) tais que $u \in S$ e $v \in T$ e é denotado por $|(S, T)|$. Um corte mínimo é um corte que tem valor mínimo.

Teorema 9.1. *Em uma rede D, o valor do corte mínimo é igual ao valor do fluxo máximo, ou seja, se f_{max} é um fluxo máximo em D e (S_{min}, T_{min}) é um corte mínimo em D, então $|f_{max}| = |(S_{min}, T_{min})|$.*

Demonstração. Primeiro vamos provar que $|f_{max}| \geqslant |(S_{min}, T_{min})|$. Como não existe caminho de s para t na rede residual do fluxo f_{max}, então todas as arestas de S_{min} para T_{min} estão saturadas, e todas as arestas de T_{min} para S_{min} tem fluxo zero, de modo que $|f_{max}| \geqslant |(S_{min}, T_{min})|$.

Agora vamos provar que, se f é um fluxo e (S, T) é um corte, então $|f| \leqslant |(S, T)|$. Consequentemente, $|f_{max}| \leqslant |(S_{min}, T_{min})|$. Pela conservação do fluxo, se somarmos o valor de um fluxo f nas arestas de S para T de um corte (S, T) qualquer, obtemos exatamente $|f|$. Como este fluxo de S para T não pode ser maior que soma das capacidades das arestas de S para T, que é o valor do corte, a afirmação é verdadeira. □

O algoritmo de Ford-Fulkerson pode aumentar sucessivamente o valor do fluxo usando caminhos na rede residual. Além disso, pelo teorema acima, caso o algoritmo alcance um fluxo que não consegue mais aumentar, ou seja, não há caminho de s para t na rede residual, então

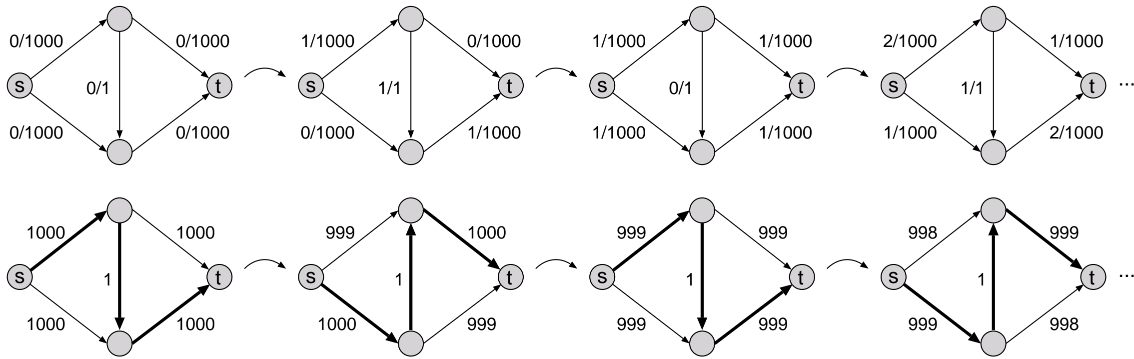


FIGURA 9.5. Caso ruim do algoritmo de Ford-Fulkerson. Os fluxos estão representados na linha de cima e a rede residual correspondente está representada abaixo.

o fluxo obtido é máximo. Porém, precisamos provar que o algoritmo sempre termina em um tempo finito.

TEOREMA 9.2. *O algoritmo apresentado leva tempo $O(m|f_{max}|)$ em uma rede D com m arestas com capacidades inteiras, onde $|f_{max}|$ é o valor do fluxo máximo em D.*

DEMONSTRAÇÃO. Claramente, a rede residual pode ser construída em tempo $O(m)$ a cada iteração. Um caminho de s para t na rede residual pode ser encontrado em tempo $O(m)$ usando busca. Afirmamos que o valor do fluxo f a cada iteração é um número inteiro. Nesse caso, o número de iterações é no máximo $|f_{max}|$, já que o valor de f aumenta a cada iteração.

Para provarmos que $|f|$ é sempre inteiro, usamos um argumento indutivo. O valor inicial de $|f|$ é 0, portanto inteiro. Como todas as arestas de D tem capacidades inteiras e as capacidades das arestas de D_f são obtidas através de diferenças entre capacidades das arestas em D e fluxos de f, as arestas de D_f também tem capacidades inteiras. Como $|f|$ é aumentado no valor da capacidade de uma aresta de D_f, então $|f|$ é aumentado de um número inteiro, a cada iteração, se mantendo inteiro. □

Dependendo dos caminhos escolhidos na rede residual este algoritmo pode ser bastante lento caso $|f_{max}|$ seja grande. Um exemplo ruim está ilustrado na figura 9.5.

Entrada:
D: Digrafo com capacidades associadas as arestas.
s: Vértice origem. Deve ser uma fonte em D.
t: Vértice destino. Deve ser um sumidouro em D.
Saída:
f: Fluxo máximo de s para t em D, onde $f[e]$ é o fluxo pela aresta e.

FluxoMáximo(D,s,t)
    $f \leftarrow$ fluxo nulo em todas as arestas
    Enquanto existir caminho de s para t em D_f
        $p \leftarrow$ caminho de s a t em D_f com número mínimo de arestas
        $min \leftarrow$ capacidade mínima dentre arestas de p
        Para toda aresta e de p
            $f[e] = f[e] + min$
    Retorne f

FIGURA 9.6. Pseudo-código do algoritmo de Edmonds-Karp para fluxo máximo em redes.

Para obter um algoritmo mais eficiente devemos escolher os caminhos na rede residual de modo mais cuidadoso. Podemos escolher sempre um caminho de s a t que contenha o menor número possível de arestas, como ilustra o pseudo-código da figura 9.6. Este algoritmo é um refinamento do algoritmo mais geral apresentado anteriormente e se chama *algoritmo de Edmonds-Karp.* Podemos obter um caminho com o número mínimo de arestas usando busca em largura. Para analisarmos a complexidade do algoritmo de Edmonds-Karp, precisamos estudar algumas propriedades das redes residuais obtidas pelo algoritmo.

LEMA 9.3. *Sejam $f_1, f_2, \ldots f_k$ os fluxos calculados pelo algorititmo, desde seu início até seu término na k-ésima iteração e $D(f_1), D(f_2), \ldots, D(f_k)$ as redes residuais correspondentes. Seja l_i o número de arestas do caminho mais curto de s a t na rede residual $D(f_i)$. Então, vale que $n \geqslant l_1 \geqslant l_2 \geqslant \ldots \geqslant l_{k-1}$.*

DEMONSTRAÇÃO. Denotamos por p_i o caminho aumentante escolhido pelo algoritmo na i-ésima iteração. Se $l_{i+1} < l_i$, então p_{i+1} precisa usar arestas que estejam em $D(f_{i+1})$, mas não em $D(f_i)$. Ao compararmos as redes residuais $D(f_i)$ e $D(f_{i+1})$, notamos que, se existe aresta direcionada (u, v) que não pertence a $D(f_i)$, mas pertence a $D(f_{i+1})$, então (v, u) é uma aresta de p_i. Como (u, v) pertence ao caminho mais curto de s a t, certamente, a inclusão de uma aresta (v, u) não pode reduzir o comprimento deste caminho, provando o lema. □

TEOREMA 9.4. *O algoritmo de Edmonds-Karp leva tempo $O(nm^2)$ em uma rede com m arestas.*

DEMONSTRAÇÃO. Pelo lema anterior, o número de arestas no caminho aumentante p_i é menor ou igual ao número de arestas no caminho aumentante p_{i+1}. Afirmamos (e provamos a seguir) que o algoritmo obtém no máximo m caminhos aumentantes com o mesmo número de arestas. Assim, o número de iterações do algoritmo não ultrapassa $nm + 1$. Como cada iteração pode ser realizada em tempo $O(m)$, se usarmos busca em largura para encontrar o caminho com o menor número possível de arestas, então a complexidade do algoritmo é $O(nm^2)$.

Para provarmos nossa afirmação de que o algoritmo obtém no máximo m caminhos aumentantes com o mesmo número de arestas, usamos o seguinte argumento. Cada caminho aumentante p_i usado pelo algoritmo possui uma aresta e_i de capacidade máxima na rede residual. Esta aresta direcionada certamente não aparece na rede residual $D(f_{i+1})$. Como as arestas que pertencem a $D(f_{i+1})$ mas não pertencem a $D(f_i)$ não podem ser usadas em nenhum caminho aumentante de comprimento l_i, o número de caminhos aumentantes de tamanho l_i é limitado por m. □

9.2. Resumo e Observações Finais

Apresentamos dois algoritmos para o problema do fluxo máximo em uma rede. O Algoritmo de Ford-Fulkerson é consequência do Teorema do fluxo máximo-corte mínimo. Uma pequena modificação introduzida por Edmonds-Karp fornece um algoritmo polinomial de ordem $O(nm^2)$. São muitas e variadas as aplicações de fluxo máximo. Na lista de exercícios, exemplificamos problemas cuja solução é obtida por redução ao problema do fluxo máximo.

Exercícios

9.1) Descreva a redução do problema Emparelhamento Máximo em um grafo bipartido para o problema Fluxo Máximo em uma rede. (construa a partir do grafo bipartido uma rede correspondente, estabeleça uma correspondência entre as soluções viáveis dos dois problemas, i.e. entre emparelhamentos e fluxos, estabeleça uma correspondência entre os máximos dos dois problemas.)

9.2) O seguinte problema pode ser resolvidos por redução ao problema de fluxo máximo em redes. Queremos construir, caso exista, um grafo direcionado simples, não necessariamente conexo, com n vértices $v_1, v_2, \ldots, v_n$ onde cada vértice v_i tem um grau externo

especificado $d^+(v_i)$ e um grau interno especificado $d^-(v_i)$. Mostre que a seguinte construção resolve este problema.

Construa uma rede N com $V = \{X, Y, a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n\}$ e arestas E contendo Xa_i, para todo i; b_iY, para todo i; e a_ib_j, para todo $i \neq j$. As capacidades são: $Xa_i = d^+(v_i)$, $b_iY = d^-(v_i)$, $a_ib_j = 1$.

Maximize o fluxo de X para Y. Se um fluxo máximo satura todas as arestas a partir de X e todas as arestas que chegam em Y, então o grafo direcionado satisfazendo as propriedades sobre os graus existe. Este grafo direcionado é obtido da rede N removendo X e Y, e identificando vértices a_i e b_i.

9.3) O seguinte problema pode ser resolvidos por redução ao problema de fluxo máximo em redes. No Problema da Excursão R famílias partem numa excursão em S veículos. Existem f_i pessoas na família i e v_j lugares no veículo j. Será possível organizar os veículos de modo que duas pessoas da mesma família não estão num mesmo veículo?

CAPíTULO 10

Problemas NP-Completos

Ao longo deste livro, estudamos técnicas para desenvolver algoritmos eficientes para diversos problemas. Porém, existem vários problemas para os quais não é conhecido nenhum algoritmo eficiente. Pergunta-se: até que ponto vale a pena tentar encontrar um algoritmo eficiente para um problema? Afinal, pode ser que tal algoritmo sequer exista. Por isso, é importante conhecer problemas para os quais não existe algoritmo eficiente, de modo a evitar esforços em vão.

Neste capítulo, estudamos uma classe de problemas para os quais acredita-se que não é possível obter algoritmos eficientes. Embora ninguém tenha conseguido provar este fato, apresentamos evidências que mostram que é pouco provável que exista algoritmo eficiente para resolver qualquer um desses problemas, chamados de problemas NP-Difíceis (um subconjunto dos problemas NP-Difíceis são os problemas NP-Completos).

10.1. Tempo Polinomial no Tamanho da Entrada

Ao longo do livro, usamos vários parâmetros da entrada, e até mesmo da saída, para expressar a complexidade de tempo dos algoritmos. Quando a entrada é um grafo, usualmente expressamos a complexidade de tempo em função de n e m, os números de vértices e de arestas do grafo. Ao analisarmos o problema de determinar se um número p é primo, seria natural expressar a complexidade em função do valor p. Assim, o algoritmo que testa dividir p por todos os números naturais de 2 até $\lfloor \sqrt{p} \rfloor$, tem complexidade de tempo $O(\sqrt{p})$. Porém, ao compararmos a complexidade de tempo de algoritmos para problemas diferentes, não podemos dizer que um algoritmo $O(\sqrt{p})$ para testar primalidade é mais eficiente ou menos eficiente que um algoritmo $O(n + m)$ para um problema em grafos. Felizmente, existe uma propriedade natural da entrada de todos os problemas que permite comparar complexidades de tempo de algoritmos para problemas diferentes.

O tamanho da entrada de um problema é o número de bits gastos para descrever esta entrada. Para representarmos um número p em uma máquina binária, precisamos de $n = O(\lg p)$ bits. A complexidade de tempo do algoritmo que testa primalidade, se descrita em função do tamanho da entrada n, é $O(2^n)$.

Um algoritmo é dito polinomial se sua complexidade de tempo é limitada por um polinômio no tamanho da entrada. Por exemplo, um algoritmo $O(n^2)$, onde n é o tamanho da entrada, é claramente polinomial. Um algoritmo $O(n^2 \lg n)$ também é polinomial, pois $O(n^2 \lg n) = O(n^3)$. O algoritmo que testa primalidade em tempo $\Theta(2^n)$ não é polinomial. Denotamos por $poli(n)$ um polinômio qualquer em n.

Ao invés de representar os números em notação binária, podemos representá-los em notação unária. Deste modo, um número p gasta $O(p)$ bits para ser representado, e não $O(\lg p)$. Um algoritmo é dito pseudo-polinomial, se a sua complexidade de tempo for $O(poli(n))$, onde n é o tamanho da entrada com todos os números escritos em notação unária. Deste modo, o algoritmo que apresentamos para testar primalidade é pseudo-polinomial, embora não seja polinomial. Neste texto, consideramos que todos os números são escritos em notação binária, a não ser quando dizemos o contrário.

Porque é útil separar os algoritmos em polinomiais e não polinomiais? Consideramos que os algoritmos polinomiais são eficientes, tendo complexidade de tempo aceitável para a maioria das aplicações práticas e consideramos que algoritmos não polinomiais não são eficientes, tendo pouca utilidade prática. A realidade é um pouco diferente. De fato um algoritmo $O(n^8)$ não é muito interessante na prática. Além disso, existem diversos algoritmos com complexidade

de tempo até mesmo linear no tamanho da entrada que, devido às grandes constantes ocultas pela notação O, não tem qualquer utilidade prática. Por outro lado, existem algoritmos não polinomiais com excelente desempenho prático, dos quais o mais famoso é o método simplex usado em programação linear. Embora o método simplex tenha complexidade exponencial no pior caso, na maioria dos casos encontrados na prática este método é bastante rápido.

Ainda assim, a grande maioria dos algoritmos polinomiais tem boa performance prática e a grande maioria dos algoritmos não polinomiais tem péssima performance prática. É raro encontrar um algoritmo com complexidade de tempo $O(n^8)$. Poucos são os algoritmos polinomiais que tem complexidade de tempo $\Omega(n^4)$.

Até aqui, a separação dos algoritmos em polinomiais e não polinomiais ainda pode parecer arbitrária. Poderíamos dividir os algoritmos em algoritmos com complexidade até $O(n^4)$ e algoritmos que não tem complexidade $O(n^4)$, por exemplo, e dizer que os primeiros são eficientes enquanto os últimos não são. Mesmo que esta divisão fosse razoável, não conseguiríamos desenvolver a teoria com base nela. A facilidade matemática de separar os algortimos em polinomias e não polinomiais ficará clara na próxima sessão.

10.2. Problemas de Decisão e Reduções

Um problema de decisão é um problema que possui apenas duas respostas: *sim* e *não*. Neste capítulo, nos restringimos a problemas de decisão. Indiretamente, porém, tratamos de outros tipos de problemas. Por exemplo, se não existir algoritmo polinomial que diz se um grafo possui conjunto independente com pelo menos k vértices, então certamente não existe algoritmo polinomial que encontra o maior conjunto independente em um grafo. Afinal, a existência de um algoritmo polinomial para o problema de otimização implicaria em um algoritmo polinomial para o problema de decisão.

Outra maneira de entender problemas de decisão é como reconhecimento de linguagens. Todo problema de decisão pode ser visto como, dada uma entrada x, decidir se $x \in L$ para uma linguagem específica L. Por exemplo, se L é o conjunto dos números primos, decidir se x é primo é equivalente a decidir se $x \in L$. Por causa dessa correspondência entre problemas de decisão e linguagens, alternamos livremente entre um e outro. Denotamos por L_π a linguagem correspondente ao problema π, isto é, a linguagem que contém todas as entradas para as quais a resposta do problema π é *sim*. Denotamos por $\pi(x)$ a resposta do problema π para a entrada x. Denotamos por $A(x)$ a saída do algoritmo A para a entrada x.

Dados dois problemas π e π', dizemos que π reduz polinomialmente a π' se existe algoritmo polinomial que transforma uma entrada x de π em uma entrada x' de π' tal que $x \in L_\pi \leftrightarrow x' \in L_{\pi'}$. Em outras palavras, π reduz polinomialmente a π' se existe algoritmo polinomial T tal que $\pi(x) = \pi'(T(x))$. O tamanho da saída $T(x)$ deve ser limitado por um polinômio no tamanho de x. Chamamos o algoritmo T de transformação. Usamos a notação $\pi' \leqslant_P \pi$ para dizer que π' se reduz polinomialmente a π.

O seguinte teorema mostra a utilidade das reduções polinomiais.

Teorema 10.1. *Dados dois problemas π e π' onde $\pi \leqslant_P \pi'$, se existe algoritmo polinomial para resolver π', então existe algoritmo polinomial para resolver π. Analogamente, se não existe algoritmo polinomial para resolver π então não existe algoritmo polinomial para resolver π'.*

Demonstração. Provaremos que, se existir algoritmo polinomial para resolver π', então existe algoritmo polinomial para resolver π. Como $\pi \leqslant_P \pi'$, podemos resolver π fazendo uma redução polinomial da entrada de π para a entrada de π' e, em seguida, rodando o algoritmo polinomial que resolve π'. A primeira etapa, que consiste em executar o algoritmo de transformação, leva tempo polinomial. A segunda etapa leva tempo polinomial no tamanho da entrada de π', que, por sua vez, é um polinomio no tamanho da entrada de π (já que o algoritmo de transformação é polinomial). Como $O(poli(poli(n)) = O(poli(n))$, o teorema segue. □

Além disso, a relação de redutibilidade polinomial é transitiva:

Teorema 10.2. *Se $\pi \leqslant_P \pi'$ e $\pi' \leqslant_P \pi''$, então $\pi \leqslant_P \pi''$.*

DEMONSTRAÇÃO. Seja T o algoritmo polinomial que transforma a entrada de π na entrada de π' e T' o algoritmo polinomial que transforma a entrada de π' na entrada de π''. O algoritmo $T'(T(x))$ é polinomial e reduz π a π''. □

10.3. Certificados Polinomiais e a Classe NP

Um ciclo Hamiltoniano em um grafo é um ciclo que contém todos os vértices do grafo. Considere o problema de decisão a seguir, para o qual não é conhecido nenhum algoritmo polinomial:

PROBLEMA 23. *Dado um grafo G, dizer se G possui ciclo Hamiltoniano.*

Digamos que uma raça alienígena possua poder de computação ilimitado, podendo executar qualquer algoritmo, polinomial ou não, instantaneamente. Nós terráqueos, entretanto, estamos limitados a executar algoritmos polinomiais e possuimos dois grafos G_1 e G_2, com milhares de vértices cada um, que desejamos saber se possuem ciclo Hamiltoniano. Então, perguntamos aos alienígenas se o grafo G_1 possui ciclo Hamiltoniano. Recebemos como resposta um sonoro *sim*.

Neste momento, surge uma dúvida: "Será que os alienígenas falam a verdade?" Para esclarecer esta dúvida, um terráqueo tem a seguinte idéia: "Peça para eles nos mostrarem o ciclo." Então, os alienígenas fornecem uma seqüência de milhares de vértices que corresponde ao ciclo Hamiltoniano. Com algum trabalho, verificamos que esta seqüência tem todos os vértices do grafo exatamente uma vez e todas as arestas do ciclo de fato existem. Afinal, esta verificação pode ser feita em tempo polinomial. Assim, temos certeza que os alienígenas forneceram a resposta certa com relação ao grafo G_1.

Apresentamos, então, o grafo G_2 aos alienígenas, que desta vez respondem *não*. Neste caso, não podemos pedir para os alinígenas exibirem o ciclo Hamiltoniano que não existe. Ficamos para sempre na dúvida se eles disseram ou não a verdade sobre o grafo G_2.

Esta história ilustra a classe de problemas chamada de NP, à qual o problema ciclo Hamiltoniano pertence. Para os problemas na classe NP, existe um certificado polinomial para a resposta *sim*. Mais formalmente, se $\pi \in$ NP então, para toda entrada $x \in L_\pi$ existe uma sequência de bits c com $|c| = O(poli(|x|))$, chamada de certificado polinomial, tal que existe algoritmo polinomial que, recebendo como entrada x e c, verifica que $x \in L_\pi$. No caso do ciclo Hamiltoniano o certificado polinomial é o próprio ciclo (figura 10.1). No problema de dizer se um número é composto, o certificado polinomial pode ser um fator do número.

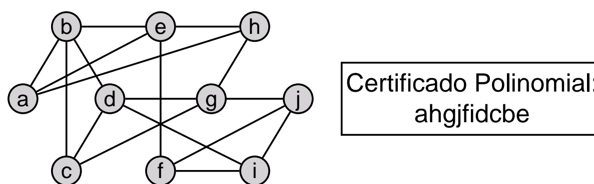


FIGURA 10.1. Grafo que possui ciclo hamiltoniano com o certificado polinomial.

Entretanto, para a resposta *não*, isto é, quando $x \notin L_\pi$, não é necessário que exista este certificado. No caso, não temos necessariamente como provar que um grafo não possui ciclo Hamiltoniano. Quando um número é primo, também não parece óbvio que exista certificado polinomial para dizer que o número é primo (embora exista quando o número é composto). De fato, existe um certificado polinomial que diz que um número é primo, mas este certificado não é simples e não entraremos em detalhes aqui.

Outra classe de problemas é chamada de CO-NP. Um problema pertece a CO-NP quando existe certificado para a resposta *não*. Todo problema pertencente a NP possui um problema simétrico em CO-NP. Dizer se um grafo *não* possui ciclo Hamiltoniano é um problema em CO-NP. Como mencionamos, o problema de dizer se um número é primo, ou, simetricamente, dizer se um número é composto, pertence simultaneamente a NP e a CO-NP, pois possui certificado polinomial tanto para o *sim* quanto para o *não*.

Todos os problemas de decisão que examinamos nos capítulos anteriores deste livro estão tanto em NP quanto em CO-NP. Isto acontece porque estes problemas estão em P, que é a classe dos problemas de decisão que podem ser resolvidos por um algoritmo polinomial no tamanho da entrada. Quando um problema pertence a P, o certificado vazio é um certificado polinomial válido tanto para o *sim* quanto para o *não*. Afinal, fornecendo apenas a entrada do problema, é possível decidir em tempo polinomial se a resposta é *sim* ou *não*. O diagrama com as classes P, NP e CO-NP está representado na figura 10.2. Claramente P está na interseção de NP e CO-NP, porém, não se sabe se P é a interseção de NP e CO-NP, ou seja, se existe algum problema π tal que $\pi \in$ NP $\cap$ CO-NP e $\pi \notin$ P. Além disso, não se sabe se de fato P = NP = CO-NP, ou seja, todos os problemas em NP e CO-NP de fato podem ser resolvidos em tempo polinomial. Embora a grande maioria dos estudiosos do assunto acredite que está igualdade não é verdade, até hoje ninguém conseguiu provar este fato.

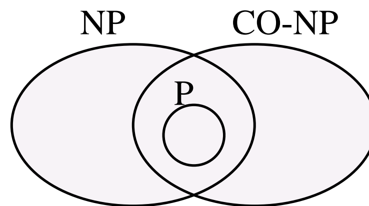


FIGURA 10.2. Diagrama das classes P, NP e CO-NP.

Muitos leigos acreditam que, assim como um problema em P é um problema polinomial, um problema em NP é um problema não polinomial. Isto está completamente errado, afinal os problemas polinomiais estão certamente na classe NP. A sigla NP surgiu de *non-deterministic polinomial time* (tempo polinomial não determinístico), pois uma outra definição para a classe NP é a classe dos problemas que podem ser resolvidos em tempo polinomial por uma máquina de Turing não determinística. Sem entrarmos em muitos detalhes, uma máquina de Turing não determinística é um modelo para um computador que pode, a cada instrução executada, se bifurcar em dois caminhos de processamento distintos, respondendo *sim* caso algum caminho responda *sim* e *não*, caso todos os caminhos respondam não. A máquina de Turing não determinísitica, diferente da máquina de Turing determinística, modela um computador que, pelo menos por enquanto, não sabemos como construir fisicamente. Um exemplo de algoritmo não determinístico que resolve ciclo Hamiltoniano em tempo polinomial está na figura 10.3.

10.4. Os Problemas NP-Completos

Cada classe de problemas contém infinitos problemas bastante diferentes entre si. É natural que alguns problemas sejam mais difíceis de resolver que outros da mesma classe, segundo algum critério. Na classe NP, existe um conjunto de problemas, chamados de NP-Completos (ou apenas NPC) que são os problemas mais difíceis de resolver da classe NP. Um problema $\Pi \in$ NP é NP-Completo se, para todo o problema $\pi \in$ NP, $\pi \leqslant_P \Pi$. Deste modo, se for descoberto um algoritmo polinomial para algum problema NP-Completo, então P = NP = CO-NP.

Não parece simples provar que um problema é NP-Completo. De fato, é relativamente complicado provar que um *primeiro* problema é NP-Completo. Porém, a partir do momento que foi provado que um problema Π é NP-completo, é bem simples provar que Π' também é NP-Completo: basta provar que $\Pi \leqslant_P \Pi'$.

TEOREMA 10.3. *Se* $\Pi \in NPC$, $\Pi' \in NP$ *e* $\Pi \leqslant_P \Pi'$, *então* $\Pi' \in NPC$.

DEMONSTRAÇÃO. Se $\Pi \in$ NPC, então para todo $\pi \in NP$, $\pi \leqslant_P \Pi$. Como $\Pi \leqslant \Pi'$, pelo teorema 10.2, para todo $\pi \in NP$, $\pi \leqslant_P \Pi'$. Como $\Pi' \in$ NP, o teorema segue. □

Uma maneira mais sucinta de definir a classe NPC é definir primeiro a classe dos problemas NP-Difíceis. Um problema Π é NP-Difícil se, para todo o problema $\pi \in$ NP, $\pi \leqslant_P \Pi$.

Entrada:

G: Grafo conexo.

Saída:

Resposta *sim* ou *não* para a questão se G possui ciclo Hamiltoniano.

Observações:

Este pseudo-código é para um modelo não determinístico de computação, respondendo *sim* caso alguma ramificação responda *sim* e *não* caso todas as ramificações respondam *não*.

```
CicloHamiltoniano(G)
    v0 ← v ← vértice qualquer de V(G)
    c ← 1
    Enquanto c ⩽ |V(G)|
        Marcar v
        c ← c + 1
        Se existir vértice v' não marcado tal que (v, v') ∈ E(G)
            Para todo vértice v' não marcado tal que (v, v') ∈ E(G)
                Crie uma ramificação do processamento onde v ← v'
        Senão
            Retorne não
    Se (v, v0) ∈ E(G)
        Retorne sim
    Senão
        Retorne não
```

FIGURA 10.3. Algoritmo que resolve ciclo Hamiltoniano em tempo polinomial em uma máquina não determinística.

Um problema NP-Difícil não precisa pertencer a classe NP. De fato, um problema NP-Difícil não precisa nem mesmo ser um problema de decisão, mas não entraremos nesse assunto aqui. Com esta definição, pode-se definir a classe NPC como NPC = NP-Difícil $\cap$ NP. Analogamente, pode-se definir a classe de problemas CO-NP-Completos (ou CO-NPC) como CO-NPC = NP-Difícil $\cap$ CO-NP. Um diagrama dessas classes, como a maioria dos estudiosos do assunto acredita que se relacionem, está na figura 10.4. Lembramos que, até hoje, ninguém conseguiu provar que P $\neq$ NP.

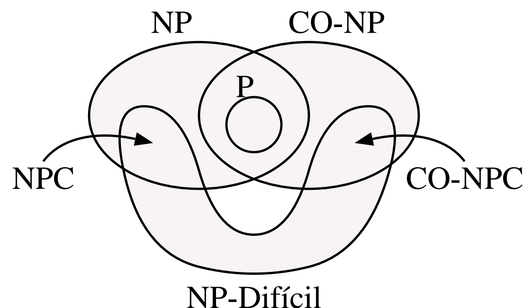


FIGURA 10.4. Diagrama das classes P, NP e CO-NP, NPC, CO-NPC e NP-Difícil.

Nas próximas sessões, provamos que alguns problemas são NP-Completos. Estas demonstrações são baseadas no terorema 10.3, fazendo redução de um problema para outro. A figura 10.5 mostra, em ordem, as reduções que são apresentadas.

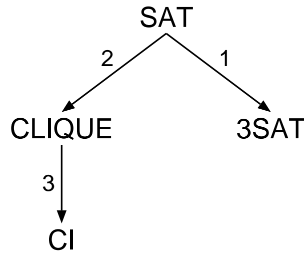


FIGURA 10.5. Reduções entre problemas NP-Completos, numeradas segundo a ordem com que são apresentadas neste capítulo.

10.5. Satisfabilidade

O primeiro problema que foi provado NP-Completo é chamado de satisfabilidade, ou simplesmente SAT. Neste problema, é fornecida uma expressão lógica na forma normal conjuntiva e deseja-se saber se a expressão é satisfatível. A forma normal conjuntiva é formada por um conjunto de cláusulas *ou* (representado pelo operador $\vee$) unidas pelo operador *e* (representado por $\wedge$). Um exemplo de expressão na forma normal conjuntiva é:

$$(a \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{c} \vee d) \wedge (b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee \overline{d}).$$

Nestas expressões, o literal $\overline{a}$ representa a negação do literal a, ou seja, a é verdadeiro se e só se $\overline{a}$ é falso. A expressão é satisfatível se existir atribuição de valores verdadeiro e falso aos literais de modo que a expressão seja verdadeira. A expressão acima é satisfatível, podendo ser satisfeita pela atribuição:

$$a = verdadeiro,\ b = verdadeiro,\ c = verdadeiro\ d = falso.$$

Um exemplo mínimo de uma expressão na forma normal conjuntiva não satisfatível é $(a) \wedge (\overline{a})$. Eis o problema SAT:

PROBLEMA 24. *(SAT) Dada uma expressão lógica na forma normal conjuntiva, dizer se a expressão é satisfatível.*

O teorema a seguir foi provado por Cook, mas prová-lo foge do escopo deste livro. Nos contentamos em justificar que SAT $\in NP$, pois a atribuição de variáveis é um certificado polinomial para a resposta *sim*.

TEOREMA 10.4. *SAT* $\in NPC$

Uma variação do problema SAT é chamada de 3SAT.

PROBLEMA 25. *(3SAT) Dada uma expressão lógica na forma normal conjuntiva, com no máximo* 3 *literais por cláusula, dizer se a expressão é satisfatível.*

Um exemplo de expressão de 3SAT é:

$$(a \vee b \vee \overline{d}) \wedge (\overline{a} \vee c \vee \overline{d}) \wedge (b \vee d) \wedge (\overline{b} \vee \overline{c} \vee \overline{d}).$$

Certamente o problema 3SAT não é mais difícil de resolver que o problema SAT. Afinal, o problema 3SAT é um caso específico do problema SAT. Seria extremamente simples provar que 3SAT $\leqslant_P$ SAT, porém queremos provar a direção contrária.

TEOREMA 10.5. *3SAT* $\in NPC$

DEMONSTRAÇÃO. Claramente, 3SAT $\in NP$, pois uma atribuição de valores aos literais é um certificado polinomial para o *sim*. Pelo teorema 10.3, basta provarmos que SAT $\leqslant_P$ 3SAT.

Podemos transformar uma cláusula C com $n > 3$ literais em duas cláusulas C_1 e C_2 com $n-1$ e 3 literais, respectivamente, pelo processo que definimos a seguir. A aplicação sucessiva

deste método permite que uma cláusula com um número arbitrariamente grande de literais seja reduzida a várias cláusulas com 3 literais por cláusula.

Sejam $x_1, \ldots, x_n$ os literais de uma clásula $C = (x_1 \vee \ldots \vee x_n)$ com $n > 3$ literais. Criamos uma variável adicional y e definimos as duas cláusulas como:

$$C_1 = (x_1 \vee \ldots \vee x_{n-2} \vee y) \text{ e } C_2 = (x_{n-1}, x_n, \overline{y}).$$

Precisamos provar que a aplicação dessa transformação não altera a satisfabilidade da expressão.

Dada uma atribuição de valores às variáveis, caso a cláusula C seja verdadeira, algum literal x_i é verdadeiro. Então, ou x_i está em C_1 ou x_i está em C_2. Caso x_i esteja em C_1, podemos satisfazer as duas cláusulas criadas fazendo $y = falso$. Caso x_i esteja em C_2, podemos satisfazer as duas cláusulas criadas fazendo $y = verdadeiro$.

Caso a cláusula C seja falsa, não existe literal x_i verdadeiro. Neste caso, não importa se $y = verdadeiro$ ou $y = falso$, uma das duas cláusulas C_1 ou C_2 não será satisfeita. Deste modo, a expressão inteira não será satisfeita.

Claramente esta transformação leva tempo polinomial no tamanho da entrada. □

Deste modo, podemos transformar a expressão de SAT:

$$(a \vee \overline{b} \vee \overline{c} \vee d \vee \overline{e}) \wedge (b \vee \overline{c} \vee d \vee e) \wedge (a \vee c) \wedge (\overline{a} \vee \overline{d} \vee e)$$

na expressão de 3SAT:

$$(a \vee \overline{b} \vee y_3) \wedge (\overline{c} \vee y_1 \vee \overline{y_3}) \wedge (d \vee \overline{e} \vee \overline{y_1}) \wedge (b \vee \overline{c} \vee y_2) \wedge (d \vee e \vee \overline{y_2}) \wedge (a \vee c) \wedge (\overline{a} \vee \overline{d} \vee e).$$

10.6. Clique e Conjunto Independente

Uma clique em um grafo é um subconjunto de seus vértices cujo subgrafo induzido é completo. Em outras palavras, uma clique em um grafo G é um subconjunto $Q \subseteq V(G)$ tal que, para todo par de vértices distintos $v_1, v_2 \in Q$, a aresta $(v_1, v_2) \in E(G)$. Um exemplo de clique está na figura 10.6.

Problema 26. *(CLIQUE) Dados um grafo G e um inteiro k, dizer se G possui clique com pelo menos k vértices.*

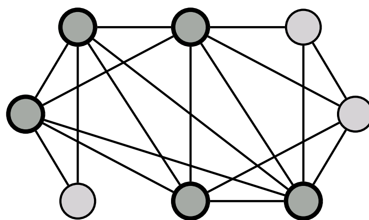


Figura 10.6. Grafo com uma clique de 5 vértices em destaque.

Provaremos que CLIQUE é NP-Completo fazendo uma redução polinomial de SAT a CLIQUE. Note que estamos reduzindo problemas que não parecem ter qualquer relação. O problema CLIQUE é um problema de grafos, enquanto o problema SAT é um problema de lógica.

Teorema 10.6. *CLIQUE $\in NPC$*

Demonstração. Claramente, CLIQUE $\in NP$, pois a própria clique é um certificado polinomial para o *sim*. Pelo teorema 10.3, basta provarmos que SAT $\leqslant_P$ CLIQUE.

A nossa transformação é definida da seguinte maneira. Para cada literal x_i em cada cláusula c criamos um vértice correspondente x_i^c no grafo. As arestas são colocadas sempre entre vértices de cláusulas distintas, desde que estes vértices não correspondam a um literal e sua negação. Um exemplo desta transformação está na figura 10.7. O valor de k é definido como o número de cláusulas.

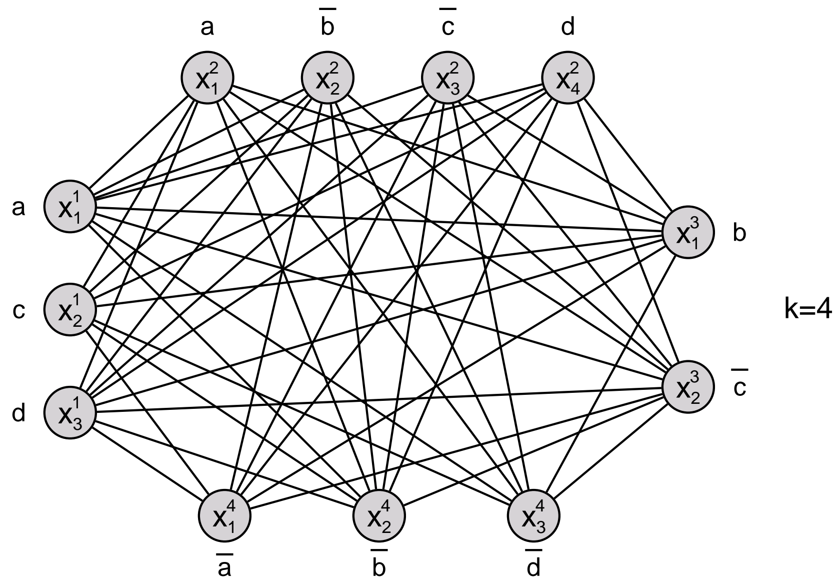


FIGURA 10.7. Grafo obtido pela redução da expressão $(a \vee c \vee d) \wedge (a \vee \overline{b} \vee \overline{c} \vee d) \wedge (b \vee \overline{c}) \wedge (\overline{a} \vee \overline{b} \vee \overline{d})$.

Claramente esta transformação pode ser feita em tempo polinomial no tamanho da entrada, embora este tempo não seja linear, mas sim quadrático. Precisamos provar que o grafo obtido pela transformação possui clique de tamanho pelo menos k se e só se a expressão lógica é satisfatível.

Suponha que o grafo possui uma clique com pelo menos k vértices. Como os vértices provenientes da mesma cláusula não possuem arestas entre si, certamente a clique possui um vértice vindo de cada cláusula. Certamente, não há na clique vértices correspondentes a um literal e sua negação, pois estes vértices não possuiriam aresta entre eles. Então, podemos atribuir valor verdadeiro a todos os literais correspondentes aos vértices da clique. Esta atribuição satisfaz a todas as cláusulas, pois tem pelo menos um literal verdadeiro em cada cláusula.

Para provar a outra direção, suponha que a expressão lógica é satisfatível e fixe uma atribuição de valores que a satisfaça. Então, cada cláusula possui pelo menos um literal verdadeiro. Defina Q como um conjunto de vértices correspondente a um literal verdadeiro de cada cláusula. Por definição, Q possui k vértices, um de cada cláusula. Além disso, como não há em Q vértices correspondentes a um literal e sua negação, então Q é uma clique. □

Um conjunto independente em um grafo é um subconjunto de seus vértices tal que não exista aresta entre qualquer par de vértices do subconjunto. O problema abaixo é extremamente semelhante ao problema clique.

PROBLEMA 27. *(CI) Dados um grafo G e um inteiro k, dizer se G possui conjunto independente com pelo menos k vértices.*

Podemos provar que CI é NP-Completo fazendo uma redução simples de CLIQUE para CI.

TEOREMA 10.7. *CI $\in NPC$*

DEMONSTRAÇÃO. Claramente, CI $\in NP$, pois o próprio conjunto independente é um certificado polinomial para o *sim*. Pelo teorema 10.3, basta provarmos que CLIQUE $\leqslant_P$ CI.

A transformação polinomial de CLIQUE para CI é bastante simples. Basta mantermos o valor de k inalterado e gerarmos o grafo $\overline{G}$ como o complemento do grafo G. O conjunto Q é uma clique em G se e só se o conjunto Q é um conjunto independente em $\overline{G}$ (figura 10.8). Esta redução é claramente polinomial. □

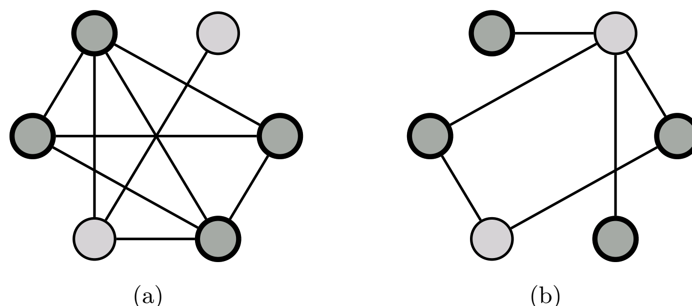


FIGURA 10.8. (a) Grafo G com uma clique de 4 vértices destacada. (b) Grafo $\overline{G}$ com o conjunto independente correspondente a clique da figura (a) destacado.

10.7. Resumo e Observações Finais

Neste capítulo estudamos uma classe de problemas de decisão que não parecem poder ser resolvidos por nenhum algoritmo polinomial, embora ninguém tenha conseguido provar esta afirmação.

Uma redução polinomial de um problema π a um problema π' consiste de um algoritmo polinomial que transforma a entrada de π em uma entrada de π' tal que a resposta dos respectivos problemas para estas entradas seja a mesma. Se existe redução polinomial de π para π', dizemos que π reduz polinomialmente a π' e denotamos por $\pi \leqslant_P \pi'$.

Os problemas de decisão da classe NP são aqueles cuja resposta *sim* pode ser verificada em tempo polinomial. Analogamente, os problemas de decisão da classe CO-NP são aqueles cuja resposta *não* pode ser verificada em tempo polinomial. Um problema Π é NP-Difícil se, para todo problema $\pi \in$ NP, $\pi \leqslant_P \Pi$. A classe NP-Completo, ou NPC, é definida como NPC $=$ NP $\cap$ NP-Difícil. A classe CO-NP-Completo, ou CO-NPC, é definida como CO-NPC $=$ CO-NP $\cap$ NP-Difícil.

O primeiro problema provado NP-Completo foi o problema SAT. Provamos que outros problemas são NP-Completos reduzindo-os polinomialmente a SAT. Provamos que 3SAT, CLIQUE e CI são NP-Completos.

Exercícios

10.1) Prove que todo problema na classe NP pode ser resolvido por um algoritmo com complexidade de tempo exponencial no tamanho da entrada.

10.2) Prove que todo problema de decisão que pode ser resolvido em tempo polinomial por uma máquina não determinística com as características descritas a seguir pertence a classe NP. A máquina não determinística retorna *sim* caso alguma ramificação do processamento retorne *sim* e retorna *não* caso todas as ramificações do processamento retornem *não*. Como deveria ser uma máquina não determinística de modo a provar que todo problema de decisão resolvido por ela em tempo polinomial pertença a classe CO-NP?

10.3) O problema EXATAMENTE3SAT consiste em, dada um expressão lógica na forma normal conjuntiva com *exatamente* 3 literais por cláusula, decidir se a expressão é satisfatível. Prove que EXATAMENTE3SAT $\in NPC$. Sugestão: prove que 3SAT $\leqslant_P$ EXATAMENTE3SAT.

10.4) O problema 2SAT consiste em, dada um expressão lógica na forma normal conjuntiva com até 2 literais por cláusula, decidir se a expressão é satisfatível. Este problema é polinomial. Tente provar que 2SAT é NP-Completo. Explique porque não é possível fazer uma pequena adaptação na prova que 3SAT é NP-Completo de modo a provar que 2SAT é NP-Completo.

10.5) O problema MAXV2SAT consiste em, dados uma expressão lógica na forma normal conjuntiva com até 2 literais por cláusula e um inteiro k, decidir se a expressão pode ser satisfeita por uma atribuição de valores lógicos as variáveis onde pelo menos k varíaveis possuem valor verdadeiro. O problema MAX2SAT consiste em, dados uma expressão lógica na forma normal conjuntiva com até 2 literais por cláusula e um inteiro k, decidir se é possível satisfazer pelo menos k cláusulas da expressão. Prove que MAXV2SAT e MAX2SAT são NP-Completos. Sugestão: prove que CI $\leqslant_P$ MAXV2SAT $\leqslant_P$ MAX2SAT.

10.6) Uma cobertura de vértice em um grafo G é um subconjunto C de $V(G)$ tal que, para toda aresta $(v, v') \in E(G)$, $v \in C$ ou $v' \in C$ (possivelmente v e v' pertencem a C). Prove que decidir se um grafo possui cobertura de vértice com no máximo k vértices é NP-Completo.

10.7) Um caminho Hamiltoniano em um grafo G é um caminho que contém todos os vértices de G. Prove que decidir se um grafo G possui caminho Hamiltoniano é NP-Completo. Considere provado que *ciclo* Hamiltoniano é NP-Completo.

10.8) Na sessão 8.3, definimos o problema de programação linear. Prove que a versão de decisão do problema de programação linear, com um número livre de variáveis, com a restrição adicional de que as variáveis só podem possuir valor 0 ou 1, é NP-Completa. Na versão de decisão, o problema de programação linear consiste em obter uma atribuição de variáveis tal que a função objetivo tenha valor pelo menos k, onde k é parte da entrada do problema.

*10.9) Prove que decidir se um grafo possui Ciclo Hamiltoniano é NP-Completo.

*10.10) Um grafo G é 3-colorível se existe uma atribuição de cores aos seus vértices, dentre um conjunto de 3 cores, tal que quaisquer dois vértices adjacentes possuam cores distintas. Prove que decidir se um grafo é 3-colorível é um problema NP-Completo.

Índice

árvore
  centro, 65
  conjunto independente máximo, 54, 72
  geradora mínima, 38
árvore de Huffman, 41
árvore enraizada, 18
árvore livre, 17
árvores
  AVL, 25
  binárias de busca, 23
  de difusão, 25
  rubro-negras, 25

agendamento de tarefas, 49
algoritmo de
  Dijkstra, 49
  Edmonds-Karp, 87
  Euclides, 71
  Ford-Fulkerson, 85
  Graham, 75
  Huffman, 41
  Jarvis, 37
  Kruskal, 48
  Prim, 38
  Strassen, 55
altura em árvores, 18
amortizada, complexidade de tempo, 23
ancestrais, vértices, 18

busca binária, 28–36
busca ilimitada, 34

caminho aumentante, 85
caminho em grafo, 17
caminhos mais curtos, 49
casar antes de conquistar, 58
centro de árvores, 65
certificado polinomial, 91
ciclo em grafo, 17
ciclo Hamiltoniano, 91
classe
  CO-NP, 91
  CO-NPC, 92
  NP, 91
  NP-Difícil, 92
  NPC, 92
  P, 91
CO-NP, 91
CO-NPC, 92
combinar antes de conquistar, 58
compactação de dados (Huffman), 41
compactação de dados (LZSS), 45
complexidade de tempo amortizada, 23
conexo, grafo, 17
conjunto independente máximo em árvores, 54, 72
construção incremental, 73–82
convexo, polígono, 30
corte mínimo, fluxo máximo, 85

dcel, 18
Delaunay, triangulação de, 59
descendentes, vértices, 18
digrafo, 16
Dijkstra, algoritmo de, 49
distância em grafo, 17
divisão e conquista, 50–59
dizimar, 65–72
doubly connected edge list, 18

Edmonds-Karp, algoritmo de, 87
embrulho para presente, 37
envelope superior, 48, 50
estruturas de dados, 15–27
Euclides, algoritmo de, 71

fecho convexo, 37, 58, 69, 75
fecho convexo no espaço, 59
fecho convexo superior, 76
Fibonacci, heap de, 23
fila, 16
filhos, vértices, 18
floresta, 17
fluxo em redes, 83–87
fluxo máximo, corte mínimo, 85
Ford-Fulkerson, algoritmo de, 85
fortemente conexo, grafo, 17
fracamente conexo, grafo, 17

grafo, 16
grafo conexo, 17
Graham, algoritmo de, 75
guloso, método, 37–49

heap, 19
heap binário, 19
heap de Fibonacci, 23
Huffman, árvore de, 41

interpolação, 34

Jarvis, algoritmo de, 37

Kruskal, algoritmo de, 48

linha de varredura, 50
lista encadeada, 16
listas de adjacências, 17
listas de prioridades, 19
LZSS, 45

máquina de Turing, 92
método
  de divisão e conquista, 50–59
  de linha de varredura, 50
  guloso, 37–49
  incremental, 73–82
módulo, 29
matriz de adjacências, 16, 17
mediana, 66
memorização, 58
multiplicação de matrizes quadradas, 55
multiplicação de matrizes, ordem de, 60

nível em árvores, 18
NP, 91
NP-completos, 89–98
NP-Difícil, 92
NPC, 92

ordem de multiplicação de matrizes, 60
ordenação topológica, 48

P, classe, 91
pai, vértice, 18
par de pontos mais próximos, 52
permutação aleatória, 79
pilha, 16
podar e buscar, 65–72
polígono convexo, 30
ponte do fecho convexo, 69
ponto extremo de polígono convexo, 30
pontos mais próximos, 52
Prim, algoritmo de, 38, 39
problema
  de decisão, 90
problemas NP-completos, 89–98
programação dinâmica, 60–64
programação linear, 77
programação linear com duas variáveis, 72

recorrência, 11
rede residual, 84
redução polinomial, 90
refinamento de solução, 83–88
relação de recorrência, 11
reta suporte, 69
rotações em árvores binárias de busca, 25

seleção do k-ésimo, 66
simplificação, 65–72
Strassen, algoritmo de, 55

teorema da vizinhança, 74
triangulação de Delaunay, 59
Turing, máquina de, 92

vetor, 15
  cíclico, 15
  com índices módulo n, 15
vetor ciclicamente ordenado, 30