Chapter 1

Introdução à Otimização Combinatória

Flávio K. Miyazawa e Cid C. de Souza

Abstract

Most of the computational problems are discrete in nature and involve the search of a solution satisfying certain properties, with possible alternatives growing in a combinatorial way. A classic example is the sorting problem, where one must find a permutation of a sequence in non-decreasing order. Moreover, real problems of this type become more complex, when we consider that available resources are limited and the need to optimize their use.

Consider, for example, a city road network and a set of bus stops where employees of a company pick up a bus having sufficient capacity to transport them all. The bus route starts in the garage, pass through the points and take the employees to the work place. However, when we consider the transportation cost, usually proportional to the total traveled distance, it is desirable that the bus go through the shortest route. This is a typical combinatorial optimization problem. In addition to vehicle routing problems, several other applications can be found in various fields of knowledge.

In this mini-course, we present some of the main algorithmic techniques for combinatorial optimization problems.

Resumo

Grande parte dos problemas computacionais são de natureza discreta e envolvem a busca por uma solução atendendo certas propriedades, sendo que as possíveis alternativas crescem de forma combinatória. Um exemplo clássico é o problema de ordenação, onde se deseja encontrar uma permutação de uma sequência com ordenação não-decrescente. Além disso, problemas reais deste tipo tornam-se ainda mais complexos quando existe limitação nos recursos disponíveis e se quer otimizar alguma medida específica.

Considere, por exemplo, a malha viária de uma cidade e pontos onde funcionários tomam um ônibus fretado com capacidade suficiente para transportá-los até uma empresa. O ônibus deve sair da garagem, pegar os funcionários nos pontos, e levá-los ao trabalho. Sendo a malha conexa, qualquer permutação dos pontos define uma possível rota para o ônibus. Porém, quando consideramos o custo do transporte, usualmente proporcional à distância total percorrida pelo veículo, é desejável que o ônibus siga por uma rota de menor percurso. Este é um típico problema de otimização combinatória. Além do roteamento de veículos, inúmeras outras aplicações podem ser encontradas nas mais diversas áreas do conhecimento.

O texto apresenta algumas das principais técnicas algorítmicas para tratamento de problemas de otimização combinatória.

1.1. Introdução

O texto contempla uma breve introdução à área de Otimização Combinatória. Trata-se de uma área que engloba grande quantidade de problemas e que busca por soluções que façam melhor uso dos recursos envolvidos. Nestes problemas, não basta ter uma das soluções possíveis, mas uma que também otimize os objetivos de interesse. Alguns objetivos típicos dos problemas de otimização combinatória, consistem no geral em aproveitar melhor os materiais no processo de produção, otimizar o tempo para realização de ações e operações, transportar mais materiais pelas melhores rotas, diminuir chances de se obter prejuízos, aumentar lucros e diminuir gastos, etc. Para ilustrar a diversidade de áreas, os problemas ocorrem no projeto de redes de telecomunicações, projeto de redes de escoamento de produtos, projeto de circuitos VLSI, corte de materiais, empacotamento em containers, localização de centros distribuidores, escalonamento de tarefas, roteamento de veículos, sequênciamento de DNA, etc.

Os problemas de otimização combinatória podem ser de minimização ou de maximização. Em ambos os casos, temos uma função aplicada a um domínio finito, que em geral é enumerável. Apesar de finito, o domínio da função é em geral grande e algoritmos ingênuos que verificam cada elemento deste domínio se tornam impraticáveis. Desta maneira, surge a necessidade de usar técnicas mais elaboradas para encontrar soluções de valor ótimo, que pode ser de valor mínimo, caso o problema seja de minimização, ou máximo, caso o problema seja de maximização.

Neste texto apresentamos uma breve introdução a algumas técnicas e conceitos básicos da área de Otimização Combinatória.

Alguns problemas de otimização combinatória podem ser bem resolvidos, mas para muitos outros não são esperados algoritmos exatos e rápidos para problemas de grande porte. Muitos destes problemas são importantes e necessitam de soluções, mesmo que não sejam de valor ótimo (de preferência que tenham valor próximo do ótimo) ou sejam obtidas após muito processamento computacional.

O texto está organizado da seguinte forma. Na Seção 1.1.1 a seguir, apresenta-se

a notação usada no texto. Na Seção 1.3 apresenta-se a técnica de projetar algoritmos usando indução matemática. Nas seções 1.4 e 1.5 discutem-se, respectivamente os algoritmos gulosos e os de programação dinâmica, enfatizando sua aplicações na resolução de problemas de Otimização Combinatória. A Seção 1.6 é dedicada a uma discussão sobre Programação Linear e Programação Inteira. As seções 1.7 e 1.8 tratam do desenvolvimento de algoritmos enumerativos usando, respectivamente, os paradigmas de *backtracking* e de *branch and bound*. Finalmente, na Seção 1.9 são feitas algumas considerações finais sobre técnicas e abordagens que não puderam ser contempladas no texto, mas seria interessante contemplá-las em um primeiro curso.

Por ser um texto resumido, é recomendado que o leitor consulte outros livros na literatura, como [Cormen et al. 2009, Dasgupta et al. 2008, Ferreira e Wakabayashi 1996, Manber 1989, Nemhauser e Wolsey 1988, Papadimitriou e Steiglitz 1982, Schrijver 1986, Szwarcfiter 1988, Wolsey 1998] e [Ziviani 2011].

1.1.1. Notação

Conjuntos, Vetores e Funções

Denota-se por $\mathbb{N}$ (resp. $\mathbb{Q}$ e $\mathbb{R}$) o conjunto dos números naturais $\{1,2,\ldots\}$ (resp. conjunto dos números racionais e reais).

Denota-se por $\mathbf{0}^d$ o vetor d-dimensional contendo apenas zeros. Quando a dimensão estiver clara pelo contexto, tal vetor será denotado apenas por $\mathbf{0}$. Dados vetores $u = (u_1,\ldots,u_n) \in \mathbb{R}^n$ e $v = (v_1,\ldots,v_m) \in \mathbb{R}^m$, para inteiros positivos n e m, o *produto interno* $u \cdot v$ entre u e v será denotado por $u \cdot v = \sum_{i=1}^{\min\{n,m\}} u_i v_i$ e a *concatenação* de u e v é dada pelo vetor $u\|v = (u_1,\ldots,u_n,v_1,\ldots,v_m)$.

Dada uma função numérica $f : D \to \mathbb{R}$, denota-se por $f(S)$ a soma dos valores de f nos elementos de D, i.e., $f(S) := \sum_{e \in S} f(e)$ para todo $S \subseteq D$. Dado conjuntos S e T, denota-se por $S - T$ (resp. $S + T$) o conjunto $S \setminus T$ (resp. $S \cup T$). O conjunto formado por apenas um elemento $\{e\}$ pode ser representado apenas pelo elemento e. Assim, $S - e$ representa o conjunto $S \setminus \{e\}$.

Complexidade Assintótica

Ao medir a complexidade de tempo dos algoritmos, a maneira padrão é defini-la em termos do tamanho (*e.g.* número de bits) da entrada. No caso, deve-se apresentar uma função que, dado o tamanho da entrada, devolve o número de passos do algoritmo dentro do modelo computacional considerado. Na maioria das vezes, basta saber a ordem de grandeza destes números, sem no entanto saber exatamente o número de passos ou ciclos realizados para cada entrada. Ademais, a codificação de um algoritmo em linguagem de máquina para dois computadores diferentes, pode levar a quantidade diferente de ciclos.

Assim, na análise da complexidade de tempo de um algoritmo, será usada a no-

tação assintótica, que permite restringir a análise nos termos que crescem, de maneira assintótica, mais rapidamente, sem se preocupar com constantes multiplicativas ou termos menores. Por exemplo, se o número de ciclos em um computador for dado por uma função $T(n) = 10n^2 + 1000n + 20$, o termo que cresce mais rápido e domina os outros termos é $10n^2$. Por mais que termos de ordem menor tenham valores grandes no início o termo de ordem maior, como o $10n^2$ domina os demais a medida que n cresce. Para isto, note que $\frac{T(n)}{10n^2}$ tende a 1 a medida que n cresce. Desconsiderando as constantes multiplicativas, que podem ser diferentes de um computador para o outro, temos que $T(n)$ é da ordem de n^2, ou como definido abaixo, $T(n) \in O(n^2)$.

Dada função não negativa $g(n)$, denotamos por $O(g(n))$ o conjunto $O(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tal que } 0 \leq f(n) \leq cg(n) \text{ para todo } n \geq n_0\}$, por $\Omega(g(n))$ o conjunto $\Omega(g(n)) = \{f(n) : \text{existem constantes positivas } c \text{ e } n_0 \text{ tal que } 0 \leq cg(n) \leq f(n) \text{ para todo } n \geq n_0\}$ e por $\Theta(g(n))$ o conjunto $\Theta(g(n)) = \{f(n) : \text{existem constantes positivas } c_1, c_2 \text{ e } n_0 \text{ tal que } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ para todo } n \geq n_0\}$.

Para funções que tem crescimento muito rápido, como as funções exponenciais em n, é comum utilizar a notação $O^*(g(n))$, definida como o conjunto $O^*(g(n)) = \{f(n) : \text{existe constante } c \text{ tal que } f(n) \in O(g(n)n^c)\}$. Esta notação remove os polinômios multiplicativos. Para entender porque a notação se justifica, considere uma função $f(n) = p(n)a^n$, onde $p(n)$ é um polinômio e a uma constante maior que 1. Note que $f(n)$ é assintoticamente menor que a função $g(n) = (a+\varepsilon)^n$ para qualquer polinômio $p(n)$ e constante $\varepsilon > 0$ (que pode ser bem pequena).

Teoria dos Grafos

Um *grafo* $G = (V,E)$ é uma estrutura composta por dois tipos de objetos, os *vértices*, dado por um conjunto V, e as *arestas*, dado por um conjunto E tal que $E \subseteq \{\{u,v\} : u,v \in V\}$ e $V \cap E = \emptyset$.

Se $G = (V,E)$ e $e = \{u,v\} \in E$ é uma aresta de G, então, u e v são *adjacentes* entre si e ditas *extremidades* de e. Neste caso, e é dito incidir em u e em v. Note que nesta definição não é permitido ter mais do que uma aresta contendo dois vértices adjacentes ou ter uma aresta incidente em apenas um vértice (estes grafos também são conhecidos como *grafos simples*). O conjunto dos vértices adjacentes a v é dado por $\text{Adj}(v)$. Dado vértice $v \in V$ (resp. conjunto $S \subseteq V$), o conjunto das arestas com exatamente uma extremidade em v (resp. em S) é denotado por $\delta(v)$ (resp. $\delta(S)$). O *grau* de v é o número de arestas que incidem em v. Um grafo pode ser representado de maneira gráfica em um plano, fazendo com que cada vértice seja um ponto no plano e vértices adjacentes são ligados por uma linha. Algumas operações usadas para conjuntos são estendidas para grafos. Se $G = (V,E)$ é um grafo, então $v \in G$ (resp. $e \in G$) denota o mesmo que $v \in V$ (resp. $e \in E$). O grafo $G - v$ é o grafo obtido a partir de G removendo-se v e as arestas incidentes a ele. Da mesma forma, para um conjunto de vértices $S \subseteq V$, o grafo $G - S$

é o grafo obtido a partir de G removendo cada vértice de S. Se $\mathcal{S} = \{S_1, \ldots, S_k\}$ são subconjuntos disjuntos de V então $E(S_1, \ldots, S_k)$ é o conjunto de arestas com extremos em partes distintas. Em particular, se $\mathcal{S}$ é uma partição de V com k partes, o conjunto $\mathcal{S}$ é dito ser um k-corte de G. Um 2-corte também é chamado pelo termo *corte*.

Dado grafo $G = (V, E)$, o grafo $H = (V_H, E_H)$ é um *subgrafo* de G se $V_H \subseteq V$ e $E_H \subseteq E$. Um *passeio* em um grafo $G = (V, E)$ é uma sequência $P = (v_0, e_1, v_1, \ldots, e_t, v_t)$, onde $v_i \in V$, $e_i \in E$ e $e_i = \{v_{i-1}, v_i\}$ para $i = 1, \ldots, t$. Neste caso, P é um passeio de v_0 a v_t, sendo estes também seus *extremos*. Um *caminho* em G é um passeio onde todos os vértices são distintos. Um *ciclo* é um passeio onde todos os vértices são distintos, exceto o último que é igual ao primeiro. Quando o interesse for apenas no conjunto de vértices e arestas destes grafos, estas estruturas poderão ser consideradas simplesmente como grafos ou subgrafos, pelo conjunto de vértices e arestas. Um grafo é dito *conexo* se para todo par de vértices u, v em V, existe um caminho em G de u a v, caso contrário, é dito desconexo. Uma *árvore* é um grafo conexo sem ciclos. Um subgrafo que é ciclo ou caminho de G é dito *hamiltoniano* se contém todos os vértices de G. Um subrafo H de G é dito *gerador* se H contém todos os vértices de G.

Para muitos problemas apresentados, os grafos tem pesos ou custos nas arestas. Assim, a tupla $G = (V, E, c)$ representa um grafo ponderado nas arestas, onde $c(e)$ (ou c_e) é o peso ou custo da aresta e.

Um *grafo orientado* (ou *direcionado*) $G = (V, A)$ é uma estrutura dada por dois tipos de objetos, os *vértices*, dado pelo conjunto V, e os arcos, dado pelo conjunto A, tal que $A \subseteq V \times V$ e $V \cap A = \emptyset$. Diferente dos grafos (não-orientados), um arco $a = (u, v)$, é um par ordenado. Neste caso, o vértice u é a *origem* de a e v é o destino de a, ou analogamente, diz-se que a *sai* de u e *entra* em v. Denote por $\text{Adj}^+(u)$ o conjunto de vértices v para os quais há um arco $(u, v) \in A$. Dado conjunto de vértices $S \subseteq V$, denota-se $\delta^+(S)$ (resp. $\delta^-(S)$) o conjunto de arcos que saem de (resp. entram em) algum vértice de S e entram em (resp. saem de) algum vértice de $V \setminus S$.

Dado um grafo orientado $G = (V, A)$, o grafo $H = (V_H, A_H)$ é um *subgrafo* de G se $V_H \subseteq V$ e $A_H \subseteq A$. As definições de passeios, caminhos e ciclos, são análogas ao caso não-orientado, porém respeitando uma orientação das arestas. Um *passeio* em um grafo orientado $G = (V, A)$ é uma sequência $P = (v_0, a_1, v_1, \ldots, a_t, v_t)$, onde $v_i \in V$, $a_i \in A$ e $a_i = (v_{i-1}, v_i)$ para $i = 1, \ldots, t$. Neste caso, P é um passeio de v_0 a v_t, sendo estes também seus *extremos*. As definições de caminhos e ciclos, bem como as versões hamiltonianas, para grafos orientados são definidas analogamente.

Ao leitor interessado em saber mais sobre a teoria dos grafos, recomenda-se os livros [Feofiloff et al. 2004] e [Bondy e Murty 2008].

1.2. Otimização Combinatória

Considere o seguinte problema. Suponha que uma empresa, chamada BUBBLE, é dona de uma máquina de busca e permite que usuários, gratuitamente, entrem com uma ou

mais palavras e recebam endereços da web associados às palavras. Apesar do serviço gratuito, a BUBBLE lucra apresentando propagandas de outras empresas que pagam para aparecer nas buscas relacionadas a elas. Para isto, a página de busca dispõe de uma região retangular na lateral da tela de busca. Tal região é estreita, de largura fixa, e preenche toda a altura da tela. Todas as propagandas devem ter a mesma largura, porém, podem ter alturas distintas, contanto que não passem da altura da região. Suponha que a busca realizada por um usuário define um conjunto de n propagandas $P = \{1, \ldots, n\}$, associadas à sua busca. Cada propaganda $i \in P$, usa uma certa altura p_i da região e caso seja apresentada ao usuário, fornece um lucro de v_i reais para a BUBBLE. Sabendo que a altura da região é B, quais propagandas devem ser apresentadas de maneira a maximizar o valor recebido pela BUBBLE ?

Este é um típico problema de otimização combinatória. Os possíveis candidatos a solução são os subconjuntos das propagandas, mas são viáveis apenas aquelas que podem ser colocadas na região, sem ultrapassar sua altura. Naturalmente, procura-se uma que maximize o valor arrecadado. O problema de otimização subjacente é o Problema da Mochila Binária, definido formalmente a seguir.

Problema da Mochila Binária (PMB): Dados conjunto de itens $\{1, \ldots, n\}$, cada item i com peso $p_i \geq 0$ e valor $v_i \geq 0$, ambos inteiros, e inteiro B, encontrar um conjunto $S \subseteq \{1, \ldots, n\}$ tal que $\sum_{i \in S} p_i \leq B$ e $\sum_{i \in S} v_i$ é máximo.

De maneira geral, uma entrada I para um problema de otimização combinatória Π, deve ter um conjunto de possíveis soluções $\mathcal{S}_I$ e uma função v, tal que $v(S)$ é o valor da solução S, para cada $S \in \mathcal{S}_I$. O objetivo é encontrar uma solução $S^* \in \mathcal{S}_I$ tal que $v(S^*)$ é máximo, se Π é de maximização, ou mínimo, se Π é de minimização. Neste caso, S^* é chamada *solução ótima* de I. Os elementos de $\mathcal{S}_I$ (não necessariamente soluções ótimas) são chamados de *soluções viáveis*, ou simplesmente *soluções*. Qualquer elemento de $\mathcal{S}_I$ poderia ser solução ótima de I, não o sendo apenas por existir outro elemento de $\mathcal{S}_I$ com valor melhor para a função objetivo. Por vezes, define-se também um conjunto $\mathcal{U}_I$ contendo os candidatos a soluções, com $\mathcal{S}_I \subseteq \mathcal{U}_I$, mas não necessariamente um elemento $U \in \mathcal{U}_I$ é uma solução viável de I. No exemplo da mochila binária, o conjunto $\mathcal{U}_I$ poderia ser o conjunto de todos os subconjuntos de itens possíveis. Neste caso, nem todos os subconjuntos são possíveis de ser solução, uma vez que a soma dos pesos pode ser maior que a capacidade B da mochila. Os conjuntos em $\mathcal{U}_I$ cuja soma dos pesos é no máximo B são as soluções viáveis de $\mathcal{S}_I$. Dentre estas, deve haver pelo menos uma solução que é ótima.

A seguir, será visto uma primeira técnica para resolver problemas de otimização combinatória.

1.2.1. Algoritmos Força-Bruta

Os algoritmos por força-bruta, são algoritmos que, como o nome diz, utilizam de muito esforço computacional para encontrar uma solução, sem se preocupar em explorar as

estruturas combinatórias do problema. De fato, estes algoritmos enumeram todas as possíveis soluções, verificando sua viabilidade e, caso satisfaça, seu potencial para ser uma solução ótima. Estão entre os algoritmos mais simples, porém, podem demandar grande quantidade de recursos computacionais sendo, no geral, consideradas apenas para entradas de pequeno porte.

Como exemplo, considere o Problema da Mochila Binária. Qualquer subconjunto de elementos é um potencial candidato a solução do problema, bastando para isso, que seu peso total não ultrapasse a capacidade da mochila. Assim, um algoritmo simples, poderia percorrer todos os subconjuntos possíveis e devolver, dentre aqueles que representam soluções viáveis, um de maior valor.

O algoritmo seguinte gera uma solução ótima para o problema da Mochila. As soluções são representadas por vetores binários, onde a i-ésima posição tem valor 1 se o item i pertence à solução e 0 caso contrário. O algoritmo é recursivo e além da entrada (B, p, v, n), recebe outros dois parâmetros: x^* e x. O parâmetro x^* é um vetor binário n-dimensional que representa a melhor solução encontrada até o momento, e com isso, todas as referências a x^* são para a mesma variável. O parâmetro x é um vetor binário que a cada chamada recursiva, é acrescido de uma nova posição, que representa a pertinência de um item na solução. A primeira chamada do algoritmo é feita com os parâmetros $(B, p, v, n, x^*, ())$, onde x^* armazena a melhor solução obtida a cada momento, começando com o vetor $\mathbf{0}$, que representa uma solução sem itens. O último parâmetro indica um vetor sem elementos. A base do algoritmo recursivo ocorre quando x contém n elementos binários, momento que representa um subconjunto dos itens da entrada.

Algoritmo: Mochila-FB(B, p, v, n, x^*, x)

Saída : Atualiza x^* se encontrar solução que completa x, e tem valor melhor

```
1 se |x| = n então
2     se p·x ≤ B  e  v·x > v·x* então x* ← x
3 senão
4     Mochila-FB(B, p, v, x*, x||(0))
5     Mochila-FB(B, p, v, x*, x||(1))
```

No início de cada chamada, os elementos que já tiveram sua pertinência na atual solução definida, estão representados em x, faltando considerar os próximos $n - |x|$ itens. A execução deste algoritmo pode ser representada por uma árvore binária de enumeração, onde cada nó representa uma chamada recursiva e as arestas que ligam seus filhos, representam as duas situações possíveis, quando o item pertence à solução (ramo com $x_i = 1$) ou não (ramo com $x_i = 0$). Note que este algoritmo sempre gera uma árvore de enumeração completa.

No início de cada chamada recursiva, o Algoritmo Mochila-FB precisa considerar a pertinência dos próximos $n-|x|$ itens, sendo que x é acrescido de uma coordenada a cada chamada recursiva. O algoritmo pode ser implementado de maneira a ter complexidade de tempo limitada pela recorrência $T(n) = 2T(n-1) + O(1)$, cuja resolução mostra que $T(n)$ é $O(2^n)$. Note que o passo 1 pode ser implementado em tempo constante, mantendo para cada vetor que representa uma solução, uma variável contendo o valor total dos itens atribuídos. Tal passo verificará todas os subconjuntos possíveis de itens, levando a um algoritmo bastante custoso na prática.

Um outro problema importante para a área de Otimização Combinatória é o Problema do Caixeiro Viajante. Trata-se de um problema clássico de grande importância, não apenas por suas várias aplicações, mas também por ser um problema para o qual diversas teorias foram desenvolvidas, testadas e difundidas.

Problema do Caixeiro Viajante (PCV): Dados um grafo completo $G = (V,E)$, cada aresta $e \in E$ com custo c_e, determinar um ciclo hamiltoniano C em G tal que $\sum_{e \in C} c_e$ é mínimo.

Um algoritmo força-bruta para o problema do caixeiro viajante poderia listar todas as sequências possíveis e armazenar a melhor. Como o grafo é completo, qualquer permutação dos vértices do grafo gera uma possível solução.

Algoritmo: Caixeiro-Viajante-FB(V,E,c)

Saída : Encontra um circuito gerador de custo mínimo

1 $\varphi^* \leftarrow \infty$
2 **para** *cada permutação* $(v_1,\ldots,v_n)$ *dos vértices de* V **faça**
3 $\quad C \leftarrow \{\{v_i, v_{i+1}\} : \ i \in \{1,\ldots,n-1\}\} \cup \{\{v_1, v_n\}\}$
4 $\quad \varphi \leftarrow \sum_{e \in C} c_e$
5 $\quad$ **se** $\varphi < \varphi^*$ **então**
6 $\quad\quad \varphi^* \leftarrow \varphi$
7 $\quad\quad C^* \leftarrow C$
8 **devolva** C^* , φ^*

Como o Algoritmo Caixeiro-Viajante-FB percorre por toda as permutações de vértices, faz claramente $n!$ iterações, para $n = |V|$. Demais operações em uma mesma iteração, são realizadas em tempo polinomial. Assim, a complexidade de tempo deste algoritmo é $\Omega(n!)$, e é assintoticamente maior que qualquer função do tipo a^n, para qualquer constante $a \geq 1$.

Para dar uma idéia do crescimento destas funções, considere um computador com velocidade de 1 Terahertz (mil vezes mais rápido que um computador de 1 Gigahertz) e funções que para cada tamanho n dão o número de instruções executadas

neste computador. A Tabela 1.1 mostra os tempos obtidos para algumas funções polinomiais e exponenciais, onde os tempos são dados em segundos (seg), dias e séculos (séc). Nota-se, para este exemplo, que o tempo computacional dado pelas funções exponenciais cresce muito mais rapidamente que as funções de tempo polinomial.

$f(n)!$	$n = 20$	$n = 40$	$n = 60$	$n = 80$	$n = 100$
n	$2{,}0\times10^{-11}$seg	$4{,}0\times10^{-11}$seg	$6{,}0\times10^{-11}$seg	$8{,}0\times10^{-11}$seg	$1{,}0\times10^{-10}$seg
n^2	$4{,}0\times10^{-10}$seg	$1{,}6\times10^{-9}$seg	$3{,}6\times10^{-9}$seg	$6{,}4\times10^{-9}$seg	$1{,}0\times10^{-8}$seg
n^3	$8{,}0\times10^{-9}$seg	$6{,}4\times10^{-8}$seg	$2{,}2\times10^{-7}$seg	$5{,}1\times10^{-7}$seg	$1{,}0\times10^{-6}$seg
2^n	$1{,}05\times10^{-6}$seg	1,1seg	13,3 dias	383,1séc	$4{,}0\times10^{8}$séc
3^n	$3{,}5\times10^{-3}$seg	140,7 dias	$1{,}3\times10^{7}$séc	$4{,}7\times10^{16}$séc	$1{,}6\times10^{26}$séc
$n!$	28,1dias	$2{,}6\times10^{26}$séc	$2{,}6\times10^{60}$séc	$2{,}3\times10^{97}$séc	$2{,}9\times10^{136}$séc

Tabela 1.1. Comparação de algumas funções de tempo computacional.

Os tempos computacionais indicados na Tabela 1.1 sugerem que algoritmos de tempo exponencial servirão apenas para entradas de tamanho muito pequeno e com utilidade bastante limitada. Por outro lado, dão também uma pequena amostra dos desafios encontrados na área de Otimização Combinatória. É uma área que contempla inúmeras aplicações práticas, onde mesmo pequenas melhorias algorítmicas podem representar uma economia de grande volume de recursos. Além disso, se torna envolvente na medida que mostra ser necessário explorar as estruturas combinatórias destes problemas de maneira a cercar as soluções ótimas ou mesmo as que apresentam proximidade em relação às soluções ótimas. Um típico exemplo é o Problema do Caixeiro Viajante, para o qual há algoritmos exatos com complexidade de tempo exponencial que encontraram soluções ótimas para entradas de 85900 vértices [Applegate et al. 2007]. Tal grandeza, confrontada com os tempos da Tabela 1.1, mostram claramente a importância de se investigar as estruturas combinatórias dos problemas na busca das melhores soluções.

1.3. Indução Matemática no Projeto de Algoritmos

A indução matemática é uma técnica poderosa de demonstração de resultados teóricos (teoremas, proposições, lemas, etc). O intuito desta seção é chamar a atenção para a analogia que existe entre o projeto de algoritmos e a prova de um resultado teórico usando indução matemática. Inicialmente, relembram-se os conceitos básicos de indução.

Uma prova por indução pode ser resumida do seguinte modo. Seja T um teorema que se quer demonstrar e suponha que o enunciado do resultado inclui um parâmetro n que é um número inteiro não-negativo. Genericamente, se Π é uma propriedade qualquer que depende de n, T pode ser enunciado por:

Para todo inteiro $n \geq n_0$, com $n_0 > 0$, a propriedade Π é verdadeira.

A técnica de prova por **indução fraca** para demonstrar T, se resume nos passos a seguir:

1. **base da indução**: mostrar que Π é verdadeira para n_0;
2. **hipótese de indução (fraca)**: para $n > n_0$, supor que Π é verdadeira para $n-1$;
3. **passo da indução**: a partir da hipótese de indução, provar que Π é verdadeira para n.

Note que a base garante a veracidade de Π para n_0. A partir daí, o passo de indução mostra que a propriedade também é correta para n_0+1, já que a hipótese de indução vale para n_0. Isso permite avançar um passo adiante pois, ao usar a validade de Π para n_0+1 no passo de indução, prova-se que Π é verdadeira para n_0+2. Fica claro que este argumento pode ser repetido indefinidamente, assegurando, então, que a propriedade vale para qualquer $n > n_0$. Alternativamente, a hipótese de indução pode ser alterada de modo a se exigir que ela valha para todo inteiro k tal que $n_0 \leq k \leq n-1$. Caso isso seja feito, ela toma a forma abaixo:

($2'$) **hipótese de indução (forte)**: para $n > n_0$, supor que Π é verdadeira para todo $n_0 \leq k \leq n-1$;

Essencialmente a estrutura da prova não muda, exceto que, no passo de indução, há uma possibilidade maior de uso da hipótese, uma vez que ela pode ser aplicada a diferentes valores de k. Considere o exemplo abaixo que, embora bastante simples, serve bem para ilustrar os conceitos.

Exemplo 1.3.1 *O serviço postal de um país hipotético tem selos de 4 e 5 centavos. Prove que qualquer valor de postagem n (em centavos) de uma carta, em que $n \geq 12$, pode ser conseguido usando-se apenas os selos de 4 e 5 centavos.*

Solução usando indução fraca:

- Base: o valor $n = 12$ pode ser conseguido usando-se 3 selos de 4 centavos.
- Hipótese: para $n > 12$, o valor de postagem $n-1$ pode ser obtido usando-se p' selos de 4 e q' selos de 5 centavos.
- Passo: provar que existem valores p e q inteiros não-negativos, para os quais $n = 4p+5q$, sendo $n > 12$. Primeiro, analisa-se o que ocorre quando $p' > 0$ na hipótese indutiva. Neste caso, faz-se $p = p'-1$ e $q = q'+1$, o que resulta em:

$$4p+5q = 4(p'-1)+5(q'+1) = 4p'+5q'+1 = (n-1)+1 = n.$$

No segundo caso, quando $p' = 0$, $n-1$ deve ser múltiplo de 5. Como $n > 12$, tem-se que $n-1 \geq 15$ e, portanto, $q' \geq 3$. Logo, fazendo-se $p = 4$ e $q = q'-3$, tem-se que

$$4p+5q = 4\times 4+5(q'-3) = 5q'+1 = (n-1)+1 = n. \quad \square$$

Solução usando indução forte:

- Base: para $n = 12$ o valor pode ser conseguido usando-se 3 selos de 4 centavos. Nada impede que o resultado possa ser mostrado para outros valores pequenos de n. Isso é feito aqui até $n = 15$ e as razões para tal são explicadas mais abaixo. Assim, tem-se que: $n = 13 = 2 \times 4 + 1 \times 5$, $n = 14 = 1 \times 4 + 2 \times 5$ e $n = 15 = 0 \times 4 + 3 \times 5$.
- Hipótese: para $n > 15$, qualquer valor de postagem k, com $15 \leq k \leq n-1$ pode ser obtido usando-se selos de 4 e de 5 centavos.
- Passo: provar que existem valores p e q inteiros não-negativos, para os quais $n = 4p + 5q$, sendo $n > 15$. Como $n > 15$, $n - 4 \geq 12$. Nesta situação, a hipótese de indução garante que existem inteiros não-negativos p' e q' tais que $n - 4 = 4p' + 5q'$. Logo, se $p = p' + 1$ e $q' = q$, tem-se

$$4p + 5q = 4(p' + 1) + 5q' = 4p' + 5q' + 4 = (n-4) + 4 = n. \quad \square$$

Na prova por indução forte, pode-se perguntar por que a demonstração da base se estendeu até $n = 15$. É muito importante perceber que isto ocorreu porque no passo de indução foi usado o fato de que o valor de postagem $n - 4$ pode ser pago usando apenas os selos de 4 e 5 centavos. Se na base não tivessem sido mostrados os casos para $n = 13, 14$ e 15, o passo de indução não teria provado o resultado para os valores de postagem, 17, 18 e 19 ! Com isto, não estaria demonstrada a veracidade da proposição para os valores $\{21, 22, 23, 25, 26, 27, \ldots\}$. Portanto, é preciso estar atento para que a base da indução (ou seja, a prova da validade da proposição para pequenos valores de n) seja feita até um valor grande o suficiente para poder ser usado na prova do passo.

Uma vez repassados os conceitos preliminares de indução matemática, examina-se a seguir como usar a ténica no projeto de algoritmos. De acordo com [Manber 1989], o princípio básico ao desenvolver um algoritmo para um dado problema pode ser resumido assim. *Não é preciso projetar todos os passos requeridos para resolver o problema "do zero". Na verdade, é suficiente garantir que* (a) *sabe-se como resolver pequenas instâncias do problema (caso base), e* (b) *a solução para a instância original do problema pode ser obtida a partir da solução de instâncias menores do mesmo problema (passo indutivo).* Como será visto, a prova por indução de que é sabido como resolver o problema para uma instância de tamanho, por exemplo, n, dá origem de modo bastante natural a um algoritmo recursivo. As chamadas recursivas são feitas para obter as soluções das instâncias de menor tamanho, enquanto o caso base da recursão confunde-se com o caso base da indução. O passo de indução diz como, a partir das soluções das instâncias menores, pode ser feito o cálculo que leva à solução da instância original. Três observações são importantes neste processo. Primeiramente, perceba que o fato do algoritmo obtido da prova por indução ser recursivo em nada restringe a técnica, até porque, sabe-se que todo algoritmo recursivo pode ser implementado de forma iterativa,

muitas vezes sem a necessidade de uso de pilhas. A segunda observação diz respeito à análise da complexidade do algoritmo obtido a partir da prova por indução a qual, pela natureza recursiva do algoritmo, é expressa por uma fórmula de recorrência. Finalmente, destaca-se que, como foi visto, há duas formas de prova por indução dependendo se a hipótese indutiva usada for a fraca ou a forte. Isso se reflete no projeto do algoritmo pois, para a indução fraca, ele seguirá o *paradigma incremental*, enquanto que, para a indução forte, ele seguirá o *paradigma de divisão e conquista*.

Para ilustrar as ideias discutidas no parágrafo anterior, aplica-se a técnica de projeto por indução ao *problema do subvetor de soma máxima* (PSSM) descrito a seguir.

Problema do Subvetor de Soma Máxima (PSSM): Dado vetor $A[1,\ldots,n]$ de inteiros irrestritos em sinal. encontrar o valor correspondente à maior soma de elementos que pode ter um subvetor de A.

Antes de prosseguir, é importante lembrar que, para todo $1 \leq i \leq j \leq n$, tem-se que $A[i,\ldots,j]$ é um subvetor de A, cuja soma dos elementos será denotada por $s(i,j)$, i.e., $s(i,j) = \sum_{k=i}^{j} A[k]$. Consequentemente, o que se quer é encontrar o valor $s(i^\star, j^\star)$, onde o par $(i^\star, j^\star)$ é tal que $s(i^\star, j^\star) = \max_{1\leq i\leq j\leq n}\{s(i,j)\}$. Por convenção, se um vetor só possui elementos negativos, a soma máxima tem valor nulo.

Claro que este problema admite uma solução ingênua em que se calcula o valor de $s(i,j)$ para todos pares (i,j) escolhidos de modo que $1 \leq i \leq j \leq n$. É fácil ver que a complexidade deste algoritmo é $O(n^3)$. Como será visto, o projeto de um algoritmo para o PSSM tanto pelo paradigma incremental (indução fraca) quanto pelo paradigma de divisão e conquista (indução forte) irá resultar em um algoritmo de complexidade menor que a do algoritmo ingênuo. Em ambos os casos, a princípio, a proposição a ser mostrada é a seguinte:

Sabe-se resolver o PSSM para todo vetor de tamanho $n \geq 1$.

Evidentemente, a indução será feita em n. Contudo, uma tentativa direta de prova desta proposição não conduz a um algoritmo com complexidade menor do que aquela do algoritmo ingênuo. Para conseguir alguma melhoria, deve-se *reforçar a hipótese*, ou seja, exigir que além de resolver o PSSM, devolvendo a sua solução, outras informações possam ser devolvidas pelo algoritmo também. O *reforço da hipótese* é uma técnica muito comum em provas por indução. Embora em um primeiro momento possa parecer contra-intuitivo que, ao se exigir provar um resultado mais forte a prova acabe se simplificando, isso pode ocorrer devido à natureza da prova por indução. Repare que a hipótese reforçada é carregada para o passo da indução, fazendo com que mais informações possam ser usadas neste último. Tendo este fato em mente, seguem-se os exemplos.

Exemplo 1.3.2 (Projeto de um algoritmo incremental para o PSSM)
Considere a seguinte proposição já com a hipótese reforçada:

> *Sabe-se resolver o PSSM para todo vetor de tamanho $n \geq 1$ e também calcular o valor da maior soma de um sufixo deste vetor.*

O sufixo (prefixo) de um vetor é um subvetor que termina (começa) na última (primeira) posição do mesmo. O valor a ser devolvido pelo PSSM será denotado por v e o valor do maior sufixo por s. Feitos estes esclarecimentos, a prova seria a seguinte.

Base: para $n = 1$, se $A[1] \geq 0$, $v = s = A[1]$, caso contrário, $v = s = 0$.

Hipótese (indução fraca): Para $n > 1$, sabe-se resolver o PSSM para todo vetor de tamanho $n-1$ e também calcular o valor da maior soma de um sufixo deste vetor.

Passo: mostra-se que a proposição é verdadeira para n usando a hipótese. Para tanto, sejam v' e s' os valores devolvidos pela aplicação da hipótese ao prefixo de tamanho $n-1$ do vetor A dado na entrada. Neste caso, os cálculos de v e s são simples. Primeiramente, v é computado por $v = \max\{v', s' + A[n]\}$. Isto ocorre porque, ou o subvetor que corresponde à solução ótima do prefixo $A[1, \ldots, n-1]$ continua sendo solução ótima para A e, portanto, não é sufixo de A, ou o sufixo de maior valor terminando em $A[n]$ é que fornece a solução ótima de A. Já o valor de s é dado por $\max\{s' + A[n], 0\}$ já que, o melhor sufixo ou é estendido com a inclusão de $A[n]$, ficando com soma não negativa, ou a inclusão deste elemento torna a soma negativa, fazendo com que o melhor sufixo seja o subvetor vazio, cuja soma é nula por definição. □

O pseudocódigo do algoritmo recursivo que resulta da prova por indução fraca é mostrado no Algoritmo PSSM. A complexidade deste algoritmo é dada pela fórmula de recorrência

$$T(n) = T(n-1) + O(1),$$

e, portanto, $T(n) \in O(n)$, bem mais eficiente do que o algoritmo ingênuo.

Exemplo 1.3.3 (Projeto de um algoritmo de divisão e conquista para o PSSM)
Considere a seguinte proposição já com a hipótese reforçada:

> *Sabe-se resolver o PSSM para todo vetor de tamanho $n \leq 1$ e também calcular (i) o valor da maior soma de um sufixo deste vetor, (ii) o valor da maior soma de um prefixo deste vetor, e (iii) o valor da soma de todos os elementos do vetor.*

O valor a ser devolvido pelo PSSM será denotado por v, o valor do maior sufixo (prefixo) por s (p) e a soma dos elementos por S.

Base: para $n = 1$, se $A[1] \geq 0$, $v = s = p = A[1]$, caso contrário, $v = s = p = 0$. Em qualquer caso, $S = A[1]$.

Algoritmo: PSSM(A,n,v,s) - Algoritmo incremental para o PSSM

Saída : Valor do maior sufixo, v, e prefixo s.

```
1  se n = 1 então
2      ▷ caso base
3      se A[1] ≥ 0 então v ← s ← A[1]
4      senão v ← s ← 0
5  senão
6      PSSM(A,n−1,v′,s′)                    ▷ hipótese indutiva
7      ▷ passo de indução
8      v ← max{v′, s′ + A[n]}
9      s ← max{s′ + A[n], 0}
10 devolva (v,s)
```

Hipótese (indução forte): Para $n > 1$, sabe-se resolver o PSSM para todo vetor de tamanho k, para qualquer k satisfazendo $1 \leq k \leq n-1$ e também calcular o valor da maior soma de um sufixo deste vetor, assim como a de um prefixo e a soma total dos seus elementos.

Passo: mostra-se que a proposição é verdadeira para n usando a hipótese. Seja $m = \lfloor \frac{n}{2} \rfloor$. Suponha, pela hipótese indutiva, que o PSSM foi resolvido para o prefixo $A[1,\ldots,m]$, devolvendo os valores v_e, s_e, p_e e S_e correspondentes às maiores somas de um subvetor qualquer, de um sufixo e de um prefixo, além da soma total dos seus elementos, respectivamente. Da mesma maneira, a hipótese indutiva aplicada ao sufixo $A[m+1,\ldots,n]$ devolve os valores v_d, s_d, p_d e S_d. A partir das informações devolvidas pelos dois subproblemas, os valores respectivos de v, s, p e S a serem devolvidos pelo PSSM quando aplicado a A podem ser computados conforme explicado a seguir.

Obviamente, tem-se que $S = S_e + S_d$. É fácil ver que o prefixo de maior soma de A ou é o mesmo de $A[1,\ldots,m]$ ou contém todo este subvetor e mais o maior prefixo de $A[m+1,\ldots,n]$. Consequentemente, chega-se a $p = \max\{p_e, S_e + p_d\}$. Usando argumentos simétricos para o sufixo, pode-se concluir que $s = \max\{s_d, S_d + s_e\}$. Finalmente, para o cálculo de v, observa-se que o subvetor de maior soma em A só pode ser: (a) o mesmo de $A[1,\ldots,m]$ ou (b) o mesmo de $A[m+1,\ldots,n]$ ou (c) um subvetor que começa em uma posição que está em $A[1,\ldots,m]$ e termina em uma posição que está em $A[m+1,\ldots,n]$. Caso a condição (c) seja a que prevalece, percebe-se que o subvetor de soma máxima está dividido em duas partes, uma que é sufixo de $A[1,\ldots,m]$ e outra que é prefixo de $A[m+1,\ldots,n]$. Porém, a maior soma atingida por um subvetor nesta situação tem que corresponder à soma do valor máximo de um sufixo de $A[1,\ldots,m]$ com o valor máximo de um prefixo de $A[m+1,\ldots,n]$. Em outras palavras, na condição (c) acima, $v = s_e + p_d$. Assim, no caso geral, tem-se que $v = \max\{v_e, v_d, s_e + p_d\}$. □

O pseudocódigo do algoritmo recursivo que resulta da prova por indução forte é mostrado na Figura 1.1 Resta analisar a complexidade do algoritmo de divisão e con-

```
1.  PSSM-DC(A,i,j,v,s,p,S)
    (* resolve o PSSM para o subvetor A[i,...,j] de A *)
    (* v, s, p e S são devolvidos pelo algoritmo *)
2.  se i = j então (* caso base *)
3.      se A[i] ≥ 0, então v = s = p = A[i]
4.      se não v = s = p = 0
5.      S ← A[i]
6.  se não
7.      m ← (i + j)/2 (* m: posição mediana *)
8.      PSSM-DC(A,i,m,v_e,s_e,p_e,S_e)        (* hipótese indutiva *)
9.      PSSM-DC(A,m + 1,j,v_d,s_d,p_d,S_d) (* hipótese indutiva *)
        (* passo de indução *)
10.     v ← max{v_e, v_d, s_e + p_d}
11.     s ← max{s_d, S_d + s_e}
12.     p ← max{p_e, S_e + p_d}
13.     S ← S_e + S_d
14. fim.
```

Figura 1.1. Algoritmo de divisão e conquista para o PSSM.

quista para o PSSM. Aproveita-se esta oportunidade para apresentar o chamado *Teorema Mestre*, que é muito útil no cálculo de equações de recorrência como as que representam a complexidade de algoritmos projetados através da estratégia de divisão e conquista.

Suponha que se tenha um algoritmo de divisão e conquista para um problema cuja entrada é medida pelo parâmetro n, um número inteiro não negativo. Tirando os casos *base* correspondentes a instâncias pequenas em que o problema é resolvido de forma simples e em tempo $O(1)$, para duas constantes a e b, $a \geq 1$ e $b > 1$, o algoritmo encontra a solução da instância original subdividindo-a em a instâncias menores de tamanho aproximadamente n/b que são resolvidas recursivamente. Ao final, as soluções dos a subproblemas são combinadas por um algoritmo de complexidade $f(n)$ que calcula a solução da instância original. Então, se $T(n)$ denota a complexidade do algoritmo de divisão e conquista, tem-se que $T(n) = aT(n/b) + f(n)$ para todo $n \geq n_0$ para algum $n_0 > 0$. Neste contexto, o resultado abaixo é passível de ser aplicado.

Teorema 1.3.1 (Teorema Mestre (cf., [Cormen et al. 2009])) *Sejam $a \geq 1$ e $b \geq 2$ constantes, $f(n)$ uma função e $T(n)$ definida para os inteiros não-negativos pela relação de recorrência*

$$T(n) = aT(n/b) + f(n).$$

Então $T(n)$ pode ser limitada assintoticamente da seguinte maneira:

1. *Se* $f(n) \in O(n^{\log_b a - \varepsilon})$ *para alguma constante* $\varepsilon > 0$*, então* $T(n) \in \Theta(n^{\log_b a})$.
2. *Se* $f(n) \in \Theta(n^{\log_b a})$*, então* $T(n) \in \Theta(n^{\log_b a} \log n)$.
3. *Se* $f(n) \in \Omega(n^{\log_b a + \varepsilon})$*, para algum* $\varepsilon > 0$ *e se* $af(n/b) \leq cf(n)$*, para alguma constante* $c < 1$ *e para n suficientemente grande, então* $T(n) \in \Theta(f(n))$.

Neste teorema, o valor de n/b na fórmula de recorrência pode ser substituído indistintamente por $\lfloor n/b \rfloor$ ou $\lceil n/b \rceil$ em qualquer uma das a parcelas, sem que o resultado final seja alterado.

Assim, para o algoritmo de divisão e conquista desenvolvido acima para o PSSM, tem-se que $T(n) = 2T(n/2) + O(1)$, fazendo com que $a = 2$, $b = 2$ e $f(n) \in O(1)$ (onde $T(n/2)$ é a complexidade de resolver o PSSM em cada uma das suas metades). Desta forma, $\log_b a = 1$ e, assim, chega-se a $f(n) \in O(n^{\log_b a - \varepsilon})$, para qualquer ε no intervalo $(0,1)$. Conclusão: a recorrência acima cai no caso 1 do Teorema Mestre, o que implica que $T(n) = \Theta(n)$. Ou seja, a complexidade assintótica do algoritmo de divisão e conquista é idêntica àquela do algoritmo incremental.

Na análise da complexidade de tempo de alguns algoritmos apresentados no texto, será necessário resolver recorrências lineares homogêneas e não-homogêneas. As recorrências homogêneas são equações lineares onde um termo da recorrência é definido a partir de outros valores da própria recorrência, possivelmente multiplicadas por constantes, mas não contém termos sem referência à recorrência. Uma recorrência homogênea f_n definida através de k termos anteriores, têm a seguinte forma:

$$c_n f_n + c_{n-1} f_{n-1} + \cdots + c_{n-k} f_{n-k} = 0,$$

onde f_i é a recorrência para um valor i e c_i são termos que independem da recorrência, com c_n e c_{n-k} não-nulos. A equação característica desta recorrência é dada por

$$c_k x^k + c_{k-1} x^{k-1} + \cdots + c_0 = 0.$$

Como a recorrência acima possui $k+1$ termos, é necessário que a recorrência seja definida também com os k primeiros termos de maneira direta, para que os próximos termos possam ser obtidos através da relação de recorrência. Para a obtenção de uma cota superior, pode-se utilizar o seguinte teorema.

Teorema 1.3.1 *(*cf. *[Brassard e Bratley 1988, Rosen 2012]) Seja* f_n *uma recorrência linear homogênea e* $p(n)$ *sua equação característica. Se* $p(n)$ *tem raízes* $a_1, \ldots, a_r$*, cada raiz* a_i *com multiplicidade* m_i*, então* f_n *é dada por*

$$f_n = \sum_{i=1}^{r} \sum_{j=0}^{m_i - 1} c_{i,j} n^j a_i^n,$$

onde os termos $c_{i,j}$ *são constantes.*

Exemplo 1.3.4 *Considere a recorrência de Fibonacci, $F(n) = F(n-1) + F(n-2)$, para $n \geq 2$. Reorganizando a recorrência, temos $F(n) - F(n-1) - F(n-2) = 0$. A equação característica é dada por $a^2 - a - 1 = 0$ cujas soluções são $a_1 = (1+\sqrt{5})/2$ e $a_2 = (1-\sqrt{5})/2$, ambas com multiplicidade $m_1 = m_2 = 1$. Assim, a recorrência $F(n)$ é dada por $F(n) = c_1 n^0 a_1^n + c_2 n^0 a_2^n = c_1 a_1^n + c_2 a_2^n$. De maneira assintótica temos que $F(n)$ é $O\left(\frac{1+\sqrt{5}}{2}\right)^n$. De maneira mais precisa, podemos utilizar os valores de $F(0)$ e $F(1)$ para determinar os valores de c_1 e c_2, pelo sistema de equações*

$$\begin{aligned} F(0) &= c_1 + c_2 = 0 \\ F(1) &= c_1 \frac{1+\sqrt{5}}{2} + c_2 \frac{1-\sqrt{5}}{2} = 1, \end{aligned}$$

que resolvendo, nos dá $c_1 = 1/\sqrt{5}$ e $c_2 = -1/\sqrt{5}$. Portanto, os números de Fibonacci são dados por

$$F(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n.$$

Para se obter limitantes para as recorrências não-homogêneas, pode-se utilizar o seguinte teorema.

Teorema 1.3.2 (cf. *[Brassard e Bratley 1988]) Seja f_n uma recorrência linear não-homogênea dada por*

$$c_k f_n + c_{k-1} f_{k-1} + \cdots + c_0 f_{n-k} = b_1^n q_1(n) + b_2^n q_2(n) + \cdots + b_t q_t(n), \qquad (1)$$

onde (i) $c_0, \ldots, c_k$ são constantes; (ii) $b_1, \ldots, b_t$ são constantes distintas e $q_1, \ldots, q_t$ são polinômios de grau $d_1, \ldots, d_t$, respectivamente. O polinômio característico desta recorrência é dado por

$$p(n) = (c_k x^k + c_{k-1} x^{k-1} + \cdots + c_0)(x-b_1)^{d_1+1}(x-b_2)^{d_2+1} \ldots (x-b_t)^{d_t+1}.$$

Se $p(n)$ tem r raízes distintas, $a_1, \ldots, a_r$, cada raiz a_i com multiplicidade m_i, então f_n é dada por

$$f_n = \sum_{i=1}^{r} \sum_{j=0}^{m_i-1} e_{i,j} n^j a_i^n,$$

onde os termos $e_{i,j}$ são constantes.

Note que o polinômio característico $p(n)$ possui uma parte que representa o polinômio característico da recorrência homogênea, obtida removendo os termos do lado direito da recorrência (1), e outra advinda dos termos do lado direito da recorrência.

O próximo exemplo é apresentado em [Brassard e Bratley 1988].

Exemplo 1.3.5 *Vamos resolver a recorrência $t_n = 2t_{n-1} + n + 2^n$ quando $n \geq 1$ e $t_0 = 0$. Podemos reescrevê-la como*

$$t_n - 2t_{n-1} = n + 2^n. \tag{2}$$

A recorrência homogênea obtida só pelo lado esquerdo de (2) é dada por $t_n - 2t_{n-1} = 0$, que possui equação característica $x - 2 = 0$, que claramente tem raiz 2.

Considere o lado direito de (2). Vamos colocar na forma requerida pelo Teorema 1.3.2. Podemos escrever $n + 2^n$ como

$$\begin{aligned} n + 2^n &= 1^n n + 2^n 1 \\ &= b_1^n p_1(n) + b_2^n p_2(n), \end{aligned}$$

onde $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$ e $p_2(n) = 1$. O grau do polinômio $p_1(n)$ é $d_1 = 1$ e de $p_2(n)$ é $d_2 = 0$, temos a seguinte equação característica para a recorrência não-homogênea: $(x-2)(x-1)^{1+1}(x-2)^{0+1} = 0$, que simplificando, nos dá

$$(x-2)^2(x-1)^2 = 0.$$

Como temos duas raízes com valor 1 e duas raízes com valor 2, ambas com multiplicidade 2, podemos escrever a recorrência na forma

$$\begin{aligned} t_n &= c_1 1^n + c_2 n 1^n + c_3 2^n + c_4 n 2^n \\ &= c_1 + c_2 n + c_3 2^n + c_4 n 2^n. \end{aligned}$$

Com isso, já podemos concluir que t_n é $O(n2^n)$. Para calcular a recorrência exata, devemos calcular os valores de c_1, c_2, c_3 e c_4, que podem ser obtidos resolvendo-se um sistema obtido com os quatro primeiros valores da recorrência: $t_0, \ldots, t_3$.

$$\begin{array}{llllll} c_1+ & & c_3 & & = & 0 \quad \textit{para } n = 0 \\ c_1+ & c_2+ & 2c_3+ & 2c_4 & = & 3 \quad \textit{para } n = 1 \\ c_1+ & 2c_2+ & 4c_3+ & 8c_4 & = & 12 \quad \textit{para } n = 2 \\ c_1+ & 3c_2+ & 8c_3+ & 24c_4 & = & 35 \quad \textit{para } n = 4. \end{array}$$

Tal sistema nos dá $c_1 = -2$, $c_2 = -1$, $c_3 = 2$ e $c_4 = 1$. Portanto, a recorrência é dada por

$$t_n = -2 - n + 2^{n+1} + n2^n.$$

Para mais detalhes sobre recorrências, veja [Brassard e Bratley 1988, Manber 1989] e [Cormen et al. 2009].

1.4. Algoritmos Gulosos

O projeto por algoritmos gulosos está, em geral, associado a técnicas simples, intuitivas e, em geral, fáceis de implementar. Sendo assim, uma das técnicas mais utilizadas no projeto de algoritmos para problemas de otimização combinatória. Para aplicar esta técnica, a solução deve ser construída a partir de elementos básicos. Em um problema de minimização, cada elemento possui um custo e sua escolha fornece uma contribuição para a construção de uma solução. Analogamente, em um problema de maximização, cada elemento gera um benefício, porém, consome parte dos recursos disponíveis. A idéia básica do projeto de um algoritmo guloso é, iterativamente selecionar um elemento que melhor contribua para a construção de uma solução, considerando seu custo ou benefício. Elementos cuja escolha não levam a uma solução viável são descartados. O algoritmo termina quando os elementos selecionados formarem uma solução ou contiverem uma solução, que pode ser extraída de maneira direta.

A técnica é exemplificada com os problemas da Árvore Geradora de Custo Mínimo, do Caminho Mínimo e da Mochila Fracionária.

1.4.1. Algoritmo Guloso para o Problema da Árvore Geradora de Custo Mínimo

Esta seção trata do problema da Árvore Geradora de Custo Mínimo, que tem várias aplicações em estruturas que envolvem conectividade. Exemplos contemplam os projetos de redes de computadores, de circuitos VLSI, de tubulações, de saneamento, de telecomunicações, etc. Além disso, o problema aparece como subproblema de vários problemas de otimização.

Problema da Árvore Geradora de Custo Mínimo (PAGM): Dado grafo conexo, não-orientado $G = (V, E)$, cada aresta $e \in E$ com custo c_e, encontrar uma árvore geradora T de G, tal que $\sum_{e \in T} c_e$ é mínimo.

Um dos algoritmos que resolvem este problema na otimalidade, de maneira elegante, é o Algoritmo de Prim. A estratégia usada por este algoritmo, é começar uma árvore a partir de um vértice qualquer e a cada iteração uma aresta é acrescida, conectando a árvore corrente com um novo vértice.

A escolha de cada aresta é feita utilizando a estratégia gulosa. Isto é, dentre as arestas que acrescentam mais um vértice à árvore, é escolhida uma de custo mínimo. Interpretando o custo de uma aresta como distância, o algoritmo iterativamente anexa um vértice que está mais próximo da árvore. O algoritmo termina quando todos os vértices forem anexados à árvore. A seguir, as iterações são descritas de maneira mais precisa.

Denote por T_i a árvore construída até o início da i-ésima iteração e Q_i os vértices que não pertencem a T_i. No decorrer da iteração, os vértices de T_i são denominados internos e os de Q_i de externos. As arestas que ligam vértices de T_i a vértices de Q_i são chamadas arestas do corte. Dentre estas, usando estratégia gulosa, a aresta escolhida para fazer parte da solução na i-ésima iteração é uma aresta do corte que tem custo

mínimo.

Para que a escolha da aresta do corte possa ser feita rapidamente, o algoritmo usa vetores pred$[\cdot]$ e dist$[\cdot]$ para representar as arestas da árvore e agilizar sua busca. Se $u \in Q_i$ então dist$[u]$ contém o custo de uma aresta (do corte) que liga u a algum vértice $w \in T_i$, e neste caso guarda tal ligação fazendo pred$[u] = w$. Note que a representação das arestas definidas por pred, definem uma árvore enraizada no primeiro vértice da árvore. Com isso, a busca pela aresta do corte de custo mínimo recai na busca de um vértice $u \in Q_i$ com dist$[u]$ mínimo. Para isto, usa-se uma fila de prioridade em dist, que permite remover um vértice com menor dist$[u]$ e diminuir o valor de dist$[u]$ quando for encontrada aresta mais leve ligando u à T_i.

Na primeira iteração a árvore T_1 começa vazia. A cada iteração, o vértice externo a ser anexado, possui menor dist$[u]$. Note que na primeira iteração todos os vértices começam com dist$[v] = \infty$, exceto pela raiz r que começa com dist$[r] = 0$, sendo portanto o primeiro vértice inserido em T_1. Após incluir um vértice u em T_i, os valores de dist$[v]$ são atualizados para os vértices v que são externos e adjacentes a u, já que as arestas adjacentes a u serão as únicas da iteração a sair ou entrar no corte. Após esta operação, os vetores pred$[\cdot]$ e dist$[\cdot]$ podem ficar incorretos para os vértices de Q_i adjacentes a u. Assim, para cada aresta do corte que liga u a um vértice $v \in Q_i$, atualizamos, se necessário, pred$[v]$ e dist$[v]$.

Algoritmo: Prim(G,r) onde $G = (V,E,c)$ é um grafo ponderado e $r \in V$.

Saída : Árvore geradora de custo mínimo.

```
1  para v ∈ V faça
2      dist[v] ← ∞
3      pred[v] ← nil
4  dist[r] ← 0
5  Q ← V
6  enquanto Q ≠ ∅ faça
7      u ← Extrai-Min(Q)
8      para cada v ∈ Adj(u) ∩ Q faça
9          se c(u,v) < dist[v] então
10             pred[v] ← u
11             dist[v] ← c(u,v)
12 devolva (pred)
```

Lema 1.4.1 *Seja T_{i-1} uma árvore que pode ser estendida para uma árvore geradora de custo mínimo. Se e é uma aresta de custo mínimo com exatamente uma extremidade em T_{i-1}, então, a árvore $T_i = T_{i-1} + e$, pode ser estendida para uma árvore geradora de custo mínimo de G.*

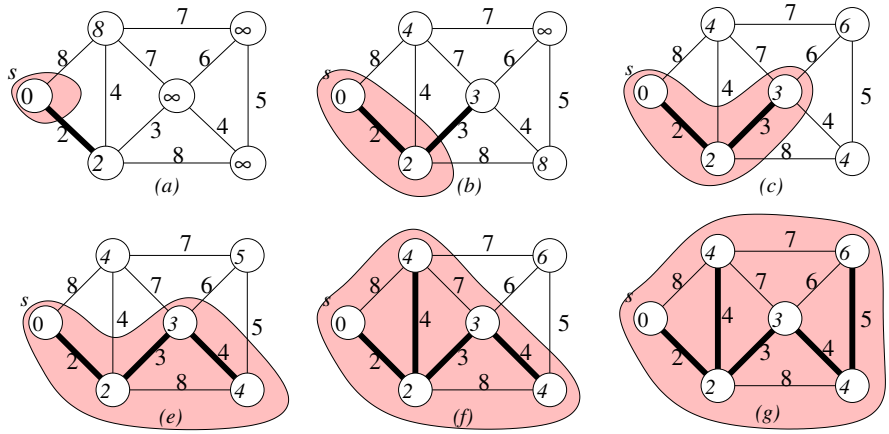


Figura 1.2. Simulação da execução do Algoritmo de Prim.

Prova. Seja T_{i-1} uma árvore que pode ser estendida para uma solução ótima. Seja T^* uma árvore geradora de custo mínimo que contém T_{i-1}.

A árvore $T^* + e$ possui um circuito passando por e. Como e liga um vértice de T_{i-1} a um vértice fora de T_{i-1}, existe uma aresta f do circuito que também tem exatamente uma extremidade em T_{i-1} Portanto, a árvore $T^+ = T^* + e - f$ é uma árvore geradora. Pela escolha gulosa da aresta e vale que $c(e) \leq c(f)$, e portanto $c(T^+) \leq c(T^*)$. Como T^* é de custo mínimo, temos que T^+ também é de custo mínimo e contém a árvore T_i. □

Teorema 1.4.2 *Para todo grafo ponderado e conexo, o Algoritmo de Prim encontra uma árvore de custo mínimo.*

Prova. Considere a seguinte afirmação:

No fim da iteração i, as seguintes propriedades são válidas, para $i = 1, \ldots, n$.

- Se T_i é a árvore obtida no fim da iteração i pelo Algoritmo de Prim, então T_i pode ser estendida para uma árvore geradora de custo mínimo.
- Para todo vértice $v \notin T_i$, tem-se que dist$[v]$ é o menor custo de uma aresta que liga v à algum vértice de T_i, através de uma aresta $\{v, pred[v]\}$.

A prova será por indução no número da iteração i, para $i = 1, \ldots, n$.
Base: para $i = 1$, a árvore T_1, obtida ao final da primeira iteração, é a árvore contendo apenas o vértice raiz, que está contido em qualquer árvore geradora de custo mínimo. Ao inserir o vértice raiz, as únicas arestas necessárias para atualização são as arestas bincidentes à raiz.

Hipótese (indução fraca): No fim da iteração i, temos uma árvore T_i que pode ser estendida para uma árvore geradora de custo mínimo. Os vetores pred e dist indicam uma aresta de menor custo que liga $w \notin T_i$ à algum vértice de T_i. O custo desta aresta é dado por dist$[w]$.

Passo: considere uma árvore T_i obtida ao fim da iteração i. Esta é a mesma árvore obtida no início da iteração $i+1$. Seja u o vértice escolhido pelo algoritmo. Pela hipótese de indução, temos que $\{u, pred[u]\}$ é uma aresta de custo mínimo com exatamente uma extremidade em T_i. Ao incluir o vértice u, o custo de uma aresta com exatamente uma extremidade em T_{i+1} pode alterar apenas se for adjacente a u. Assim, basta atualizar, se necessário, as distâncias para os vértices adjacentes à u com vértices fora de T_{i+1}. Como a aresta $\{u, pred[u]\}$ tem custo mínimo, dentre as arestas que ligam T_i a Q_i, temos pelo Lema 1.4.1 que existe árvore geradora de custo mínimo que contém T_{i+1}. □

Nem sempre é possível garantir que um algoritmo guloso produza uma solução ótima para um problema. Mas quando o problema admite certas propriedades, é possível que tais algoritmos produzam soluções ótimas. Duas propriedades que costumam aparecer no projeto de algoritmos gulosos que produzem soluções ótimas, são a da subestrutura ótima, que costuma aparecer nos problemas que podem ser resolvidos por projeto de indução, e da escolha gulosa.

Na resolução de um problema pelo projeto por indução, definimos o problema de maneira indutiva, ou um outro cuja resolução nos leve a uma solução ótima do problema original. Para se ter a propriedade de subestrutura ótima, uma solução ótima de um problema deve conter soluções ótimas para seus subproblemas. Note que o problema PAGM admite subestrutura ótima. O problema poderia ser definido com a seguinte subestrutura: Dado grafo G e subárvore T de G, encontrar uma árvore geradora de G de custo mínimo que contém T. A outra propriedade é a da escolha gulosa. Esta propriedade diz que ao escolher um próximo elemento a de maneira gulosa, ainda temos condições de atingir uma solução ótima. De fato, esta foi a propriedade usada no Lema 1.4.1, que garante que a cada iteração fazemos uma escolha gulosa que pertencerá a uma árvore geradora de custo mínimo.

A complexidade de tempo do Algoritmo de Prim depende da implementação de Q. Abaixo, temos a análise por duas estruturas de dados, o heap binário e o heap de Fibonacci. Para mais informações sobre estas estruturas de dados, veja [Cormen et al. 2009, Manber 1989]. Se for usado um heap binário, pode-se construir o heap da linha 5 em tempo $O(|V|)$. A cada iteração do laço do passo 6 é removido um elemento de Q. Como o laço é iterado por $|V|$ vezes e cada chamada da rotina Extrai-Min(Q) é feita em tempo $O(\log(|V|))$, o passo 7 é executado em $O(|V|\log|V|)$. O laço do passo 6 juntamente com o laço do passo 8 percorre todas as arestas do grafo, visitando cada uma duas vezes, uma para cada extremidade. Portanto os passos 8–10 são executados $O(|E|)$ vezes. Já o passo 11 é uma operação de decrementação, que em um heap binário pode ser implementado com tempo $O(\log|V|)$. Portanto o tempo total desta implementação

é $O(|V|\log|V| + |E|\log|V|)$, i.e., $O(|E|\log|V|)$.

No caso de se usar um heap de Fibonacci, a operação Extrai-Min pode ser feita em tempo amortizado $O(\log V)$ e a operação que decrementa um valor do heap pode ser feita em tempo amortizado $O(1)$. Assim, esta implementação tem complexidade de tempo $O(|E| + |V|\log|V|)$.

1.4.2. Algoritmo Guloso para o Problema dos Caminhos Mínimos

Esta seção aborda o problema de se encontrar caminhos de custo mínimo de um vértice aos demais vertices em um grafo orientado com custos não-negativos nos arcos. Trata-se de um dos problemas básicos em grafos que pode ser resolvido por um algoritmo guloso e que também contempla características de um algoritmo por programação dinâmica, que será visto na Seção 1.5.

Problema dos Caminhos Mínimos (PCM): Dados grafo orientado $G = (V,E)$, onde cada arco $e \in E$ tem custo não-negativo c_e, um vértice $s \in V$, encontrar para cada $t \in V$, um caminho $P_{s,t}$ de s a t tal que $\sum_{e \in P_{s,t}} c_e$ é mínimo.

A estratégia utilizada pelo algoritmo a ser visto é muito próxima da utilizada no Algoritmo de Prim, para construção de uma árvore geradora de custo mínimo. No Algoritmo de Prim, a árvore começa com apenas o vértice raiz e a cada iteração a árvore é expandida anexando um novo vértice externo v. A estratégia gulosa usada pelo Algoritmo de Prim é iterativamente escolher um vértice que está mais próximo da árvore corrente.

No que segue, apresenta-se o Algoritmo de Dijkstra, que usa a mesma idéia do Algoritmo de Prim, mas em vez de anexar um vértice mais próximo da árvore, anexamos um que esteja mais próximo do vértice de origem s, uma vez que o problema busca por caminhos mínimos com origem em s. Com isto, o vetor dist$[v]$ mantém a melhor distância encontrada do vértice s até v. Inicialmente todos os vértices começam com valor dist$[v] = \infty$, exceto pelo vértice de origem que começa com dist$[s] = 0$. Estes valores iniciais garantem que o primeiro vértice a ser removido de Q é justamente o vértice de origem. Enquanto v está em Q, dist$[v]$ é atualizado sempre que se encontra um caminho melhor. Quando um vértice v é removido de Q, o caminho mínimo de s a v é definido e este não mais se altera.

Cada iteração do Algoritmo de Dijkstra também começa com uma árvore T, um vetor dist indexado nos vértices e uma fila de prioridade Q que prioriza vértices com menor dist e contém os vértices cuja busca de um caminho mínimo não foi finalizada. Neste caso, dist$[v]$ começa a iteração contendo a menor distância encontrada de um caminho de s para v, para todo $v \notin T$. Com isso, o algoritmo escolhe um vértice $v \notin T$ com dist$[v]$ mínimo. Feita a inclusão deste vértice, as estruturas de dados são atualizadas.

Em vez de representar os caminhos encontrados de s aos vértices de T, utiliza-se o vetor pred$[\cdot]$ para representar o percurso inverso, de um vértice de T a s. Se $v \in T$, o vértice pred$[v]$ é o vértice que precede v no caminho de s a v. Assim, o cami-

nho $(v, \text{pred}[v], \text{pred}[\text{pred}[v]], \ldots, s)$ representa o caminho encontrado de s a v na ordem inversa. A árvore representada pelo vetor pred, é denominada *árvore de caminhos mínimos*, que como será visto, representará ao final do algoritmo, os caminhos mínimos em ordem invertida encontrados de um vértice s até outro vértice de G.

Uma descrição deste processo é apresentada no Algoritmo de Dijkstra. Note que a única diferença com o Algoritmo de Prim ocorre no bloco da linha 11, que atualiza dist e pred.

Algoritmo: Dijkstra(G,s) grafo ponderado $G=(V,E,c)$ e vértice $s \in V$.

Saída : Árvore de caminhos mínimos com origem em s.

```
1  para v ∈ V faça
2      dist[v] ← ∞
3      pred[v] ← nil
4  dist[s] ← 0
5  Q ← V
6  S ← ∅                                   ▷ usado apenas na prova
7  enquanto Q ≠ ∅ faça
8      u ← Extrai-Min(Q)
9      S ← S ∪ {u}                         ▷ usado apenas na prova
10     para cada v ∈ Adj(u) ∩ Q faça
11         se dist[u] + c(u,v) < dist[v] então
12             pred[v] ← u
13             dist[v] ← dist[u] + c_{u,v}
14 devolva (dist, pred)
```

O algoritmo explora a subestrutura ótima deste problema, quando considera que se um caminho de s a t é mínimo, qualquer subcaminho também é mínimo para os correspondentes extremos. O seguinte lema, cuja prova deixamos para o leitor, é válido.

Lema 1.4.3 *Seja $G=(V,E)$ um grafo orientado onde cada arco $e \in E$ possui um custo não-negativo c_e. Se $P=(v_1,v_2,\ldots,v_k)$ é um caminho mínimo de v_1 a v_k, então o caminho $P'=(v_i,\ldots,v_j)$ obtido de P, é um caminho mínimo de v_i a v_j, para $1 \leq i \leq j \leq k$.*

Naturalmente este resultado também vale quando a origem dos caminhos é sempre o vértice s. O seguinte lema também é válido e pode ser provado por indução.

Lema 1.4.4 *Se $d[v] < \infty$, então a sequência $v, \text{pred}[v], \text{pred}[\text{pred}[v]], \ldots$ termina no vértice de origem s e representa a sequência inversa de um caminho de s a v de custo $d[v]$.*

Prova. Por indução no tamanho de S.
Base: Pelas atribuições iniciais, o primeiro vértice inserido em S possui $d[s] = 0$ e todos os demais possuem $d[v] = \infty$. Assim, s é o primeiro vértice inserido em S e o resultado é válido.
Hipótese: Suponha que o resultado é válido quando $|S| = k$.
Passo: Seja v o $(k+1)$-ésimo vértice inserido em S. Se $d[v] = \infty$, não há nenhum arco saindo de um vértice de S e portanto não há caminho de s para v. Assim, $d[v]$ está calculado corretamente. Caso contrário, existe arco (u,v) de maneira que u entrou em S antes de v e $\text{pred}[v] = u$ e $d[v] = d[u] + c(u,v)$. Como há caminho de s para u tem-se que $d[v]$ é a soma do custo de um caminho de s até u e do custo do arco (u,v). Portanto $d[v]$ é o custo do caminho definido de s até u e de u para v pelo arco (u,v). □

Teorema 1.4.5 *Dado grafo $G = (V,E)$, onde cada arco $e \in E$ tem custo não-negativo c_e, e vértice $s \in V$, o Algoritmo de Dijkstra(G,s) produz uma árvore de caminhos mínimos com origem em s.*

Prova. A prova usa o seguinte invariante: No início do laço da linha 7 $d[v]$ é o custo do caminho mínimo de s até v, para todo $v \in S$ e $(v, \text{pred}[v], \text{pred}[\text{pred}[v]], \ldots, s)$ é um caminho de custo $d[v]$.

No início $S = \emptyset$ e o invariante é trivialmente válido. Suponha, para efeitos de contradição, que há alguma entrada para o qual o Algoritmo de Dijkstra computa incorretamente o caminho mínimo de um vértice, quando este foi inserido em S. Considere a execução neste grafo e seja u o primeiro vértice a ser inserido em S com custo calculado incorretamente. Certamente $u \neq s$, uma vez que $d[s]$ começa com 0, tendo sido o primeiro vértice de S. Seja P um caminho mínimo de s para u. Note que P deve ter algum vértice que não pertence a $S \cup \{u\}$. Seja y o primeiro vértice no caminho de P tal que $y \notin S$ e $x \in S$ o vértice que precede y em P. Seja P_x (resp. P_y) o caminho obtido de P saindo de s e chegando em x (resp. y). Pela subestrutura ótima, P_x e P_y são caminhos mínimos para seus respectivos vértices. Como u foi escolhido para ser removido antes de y, temos que $d[u] \leq d[y]$. Pela escolha de u, o caminho obtido pelo Algoritmo de Dijkstra para x é mínimo. No momento que x foi inserido em S, houve a atualização da estrutura percorrendo os vértices adjacentes a x.

Com isso $d[y]$ teve seu custo computado corretamente, uma vez que ao processar o arco (x,y) a partir de x, $d[y]$ receberia o valor $d[x] + c(x,y)$. Se existe algum arco de y para u com custo positivo, temos que $d[u] > d[y]$ e portanto y deveria ser removido de Q antes de u, o que contradiz a escolha de y. Assim, todos os arcos em P de y a u são nulos e portanto $d[y] \leq d[u]$. Portanto, $d[u] = d[y]$ e pelo lema anterior, o caminho dado por $(u, \text{pred}[u], \text{pred}[\text{pred}[u]], \ldots, s)$ é um caminho válido de custo $d[u]$ e consequentemente também é de custo mínimo, contrariando a escolha de u. □

A análise da complexidade de tempo do Algoritmo de Dijkstra depende da estrutura de dados usada para representar a fila de prioridades Q e é análoga a realizada para

o Algoritmo de Prim, descrito para o Problema da Árvore Geradora de Custo Mínimo. Neste caso, o Algoritmo de Dijkstra tem complexidade de tempo $O((|V|+|E|)\log|V|)$ usando heap e $O(|E|+|V|\log|V|)$ usando heap de Fibonacci.

1.4.3. Algoritmos Gulosos para os Problemas da Mochila Fracionária e Binária

O problema da Mochila Binária, como definido na Seção 1.2, admite apenas soluções onde um item está completamente incluso na mochila, ou o item não é incluído. A variante, conhecida como Problema da Mochila Fracionária, é análoga ao problema da mochila binária, porém, permite que se tome qualquer quantidade fracionária de um item.

Problema da Mochila Fracionária (PMF): Dados conjunto de itens $\{1,\ldots,n\}$, cada item i com peso $p_i \geq 0$ e valor $v_i \geq 0$, ambos inteiros, e inteiro B, encontrar frações $x_i \in [0,1]$, para $i = 1,\ldots,n$, tal que $\sum_{i=1}^{n} p_i x_i \leq B$ e $\sum_{i=1}^{n} v_i x_i$ é máximo.

Note que a troca da restrição $x_i \in [0,1]$ pela restrição $x_i \in \{0,1\}$ nos dá exatamente o problema da mochila binária. Apesar de muito parecidos e ambos satisfazerem a propriedade de subestrutura ótima, os problemas da Mochila Binária e da Mochila Fracionária apresentam dificuldades computacionais bem distintas.

Considere o desenvolvimento de algoritmos gulosos para os dois problemas. Em ambos, a estratégia gulosa natural indica uma ordenação dos itens por seu peso relativo, que é o valor de um item por unidade de peso. Assim, considere que ambos já recebem itens em ordem que satisfaz $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \geq \frac{v_n}{p_n}$. Os algoritmos gulosos para estes dois problemas poderiam iterativamente colocar um item (ou a maior parte deste item, no caso fracionário) de maior valor relativo possível de ser incluído na solução. Caso um item (ou parte do item) não possa ser inserida, este é descartado. O algoritmo seguinte é um detalhamento desta idéia para o problema da mochila fracionária.

Algoritmo: Mochila-Fracionária-Guloso(B,p,v,n), onde $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \geq \frac{v_n}{p_n}$

Saída : Frações $x_i \in [0,1]$ para $i = 1,\ldots,n$.

```
1 R ← B
2 para i = 1 até n faça
3     x_i ← min{R/p_i, 1}
4     R ← R − p_i x_i
5 devolva (x_1, ..., x_n)
```

O algoritmo insere os itens na ordem dos que tem maior valor por unidade de peso primeiro. A cada iteração, o algoritmo procura colocar o máximo do item corrente e só então segue para o próximo item. Note que sempre que um item não couber em sua totalidade, a mochila é preenchida completamente e portanto todos os itens seguintes não serão colocados na solução. Assim, a propriedade da escolha gulosa é claramente

satisfeita. O próximo teorema, cuja prova fica como exercício, mostra que este algoritmo obtém uma solução ótima para o problema.

Teorema 1.4.6 *O Algoritmo Mochila-Fracionária-Guloso obtém uma solução ótima para o Problema da Mochila Fracionária.*

Agora, considere o Problema da Mochila Binária. O algoritmo guloso proposto anteriormente é descrito no Algoritmo Mochila-Guloso.

Algoritmo: Mochila-Guloso(B,p,v,n), onde $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \geq \frac{v_n}{p_n}$

Saída : Conjunto de itens S.

```
1 R ← B
2 S ← ∅
3 para i = 1 até n faça
4     se p_i ≤ R então
5         S ← S ∪ {i}
6         R ← R − p_i
7 devolva S
```

Este algoritmo pode produzir soluções com valor arbitrariamente distantes do valor das soluções ótimas. O exemplo a seguir, mostra uma construção onde isto ocorre. Neste caso, a propriedade da escolha gulosa não é satisfeita.

Exemplo 1.4.1 *Considere, por exemplo, uma entrada $I = (B,p,v,n)$ onde $B = 1$, $n = 2$, $v = (2\varepsilon,1)$ e $p = (\varepsilon,1)$, para $0 < \varepsilon < 1$. O peso relativo do primeiro item é $v_1/p_1 = 2\varepsilon/\varepsilon = 2$, enquanto o peso relativo do segundo é $v_2/p_2 = 1/1 = 1$. Ao colocar o primeiro item, não há espaço suficiente para o segundo, e temos uma solução gerada pelo algoritmo guloso de valor 2ε. A solução ótima é pegar apenas o segundo item, de valor 1. Portanto, a relação entre a solução ótima e a gerada pelo algoritmo é de $\frac{1}{2\varepsilon}$, que pode se tornar tão grande quanto se queira, bastando para isso usar ε bem pequeno.*

É interessante notar que apesar deste algoritmo guloso não fornecer nenhuma garantia sobre a solução gerada, é possível fazer uma pequena alteração para que garanta soluções com valor de pelo menos 50% do valor de uma solução ótima.

Exemplo 1.4.2 *Seja t o primeiro item que o Algoritmo Mochila-Guloso não pode colocar no espaço remanescente da mochila e S^* uma solução ótima para uma entrada. Tome $S_1 = \{1,\ldots,t-1\}$ e $S_2 = \{t\}$. É fácil ver que $\sum_{i=1}^{t} v_i \geq \sum_{i \in S^*} v_i$ é um limitante superior para o valor de uma solução ótima. Como S_1 e S_2 são soluções, ao escolher a de maior valor, temos uma garantia de pelo menos $1/2$ do limitante superior, e portanto de uma solução ótima.*

Exemplo 1.4.3 *Usando o mesmo tipo de argumento, que o usado no exemplo anterior, pode se mostrar que o algoritmo guloso produz soluções boas quando os itens não são grandes em relação à capacidade da mochila, em particular, quando $p_i \leq B/m$, para um parâmetro inteiro positivo m, que mede o quão pequeno são os itens em relação ao recipiente. Quanto maior o valor de m, menor são os itens. Neste caso, o algoritmo guloso obtém soluções com valor pelo menos $\frac{m-1}{m}$ do valor da solução ótima.*

1.5. Programação Dinâmica

O projeto de algoritmos por programação dinâmica também é um método indutivo. Com isso, poderiam ser resolvidos também pelo projeto por divisão e conquista (abordagem *top-down*). Porém, quando há subproblemas que se repetem na construção de um ou mais problemas, esta abordagem, sem maiores cuidados, pode levar a um algoritmo com complexidade de tempo muito maior que o necessário. Isto ocorre pela forma como é feita a resolução de cada um dos subproblemas, de maneira independente e sem conhecimento dos subproblemas já resolvidos. Um exemplo onde é possível perceber a velocidade como os subproblemas se multiplicam, é no cálculo da função de Fibonacci, quando esta é calculada por um algoritmo recursivo.

Exemplo 1.5.1 *A função de Fibonacci $F(n)$ é definida nos inteiros positivos como $F(0) = 0, \quad F(1) = 1 \quad e \quad F(n) = F(n-1) + F(n-2)$ se $n \geq 2$. Caso $F(n)$ seja implementado de maneira recursiva, o valor de $F(\cdot)$ será calculado repetidas vezes para várias entradas. A Figura 1.3 mostra os vários subproblemas envolvidos no cálculo de $F(4)$. Nota-se que $F(0)$ e $F(2)$ são resolvidos duas vezes cada um e $F(1)$ é resolvido por três vezes. A quantidade de operações realizadas para o cálculo do número*

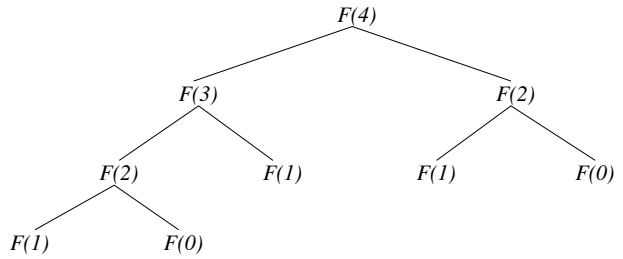


Figura 1.3. Resolução de $F(4)$ faz $F(0)$, $F(1)$ e $F(2)$ serem resolvidos mais de uma vez.

de Fibonacci pela abordagem top-down pode ser obtida resolvendo-se a recorrência de Fibonacci, que pelo Exemplo 1.3.4, mostra um crescimento exponencial para esta função. Mais precisamente, $F(n) = \Theta((1+\sqrt{5})/2)^n$.

A estratégia usada no projeto por programação dinâmica é armazenar as soluções dos subproblemas já resolvidos em uma tabela. Caso um destes problemas repetidos apareça como subproblema de outro, obtém-se o resultado já armazenado, evitando

recalcular o mesmo subproblema. Os valores da função de Fibonacci, por exemplo, podem ser calculados do menor para o maior (abordagem *bottom-up*) e seus valores podem ser armazenados em um vetor. Como o cálculo de cada posição é feito em tempo constante, este processo obtém o valor de $F(n)$ em $\Theta(n)$ passos.

Em particular, o cálculo de um termo da sequência de Fibonacci depende apenas dos dois termos anteriores. Com isso, basta manter a cada iteração os dois maiores termos, pois após calcular o próximo termo, o menor termo não é mais usado, permitindo que o cálculo de $F(n)$ seja feito utilizando quantidade constante de espaço.

Os problemas para os quais a programação dinâmica se aplica têm, em geral, as seguintes propriedades.

Subestrutura ótima: Quando uma solução ótima de um problema deve conter soluções ótimas para seus subproblemas.

Subproblemas repetidos: presença de subproblemas que se repetem na resolução de diferentes problemas ou subproblemas.

Alguns passos importantes no desenvolvimento de um algoritmo por programação dinâmica são os seguintes [Cormen et al. 2009]: *(i)* Caracterização da estrutura de uma solução ótima. *(ii)* Definição recursiva do valor de uma solução ótima. *(iii)* Resolução dos problemas de maneira *bottom-up*. *(iv)* Construção da solução ótima.

As próximas subseções apresentam dois exemplos usando essa técnica, um para o Problema da Mochila Binária e outro para o Problema da Árvore Binária de Busca Ótima.

1.5.1. Algoritmo de Programação Dinâmica para o Problema da Mochila Binária

Esta seção apresenta um algoritmo por programação dinâmica para o Problema da Mochila Binária, definido na Seção 1.2.

Como pode ser visto, o problema da mochila binária contém a propriedade de subestrutura ótima. Considere uma solução ótima S^* para um problema da mochila binária de capacidade B. Ao remover um subconjunto $T \subseteq S^*$ da solução ótima e também o espaco no recipiente ocupado por estes itens, temos que $S^* \setminus T$ é uma solução ótima para o subproblema residual, formado pelos itens não pertencentes à T e com mochila de capacidade $B - \sum_{i \in T} p_i$.

Para simplificar o uso da subestrutura ótima, podemos considerar apenas um item e recair em problemas cujas soluções contém ou não tal item. Dada entrada $I = (B, p, v, n)$ e uma solução ótima S^* para I considere a pertinência do item n em S^*. Ocorre exatamente uma das situações: ou $n \in S^*$ ou $n \notin S^*$. Caso $n \in S^*$, recai-se no subproblema de completar o espaço restante da mochila da melhor maneira possível usando os demais $n-1$ itens. Caso $n \notin S^*$, a mochila também deve ser preenchida na otimalidade com os demais $n-1$ itens. Assim, de um problema com n itens recai-se em dois de $n-1$ itens, sendo que um simples algoritmo recursivo teria complexidade

de tempo dado pela recorrência $T(n) = 2T(n-1) + O(1)$, que resolvendo se obtém $T(n) \in \Theta(2^n)$. Tal complexidade de tempo não é muito melhor que o obtido por um algoritmo força-bruta. Para obter um algoritmo com complexidade de tempo melhor, pode-se explorar o fato que há subproblemas repetidos, que seriam resolvidos várias vezes em uma abordagem *top-down*, e do fato que na prática o tamanho da mochila não é muito grande.

O seguinte exemplo apresenta uma entrada para o problema da mochila binária que recai em subproblemas repetidos. Além disso, nota-se uma melhor definição dos subproblemas.

Exemplo 1.5.2 *Para entender a estrutura do Problema da Mochila Binária, considere a estratégia top-down aplicada a um problema com $B = 8$, e pesos $p = (1,2,1,2,1,1,\ldots)$. Os valores de cada item não foram colocados para não sobrecarregar a figura. Dado uma solução ótima, um item qualquer pode ou não pertencer a esta solução. Assim, um algoritmo usando a abordagem top-down pode considerar os dois casos, quando o item está ou não na solução, devolvendo uma que possuir o maior valor. A Figura 1.4 apresenta uma parte dos nós da árvore de ramificação de um possível algoritmo usando esta estratégia. Todos os subproblemas de um mesmo nível escolhem o mesmo item para fazer sua ramificação. Com isso, os subproblemas de um mesmo nível tem o mesmo conjunto de itens. Quando dois nós tiverem a mesma capacidade para a mochila residual, os nós tratam do mesmo problema. Disso, temos que o par de nós (a) e (d) representam o mesmo problema, assim como o par (b) e (e) e o par (c) e (f).*

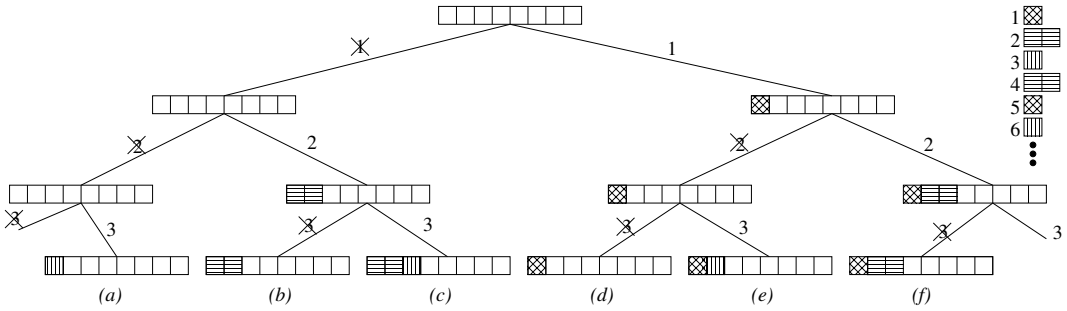


Figura 1.4. Problemas repetidos no problema da mochila: (a) e (d), (b) e (e) e (c) e (f).

Do exemplo anterior é possível notar que não basta ter o mesmo conjunto de itens para representar um mesmo subproblema. Para isto, é necessário que as capacidades das mochilas de cada subproblema também tenham o mesmo tamanho. Isto permite dizer melhor sobre o formato da entrada de cada subproblema.

Para a definição recursiva do problema, seja $I = (B, p, v, n)$ uma entrada para o problema da mochila binária. Considere o item n, que pode ou não estar em uma solução

ótima. Disso temos dois subproblemas, dados pelas entradas $I' = (B, p, v, n-1)$, caso o item n não esteja na solução ótima sendo construida ou $I'' = (B - p_n, p, v, n-1)$, quando o item n participa da solução. Os casos base se resumem a entradas onde não há itens ou a mochila não tem capacidade suficiente para novos itens.

Os subproblemas possíveis serão as entradas na forma (b, p, v, m) tal que $1 \leq b < B$ e $1 \leq m < n$. Portanto, o número de subproblemas que possuem pelo menos um item e tem capacidade não nula está limitado a nB. A definição recursiva do valor da solução ótima é a seguinte:

$$\vartheta(m,b) = \begin{cases} 0 & \text{se } m = 0 \text{ ou } b = 0, \\ \vartheta(m-1,b) & \text{se } p_m > b, \\ \max\{\vartheta(m-1,b)\,,\, \vartheta(m-1,b-p_m) + v_m\} & \text{se } p_m \leq b. \end{cases}$$

Com isso, a estratégia usada no algoritmo por programação dinâmica será resolver os subproblemas menores primeiro.

Itens	#Itens\Mochila	0	1	2	...	$b - p_m$	...	b	...	B
$\{1,\ldots,n\}$	n	0			...		...		...	$\vartheta(n,B)$
$\vdots$	$\vdots$				...		...		...	
$\{1,\ldots,m\}$	m	0			...			$\vartheta(m,b)$	...	
$\{1,\ldots,m-1\}$	$m-1$	0			...	$\vartheta(m-1,b-p_m)$	...	$\vartheta(m-1,b)$	...	
$\vdots$	$\vdots$				...		...		...	
$\{1,2\}$	2	0			...		...		...	
$\{1\}$	1	0			...		...		...	
$\emptyset$	0	0	0	0	...	0	...	0	...	0

Tabela 1.2. Tabela de programação dinâmica para Mochila Binária preenchido bottom-up.

O Algoritmo Mochila-Tabela devolve a tabela dos valores dos subproblemas de (B, p, v, n), mas não devolve o conjunto de itens que fazem parte da solução. Para encontrar os itens que compõem a solução ótima, partimos da posição (n, B), onde se encontra o valor $\vartheta(n,B)$ da solução ótima e percorremos os itens na ordem reversa, decidindo a pertinência de cada item.

Na composição da solução de valor $\vartheta(n,B)$, o item n pode ou não ser utilizado. Se o item n não foi utilizado, o valor desta solução é igual a $\vartheta(n-1,B)$ e a construção da solução continua a partir da célula $(n-1,B)$. Caso contrário, a solução usa o item n e a busca segue a partir da célula $(n-1,B-p_n)$. Isto é, podemos verificar se $\vartheta[n,B] = \vartheta[n-1,B]$, e neste caso o item n não é usado. Caso contrário, o item n faz parte da solução ótima. Este processo é repetido até o primeiro item. O Algoritmo Mochila-PD descreve este processo.

Algoritmo: Mochila-Tabela(B,p,v,n).

Saída : Tabela dos subproblemas do problema da mochila binária.

para $b \leftarrow 0$ *até* B **faça** $\vartheta[0,b] \leftarrow 0$
para $m \leftarrow 0$ *até* n **faça** $\vartheta[m,0] \leftarrow 0$
para $m \leftarrow 1$ *até* n **faça**
    para $b \leftarrow 1$ *até* B **faça**
        $\vartheta[m,b] \leftarrow \vartheta[m-1,b]$
        se $p_m \leq b \quad e \quad v_m + \vartheta[m-1,b-p_m] > \vartheta[m,b]$ **então**
            $\vartheta[m,b] \leftarrow v_m + \vartheta[m-1,b-p_m]$
devolva ϑ

Algoritmo: Mochila-PD(B,p,v,n).

Subrotina: Algoritmo Mochila-Tabela.

Saída : Solução ótima para o problema da mochila binária.

$\vartheta \leftarrow$Mochila-Tabela(B,p,v,n)
$S \leftarrow \emptyset$
$b \leftarrow B$
para $m \leftarrow n$ *decrescendo até* 1 **faça**
    se $\vartheta[m,b] \neq \vartheta[m-1,b]$ **então**
        $S \leftarrow S \cup \{m\}$
        $b \leftarrow b - p_m$
devolva $S, \vartheta(n,B)$

O próximo teorema consolida o resultado do Algoritmo Mochila-PD, cuja prova de corretuda ficará como exercício.

Teorema 1.5.1 *Dados entrada $I = (B,p,v,n)$ para o problema da mochila binária, o conjunto S, de valor $\vartheta(n,B)$ devolvido pelo Algoritmo Mochila-PD é uma solução ótima para a entrada I.*

A complexidade do Algoritmo Mochila-PD é claramente $O(nB)$. Trata-se de uma complexidade de tempo pseudo-polinomial, quando temos uma complexidade de tempo polinômial no tamanho da entrada e na magnitude do maior valor que ocorre na entrada. Quando B não é muito grande, é possível ter um algoritmo bastante eficaz na prática.

1.5.2. Algoritmo de Programação Dinâmica para o Problema da Árvore Binária de Busca Ótima

Uma árvore binária de busca, é uma estrutura de dados dinâmica que permite armazenar, fazer buscas, guardar a ordem e fazer operações básicas de manutenção de uma base de

dados, como remover, inserir e atualizar. Para tanto, seus elementos devem ter uma relação de ordem total. O conjuto dos elementos será denotado por $\mathcal{D}$.

Definida de maneira recursiva, uma *árvore binária T* é definida como: *(i)* Uma árvore vazia, representada por nil e não possui nenhum elemento nem subárvore; ou *(ii)* Um nó contendo um elemento $T.x$ e duas subárvores binárias, denotadas por subárvore esquerda de T, dada por T.esq, e subárvore direita de T, dada por T.dir. As duas subárvores são nós filhos da árvore T. A lista dos elementos representados por uma árvore binária T, denotado por $\mathcal{L}(T)$ é dado pela lista vazia, se $T = \text{nil}$ ou a lista $\mathcal{L}(T.\text{esq})\|(T.x)\|\mathcal{L}(T.\text{dir})$, caso T tenha pelo menos um elemento. Um nó também pode armazenar outras informações, porém, nos restrigiremos apenas às informações em $\mathcal{D}$. Uma *árvore binária de busca* é uma árvore binária definida sobre elementos de $\mathcal{D}$ satisfazendo as seguintes propriedades: *(a)* $\mathcal{D}$ é um conjunto com ordem total; *(b)* $T.x > e$, para todo $e \in \mathcal{L}(T.\text{esq})$; *(c)* $T.x < d$, para todo $d \in \mathcal{L}(T.\text{dir})$. Note que nesta definição não será permitido armazenar mais que um elemento de mesmo valor, caso necessário, estes deverão ser armazenados no mesmo nó.

Aplicações da árvore de busca ótima ocorrem na predição de palavras, busca em dicionários, verificadores de ortografia, tradução de textos, entre outros.

Considere, de maneira simplificada, o problema de se representar um dicionário de palavras. Neste caso, é importante que exista a possibilidade de mostrar o significado de uma palavra, mas também poder percorrer as palavras na ordem existente. Há estruturas de dados, como árvores 2-3 ou AVL que não só representam uma árvore de busca, mas também mantém a altura da árvore em ordem logarítmica no número de elementos, agilizando buscas. Porém, cada palavra em uma língua possui uma frequência em que é usada ou consultada. Com isso, é possível construir uma árvore binária de busca que leve esta frequência em consideração para produzir uma árvore que minimiza o número total de acessos aos seus nós, diminuindo com isso o tempo total das consultas.

Problema da Árvore Binária de Busca Ótima (PABO): Dados elementos $\mathcal{D} = (e_1 < e_2 < \ldots < e_n)$, onde cada elemento $e \in \mathcal{D}$ é consultado $f(e)$ vezes, construa uma árvore binária de busca, tal que o total de nós consultados é mínimo.

Exemplo 1.5.3 *Considere quatro chaves: $A < B < C < D$ e suas frequências $f(A) = 45$, $f(B) = 25$, $f(C) = 18$ e $f(D) = 12$. Há 19 árvores de busca binária válidas para estes quatro elementos. Na Figura 1.5 temos algumas destas, sendo que a primeira tem o menor custo de todos. É possível notar que há árvores com custos bem diversos.*

Definicao 1.5.1 *Se T é uma árvore binária de busca e v é um vértice de T, denotamos por $T(v)$ a subárvore enraizada em v contendo todos os vértices abaixo de v.*

Exemplo 1.5.4 *Para entender melhor a estrutura das soluções ótimas, considere uma árvore de busca contendo os elementos em $\{A,B,C,D\}$, onde $A < B < C < D$, e T é*

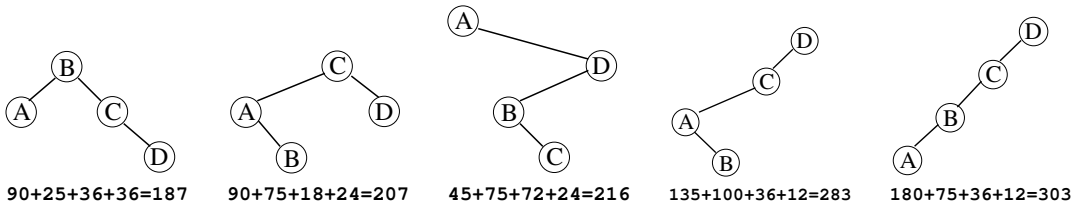


Figura 1.5. Nós acessados na primeira árvore: $2f(A)+f(B)+2f(C)+3f(D)=90+25+36+36=187$

uma árvore com três elementos, sendo o elemento $T.x = B$ como raiz e à sua esquerda, uma subárvore contendo apenas o elemento A e a sua direita uma subárvore contendo apenas o elemento D. Apesar de T representar uma árvore binária de busca, se no final todos os quatro elementos devem ser armazenados, então a árvore T não pode representar um subproblema válido. Dentre os elementos em $\mathcal{L}(T)$ falta o elemento C e em T há pelo menos um elemento maior que C e pelo menos um elemento menor que C. Com isso, se T fosse um subproblema, este deveria estar totalmente a esquerda de C ou totalmente a direita de C, porém isso não é possível, uma vez que T contém tanto elementos maiores como menores que C.

Portanto, as únicas subárvores de busca T, que representam subproblemas válidos, são as que definem $\mathcal{L}(T)$ como uma subsequência de elementos consecutivos da lista de elementos original ordenada.

Seja $\mathcal{D} = (e_1, e_2, \cdots, e_n)$, onde $e_{k-1} < e_k$ para $k = 2, \ldots, n$, a lista de elementos para o qual se deve construir uma árvore binária de busca. Do exemplo acima, conclui-se que uma árvore T só será um subproblema a ser considerado se existir i e j com $i \leq j$, tal que $\mathcal{L}(T) = (e_i, \ldots, e_j)$.

Esta observação mostra que o número de subproblemas válidos distintos não é grande. De fato, é quadrático no número de termos total, uma vez que o número de subconjuntos de tamanho pelo menos dois em $\{e_1, e_2, \ldots, e_n\}$ é $\binom{n}{2}$ e o número de subconjuntos de tamanho um é n. A estratégia será usar o projeto por programação dinâmica e armazenar os subproblemas de maneira que se possa computar um subproblema sabendo-se a solução dos subproblemas menores.

Denote por $T^*(e_i, \ldots, e_j)$ uma árvore ótima dos elementos $(e_i, \ldots, e_j)$. Se $i = j$, temos apenas um elemento e apenas uma árvore de busca binária pode representá-lo, e portanto é ótima. Caso contrário, sabe-se que um dos elementos em $(e_i, \ldots, e_j)$ deve ser a raiz de $T^*(e_i, \ldots, e_j)$, e para saber qual pode ser, testamos todos os elementos e_k como raiz, para $k = i, \ldots, j$.

Considere um dos testes quando e_k é raiz, onde $i \leq k \leq j$, e temos dois subproblemas filhos: $T^*(e_i, \ldots, e_{k-1})$ e $T^*(e_{k+1}, \ldots, e_j)$, como na Figura 1.6. Como cada um dos subproblemas filhos é menor, estes já foram computados e basta recuperar a solução ótima armazenada para cada um destes subproblemas. Porém, note que o valor

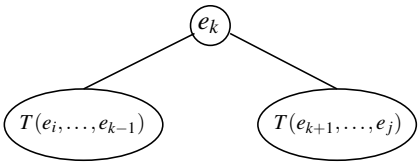


Figura 1.6. Divisão de um problema em subproblema.

armazenado nestes subproblemas não considera que cada consulta a um dos elementos nos subproblemas filhos, teve que consultar o nó e_k antes de chegar à árvore do subproblema. Portanto, deve-se acrescentar $\sum_{t=i}^{k-1} f(e_t)$ acessos ao nó e_k devido aos elementos do subproblema da esquerda e $\sum_{t=k+1}^{j} f(e_t)$ devido aos acessos dos elementos da direita. Além disso, há também os acessos ao nó e_k, igual a $f(e_k)$. Portanto, o número de acessos ao nó e_k é igual a $\sum_{t=i}^{k-1} f(e_t) + f(e_k) + \sum_{t=k+1}^{j} f(e_t)$. Agregando as somas, temos $\sum_{t=i}^{j} f(e_t)$ acessos ao nó e_k além dos acessos em cada subárvore/subproblema. Assim, se e_k é raiz de $T(e_i,\ldots,e_j)$, temos um total de acessos para esta possibilidade igual ao obtido em $T^*(e_i,\ldots,e_{k-1}) + T^*(e_{k+1},\ldots,e_j)$. Por fim, basta testar cada possibilidade de e_k ser raiz de uma subárvore com os elementos $\{e_i,\ldots,e_j\}$ e escolher a melhor configuração.

Sabendo-se a estrutura de uma solução ótima, pode-se escrevê-la recursivamente. Como os subproblemas são dados por subsequências de elementos consecutivos, seja $A[i,j]$ a árvore de busca ótima do conjunto $\{e_i,\ldots,e_j\}$.

$$A[i,j] = \begin{cases} 0 & \text{se } i > j, \\ f(e_i) & \text{se } i = j, \\ \min\left\{A[i,k-1] + A[k+1,j] + \sum_{t=i}^{j} f(e_t) :\ k \in \{i,\ldots,j\}\right\} & \text{se } i < j. \end{cases}$$

Para preencher a tabela $A[i,j]$ deve se tomar o cuidado para preencher na ordem dos menores para os maiores. Para isso, pode-se resolver os subproblemas de tamanho 1 (base), em seguida os subproblemas de tamanho 2, seguindo assim até o maior subproblema contendo n elementos.

Teorema 1.5.2 *O Algoritmo Árvore-Busca encontra o valor da árvore de busca ótima em tempo* $O(n^3)$.

A prova de corretude deste algoritmo bem como a construção da solução ficam como exercício.

1.6. Programação Linear e Inteira

Muitos problemas de otimização podem ser formulados matematicamente de modo que o objetivo se torne maximizar ou minimizar uma função linear definida sobre um conjunto de variáveis, as quais devem satisfazer um sistema de equações e inequações lineares. Neste caso, o sistema linear modela as restrições do problema que, sob esta forma,

Algoritmo: Árvore-Busca$(e_1,\ldots,e_n,f)$ onde $e_1 < e_2 < \cdots < e_n$

Saída : Peso da melhor árvore de busca.

```
1 para i ← 1 até n faça A[i,i] ← f(e_i)
2 para t ← 2 até n faça
3     para i ← 1 até n−t+1 faça
4         j ← i+t−1
5         A[i,j] ← min{A[i,k−1]+A[k+1,j] : k ∈ {i+1,...,j−1};
6                       A[i+1,j]; A[i,j−1]} + Σ_{t=i}^{j} f(e_t)
7 devolva A[1,n]
```

define um problema de *programação linear*, no qual a função ser otimizada recebe o nome de *função objetivo*. Caso, adicionalmente, seja exigido que algumas ou todas as variáveis assumam apenas valores inteiros, tem-se um problema de *programação linear inteira*. A Programação Linear (PL) e a Programação Linear Inteira (PLI) constituem-se em poderosas ferramentas para a resolução de problemas de Otimização Combinatória. O intuito desta seção não é fazer uma apresentação exaustiva destes temas, mas sim destacar alguns aspectos relevantes destas técnicas para a Otimização Combinatória. Em particular, será enfatizada a modelagem de problemas combinatórios usando a PL e a PLI. Inicialmente, exemplifica-se o uso de PL na modelagem de um problema de otimização.

Exemplo 1.6.1 *Suponha que uma companhia de geração de energia elétrica está planejando instalar uma usina termoelétrica em uma localidade. A maior dificuldade da empresa está em atender às exigências impostas pelas leis de proteção ambiental. Uma delas refere-se aos poluentes emitidos na atmosfera. O carvão necessário para aquecer as caldeiras deverá ser fornecido por três minas. As propriedades dos diferentes tipos de carvão produzidos em cada uma das minas estão indicadas na tabela abaixo. Os valores mostrados são relativos à queima de uma tonelada de carvão.*

Mina	*Enxofre (em* ppm*)*	*Poeira de Carvão (em* Kg*)*	*Vapor produzido (em* Kg*)*
1	*1100*	*1.7*	*24000*
2	*3500*	*3.2*	*36000*
3	*1300*	*2.4*	*28000*

Os 3 tipos de carvão podem ser misturados em qualquer proporção. As emissões de poluentes e de vapor de uma mistura qualquer são proporcionais aos valores indicados na tabela. As exigências ambientais requerem que: (i) *para cada tonelada de carvão queimada a quantidade de enxofre não deve ser superior a 2.500* ppm*, e* (ii) *para cada tonelada de carvão queimada a quantidade de poeira de carvão não deve ser superior*

a 2.8 kg. *A companhia quer encontrar a proporção ótima da mistura do carvão de modo a maximizar a quantidade de vapor, portanto de energia, que é produzida com a queima de uma tonelada da mistura.*

Para modelar o problema acima como um PL, pode-se definir as variáveis x_i, $i = 1,2,3$, como sendo a proporção do carvão da mina i que será usado na mistura. Deste modo, o valor a ser maximizado pode ser descrito pela função linear $z = 24000x_1 + 36000x_2 + 28000x_3$, que corresponde a energia produzida pela queima de uma tonelada da mistura de carvão nas proporções x_1, x_2 e x_3. De forma análoga, a limitação na quantidade de enxofre produzida pela queima de uma tonelada da mistura é descrita pela inequação $1100x_1 + 3500x_2 + 1300x_3 \leq 2500$, enquanto que a limitação da quantidade de poeira de carvão é dada por $1.7x_1 + 3.2x_2 + 2.4x_3 \leq 2.8$. Restam ainda duas restrições que precisam ser consideradas. A primeira delas diz respeito à proporção da mistura, a qual se supõe aqui não conter impurezas. Com isto, a seguinte igualdade precisa ser satisfeita: $x_1 + x_2 + x_3 = 1.0$. Para completar, todas as proporções devem ser não-negativas. Assim, o modelo final resultante é:

$$\begin{array}{lrcrcrcrcr}
\max\ z = & 24000 & x_1 & + & 36000 & x_2 & + & 28000 & x_3 & \\
\text{sujeito a} & 1100 & x_1 & + & 3500 & x_2 & + & 1300 & x_3 & \leq & 2500, \\
& 1.7 & x_1 & + & 3.2 & x_2 & + & 2.4 & x_3 & \leq & 2.8, \\
& & x_1 & + & & x_2 & + & & x_3 & = & 1.0, \\
& & x_1, & & & x_2, & & & x_3 & \geq & 0.
\end{array}$$

De um modo geral, um problema de PL assume a forma abaixo:

$$\begin{array}{lll}
\max\ z = & c_1x_1 + c_2x_2 + \ldots + c_nx_n & \\
\textit{sujeito a} & a_{11}x_1 + a_{12}x_2 + \ldots + a_{1n}x_n & \bowtie\ b_1, \\
& a_{21}x_1 + a_{22}x_2 + \ldots + a_{2n}x_n & \bowtie\ b_2, \\
& \ldots & \\
& a_{m1}x_1 + a_{m2}x_2 + \ldots + a_{mn}x_n & \bowtie\ b_m, \\
& x \in \mathbb{R}^n_+, & (3)
\end{array}$$

onde o símbolo "$\bowtie$" pode representar uma das seguintes relações: "$=$", "$\leq$" ou "$\geq$". Um modo mais compacto de representar um PL é a forma matricial dada por

$$\max\{cx\ :\ Ax = b, x \in \mathbb{R}^n_+\},$$

onde supõe-se que todas as restrições são de igualdade e as dimensões da matriz e dos vetores são: $A : m \times n$, $x : n \times 1$, $b : m \times 1$ e $c : 1 \times n$. Vale notar que, a rigor, qualquer problema de PL pode ser escrito de modo que todas restrições estejam sob a forma de igualdades ou sob a forma de desigualdades "$\leq$" (ou "$\geq$", tanto faz). Isso é importante

pois alguns resultados para PL são demonstrados supondo que todas as relações são de um mesmo tipo, o que, pelo que acaba de ser dito, não os torna mais restritivos.

Para que um problema admita uma modelagem por PL, algumas hipóteses importantes devem ser satisfeitas. São elas: *(i)* proporcionalidade: a contribuição de uma variável na função objetivo e em uma restrição é multiplicada por k se o valor da variável também for multiplicado por k. *(ii)* aditividade: as contribuições individuais das variáveis se somam na função objetivo e nas restrições e são independentes, e *(iii)* divisibilidade: as variáveis podem ser divididas em qualquer fração.

Existem algoritmos que permitem resolver problemas de PL. O primeiro a surgir, e possivelmente o mais famoso e largamente empregado, é o método SIMPLEX desenvolvido por Dantzig em 1947 (cf., [Bazaraa et al. 2009]). Do ponto de vista da Teoria da Computação, o SIMPLEX apresenta um desvantagem por não ter complexidade polinomial, embora seu desempenho na prática seja altamente satisfatório. Passaram-se décadas desde o nascimento da PL até que aparecessem os primeiros algoritmos polinomiais, começando pelo *método dos elipsóides* [Khachiyan 1979] e com os *algoritmo de pontos interiores* [Karmarkar 1984]. Na prática, o SIMPLEX, mesmo tendo uma complexidade exponencial, ainda é competitivo, muitas vezes superando, algoritmos de pontos interiores. Existem bons resolvedores comerciais e de domínio público para resolver programas lineares, o que torna atrativo e viável o uso da técnica para a resolução de problemas de Otimização Combinatória.

A descrição dos algoritmos para PL foge ao escopo do presente texto. Contudo, é instrutivo que se tenha uma ideia sobre a geometria do problema, o que pode ser ilustrado no espaço bidimensional, ou seja, quando existem apenas duas variáveis x_1 e x_2. Na Figura 1.7, o polígono hachurado corresponde às soluções do programa linear cujas restrições são $x_1 \geq 0$, $x_2 \geq 0$, $2x_1 + 6x_2 \leq 15$ e $2x_1 - 2x_2 \leq 3$. A função objetivo é dada por max $z = 2x_1 + 3x_2$, cujo vetor gradiente está representado pela seta no alto à direita da figura. Como se sabe, à medida que se desloca na direção do gradiente, o valor da função z irá aumentar. Isso pode ser visto pelas linhas tracejadas que representam as retas da forma $2x_1 + 3x_2 = z$ para diferentes valores de z (indicados na figura). Ao se deslocar a reta na direção apontada pelo vetor gradiente, vê-se que o valor de z aumenta. Em um dado momento neste deslocamento, a reta atinge o ponto extremo do polígono cujo par de coordenadas é $(3, \frac{3}{2})$. A partir daí, se a reta continuar a se mover, ela não mais interceptará uma solução do programa linear. Conclui-se, portanto, que o ponto extremo em questão é uma solução ótima do programa linear e que o valor ótimo é $z = 10\frac{1}{2}$. Conforme ilustrado por este exemplo, o conjunto de vetores que satisfazem às restrições de um PL, chamadas de *soluções viáveis*, formam um polígono ou, mais geralmente, um poliedro (para dimensões maiores que 2). O conjunto de todas estas soluções (o poliedro) é chamdo de *região de viabilidade* do PL. Além disso, como sugerido pelo exemplo, quando o valor ótimo for finito, existe pelo menos uma solução ótima que é um ponto extremo (vértice) do poliedro. Esta observação é a base do desenvolvimento

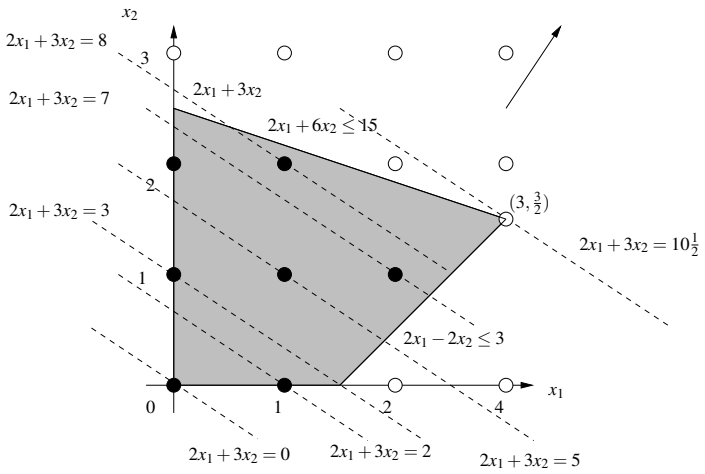


Figura 1.7. Representação Gráfica de um PL em duas variáveis.

do método SIMPLEX, o qual percorre sucessivamente vértices vizinhos da região de viabilidade, verificando se eles atendem a um critério de otimalidade.

Como visto acima, um PL pode ser resolvido em tempo polinomial. Contudo, quando se requer que uma ou mais variáveis tomem valores inteiros, a situação torna-se bem distinta. Isto porque, em sua forma geral, PLI está em $\mathbb{NP}$-difícil, o que significa que, a menos que $\mathbb{P} = \mathbb{NP}$, não será possível encontrar um algoritmo polinomial para a PLI. Portanto, para resolver um problema de PLI na prática, recorre-se a algoritmos enumerativos, como será discutido na Seção 1.8. Vale notar que a PLI fere uma das hipóteses da PL que é a *divisibilidade* das variáveis. Para ilustrar este fato, dá-se a seguir um exemplo de um problema que é formulado por um PLI.

Exemplo 1.6.2 *A administração de um hospital está montando a escala das enfermeiras que ficarão de plantão em um feriado prolongado. Estas enfermeiras serão responsáveis por procedimentos a serem realizados em pacientes que requerem cuidados especiais. O regime diário de trabalho das enfermeiras durante o feriado é tal que elas trabalham em turnos. Um período de trabalho é composto de 4 horas e um turno é composto de dois períodos de trabalho separados por uma pausa também de 4 horas, conforme exigências trabalhistas. Terminado o seu turno, uma enfermeira entra em um período de descanso de 12 horas seguidas. No total, são seis turnos cujos horários de trabalho ao longo do dia são mostrados na tabela a seguir. Nem todos os pacientes especiais passam por um procedimento em todos os períodos. Contudo, em um período qualquer, todo procedimento deve ser acompanhado o tempo todo por uma enfermeira. O número de procedimentos a serem feitos em cada período também é dado na tabela.*

Turno	*Período do dia (faixa horária)*					
	00–04	*04–08*	*08–12*	*12–16*	*16–20*	*20–24*
1	√		√			
2		√		√		
3			√		√	
4				√		√
5	√				√	
6		√				√
Procedimentos a realizar	*6*	*7*	*15*	*9*	*13*	*10*

O objetivo da administração é encontrar o número mínimo de enfermeiras que devem ficar de plantão no feriado de modo a garantir que todos os procedimentos sejam realizados.

Para modelar este problema como um PLI, usamos as variáveis x_i para denotar o número de enfermeiras que ficarão no turno i. Com isto, o número de enfermeiras que serão chamadas para o plantão é dado por $x_1+x_2+x_3+x_4+x_5+x_6$. Já a quantidade de enfermeiras no período das meia-noite às 4 da manhã será x_1+x_5 pois, para estar trabalhando neste período, uma enfermeira tem que estar sob o regime do turno 2 ou do turno 6. Raciocinando de forma análoga para os demais períodods do dia, chega-se ao seguinte modelo PLI:

$$
\begin{array}{lrcrcrcrcrcrcrcl}
\min z = & x_1 & + & x_2 & + & x_3 & & x_4 & + & & x_5 & + & & x_6 & & & \\
\text{sujeito a} & x_1 & & & & & & & & + & & x_5 & & & \geq & 6, \\
 & & & x_2 & & & & & & & & & + & x_6 & \geq & 7, \\
 & x_1 & & & + & x_3 & & & & & & & & & \geq & 15, \\
 & & & x_2 & & & + & & x_4 & & & & & & \geq & 9, \\
 & & & & & x_3 & & & & + & & x_5 & & & \geq & 13, \\
 & & & & & & & & x_4 & & & & + & x_6 & \geq & 10, \\
 & x_1, & & x_2, & & x_3, & & & x_4, & & & x_5, & & x_6 & \geq & 0, \\
 & x_1, & & x_2, & & x_3, & & & x_4, & & & x_5, & & x_6 & & \text{inteiras.}
\end{array}
$$

A última condição é a chamada *restrição de integralidade* das variáveis do problema. Claramente a restrição de integralidade destrói a hipótese de divisibilidade da PL mas é imperativa neste caso, uma vez que a quantidade de enfermeiras em um turno não pode ser medido por um valor contendo uma parte fracionária não nula.

De um modo geral, para um PLI com n variáveis representadas pelo vetor $x \in \mathbb{Z}^n_+$, o PL obtido ao se substituir a restrição de integralidade pela restrição $x \in \mathbb{R}^n_+$ é denominado de *relaxação linear* do PLI. Tipicamente, ao resolver a relaxação linear, chega-se a uma solução ótima onde existem variáveis com valor fracionário (não inteiro). O valor ótimo da relaxação será, para um problema de maximização (minimização), um limitante superior (inferior) ou *dual* para o PLI. A partir daí, percebe-se que se, por acaso, a solução ótima da relaxação contiver apenas variáveis de valor inteiro,

esta solução será ótima para o PLI. Feitas estas considerações, pode-se perguntar em que condições um PLI pode ser resolvido a partir da sua relaxação linear. Essa questão torna-se ainda mais relevante quando se analisa a complexidade computacional do problema que se está tratando. Essencialmente, se o PLI que modela um determinado problema de otimização combinatória pode ser resolvido como um PL, isto implica que o problema admite um algoritmo polinomial e, portanto, pertence à classe $\mathbb{P}$. No restante desta seção mostra-se como esta ideia pode ser aplicada à resolução de alguns problemas combinatórios conhecidos. Inicialmente, faz-se necessária a introdução de alguns conceitos adicionais.

Dada uma matriz $A : m \times n$, contendo apenas elementos 0, $+1$ ou -1, A é dita ser *totalmente unimodular* (TU) se toda submatriz quadrada de A tiver determinante 0, $+1$ ou -1. Abaixo são mostrados alguns exemplos que ilustram esta definição:

- Matrizes TU: $\begin{pmatrix} 1 & -1 & -1 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$ $\begin{pmatrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$
- Matrizes não TU: $\begin{pmatrix} 1 & -1 \\ 1 & 1 \end{pmatrix}$ $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$

A importância das matrizes TU para a PLI deriva do resultado a seguir.

Proposição 1.6.1 *Seja $S = \{x \in \mathbb{R}^n_+ : Ax = b\}$ onde A é uma matriz totalmente unimodular e b é um vetor com todas as componentes inteiras. Então os pontos extremos de S são dados por vetores com todas coordenadas inteiras. Consequentemente, se $z = \max\{cx \ : \ x \in S\}$ tem um ótimo finito para um vetor $c \in \mathbb{R}^n$, este PL tem uma solução ótima que é inteira.*

Como mencionado, se o valor ótimo de um PL é finito, existe um ponto extremo da sua região de viabilidade que é uma solução ótima. Os algoritmos de PL são capazes de encontrar este ponto extremo ótimo. Agora, perceba que se o PL está associado à relaxação linear de um PLI particular, então, ao resolvê-lo, na verdade, se está resolvendo o PLI. Porém, isto significa que o problema de PLI em questão pode ser solucionado em tempo polinomial !

Do ponto de vista prático não faz muito sentido executar um algoritmo para identificar se a matriz A correspondente às restrições de um PLI genérico é TU ou não. Se ela for TU, o PLI será computado em tempo polinomial através de sua relaxação, porém isso poderia ser feito imediatamente, sem a necessidade de responder explicitamente se A é TU ou não. Mas, se A não for TU, geralmente algum algoritmo de complexidade exponencial terá que ser utilizado para achar uma solução ótima do PLI. Como será

visto na Seção 1.8, de qualquer modo estes algoritmos já precisam resolver a relaxação linear, independente da matriz A ser TU ou não. Contudo, do ponto de vista teórico, o conhecimento de que a matriz é TU é muito relevante. Além de prover de imediato uma forma prática de resolver o problema através dos algoritmos de PL, o que raramente será a forma mais eficiente de resolução, esta informação é uma prova de que o problema sendo investigado esta na classe $\mathbb{P}$. Antes de prosseguir com exemplos em que esta situação ocorre, faz-se algumas considerações.

São conhecidas condições necessárias e suficientes para uma dada matriz ser TU. Contudo, neste texto apenas as condições necessárias expressas no resultado a seguir serão utilizadas para exemplificar situações em que um PLI pode ser resolvido por um algoritmo de PL. A importância deste resultado ficará evidente mais adiante.

Proposição 1.6.2 *Seja A uma matriz com todos elementos em $\{0,-1,+1\}$. Suponha que A contém no máximo dois elementos não-nulos por coluna e que existam dois subconjuntos disjuntos M_1 e M_2 do conjunto M dos índices das linhas de A tal que, $M_1 \cup M_2 = M$ e, para toda coluna j contendo dois coeficientes não-nulos, tem-se que $\sum_{i\in M_1} a_{ij} - \sum_{i\in M_2} a_{ij} = 0$. Então, A é TU.*

Além destas condições necessárias as propriedades a seguir também podem ser facilmente verificadas.

Proposição 1.6.3 *As seguintes afirmações são equivalentes:* (i) *A é TU;* (ii) *A^T (matriz transposta de A) é TU;* (iii) $\begin{pmatrix} A \\ I \end{pmatrix}$, *ou seja, a matriz A acrescida das linhas da matriz identidade I, é TU.*

Pela Proposição 1.6.2, pode-se verficar que a matriz A de incidência vértice-arco de um grafo direcionado $G = (V,E)$ é TU. Suponha que $|V| = n$ e $|E| = m$ de modo que A tenha dimensão $n \times m$. Dado um arco $e = (i, j)$, os elementos da e-ésima coluna de A são todos nulos, exceto aqueles correspondentes às linhas i e j cujos valores são $+1$ e -1, respectivamente. Portanto, a matriz só tem elementos 0, $+1$ e -1, uma condição necessária para ser TU, e cada coluna tem exatamente dois elementos não nulos como requerido na hipótese da Proposição 1.6.2. Agora, sendo $M = \{1,\ldots,n\}$ os índices das linhas, pode-se fazer $M_1 = M$ e $M_2 = \emptyset$. Pela definição da matriz e dos conjuntos M_1 e M_2, para toda coluna e correspondente a um arco em E, tem-se que $\sum_{i\in M_1} a_{ie} = \sum_{i\in M_2} a_{ie} = 0$. Logo $\sum_{i\in M_1} a_{ie} - \sum_{i\in M_2} a_{ie} = 0$ para toda coluna e e, sendo assim, A é TU.

Sabendo que a matriz de incidência vértice-arco de um grafo direcionado é TU, pode-se mostrar que o *Problema de Fluxo de Custo Mínimo* (PFCM) em uma rede com dados de entrada inteiros está na classe $\mathbb{P}$. Inicialmente, define-se o PFCM formalmente.

Problema do Fluxo de Custo Mínimo (PFCM):
Entrada: grafo direcionado (rede) $G = (V,E)$; demandas b_i para todo $i \in V$, satisfa-

zendo $\sum_{i \in V} b_i = 0$, se b_i for positivo (negativo) esta demanda será a quantidade de fluxo injetada (retirada) da rede no vértice i, enquanto que b_i nulo significa que i é apenas um vértice de passagem para o fluxo; custos c_{ij} por unidade de fluxo que passa no arco (i, j), para todo arco em E; capacidades h_{ij} que limitam o fluxo máximo que pode passar no arco (i, j), para todo arco em E;

Solução: deve dizer a quantidade de fluxo que passa em cada arco de modo a escoar todo fluxo injetado na rede nos vértices i com $b_i > 0$ (fontes) para que este chegue nos vértices onde haja retirada de fluxo (sorvedouros), ou seja, aqueles com $b_i < 0$. A quantidade de fluxo em cada arco é não-negativa e não pode ser superior a sua capacidade.

Objetivo: minimizar o custo total do fluxo que passa pelos arcos da rede.

Denotando por x_{ik} o fluxo que passa pelo arco (i,k) e por $V^+(i)$ ($V^-(i)$) os vértices j em $V \setminus \{i\}$ para os quais o arco (i, j) ((j,i)) está em E, o problema pode ser modelado como um PL da seguinte forma:

$$\begin{aligned}
\min \quad & z = \sum_{(i,j) \in E} c_{ij} x_{ij} & & \\
\textit{sujeito a} \quad & \sum_{k \in V^+(i)} x_{ik} - \sum_{k \in V^-(i)} x_{ki} = b_i, & \forall i \in V, & \quad (4) \\
& x_{ij} \leq h_{ij}, & \forall (i,j) \in E, & \quad (5) \\
& x_{ij} \geq 0, & \forall (i,j) \in E. & \quad (6)
\end{aligned}$$

As equações (4) são chamadas de *restrições de conservação de fluxo* nos vértices e impõem que as quantidades totais de fluxo que saem e que chegam de qualquer vértice da rede tem que ser iguais. Note que o fluxo injetado (retirado) em um vértice i com $b_i > 0$ ($b_i < 0$) entra na (sai da) rede por este vértice. As restrições (5) apenas limitam o fluxo máximo que passa em cada arco da rede, enquanto as restrições (6) dizem que este mesmo fluxo não pode ser negativo.

Observe que a submatriz formada apenas pelos coeficientes das restrições (4) corresponde exatamente à matriz de incidência vértice-arco do grafo (direcionado) G. Como visto anteriormente, esta matriz é TU. Ao anexar a esta matriz as linhas correspondentes aos coeficientes da submatriz formada pelas restrições (5), o que se faz é estender a matriz com as linhas de uma matriz identidade de ordem $|E|$. Pelo item *(ii)* da Proposição 1.6.3, tem-se que a matriz de restrições do PFCM é TU. Assim, supondo que todos os $b_i's$ e h_{ij}'s sejam inteiros, pela Proposição 1.6.1, sabemos que a solução do problema linear que modela o PFCM é inteira. De acordo com a discussão anterior, fica provado que o PFCM com demandas e capacidades inteiras pode ser resolvido em tempo polinomial.

Este último resultado tem sua relevância ampliada quando se percebe que importantes problemas combinatórios são casos particulares do PFCM. Este é o caso do *problema do caminho mais curto* entre um vértice s e um vértice t de um grafo direcionado, ou ainda, o *problema do fluxo máximo* entre os vértices s e t neste mesmo grafo.

A seguir discute-se como estes problemas são modelados pelo PFCM. Antes porém, ambos são definidos formalmente.

Dado um grafo direcionado $G = (V, E)$, dois vértices distintos s e t de V e uma função $l : E \to \mathbb{R}_+$ que associa comprimentos aos arcos de E, deseja-se encontrar um caminho de s para t em G tal que a soma dos comprimentos dos arcos neste caminho seja mínima. Recorde que um caminho de um vértice s a um outro vértice t no grafo G é uma sequência de vértices $(v_0, v_1, v_2, \ldots, v_p)$ tal que $v_0 = s$, $v_p = t$ e $(v_{i-1}, v_i) \in E$ para todo $i = 1, \ldots, p$. O *problema do caminho mais curto* (PCMC) descrito acima pode ser resolvido por algoritmos combinatórios de complexidade polinomial em $n = |V|$ e $m = |E|$, como o Algoritmo de Dijkstra, apresentado na Seção 1.4.2. Contudo, para evidenciar como a PL e as matrizes TU podem ajudar a estabelecer a complexidade de um problema, pelo menos no que diz respeito à sua pertinência à classe $\mathbb{P}$, suponha por um momento que a complexidade do PCMC é desconhecida.

Para formular o PCMC como um PFCM, considere inicialmente que o comprimento de um arco seja equivalente ao custo unitário de passar uma unidade de fluxo por ele. Agora, suponha que se deseja escoar uma unidade de fluxo de s para t. Na terminologia do PFCM, isto corresponde a dizer que $b_s = 1 = -b_t$ que e $b_i = 0$ para todo $i \in V \setminus \{s, t\}$. O fluxo que deixa s deve percorrer um caminho em G até chegar em t. O custo desta operação será a soma dos custos dos fluxos nos arcos do caminho, já que a quantidade de fluxo que passa é unitária. Contudo, por construção, esta soma é exatamente o comprimento do caminho. Logo, ao minimizar o custo do escoamento do fluxo, na verdade, também se está resolvendo o problema de encontrar o caminho de comprimento mínimo de G que liga s a t. O modelo de PL para o PCMC seria:

$$
\begin{aligned}
\min \quad & z = \sum_{(i,j)\in E} c_{ij} x_{ij} && && \\
\textit{sujeito a} \quad & \sum_{k\in V^+(i)} x_{ik} - \sum_{k\in V^-(i)} x_{ki} = 0, && \forall i \in V \setminus \{s,t\}, && (7)\\
& \sum_{k\in V^+(s)} x_{ik} - \sum_{k\in V^-(s)} x_{ki} = 1, && && (8)\\
& \sum_{k\in V^+(i)} x_{ik} - \sum_{k\in V^-(i)} x_{ki} = -1, && && (9)\\
& x_{ij} \leq 1, && \forall (i,j) \in E, && (10)\\
& x_{ij} \geq 0, && \forall (i,j) \in E. && (11)
\end{aligned}
$$

Conforme argumentado no modelo geral do PFCM, a matriz dos coeficientes das restrições da formulação acima é TU. Como o vetor correspondente ao termo independente das restrições (o "lado direito" das mesmas) possui todas componentes inteiras, pela Proposição (1.6.1), este modelo admite uma solução ótima que é inteira, no caso, um vetor binário. Ou seja, o caminho mais curto será composto pelos arcos (i, j) para os

quais a variável x_{ij} tem valor 1 na solução ótima. A consequência deste fato é que o PCMC está em $\mathbb{P}$. Isto porque ele tem um número de restrições e variáveis que e polinomial em n e m e todos os coeficientes da matriz de restrições e do vetor de termos independentes estão em $\{0,+1,-1\}$. Logo, o modelo pode ser resolvido por um algoritmo de pontos interiores, ou pelo método dos elipsóides em tempo polinomial no tamanho da entrada do PCMC.

No problema do fluxo máximo tem-se o seguinte cenário. Na entrada é dado um grafo direcionado $G = (V,E)$ e dois vértices distintos s e t em V. Por hipótese, supõe-se que o arco (t,s) não esteja em E (veja em [Ahuja et al. 1993] formas simples de alterar a rede no caso em que este arco está E). Além disso, para cada arco (i,j) de E, é dada uma capacidade positiva h_{ij}. O *problema do fluxo máximo* (PFM) pede que seja encontrada a quantidade máxima de fluxo que ao ser injetada para dentro da rede através do vértice s consegue atingir o vértice t. O fluxo injetado em s percorrerá os arcos da rede, sempre respeitando suas capacidades máximas. Como no caso anterior, este problema admite algoritmos de complexidade polinomial no tamanho da entrada mas será suposto, apenas para fins didáticos, que não é sabido que PFM está em $\mathbb{P}$.

Ao tentar modelar este problema como um PFCM, surge uma dificuldade, já que as demandas de fluxo nos vértices não são dados de entrada. A rigor, o que se busca é um valor máximo para b_s ou, equivalentemente para $-b_t$, de modo que, se $b_i = 0$ para todos os demais vértices, o PFCM ainda admita uma solução viável. Para contornar esta situação, faz-se uma pequena alteração no grafo G, acrescentando-se a E o arco (t,s) com capacidade de fluxo ilimitada. Denote-se por $G' = (V,E' = E \cup \{t,s\})$ o grafo resultante desta operação. Além disso, fixe $b_i = 0$ para todo vértice i de V, incluindo s e t. Finalmente, atribua custo nulo para a circulação de uma unidade de fluxo a todos os arcos, exceto ao novo arco (t,s) ao qual se atribui o custo 1. Note que ao fazer isso, se na rede original G for escoado um fluxo de f unidades de s para t, na rede G', esta mesma quantidade de fluxo deverá retornar a s através do arco (t,s), já que a rede está isolada do mundo externo pelo fato das demandas em todos os vértices serem nulas. Seguindo esta interpretação, o PFM pode ser resolvido se encontrarmos o maior valor possível para f, respeitadas as capacidades dos arcos originais de G. Logo, definido-se x_{ij} como sendo a quantidade de fluxo que passa no arco (i,j) para todo $(i,j) \in E'$, o PFM pode ser modelado pelo seguinte problema de PL:

$$\begin{aligned}
\max \quad & z = x_{ts} & & \\
\textit{sujeito a} \quad & \sum_{k \in V^+(i)} x_{ik} - \sum_{k \in V^-(i)} x_{ki} = 0, & \forall i \in V, & \quad (12)\\
& x_{ij} \leq h_{ij}, & \forall (i,j) \in E = E' \setminus \{(t,s)\}, & \quad (13)\\
& x_{ij} \geq 0, & \forall (i,j) \in E'. & \quad (14)
\end{aligned}$$

Caso as capacidades h_{ij} sejam dadas por valores inteiros, assim como ocorreu com o PCMC, tem-se um PL cuja matriz de restrições é TU. Novamente conclui-se que o

problema admite um algoritmo polinomial no tamanho de entrada via a PL. Isso seria, portanto, uma prova de que o PFM (com capacidades inteiras nos arcos) está em $\mathbb{P}$.

1.7. Algoritmos *Backtracking*

Esta seção apresenta o projeto de algoritmos por backtracking, que é bastante aplicado no desenvolvimento de algoritmos exatos para problemas de otimização combinatória. A técnica também consiste na busca de uma solução ótima por enumeração, construindo as candidatas a solução de maneira incremental, adicionando ou descartando elementos a cada chamada recursiva ou iteração. Diferente de um algoritmo força-bruta, a técnica busca limitar a quantidade de ramificações a explorar, podando ramos que não levam a soluções melhores. Para isto, é necessário investigar a estrutura combinatória do problema procurando propriedades que permitam identificar ramos não promissores. Diminuindo a quantidade de ramos percorridos de maneira efetiva, é possível encontrar uma solução ótima em menos tempo.

Nas próximas seções, a técnica é aplicada a dois problemas: ao problema da Mochila Binária e ao problema do Conjunto Independente Máximo.

1.7.1. Algoritmo *Backtracking* para o Problema da Mochila Binária

O algoritmo proposto para o PMB é recursivo e a cada chamada considera-se um item e duas situações, uma quando o item é adicionado à solução corrente e outra quando o item não pertence à solução.

Inicialmente, considere um conjunto de itens cuja soma dos pesos ultrapassa a capacidade da mochila. Naturalmente, trata-se de um conjunto inviável para o problema da Mochila Binária e ao percorrer por um subconjunto que o contém, este será igualmente inviável e também deve ser descartado. Como a técnica constrói soluções de maneira incremental, a poda dos conjuntos inviáveis é feita de maneira direta. Se a cada chamada, um item é colocado apenas se este couber no espaço remanescente, então, claramente o algoritmo fará sua enumeração passando apenas por soluções factíveis.

Outra estratégia usada para otimizar a busca, é evitar fazer a busca por direções que, apesar de possuirem soluções factíveis, estas tem valores iguais ou piores que a melhor solução corrente. Para explorar essa estratégia, é importante ter soluções boas o quanto antes, e saber estimar, durante o processo de ramificação, o quanto se pode melhorar a partir daquele ponto. Esta estimativa representa um limitante para o valor da melhor solução possível que completa a solução corrente. Caso esta estimativa indique haver apenas soluções iguais ou piores que a melhor solução obtida até o momento, evita-se fazer a busca por tal ramo. Caso não se tenha encontrado soluções até o momento, ou a atual candidata a solução não tenha boa qualidade, haverá menos chances de um ramo ser podado. Segue disto a importância de se ter boas soluções o quanto antes, podendo inclusive começar a enumeração já com o valor de uma solução candidata obtida por métodos heurísticos. Por exemplo, pode-se usar o algoritmo guloso, visto para o PMB, como heurística para a obtenção de uma primeira solução viável. De fato,

o algoritmo apresentado usa uma estratégia mais sofisticada e aplica a própria busca gulosa também como parte do processo de enumeração. Mais precisamente, a cada chamada recursiva, o algoritmo escolhe um item, dentre os que não foram considerados, com o maior valor relativo. Caso este item possa ser inserido no espaço remanescente, o algoritmo busca primeiro pelo ramo que incorpora o item na solução corrente e posteriormente, busca pelo ramo que o descarta.

Para percorrer os itens na mesma ordem que a utilizada pelas chamadas recursivas, basta ordená-los previamente por seu valor relativo, obtendo $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \frac{v_n}{p_n}$. Desta forma, a ramificação é guidada por uma direção gulosa, na esperança de encontrar soluções melhores mais rapidamente.

Para que se possa podar um ramo não-promissor, é necessário obter algum limitante para a qualidade das soluções que serão obtidas a partir deste ramo. No caso do problema da Mochila Binária, é possível obter um limitante para uma solução ótima através de uma solução para o problema da correspondente Mochila Fracionária. Assim, suponha que no início de uma chamada recursiva consideramos o m-ésimo item, para colocar ou não na solução corrente. Seja x^* a melhor solução obtida até o momento e x a solução atual. O valor e peso total dos itens representados em x^* (resp. x) é dado por $V^* = v \cdot x^*$ e $P^* = p \cdot x^*$ (resp. $Vx = v \cdot x$ e $Px = p \cdot x$). O espaço remanescente neste ponto é dado por um valor $B' = B - p \cdot x$ e um limitante superior para este espaço é obtido pela aplicação do algoritmo que resolve o problema da Mochila Fracionária com capacidade B' e itens em $\{m, \ldots, n\}$. Se a solução deste problema da Mochila Fracionária é dada por V_f, o limitante para qualquer solução que complementa a atual solução é no máximo $v \cdot x + V_f$. Caso este valor seja no máximo o valor da melhor solução corrente $v \cdot x^*$, podemos descartar os ramos que seguem do nó corrente. Este processo é descrito no Algoritmo Mochila-BT.

A primeira chamada da rotina recebe, além dos dados de entrada, dois parâmetros: um vetor n-dimensional x^* com todas as posições iguais a zero e um vetor sem elementos, x. O vetor x^* mantém a melhor solução obtida durante a execução da rotina e o vetor x representa a solução que está sendo construída. A cada nova chamada recursiva, o vetor é incrementado com mais uma posição. Observa-se que, no passo 3 pode-se utilizar uma estimativa mais fraca V'_f, definida como $V'_f = (B - p \cdot x)\frac{v_m}{p_m}$. Desta forma, o cálculo da estimativa é feito em tempo constante, e apesar de ser uma estimativa mais fraca, pode ser melhor em alguns casos, dado que o tempo computacional para seu cálculo é menor.

Apesar do Algoritmo Mochila-BT ter complexidade de tempo $O(2^n)$, o algoritmo possui um bom desempenho prático. Não é possível dizer que este algoritmo é mais rápido ou mais lento que o Algoritmo Mochila-PD, apresentado na Seção 1.5.1, que utiliza programação dinâmica, uma vez que cada um pode ser mais rápido para diferentes entradas. Por fim, pode-se implementar o Algoritmo Mochila-BT de maneira iterativa. Recomenda-se que o leitor implemente a versão iterativa deste algoritmo.

Algoritmo: Mochila-BT(B,p,v,n,x^*,x,m) onde $\frac{v_1}{p_1} \geq \frac{v_2}{p_2} \geq \cdots \frac{v_n}{p_n}$

Saída : Atualiza x^* se encontrar solução melhor que completa x

1 **se** $m \leq n$ **então**
2 **se** $v \cdot x > v \cdot x^*$ **então** $x^* \leftarrow x$
3 seja V_f o valor da solução de
   Mochila-Fracionária-AG$(B - p \cdot x, v[m,\ldots,n], p[m,\ldots,n], n-m+1)$
4 **se** $v \cdot x + V_f > v \cdot x^*$ **então**
5 **se** $p_m \leq B - p \cdot x$ **então** Mochila-BT$(B,p,v,n,x^*,x\|(1),m+1)$
6 Mochila-BT$(B,p,v,n,x^*,x\|(0),m+1)$

1.7.2. Algoritmo *Backtracking* para o Problema do Conjunto Independente

Nesta seção, investiga-se o problema do Conjunto Independente Máximo, para o qual, é apresentado uma sequência de algoritmos, onde cada um pode ser visto como uma sofisticação do anterior. São apresentados alguns algoritmos por backtracking para resolver o problema do conjunto independente máximo, onde a cada nova versão, temos um algoritmo que explora melhor a estrutura combinatória do problema. Esta seção foi baseada em [Kloks 2012, Erickson 2015].

Definicao 1.7.1 *Dado grafo $G = (V,E)$ não-orientado, um subconjunto $S \subseteq V$ é um* conjunto independente *de G se quaisquer dois vértices distintos de S não estão ligados por uma aresta de G.*

Problema do Conjunto Independente (PCIM)**:** Dado grafo não-orientado G, encontrar um conjunto independente de G de cardinalidade máxima.

Um algoritmo por força bruta para o PCIM pode ser feito de maneira análoga ao feito para o algoritmo força bruta para o PMB. Neste caso, usa-se um vetor binário $x \in \{0,1\}^{|V|}$, onde $x_v = 1$ se v está na solução e $x_v = 0$ caso contrário. Ao testar todos os vértices do grafo, verifica-se se o conjunto representado é um conjunto independente maior que o encontrado até o momento, e neste caso o atualiza. É possível verificar se o vetor x representa um conjunto independente em tempo linear no tamanho do grafo, isto é, em $O(n+m)$, que ficará como exercício para o leitor. Com isso, a complexidade de tempo deste algoritmo é $O(2^n(n+m))$, isto é, $O^*(2^n)$.

O objetivo será fazer um algoritmo com complexidade de tempo melhor que do força bruta. Considere uma versão de algoritmo backtracking para o PCIM, enumerando a pertinência de cada vértice na solução. Em cada iteração, pode se escolher um vértice qualquer $v \in V$ e encontrar dois conjuntos independentes máximos, um condicionado a não ter o vértice v e outro condicionado a tê-lo. Naturalmente um deles é um conjunto independente máximo de G. No primeiro caso, a busca recai no grafo $G-v$. No segundo

caso, v deve pertencer à solução, e consequentemente os vértices adjacentes a V não podem pertencer. Esta idéia é representada no seguinte algoritmo.

Algoritmo: ConjInd1(G)

Saída : Um conjunto independente de $G = (V,E)$ de cardinalidade máxima.

1 **se** $|V| \leq 1$ **então devolva** V
2 **senão**
3 seja $v \in V$ um vértice qualquer.
4 $S_0 \leftarrow$ ConjInd1$(G - v)$.
5 $S_1 \leftarrow$ ConjInd1$(G - \text{Adj}(v) - v) \cup \{v\}$.
6 **devolva** $S \in \{S_0, S_1\}$, com $|S|$ máximo.

Como não necessariamente há vértices adjacentes a v, ambas as chamadas para a rotina ConjInd1, nos passos 4 e 5, podem ocorrer para um grafo com $n-1$ vértices, onde $n = |V|$. Com isso, a complexidade computacional do Algoritmo ConjInd1 é dada por $T_1(n) = 2T_1(n-1) + p_1(n)$, onde $p_1(n)$ é a função de complexidade computacional que delimita o tempo gasto pela rotina ConjInd1, sem contar o tempo das chamadas recursivas. Considerando estruturas adequadas para representar G é possível que $p_1(n)$ seja polinomial em n. Portanto, pelo Teorema 1.3.2, temos que $T_1(n)$ é $O^*(2^n)$, o que não leva a melhorias no termo exponencial, em relação ao algoritmo força bruta.

Para desenvolver um algoritmo com complexidade de tempo melhor, deve-se explorar melhor a estrutura combinatória do problema. De fato, para cada uma das chamadas recursivas das linhas 4 e 5, considerou-se que os grafos das chamadas recursivas diminuiu apenas um vértice. Porém, na chamada da linha 5, isto só ocorre quando v é escolhido ser vértice isolado, sem arestas incidentes. Naturalmente, todos os vértices isolados pertencem a um conjunto independente de cardinalidade máxima. Com isso, sempre que há tais vértices, estes podem ser incorporados ao conjunto sendo construído.

Considere um algoritmo obtido do Algoritmo ConjInd1, denominado aqui por ConjInd2, que, logo antes da escolha do vértice v, incorpora todos os vértices isolados na solução. Com isto, v será adjacente a pelo menos um vértice e portanto, a chamada da linha 5 será feita para um grafo com no máximo $n-2$ vértices. Assim, a recorrência que representa o comportamento assintótico deste algoritmo é dada por

$$T_2'(n) \leq \max\{T_2'(n-1)\ ;\ T_2'(n-1) + T_2'(n-2)\} + p_2(n),$$

onde $p_2(n)$ é a função de complexidade computacional do tempo gasto pela rotina ConjInd2, sem contar o tempo das chamadas recursivas. Naturalmente, a recorrência $T_2'(n)$ pode ser limitada pela recorrência $T_2(n) = T_2(n-1) + T_2(n-2)\} + p_2(n)$, que aplicando o Teorema 1.3.2, tem-se que $T_2(n)$ é $O^*(1.618\ldots^n)$, uma melhora significativa em relação ao primeiro algoritmo.

Buscando melhorar o Algoritmo ConjInd2, considere o caso crítico deste algoritmo. Na análise acima, este ocorre quando o vértice v possui exatamente um vértice adjacente. Seja u o vértice adjacente a v e S um conjunto independente de cardinalidade máxima de G. Note que necessariamente u ou v deve pertencer a S, mas não ambos. Se u pertence a S, podemos obter um novo conjunto independente de mesma cardinalidade removendo u e adicionando v. Assim, concluímos que neste caso, sempre há um conjunto independente máximo contendo v. Logo, podemos incorporá-lo na solução e fazer a chamada recursiva para o grafo sem os vértices u e v. A descrição deste algoritmo é dada a seguir.

Algoritmo: ConjInd3(G)

Saída : Um conjunto independente de $G = (V, E)$ de cardinalidade máxima.

1 **se** $|V| \leq 1$ **então devolva** V
2 **senão**
3 **se** *existe* $v \in V$*:* grau$(v) \leq 1$ **então**
4 **devolva** ConjInd3$(G - \text{Adj}(v) - v) \cup \{v\}$.
5 **senão**
6 Seja $v \in V$ um vértice qualquer.
7 $S_0 \leftarrow$ ConjInd3$(G - v)$.
8 $S_1 \leftarrow$ ConjInd3$(G - \text{Adj}(v) - v) \cup \{v\}$.
9 **devolva** $S \in \{S_0, S_1\}$, com $|S|$ máximo.

Analisando a complexidade de tempo do Algoritmo ConjInd3, considerando que o grafo usado na chamada da linha 8 tem no máximo $n-2$ vértices, é possível limitar a complexidade de tempo deste algoritmo, de maneira análoga a feita para delimitar T_2' por T_2, pela recorrência

$$T_3(n) = T_3(n-1) + T_3(n-3) + p_3(n),$$

onde $p_3(n)$ é um polinômio em n. A resolução desta recorrência nos dá que $T_3(n)$ é $O^*(1.465\ldots^n)$.

Neste ponto está claro que teremos uma complexidade de tempo melhor, se garantirmos que o vértice v, escolhido para ser considerado dentro ou fora do conjunto independente, tem grau tão grande quanto possível. O caso crítico do Algoritmo ConjInd3 ocorre quando o vértice v, escolhido no passo 6, tem grau 2. Agora, considere uma versão de algoritmo que escolhe um vértice v de grau máximo. Enquanto o vértice escolhido tiver grau pelo menos 3, a complexidade de tempo do algoritmo deve levar a uma função de complexidade de tempo melhor que a apresentada por T_3. Mas, inevitavelmente podemos chegar a situação onde não há vértices de grau pelo menos 3. Por outro lado, tal grafo possui estrutura bastante restrita. O seguinte lema, cuja prova fi-

cará como exercício, mostra que de fato é possível encontrar um conjunto independente neste grafo de maneira eficiente.

Lema 1.7.1 *Se G é um grafo onde todos os vértices tem grau no máximo 2, então é possível obter um conjunto independente de cardinalidade máxima em tempo linear.*

Algoritmo: ConjInd4(G)

Saída : Um conjunto independente de $G = (V,E)$ de cardinalidade máxima.
se $|V| \leq 1$ **então devolva** V
senão
    se grau$(v) \leq 2$ *para todo* $v \in V$ **então**
        Seja S um conjunto independente de cardinalidade máxima de G.
        devolva S.
    senão
        Seja $v \in V$ um vértice de G de grau máximo.
        $S_0 \leftarrow$ ConjInd4$(G - v)$.
        $S_1 \leftarrow$ ConjInd4$(G - \text{Adj}(v) - v) \cup \{v\}$.
        devolva $S \in \{S_0, S_1\}$, com $|S|$ máximo.

De forma análoga, o tempo de execução deste algoritmo pode ser limitado pela seguinte recorrência

$$T_4(n) = T_4(n-1) + T_4(n-4) + p_4(n),$$

onde $p_4(n)$ é um polinômio em n. Resolvendo esta recorrência, temos que T_4 é $O^*(1.380\ldots^n)$.

Alguns pontos de interesse das análises de complexidade de tempo dos algoritmos de tempo exponencial, descritos acima.

- Alterar o esforço computacional para dividir e/ou conquistar os subproblemas, mantendo este esforço em tempo polinomial, sem alterar o tamanho dos subproblemas, não mudará a base da complexidade de tempo exponencial.

- Porém, se o esforço investido diminuir o tamanho das entradas usadas nas chamadas recursivas, pode-se diminuir a base da complexidade de tempo exponencial.

Do ponto de vista teórico, estes pontos indicam que é melhor investir na redução do tamanho dos subproblemas, contanto que esta redução seja feita em tempo polinomial.

1.8. Algoritmos *Branch and Bound*

O *branch-and-bound* é uma técnica empregada na resolução de problemas difíceis de otimização combinatória, i.e., aqueles para os quais não se conhecem algoritmos polinomiais capzes de resolvê-los. Um algoritmo *branch-and-bound* enumera *implicitamente* todas as soluções do problema. Evidentemente, como o conjunto de soluções é, na maioria dos casos, exponencial no tamanho da entrada, esta enumeração deve ser *ímplicita* pois, do contrário, o algoritmo seria de pouca utilidade prática, equivalendo-se à resolução do problema pela *força bruta*. Os algoritmos *branch-and-bound* são exemplos de aplicação do paradigma de divisão-e-conquista uma vez que a prova de otimalidade da solução retornada por eles se baseia na partição sucessiva do conjunto de soluções. O termo *branch*, palavra inglesa que pode ser traduzida por *ramificação*, diz respeito a este processo de partição. Já o termo *bound* refere-se ao fato dos algoritmos *branch-and-bound* valerem-se do cálculo de limitantes duais e primais para o valor ótimo procurado. São estes limitantes que permitem que se possa fazer uma busca inteligente por uma solução ótima, evitando a enumeração exaustiva de todas as possíveis soluções. Para melhor entender o funcionamento de um algoritmo *branch-and-bound* é necessário introduzir algumas definições e terminologias.

O processo de particionamento do conjunto de soluções efetuado por um algoritmo *branch-and-bound* pode ser representado por uma *árvore de enumeração*, também denominada por *árvore de espaço de estados*. Nesta árvore, os nós folha representam as soluções enquanto os nós internos dizem respeito aos subconjuntos de soluções que são considerados à medida que o processo de particionamento sucessivo avança. Assim, um nó interno está associado ao subconjunto de soluções que correspondem às folhas da subárvore com raiz naquele nó. Consequentemente, de acordo com esta intepretação, o nó raiz representa o conjunto de todas as possíveis soluções do problema.

Agora, considere o problema de otimização combinatória sob a forma de maximização[1] dado por: $z = \max\{cx : x \in S\}$, onde S é o conjunto de todas as soluções viáveis do problema. Suponha uma partição $\{S_1, S_2, \ldots, S_K\}$ de S de modo que $z^k = \max\{cx : x \in S_k\}$, para todo $k = 1, \ldots, K$. Não é difícil ver que $z = \max_k z^k$, ou seja, o valor ótimo global é o máximo dos valores ótimos computados sobre os elementos da partição. Além disso, suponha que, para todo $k = 1, \ldots, K$ sejam conhecidos um limitante dual (superior) $\overline{z}^k$ e um limitante primal (inferior) $\underline{z}^k$ para z^k. Pode-se verificar que os valores de $\overline{z}$ e $\underline{z}$ abaixo são limitantes superior e inferior de z, respectivamente,

$$\overline{z} = \max_k\{\overline{z}^k\} \qquad \text{e} \qquad \underline{z} = \max_k\{\underline{z}^k\}.$$

A observação anterior permite entender o princípio básico do algoritmo *branch-and-bound*. Partindo do nó raiz, o algoritmo faz uma exploração da árvore de enumeração. Supondo que ele compute limitantes inferiores e superiores a cada nó que explore,

[1] A maior parte desta seção trata de problemas de maximização. Não é difícil adaptar os resultados para o caso de minimização.

pelo resultado acima ele terá constantemente à sua disposição um limitante dual e um primal para o valor ótimo z. Este último pode ser calculado, por exemplo, por meio de heurísticas. A rigor, se estes limitantes forem iguais[2], isto significa que a solução que fornece o limitante primal $\underline{z}^k$ é ótima. Nesta situação, o algoritmo para, interrompendo-se a exploração da árvore de enumeração. Evidentemente que quanto menor a quantidade de nós explorados, menor será o tempo de execução do algoritmo e, portanto, maior será a sua eficácia. Daí a enorme importância da qualidade tanto dos limitantes duais quanto dos primais para viabilizar o uso de um algoritmo *branch-and-bound*. Esta qualidade é mensurada pelo desvio destes valores em relação ao ótimo.

Antes de prosseguir com mais detalhes relativos ao funcionamento do algoritmo, discute-se sobre as formas alternativas de efetuar o particionamento do conjunto de soluções. Dependendo como isto é feito, a árvore de enumeração poderá ter formas diferentes. Por exemplo, se o conjunto de soluções representadas por um nó interno for sempre particionado em dois subconjuntos, a árvove de enumeração será binária, já que este nó terá dois filhos. Contudo, diferentes estratégias de particionamento do conjunto de soluções de um nó interno podem ser adotadas, originando árvores onde estes nós têm um número de filhos que pode ser fixo (e diferente de 2) ou até mesmo variável. Esta ideia é ilustrada nos exemplos a seguir, onde considera-se que o nó raiz da árvore de enumeração encontra-se no nível *zero*.

Exemplo 1.8.1 *Considere o problema da mochila binária definido sobre 3 itens em que a solução é representada por um vetor binário x de tamanho 3, em que* $x_i = 1$ *se e somente se o item* $i \in \{1,2,3\}$ *é levado na mochila. A árvore de enumeração é construída de modo que, no nível i, as decisões sobre os i primeiros itens já estão fixadas. Assim, estando em um nó interno no nível i da árvore de enumeração, pode-se fazer uma ramificação em que este nó dá origem a dois filhos, um deles à esquerda que será a raiz de uma subárvore onde todas soluções não irão conter o item i, ou seja,* $x_i = 0$*. Analogamente, a subárvore à direita corresponderá ao subespaço de soluções em que o item i será incluído na mochila e, portanto,* $x_i = 1$*. A árvore de enumeração seria aquela vista na Figura 1.8. Evidentemente, neste caso a árvore é binária.*

Exemplo 1.8.2 *Considere o problema do caixeiro viajante, definido na seção 1.2.1, sobre um grafo direcionado completo contendo 4 vértices*[3]*. Inicialmente, note que a descrição de um circuito hamiltoniano pode ser feita, sem ambiguidades, pela ordem em que os vértices são visitados ao começar o percurso no vértice 1. A partir daí, define-se a árvore de enumeração de modo que em um nó interno no nível i, considera-se que já tenham sido tomadas as decisões sobre os* $i+1$ *primeiros vértices visitados no circuito. Com isto, o nó terá* $n-i-1$ *ramificações possíveis, cada uma associada*

[2]Será visto mais adiante que em algumas situações esta condição pode ser relaxada

[3]O termo "nó" será sempre usado nesta seção para referir-se à árvore de enumeração, enquato "vértices" será reservado para grafos de entrada de problemas que estejam sendo tratados.

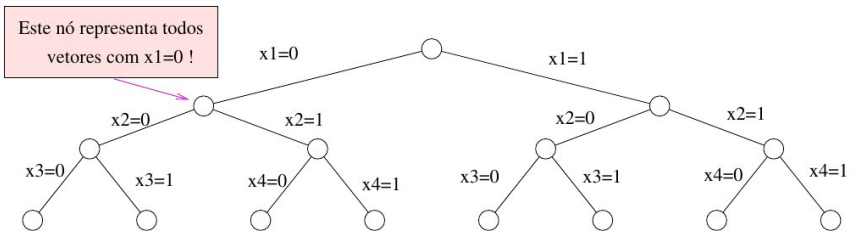


Figura 1.8. Árvore de enumeração completa para o problema binário da mochila com 3 itens.

a um possível vértice que irá suceder o último vértice do caminho parcial (no grafo) que sai do vértice 1 e corresponde às ramificações percorridas ao se sair da raiz da árvore de enumeração até chegar ao nó interno em questão. Para um grafo completo de 4 vértices, a árvore de enumeração seria aquela vista na Figura 1.9. Repare que nesta árvore, o grau de um nó no nível i é sempre igual a $n-i-1$.

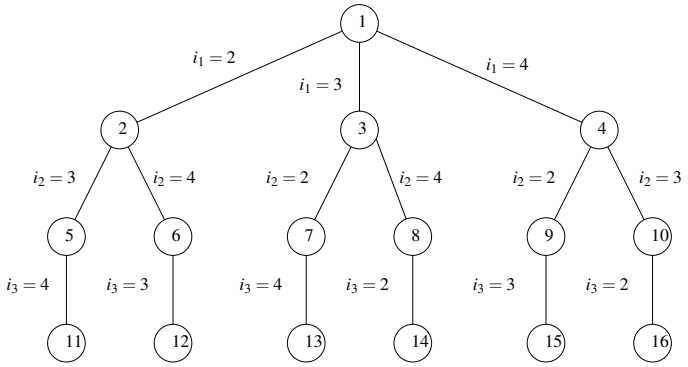


Figura 1.9. Árvore de enumeração completa para o problema do caixeiro viajante de um grafo direcionado completo com $n=4$ vértices, onde $i_0=i_4=1$.

Tendo visto exemplos de diferentes formas de particionamento do espaço de soluções e das árvores de enumeração que delas resultam, retorna-se agora à descrição do funcionamento do algoritmo *branch-and-bound*. Antes de prosseguir, deve-se ter em mente que a árvore de enumeração é uma abstração e que, obviamente, ela não é armazenada explícitamente na memória. No entanto, o fluxo do algoritmo *branch-and-bound* coincide com a porção da árvore que foi explorada que, para fins de eficácia, deve ser a menor possível. Dito isso, analisa-se agora como usar os limitantes primais e duais para *podar* ramos da árvore, evitando explorar subárvores que não contenham a solução ótima. Ou seja, ao *podar* um nó[4] impede-se que o algoritmo faça alguma

[4]Também se usa o termo *amadurecer* um nó.

ramificação a partir dele. São três os tipos de poda, a saber: por *otimalidade*, *limitante* e por *inviabilidade*. Estas 3 situações são ilustradas nos exemplos a seguir. Nestes exemplos, considera-se que a existência de um limitante primal pressupõe o conhecimento de uma solução viável cujo valor corresponde àquele limitante. Além disso, supõe-se que o problema que se esteja resolvendo seja de maximização.

Exemplo 1.8.3 *Considere a porção da árvore de enumeração mostrada na Figura 1.10, onde S representa o conjunto de soluções correspondente ao nó pai e S_1 e S_2 é uma partição de S. Os limitante primais (inferiores) e duais (superiores) encontram-se ao lado dos respectivos nós.*

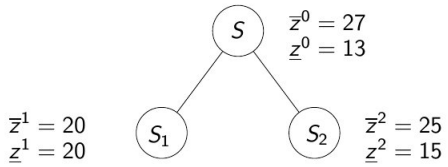


Figura 1.10. Exemplo de poda por otimalidade.

Como visto anteriormente, os limitantes superior e inferior para S são:

$$\overline{z} = \max\{\overline{z}^1, \overline{z}^2\} = \max\{20, 25\} = 25 \quad e \quad \underline{z} = \max\{\underline{z}^1, \underline{z}^2\} = \max\{20, 15\} = 20$$

Observe que em S_1 é conhecida uma solução de custo 20 ($= \underline{z}^1$) e ela é ótima *para a subárvore com raiz neste nó, pois o custo de qualquer solução de S_1 é ≤ 20 ($= \overline{z}^1$). Logo o nó que representa S_1 deve ser podado.*

Exemplo 1.8.4 *Considere agora a porção da árvore de enumeração mostrada na Figura 1.11. Neste caso, a solução de maior custo em S_1 tem custo limitado a 20 ($= \overline{z}^1$)*

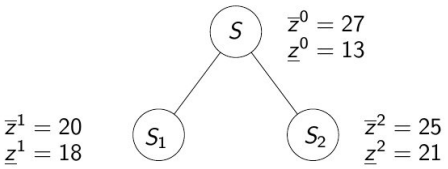


Figura 1.11. Exemplo de poda por limitante.

mas, em S_2 já é conhecida uma solução de custo 21 ($= \underline{z}^2$). Logo o nó que representa S_1 deve ser podado, pois a exploração desta subárvore só pode conduzir a soluções subótimas.

Exemplo 1.8.5 *Para ilustrar a poda por inviabilidade, considera-se a instância do problema da mochila binária para 4 itens cuja restrição é dada por* $8x_1 + 5x_2 + 3x_3 + 3x_4 \leq 12$. *Considere agora a porção da árvore de enumeração mostrada na Figura 1.12, na qual supôs-se que o espaço de estados é particionado como no Exemplo 1.8.1. O nó* S_4 *representa o subconjunto de todas as soluções que contêm simultaneamente os itens 1 e 2. Mas este subconjunto é vazio já que a soma dos pesos destes itens ultrapassa a capacidade da mochila. Consequentemente este nó deve ser podado.*

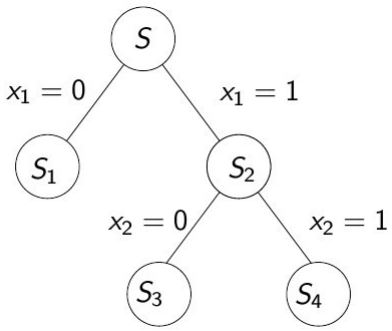


Figura 1.12. Exemplo de poda por inviabilidade.

Uma vez definidos os tipos de poda, os nós da porção da árvore de enumeração explorados pelo algoritmo *branch-and-bound* se dividem em três grupos. No primeiro deles encontram-se os nós que foram ramificados, como é o caso do nó identificado por *S* nos exemplos anteriores. No segundo grupo estão os nós podados. Finalmente, o último grupo é composto pelos chamados *nós ativos*, aqueles que não foram podados e, portanto, poderão ser ramificados.

A estratégia de exploração da árvore adotada pelo algoritmo depende fundamentalmente da escolha do próximo nó ativo a ser ramificado a cada iteração. Pode-se imaginar que os nós ativos correntes são armazenados em uma lista, inicialmente composta exclusivamente pelo nó raiz. Um pseudocódigo para um algoritmo *branch-and-bound* genérico é exibido na Figura 1.13. Nele a lista de nós ativos é representada pela variável `Ativos`. Vê-se, portanto, que a escolha do próximo nó a ramificar é feita na linha 4 do algoritmo. Comumente, na prática, tal escolha feita de acordo com a estratégia do *melhor limitante*, a qual consiste em optar pelo nó cujo limitante superior é o maior. O princípio que norteia esta estratégia é que, de acordo com os limitantes duais calculados até aquela iteração, a subárvore com raiz naquele nó é aquela que potencialmente apresenta uma solução com o maior valor. Não é difícil perceber que, ao adotar esta estratégia, nunca serão explorados nós com limitantes duais piores do que o valor ótimo. Claramente, ao se fazer isso pretende-se minimizar o número de nós explorados. Outras

critérios de escolha são propostos na literatura e que podem ser convenientes em situações específicas. É o caso, por exemplo, da adoção da busca em profundidade quando se resolve um problema onde há muita dificuldade na obtenção de uma solução primal.

1. `B&B`; (* considerando problema de **maximização** *)
2. `Ativos` $\leftarrow$ {nó raiz}; `melhor-solução` $\leftarrow$ {}; $\underline{z} \leftarrow -\infty$;
3. **Enquanto** (`Ativos` não está vazia) **faça**
4. Escolher um nó k em `Ativos` para ramificar;
5. Remover k de `Ativos`;
6. Gerar os filhos de k: $n_1, \ldots, n_q$ computando $\overline{z}_{n_i}$ e $\underline{z}_{n_i}$ correspondentes;
   (* definir $\overline{z}_{n_i}$ e $\underline{z}_{n_i}$ iguais a $-\infty$ para subproblemas inviáveis *)
7. **Para** $j = 1$ **até** q **faça**
8. **se** ($\overline{z}_{n_i} \leq \underline{z}$) **então** podar o nó n_i; (* inclui os 3 casos *)
9. **se não**
10. **Se** (n_i representa uma única solução) **então** (* atualizar melhor limitante primal *)
11. $\underline{z} \leftarrow \overline{z}_{n_i}$; `melhor-solução` $\leftarrow$ {solução de n_i};
12. **se não** adicionar n_i à lista `Ativos`.

Figura 1.13. Algoritmo *branch-and-bound* genérico.

Pelo exposto acima, nota-se que alguns aspectos são relevantes para a implementação de um algoritmo *branch-and-bound*. No caso dos limitantes, tanto duais como primais, a questão é se devem ser usados limitantes fáceis de calcular, usualmente fracos, ou limitantes fortes mas computacionalmente caros. Também deve-se pensar nas formas de decomposição do espaço de soluções e em como isto impacta o armazenamento e a manutenção da porção da árvore de enumeração explorada pelo algoritmo. Ademais, como visto acima, também é preciso definir uma ordem de percurso da árvore. Independentemente disso, existe uma quantidade mínima de informações que precisam ser armazenadas para a correta operação do algorimo. Isso incluiu os limitantes superiores de todos os nós ativos e o limitante inferior global $\underline{z}$.

Considera-se agora a aplicação do algoritmo *branch-and-bound* a 3 problemas.

1.8.1. Algoritmo *Branch-and-bound* para o Problema da Mochila Binária

Como mencionado, os limitantes duais são cruciais para o bom desempenho do algoritmo *branch-and-bound*. Tais limitantes podem ser gerados através de relaxações do problema que se quer resolver e para as quais são conhecidos algoritmos eficientes de resolução. Para o problema da mochila, uma relaxação usual é obtida ao se permitir que sejam levadas frações dos itens na mochila. Como visto na Seção 1.4 o problema da mochila fracionária pode ser resolvido em tempo polinomial usando um algoritmo guloso. Assim, um algoritmo *branch-and-bound* para a mochila binária poderia ser projetado em que a ramificação dos nós segue o esquema do Exemplo 1.8.1 e o cálculo do limi-

tante dual seria feito pelo algoritmo guloso. Considere então a instância do problema da mochila dada por:

$$\begin{array}{ll} \max & 8x_1+16x_2+20x_3+12x_4+6x_5+10x_6+4x_7 \\ & 3x_1+\ \ 7x_2+\ 9x_3+\ 6x_4+3x_5+\ \ 5x_6+2x_7 \ \ \leq 17 \\ & x_i \in \{0,1\}, i=1,\ldots,n. \end{array}$$

Note que os $n=7$ itens já se encontram ordenados em ordem não crescente da relação custo/peso, facilitando rápida identificação da solução ótima das relaxações.

A Figura 1.14 mostra a parte explorada da árvore de espaço de estados, considerando o uso da estratégia de melhor limitante (em casos de empate foram privilegiados os nós com maior profundidade e, persistindo o empate, a ordem de geração do nó). Ao lado de cada nó exibe-se uma tripla da forma $(W', C, \overline{z}_{n_i})$ onde W' é a capacidade residual da mochila, C é o custo da solução parcial correspondente ao nó e $\overline{z}_{n_i}$ é o valor do limitante obtido pelo algoritmo guloso naquele nó (já consideradas as variáveis fixadas pelas ramificações que levam até ele). Nesta figura, as ramificações de um nó interno no nível i significam que o item i é levado ($x[i]=1$) na solução quando se tratar do filho esquerdo e, do contrário, que ele não é levado ($x[i]=0$) no caso do filho direito. Os números no interior dos nós indicam a ordem em que eles foram explorados, começando em *zero*. A ordem de geração dos nós foi: $0,1,16,2,15,3,4,8,5,6,7,9,11,10,12,13,14$. Os nós podados por inviabilidade são representados por pequenos círculos pretos. Assim, por exemplo, saindo da raiz e tomando-se sempre o ramo da esquerda, chega-se a um nó preto que corresponde a todas as soluções em que $x_1=x_2=x_3=1$. Contudo, a soma dos pesos destes três itens é 19, ou seja superior à capacidade da mochila que é 17. Daí a poda por inviabilidade. Os nós 7, 13 e 14 correspondem a soluções inteiras (poda por otimalidade), enquanto os nós 15 e 16 foram podados por limitante. Pecebe-se claramente a vantagem do algoritmo *branch-and-bound* sobre o uso da força bruta, tendo em vista que a árvore de enumeração completa neste caso possui $2^{7+1}-1=255$ nós e só foram explorados 23 !

Note a proximidade do algoritmo por branch-and-bound com a versão backtracking. A principal diferença é na forma como os nós são explorados. No Algoritmo Mochila-BT, a ramificação é guiada por uma heurística gulosa, usando apenas a pilha de recursão para percorrer a árvore de maneira a privilegiar ramos mais promissores.

Na versão branch-and-bound há mais liberdade para se explorar os ramos. A versão apresentada dá prioridade para o nó com melhor limitante. Esta escolha favorece o estreitamento do *gap* entre o limitante superior e inferior, melhorando a garantia da qualidade da solução ao longo do tempo. Porém, nem sempre atinge soluções boas. Uma hibridização das duas formas pode buscar por ramos promissores enquanto não há uma solução primal boa e, após isso, concentrar-se em fechar o *gap* entre os limitantes.

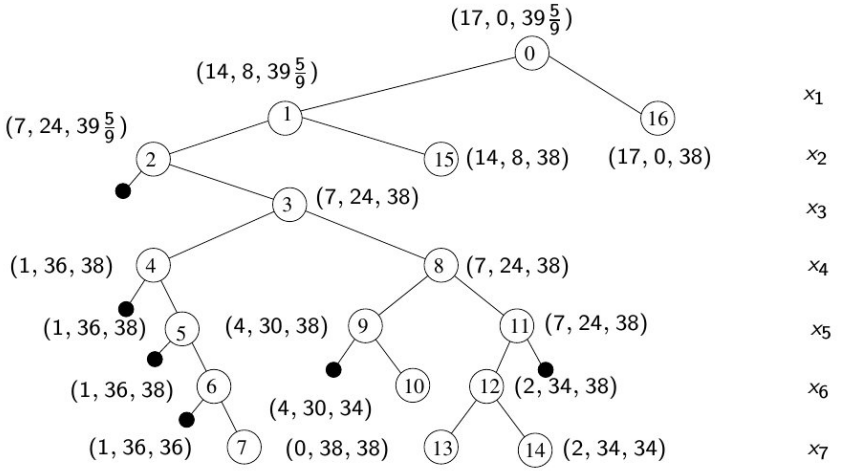


Figura 1.14. Aplicação do algoritmo *branch-and-bound* ao problema da mochila binária.

1.8.2. Algoritmo *Branch-and-bound* para um Problema de Escalonamento de Tarefas

Considere o seguinte problema de escalonamento de tarefas em máquinas. Na entrada são dados um conjunto de n tarefas $J_1, \ldots, J_n$ e duas máquinas M_1 e M_2. Cada tarefa é composta de duas operações sendo que a primeira deve ser executada em M_1 e a segunda em M_2, somente após encerrada a execução da primeira operação. O tempo de processamento da tarefa J_j na máquina M_i é dado por t_{ij}. O *tempo de término* de execução da i-ésima operação da tarefa J_j na máquina M_i é dado por f_{ij}. Cada máquina só pode executar uma operação por vez e uma vez iniciada uma operação em uma máquina ela não pode mais ser interrompida.

Deseja-se encontrar uma seqüência de execução das operações das n tarefas nas 2 máquinas, ou seja, um escalonamento das operações, de modo que a soma dos tempos de término das operações em M_2 seja mínima. Logo, a função objetivo é:

$$\min f = \sum_{j=1}^{n} f_{2j}.$$

Note que esta função objetivo é equivalente a dizer que o que se busca é minimizar o tempo médio de conclusão das tarefas. Vale observar que trata-se um problema de **minimização** e, consequentemente, o limitante dual (primal) é um limite inferior (superior) para f. Na literatura este problema é conhecido pelo seu nome em inglês *Flowshop Scheduling Problem* e, por isso, será usada a notação FSP para denotá-lo.

Dentre os resultados conhecidos para o FSP destacam-se: *(i)* a versão de decisão de FSP é $\mathbb{NP}$-completo, e *(ii)* existe um escalonamento ótimo no qual a sequência

de execução das tarefas é a mesma nas duas máquinas (conhecido por *escalonamento permutado*[5]) e no qual não há tempo ocioso *desnecessário* entre as tarefas. Este último resultado é particularmente importante pois permite que a busca por soluções ótimas fique restrita às $n!$ permutações das n tarefas, uma vez que isso define a ordem das operações nas máquinas e os seus instantes de início. O algoritmo *branch-and-bound* descrito a seguir está fundamentado neste fato.

Os conceitos anteriores são ilustrados a seguir. A tabela abaixo fornece os tempos de execução das operações para uma instância constituída de 3 tarefas.

t_{ij}	M_1	M_2
Tarefa 1	2	1
Tarefa 2	3	1
Tarefa 3	2	3

Para o escalonamento permutado $(2,3,1)$, a solução obtida é representada pelo esquema abaixo onde as linhas correspondem às máquinas e as colunas a unidades de tempo. Deste modo, na célula (i,j) está indicado o número da tarefa sendo executada na máquina M_i no instante de tempo j.

	1	2	3	4	5	6	7	8	9
M_1	2	2	2	3	3	1	1		
M_2				2		3	3	3	1

O custo desta solução é $f = f_{21} + f_{22} + f_{23} = 9 + 4 + 8 = 21$. Para esta instância, com o será visto, a escalonamento permutado ótimo é dado por $(1,3,2)$:

	1	2	3	4	5	6	7	8	9
M_1	1	1	3	3	2	2	2		
M_2			1		3	3	3	2	

O custo da solução ótima é $f = f_{21} + f_{22} + f_{23} = 3 + 8 + 7 = 18$.

Como existe uma solução ótima que é um escalonamento permutado, uma solução viável pode ser representada apenas por uma permutação de n, do mesmo modo que no problema do caixeiro viajante. A partir desta observação, uma opção no projeto do algoritmo *branch-and-bound* é utilizar a mesma forma de particionamento do espaço de estados que foi adotada no Exemplo 1.8.2. Contudo, ainda é preciso definir como calcular limitantes duais (inferiores). Como dito anteriormente, isto pode ser alcançado mediante a resolução de relaxações do FSP. Duas relaxações são consideradas aqui.

Suponha que em um dado nó u da árvore de enumeração um subconjunto R de tarefas já tenham sido escalonadas, sendo $|R| = r$. Seja t_k, $k = 1,\ldots,n$, o índice da k-ésima tarefa em qualquer escalonamento que possa ser representado por um nó na subárvore cuja raiz é o nó corrente u. O custo deste escalonamento será:

$$f = \sum_{i \in R} f_{2i} + \sum_{i \notin R} f_{2i}. \tag{15}$$

[5]Traduzido livremente do inglês "permutation schedule".

Observe que, no momento da exploração do nó u, o primeiro termo desta soma já está definido, uma vez que ele corresponde as tarefas de R, cujas posições na permutação foram definidas nas ramificações executadas pelo algoritmo desde o nó raiz até atingir u. Ou seja, caberá ao algoritmo decidir ainda sobre a ordem das tarefas que não estão em R. Fica claro então que, para calcular um limitante dual em u, precisamos encontrar um minorante para a segunda somatória, o que pode ser atingido relaxando-se as restrições do FSP.

A primeira relaxação supõe que a segunda operação de cada tarefa em $\overline{R}$ (i.e., fora de R) comece a ser executada na máquina M_2 imediatamente após o término da sua primeira operação em M_1. Para uma ordem fixa das tarefas em $\overline{R}$, um minorante para a segunda soma da equação (15) seria:

$$S_1 = \sum_{k=r+1}^{n} [f_{1,t_r} + (n-k+1)t_{1,t_k} + t_{2,t_k}]. \tag{16}$$

Pode-se chegar à equação acima percebendo que, nesta relaxação, o tempo de término da tarefa k em M_2 é dado por $f_{1,t_r} + \sum_{l=1}^{k} t_{1,t_l} + t_{2,t_k}$. Pode-se notar que S_1 é um limitante inferior para S ao ver que esta relaxação admite que tarefas sejam executadas simultaneamente em M_2.

A segunda relaxação é obtida ao se supor que cada tarefa em $\overline{R}$ começa em M_2 imediatamente depois que a tarefa precedente termina sua execução naquela máquina. Nesta hipótese, a segunda operação de uma tarefa em $\overline{R}$ poderia ser iniciada antes que a primeira operação da mesma tarefa estivesse concluída em M_1, portanto, é claramente uma relaxação do FSP. Com isto, o tempo de término da k-ésima em $\overline{R}$ seria a soma dos tempos de execução de todas as tarefas de $\overline{R}$ que a precedem mais o tempo de conclusão da tarefa r (a última de R) em M_2, i.e., f_{2,t_r}. Na verdade, alguma melhora pode ser obtida no limitante se, ao invés de f_{2,t_r}, for usado o valor $f_{1,t_r} + \min_{i \notin R}(t_{1i})$, que seria o menor tempo de término possível de uma operação de uma tarefa de $\overline{R}$ em M_1. A partir daí, um minorante para a segunda soma da equação (15) seria:

$$S_2 = \sum_{k=r+1}^{n} [\max(f_{2,t_r}, f_{1,t_r} + \min_{i \notin R} t_{1i}) + (n-k+1)t_{2,t_k}]. \tag{17}$$

Note que os minorantes obtidos nas equações (16) e (17) pressupõem uma ordem fixa das tarefas em $\overline{R}$. Porém, em cada um dos casos a ordem que leva ao menor valor da segunda soma da equação (15) pode ser computada facilmente. De fato, a minimização de S_1 é atingida ordenando-se as tarefas na ordem crescente dos valores de t_{1,t_k} enquanto a de S_2 é alcançada ordenando-se as tarefas na ordem crescente dos valores de t_{2,t_k}. Denotando-se por $\hat{S_1}$ e $\hat{S_2}$ os mínimos de S_1 e S_2 calculados como descrito acima, tem-se um *limitante inferior* para o FSP no nó corrente da árvore de enumeração dado por:

$$f \geq \sum_{i \in R} f_{2i} + \max(\hat{S_1}, \hat{S_2}). \tag{18}$$

Usando estes limitantes duais e a forma de representação do espaço de estados sugerida, a árvore de enumeração resultante da aplicação do algoritmo *branch-and-bound* usando a estratégia do melhor limitante seria aquela vista na Figura 1.15. Para entender como se chegou a esta árvore, supõe-se que, se a solução, usualmente inviável, que corresponde aos limitantes $\hat{S_1}$ e $\hat{S_2}$ em um dado nó for viável, esta será usada para atualizar o limitante superior (primal). Ademais,

note-se que a ordenação das tarefas em ordem não decrescente do tempo de execução em M_1 (M_2), que é determinante para o cálculo de $\hat{S}_1$ ($\hat{S}_2$), é dada por $\{1,3,2\}$ ($\{1,2,3\}$). Dito isto, as soluções que levam aos limitantes $\hat{S}_1$ e $\hat{S}_2$ no nó 1, em que a primeira tarefa alocada é a tarefa 1 ($\equiv R = \{1\}$) são representadas por:

$\hat{S}_1$:

	1	2	3	4	5	6	7	8	9
M_1	1	1	3	3	2	2	2		
M_2			1		3	3	3	2	

$\Longrightarrow \quad f_{2,1} + \hat{S}_1 = 18,$

$\hat{S}_2$:

	1	2	3	4	5	6	7	8	9
M_1	1	1	(3)	(3)					
M_2			1		2	3	3	3	

$\Longrightarrow \quad f_{2,1} + \hat{S}_2 = 16.$

A ociosidade observada em M_2 antes do início da tarefa 2 deve-se ao fato de que em M_1, o tempo mínimo de espera até a conclusão da próxima tarefa (o qual seria alcançado se a tarefa 3 se sucedesse à tarefa 1) termina no instante 4. Note-se ainda que, no cálculo de $\hat{S}_1$, a solução é viável e, consequentemente, ótima para aquela subárvore (limitantes primais e duais idênticos), forçando a poda deste nó por otimalidade. Para o nó 2 em que $R = \{2\}$, pode-se constatar que os cálculos de $\hat{S}_1$ e $\hat{S}_2$ corresponderiam às seguintes situações:

$\hat{S}_1$:

	1	2	3	4	5	6	7	8	9	10
M_1	2	2	2	1	1	3	3			
M_2				2		1		3	3	3

$\Longrightarrow \quad f_{2,2} + \hat{S}_1 = 20,$

$\hat{S}_2$:

	1	2	3	4	5	6	7	8	9
M_1	2	2	2	(1)	(1)				
M_2				2		1	3	3	3

$\Longrightarrow \quad f_{2,2} + \hat{S}_2 = 19.$

Como o nó 1 já deu uma solução primal de valor 18, menor do que o limitante dual de 20 computado para o nó 2, este último é podado por limitante[6]. Por fim, no nó 3 em que $R = \{3\}$, as situações associadas aos limitantes duais $\hat{S}_1$ e $\hat{S}_2$ seriam aquelas vistas abaixo:

$\hat{S}_1$:

	1	2	3	4	5	6	7	8	9
M_1	3	3	1	1	2	2	2		
M_2			3	3	3 / 1			2	

$\Longrightarrow \quad f_{2,3} + \hat{S}_1 = 18,$

$\hat{S}_2$:

	1	2	3	4	5	6	7	8	9
M_1	3	3	(1)	(1)					
M_2			3	3	3	1	2		

$\Longrightarrow \quad f_{2,3} + \hat{S}_2 = 18.$

Perceba que, no cálculo de $\hat{S}_1$ há uma sobreposição das tarefas 1 e 3 no instante 5 em M_2, o que é permitido nesta relaxação. Note ainda que haverá a execução simultânea da mesma tarefa na solução de $\hat{S}_2$. De qualquer maneira, estes limitantes não impedem a poda do nó 3 por limitante, uma vez que o limitante primal do nó 1 já é igual a 18. Logo, não existem mais nós ativos na árvore de enumeração e o valor ótimo retornado é 18.

[6] Alternativamente poder-se-ia pensar também em podar este nó por otimalidade dado que o cálculo de $\hat{S}_1$ conduz à uma solução viável.

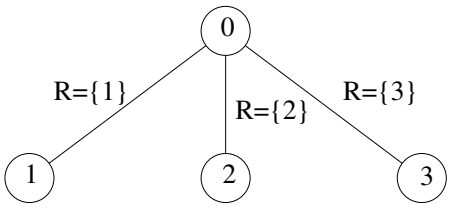


Figura 1.15. Aplicação do algoritmo *branch-and-bound* ao FSP.

1.8.3. Algoritmo *Branch-and-bound* para Programação Linear Inteira

Como mencionado na Seção 1.6, diversos problemas de otimização podem ser modelados usando Programação Linear e adicionando-se a restrição de que algumas ou todas as variáveis tomem valores inteiros (as chamadas *restrições de integralidade*). Ao fazer isso, chega-se a um problema de Programação Linear Inteira. Diferentemente do PL que pertence à classe $\mathbb{P}$, a PLI está em $\mathbb{NP}$-difícil. Por isso, os problemas de PLI são candidatos a serem resolvidos por um algoritmo de *branch-and-bound*. Neste caso, o limitante dual pode ser computado através da relaxação linear do PLI, obtida ao se remover as restrições de integralidade. Quanto à ramificação da árvore de enumeração, uma técnica comumente empregada é é escolher a variável *''mais fracionária"* na solução ótima da relaxação linear para gerar os filhos do nó corrente. Ela funciona do seguinte modo. Suponha que em um nó da árvore tenha sido resolvida a relaxação linear, obtendo-se uma solução ótima $x^* \in \mathbb{R}^n$ em que pelo menos uma variável x_i^* tem parte fracionária não nula, ou seja, $x_i^* = \lfloor x_i^* \rfloor + f_i$, com $0 < f_i < 1$ para algum $i = 1, \ldots, n$. Quando isso ocorre, seleciona-se a variável x_j em que f_j se aproxima mais do valor $0,5$ (meio) e faz-se uma ramificação em que, o filho à esquerda do nó corrente representará o conjunto das soluções em que $x_j \leq \lfloor x_j^* \rfloor$, enquanto o filho da direita está associado ao conjunto das soluções em que $x_j \leq \lceil x_j^* \rceil$. Novamente, o critério de escolha do próximo nó ativo a ser ramificado pode obedecer à estratégia do *melhor limitante (dual)* como nos exemplos anteriores.

Para se ter uma ideia do funcionamento do algoritmo, será usado o mesmo PL do exemplo da Figura 1.7 ao qual se acrescenta a restrição de que as variáveis devam ser inteiras. Ou seja, a relaxação linear deste problema é dada por

$$
\begin{aligned}
(PL0) \quad \max\ & z = 2x_1 + 3x_2 \\
\text{s. a}\quad & 2x_1 + 6x_2 \leq 15, && (19)\\
& 2x_1 - 2x_2 \leq 3, && (20)\\
& x_1, x_2 \geq 0. && (21)
\end{aligned}
$$

Na Figura 1.16 os círculos pretos representam as soluções inteiras viáveis do PLI enquanto a a região hachurada corresponde à região de viabilidade de sua relaxação linear. Foi visto na Seção 1.6 que a solução ótima deste problema está no ponto extremo de coordenadas $(x_1^*, x_2^*) = (3, \frac{3}{2})$ (círculo branco). Adotando-se a regra de ramificação discutida anteriormente, como x_2^* é a única variável com parte fracionária não nula, o nó raiz da árvore de enumeração (associado a *PL*0), dará origem a dois filhos: à esquerda onde será exigido que $x_2 \leq 1$ e à direita onde as soluções deverão atender a $x_2 \geq 2$. As relaxações lineares dos PLIs correspondentes aos dois

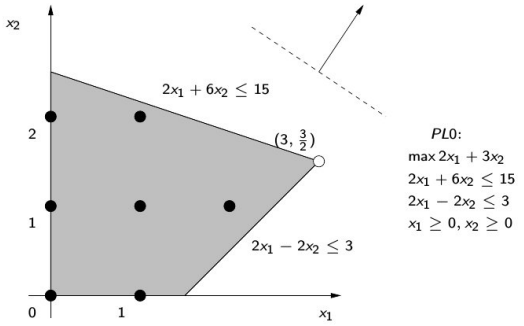


Figura 1.16. Aplicação do algoritmo *branch-and-bound* a um PLI: relaxação linear na raiz.

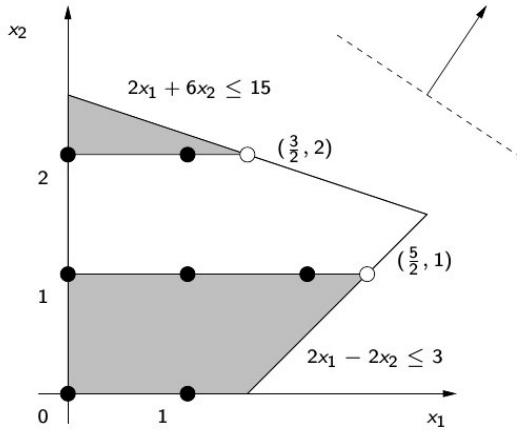


Figura 1.17. Aplicação do algoritmo *branch-and-bound* a um PLI: relaxações lineares nos filhos da raiz.

filhos estão identificadas pelas regiões hachuradas da Figura 1.17, sendo o quadrilátero associado ao filho esquerdo e o triângulo ao filho direito. Denota-se por $PL1$ e $PL2$, respectivamente, as relaxações lineares dos filhos esquerdo e direito da raiz. A solução ótima de $PL1$ é o ponto $(\frac{5}{2}, 1)$ e a de $PL2$ $(\frac{3}{2}, 2)$ como se observa na figura. Se o algoritmo *branch-and-bound* for executado até o final, a parte da árvore de enumeração explorada será aquela mostrada na Figura 1.18. Em cada nó árvore vê-se os valores de x_1 e x_2 na solução ótima da relaxação assim como o limitante dual produzido por ela. Note que pelo critério do *melhor limitante*, todos nós da subárvore à direita da raiz serão explorados antes do seu filho esquerdo (à exceção, obviamente, daqueles que levam a inviabilidades). Com isso, o filho esquerdo acaba sendo podado *por limitante*, uma vez que o nó mais à esquerda no último nível da árvore, que foi podado por *otimalidade*, gera um limitante primal de valor 8, igual ao limitante dual do filho esquerdo da raiz. As Figuras 1.19 e 1.20 mostram graficamente as demais relaxações resolvidas durante o percurso na árvore.

1.9. Considerações Finais

O texto apresenta algumas das principais técnicas usadas no desenvolvimento de algoritmos para problemas de Otimização Combinatória e seu conteúdo foi guiado pelo material usado pelos autores para ministrar as disciplinas de *Projeto e Análise de Algoritmos* nos cursos de *Ciência da Computação* e *Engenharia da Computação* na Universidade Estadual de Campinas.

Por ser introdutório e por restrições de espaço, várias técnicas e abordagens importantes não puderam ser vistas. Assim, optou-se pela abordagem exata, para a qual muitos problemas de otimização puderam ser contemplados e resolvidos por técnicas importantes da área. Dentre

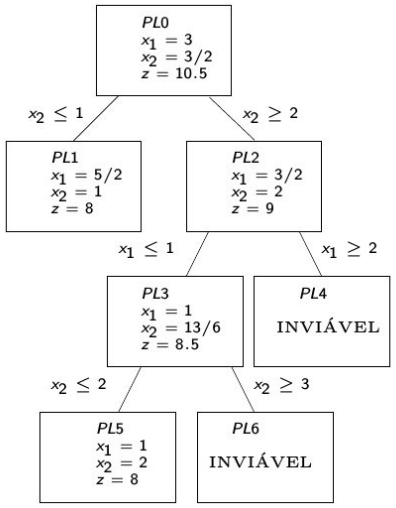
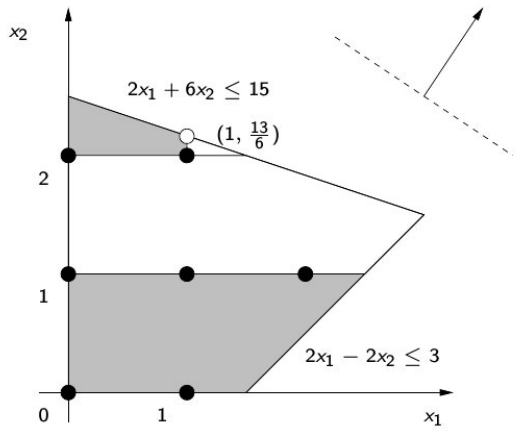
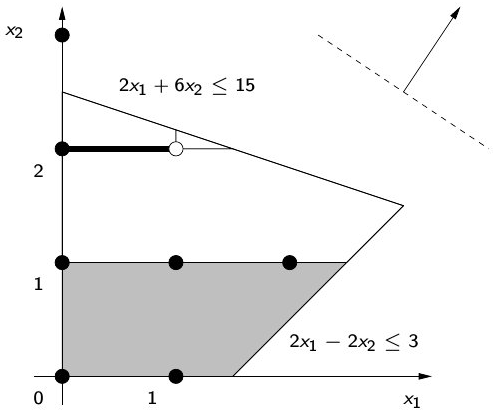


Figura 1.18. Aplicação do algoritmo *branch-and-bound* a um PLI: porção explorada da árvore de enumeração.



$PL3 = PL2 + \{x_1 \leq 1\},\ (z_3 = 8.5)$
$PL4 = PL2 + \{x_1 \geq 2\},\ (\text{inviável})$

Figura 1.19. Aplicação do algoritmo *branch-and-bound* a um PLI: relaxações lineares intermediárias (Partte I).



$PL5 = PL3 + \{x_2 \leq 2\},\ (z_5 = 8)$
$PL6 = PL3 + \{x_2 \geq 3\},\ (\text{inviável})$

Figura 1.20. Aplicação do algoritmo *branch-and-bound* a um PLI: relaxações lineares intermediárias (Parte II).

os aspectos teóricos que não pudemos explorar, mas são importantes no desenvolvimento de algoritmos da área, são os aspectos relativos à complexidade computacional, principalmente as classes de complexidade $\mathbb{NP}$-completo e $\mathbb{NP}$-difícil, sendo que esta última contempla inúmeros problemas práticos. O estudo destas classes de complexidade permite entender a dificuldade de se desenvolver algoritmos de tempo polinomial para tais problemas, bem como técnicas para reconhecê-los. Em particular, a menos que a improvável conjectura $\mathbb{P} = \mathbb{NP}$ seja válida, não existirão algoritmos de tempo polinomial para encontrar soluções ótimas de tais problemas [Garey e Johnson 1979]. Assim, é de fundamental importância conhecer a complexidade computacional dos problemas, uma vez que estas podem guiar na escolha das técnicas e abordagens a serem usadas no desenvolvimento de algoritmos para cada problema.

Dentre outras técnicas consideradas importantes no desenvolvimento de algoritmos exatos, citamos a técnica *branch-and-cut*, que leva a bons resultados práticos na resolução de vários problemas. Em particular, estão entre os melhores métodos para obter algoritmos exatos para problemas envolvendo restrições de conectividade para problemas $\mathbb{NP}$-difíceis, como o Problema do Caixeiro Viajante. Dentre as abordagens importantes que não foram exploradas no texto, temos as de *Algoritmos de Aproximação* e de *Heurísticas*. A ideia básica ao desenvolver *Algoritmos de Aproximação* é sacrificar a busca por uma solução ótima em favor de uma solução com valor próximo do ótimo, utilizando algoritmos de tempo polinomial, em contraposição aos algoritmos de tempo exponencial. *Heurísticas* são procedimentos que buscam por soluções de boa qualidade, dentro das limitações dos recursos existentes para sua obtenção. As heurísticas não necessariamente dão garantias de se encontrar soluções ótimas, ou mesmo de se encontrar uma solução que atenda aos requisitos estruturais do problema.

Agradecimentos: Os autores agradecem a Profa. Cláudia Linhares pelo apoio que tiveram para desenvolver este texto de maneira mais abrangente, o revisor pelas anotações e observações para a melhoria do texto, e ao CNPq pelo suporte financeiro.

Referências bibliográficas

[Ahuja et al. 1993] Ahuja, R. K., Magnanti, T. L. e Orlin, J. B. (1993). *Network Flows: Theory, Algorithms, and Applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Applegate et al. 2007] Applegate, D. L., Bixby, R. E., Chvatal, V. e Cook, W. J. (2007). *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA.

[Bazaraa et al. 2009] Bazaraa, M. S., Jarvis, J. J. e Sherali, H. D. (2009). *Linear Programming and Network Flows*. Wiley-Interscience. 4th edition.

[Bondy e Murty 2008] Bondy, J. e Murty, U. (2008). *Graph Theory*, volume 244 of *Graduate Texts in Mathematics*. Springer-Verlag. 654 pgs.

[Brassard e Bratley 1988] Brassard, G. e Bratley, P. (1988). *Algorithmics: Theory and Practice*. Prentice-Hall.

[Cormen et al. 2009] Cormen, T. H., Leiserson, C. E., Rivest, R. L. e Stein, C. (2009). *Introduction to Algorithms (3. ed.)*. MIT Press.

[Dasgupta et al. 2008] Dasgupta, S., Papadimitriou, C. H. e Vazirani, U. V. (2008). *Algorithms*. McGraw-Hill.

[Erickson 2015] Erickson, J. (Acessado em Maio de 2015). Algorithms and models of computation. `http://www.cs.illinois.edu/~jeffe/teaching/algorithms`.

[Feofiloff et al. 2004] Feofiloff, P., Kohayakawa, Y. e Wakabayashi, Y. (2004). Uma introdução sucinta à teoria dos grafos. `http://www.ime.usp.br/~pf/teoriadosgrafos/`. Texto da II Bienal da SBM.

[Ferreira e Wakabayashi 1996] Ferreira, C. E. e Wakabayashi, Y. (1996). *Combinatória Poliédrica e Planos-de-Corte Faciais*. 10^a Escola de Computação, Campinas.

[Garey e Johnson 1979] Garey, M. R. e Johnson, D. S. (1979). *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York.

[Karmarkar 1984] Karmarkar, N. (1984). A new polynomial-time algorithm for linear programming. *Combinatorica*, 4(4):373–395.

[Khachiyan 1979] Khachiyan, L. G. (1979). A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096.

[Kloks 2012] Kloks, T. (2012). *Advanced Graph Algorithms*.

[Manber 1989] Manber, U. (1989). *Introduction to algorithms - a creative approach*. Addison-Wesley.

[Nemhauser e Wolsey 1988] Nemhauser, G. L. e Wolsey, L. A. (1988). *Integer and combinatorial optimization*. Wiley interscience series in discrete mathematics and optimization. Wiley.

[Papadimitriou e Steiglitz 1982] Papadimitriou, C. H. e Steiglitz, K. (1982). *Combinatorial optimization: algorithms and complexity*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Rosen 2012] Rosen, K. H. (2012). *Discrete Mathematics and Its Applications: And Its Applications*. McGraw-Hill Higher Education, sétima edição.

[Schrijver 1986] Schrijver, A. (1986). *Theory of Linear and Integer Programming*. Wiley-Interscience Series in Discrete Mathematics. John Wiley, Chichester.

[Szwarcfiter 1988] Szwarcfiter, J. (1988). *Grafos e Algoritmos Computacionais*. Editora Campus Ltda, second edição.

[Wolsey 1998] Wolsey, L. A. (1998). *Integer programming*. Wiley-Interscience, New York, NY, USA.

[Ziviani 2011] Ziviani, N. (2011). *Projetos de Algoritmos com implementações em Pascal e C*. Cengage Learning.