Algoritmos gulosos: definições e aplicações

Anderson Rocha
anderson.rocha@ic.unicamp.br
Leyza Baldo Dorini
leyza.dorini@ic.unicamp.br

Campinas, 29 de abril de 2004

Introdução

De forma geral, os algoritmos relacionados com otimização lidam com uma seqüência de passos, sendo que em cada passo há um conjunto de escolhas/opções. Uma estratégia para resolver problemas de otimização são os algoritmos gulosos, os quais escolhem a opção que parece ser a melhor no momento (escolha ótima), e esperam que desta forma consiga-se chegar a uma solução ótima global. Embora nem sempre seja possível chegar a uma solução ótima a partir da utilização de algoritmos gulosos, eles são eficientes em uma ampla variedade de problemas, conforme poderemos ver neste trabalho.

Os algoritmos gulosos tomam decisões com base apenas na informação disponível, sem se preocupar com os efeitos futuros de tais decisões, isto é, eles nunca reconsideram as decisões tomadas, independentemente das conseqüências. Não há, portanto, necessidade de avaliar as alternativas nem de empregar procedimentos elaborados permitindo que decisões anteriores sejam desfeitas. Devido a tais características, de forma geral eles são fáceis de se "inventar" e implementar, e são eficientes quando funcionam [3], [2].

O Capítulo 1 apresenta uma noção introdutória do assunto, envolvendo as características gerais dos algoritmos gulosos, elementos da estratégia gulosa. Também é apresentada uma formalização dos algoritmos gulosos segundo a teoria dos *matróides*.

O Capítulo 2 apresenta o relacionamento dos algoritmos gulosos com a programação dinâmica. Julgou-se pertinente realizar tal abordagem devido aos dois temas possuírem alguns pontos em comum. Além disso, outro seminário tratou deste assunto, o que possibilitou que neste trabalho não fosse preciso fazer um levantamento teórico sobre programação dinâmica. Ainda neste capítulo foram apresentados dois problemas: o problema de seleção de atividade e o problema da mochila.

No Capítulo 3 serão apresentados mais dois exemplos. O primeiro deles foi o Código de Huffman, relacionado com a compressão de dados. O outro busca organizar tarefas de tal forma que o tempo médio que cada tarefa fica no sistema é minimizado (a definição de "tarefas" e "sistema" pode variar de um caso para outro, conforme veremos).

Os algoritmos gulosos em grafos estão no Capítulo 4. Serão abordados dois algoritmos para descobrir a árvore geradora mínima em um grafo conexo e ponderado, e um algoritmo para se computar o caminho mínimo em um grafo. O quinto e último capítulo apresenta um exemplo de como os algoritmos gulosos podem ser utilizados como heurística.

Além disso, o Anexo 1 apresenta conceitos sobre grafos, os quais são importantes em algumas partes do trabalho, especialmente no Capítulo 4.

Capítulo 1

Algoritmos Gulosos

Para possibilitar uma "noção geral" de como trabalham os algoritmos gulosos, vamos abordar um exemplo. Suponha que tenhamos disponíveis moedas com valores de 100, 25, 10, 5 e 1. O problema é criar um algoritmo que para conseguir obter um determinado valor com o menor número de moedas possível (problema do troco).

Suponha que é preciso "dar um troco" de $2.89. A melhor solução, isto é, o menor número de moedas possível para obter o valor, consiste em 10 moedas: 2 de valor 100, 3 de valor 25, 1 de valor 10 e 4 de valor 1. De forma geral, agimos tal como um algoritmo guloso: em cada estágio adicionamos a moeda de maior valor possível, de forma a não passar da quantia necessária.

Embora seja uma afirmação difícil de provar, é verdade que com os valores dados das moedas, e tendo-se disponível uma quantidade adequada de cada uma, o algoritmo sempre irá fornecer uma solução ótima para o problema. Entretanto, ressalta-se que para diferentes valores de moedas, ou então quando se tem uma quantidade limitada de cada uma, o algoritmo guloso pode vir a não chegar em uma solução ótima, ou até mesmo não chegar a solução nenhuma (mesmo esta existindo). O algoritmo para resolver o problema do troco é apresentado a seguir (próxima página).

Algoritmo 1 Algoritmo que "dá o troco" para n unidades usando o menor número possível de moedas

```
1:  função TROCO(n)
2:      const C ← {100, 25, 10, 5, 1}          ▷ C é o conjunto de moedas
3:      S ← ∅                                  ▷ S é o conjunto que irá conter a solução
4:      s ← 0                                  ▷ s é a soma dos itens em S
5:      enquanto s ≠ n faça
6:          x ← o maior item em C tal que s + x ≤ n
7:          se este item não existe então
8:              retorne "Não foi encontrada uma solução!"
9:          fim se
10:         S ← A ∪ {uma moeda de valor x}
11:         s ← s + x
12:     fim enquanto
13:     retorne S
14: fim função
```

O algoritmo apresentado é caracterizado como guloso porque a cada passo ele escolhe o maior valor possível, sem refazer suas decisões, isto é, uma vez que um determinado valor de moeda foi escolhido, não se retira mais este valor do conjunto solução [2].

Uma outra estratégia para resolver este problema é a programação dinâmica, a qual irá sempre obter um resultado. Entretanto, segundo [2] os algoritmos gulosos são mais simples, e quando tanto o algoritmo guloso quanto o algoritmo utilizando programação dinâmica funcionam, o primeiro é mais eficiente. A seguir serão apresentadas as características gerais dos algoritmos gulosos, propostas por [2].

1.1. Características gerais dos algoritmos gulosos

De forma geral, os algoritmos gulosos e os problemas por eles resolvidos são caracterizados pelos itens abordados a seguir. Para facilitar o entendimento, cada um dos itens será relacionado ao exemplo exposto anteriormente (problema do troco):

- há um problema a ser resolvido de maneira ótima, e para construir a solução existe um conjunto de candidatos. No caso do problema do troco, os candidatos são o conjunto de moedas (que possuem valor 100, 25, 10, 5 e 1), com quantidade de moedas suficiente de cada valor;
- durante a "execução" do algoritmo são criados dois conjuntos: um contém os elementos que foram avaliados e rejeitados e outro os elementos que foram analisados e escolhidos;
- há uma função que verifica se um conjunto de candidatos produz uma solução para o problema. Neste momento, questões de otimalidade não são levadas em consideração. No caso do exemplo, esta função verificaria se o valor das moedas já escolhidas é exatamente igual ao valor desejado.
- uma segunda função é responsável por verificar a viabilidade do conjunto de candidatos, ou seja, se é ou não possível adicionar mais candidatos a este conjunto de tal forma que pelo menos uma solução seja obtida. Assim como no item anterior, não há preocupação com otimalidade. No caso do problema do troco, um conjunto de moedas é viável se seu valor total não excede o valor desejado;
- uma terceira função, denominada função de seleção, busca identificar qual dos candidatos restantes (isto é, que ainda não foram analisados e enviados ou para o conjunto dos rejeitados ou dos aceitos) é o melhor (o conceito de melhor dependerá do contexto do problema). No exemplo, a função de seleção seria responsável por escolher a moeda com maior valor entre as restantes;
- por fim, existe a função objetivo, a qual retorna o valor da solução encontrada. No exemplo, esta função seria responsável por contar o número de moedas usadas na solução.

Busque identificar estas características no Algoritmo 1 exposto anteriormente.

Vamos agora definir um problema geral: a partir de um conjunto C, é desejado determinar um subconjunto $S \subseteq C$ tal que [7]:

(i) S satisfaça a uma determinada propriedade P;
(ii) S é máximo (ou mínimo, dependendo do contexto do problema)em relação a um critério α, isto é,S é o subconjunto de C que possui o maior (ou o menor) tamanho, de acordo com α que satisfaz a propriedade P.

De uma forma mais detalhada, para resolver o problema, busca-se um conjunto de candidatos que constituam uma solução, e que ao mesmo tempo otimizem a função objetivo. O conceito

de otimização irá depender do contexto do problema. No caso do problema do troco, é desejado minimizar o número de moedas.

Desta forma, pode-se ver que a função de seleção é usualmente relacionada com a função objetivo. É importante ressaltar que às vezes pode-se ter várias funções de seleção que são plausíveis, e a escolha correta de uma delas é essencial para que o algoritmo funcione corretamente. O Algoritmo 2 ilustra o funcionamento de um algoritmo guloso genérico.

Algoritmo 2 Algoritmo Guloso genérico

```
1:  função ALGORITMOGULOSO(C: conjunto)          ▷ C é o conjunto de candidatos
2:      S ← ∅                                    ▷ S é o conjunto que irá conter a solução
3:      enquanto C ≠ ∅ e não solução(S) faça
4:          x ← seleciona C
5:          C ← C {x}
6:          se é viável S ∪ {x} então
7:              S ← S ∪ {x}
8:          fim se
9:      fim enquanto
10:     se solução(S) então
11:         retorne S
12:     senão
13:         retorne "Não existe solução!"
14:     fim se
15: fim função
```

Pode-se dizer, portanto, que um algoritmo guloso trabalha da seguinte forma: a princípio o conjunto S está vazio, ou seja, não há candidatos escolhidos. Então, a cada passo, utiliza-se a função de seleção para determinar qual é o melhor candidato (lembrando que a função de seleção considera apenas os elementos que ainda não foram avaliados).

Caso o conjunto ampliado de candidatos não seja viável, ignora-se o termo que está sendo avaliado no momento. Por outro lado, se tal conjunto é viável, adiciona-se o elemento em questão ao conjunto S. O elemento considerado, sendo ele aceito ou rejeitado, não é mais considerado pela função de seleção nos passos posteriores.

Cada vez que o conjunto de candidatos escolhidos (S) é ampliado, é verificado se a solução do problema foi obtida. De acordo com [2], quando um algoritmo guloso trabalha corretamente, a primeira solução encontrada da maneira aqui descrita é ótima.

1.2. Elementos da estratégia gulosa

A partir de uma seqüência de escolhas, um algoritmo guloso busca encontrar a solução ótima para o problema em questão. Conforme exposto, a estratégia utilizada é em cada passo escolher a solução que parece ser a melhor.

No entanto, nem sempre um algoritmo guloso consegue resolver um problema de otimização. Mas existem duas características que podem indicar que os problemas podem ser resolvidos utilizando uma estratégia gulosa. São elas a propriedade de escolha gulosa e a subestrutura ótima. Estas serão abordadas a seguir [3].

1.2.1. Propriedade de escolha gulosa

Pela propriedade de escolha gulosa, uma solução globalmente ótima pode ser alcançada fazendo-se uma escolha localmente ótima (gulosa), isto é, aquela que parece ser a melhor naquele momento, desconsiderando-se resultados de subproblemas.

É importante ressaltar a necessidade de se provar que realmente uma escolha gulosa em cada um dos passos irá levar a uma solução ótima global. O que normalmente se faz, é examinar uma solução ótima global para algum subproblema e depois mostrar que ela pode ser modificada em uma solução gulosa. Isto irá resultar em um subproblema menor, mas similar.

Freqüentemente é possível fazer escolhas gulosas de uma forma mais rápida (resultando em algoritmos mais eficientes) se uma estrutura de dados apropriada for utilizada (fila de prioridades, por exemplo), ou então se houver um pré-processamento da entrada (veremos um exemplo de um caso onde isso acontece).

1.2.2. Subestrutura ótima

Esta característica também é importante quando se trabalha com programação dinâmica. Diz-se que um problema possui uma subestrutura ótima quando uma solução ótima para o problema contém dentro dela soluções ótimas para subproblemas.

O que é preciso fazer então é demonstrar que a combinação de uma escolha gulosa já realizada com uma solução ótima para o subproblema resulta em uma solução ótima para o problema original. Segundo [1, p. 305] "esse esquema utiliza implicitamente a indução sobre os subproblemas para provar que fazer a escolha gulosa em cada etapa produz uma solução ótima".

1.3. Fundamentos teóricos para métodos gulosos

Nesta seção iremos ver um pouco da teoria sobre algoritmos gulosos. Esta teoria envolve estruturas combinatórias chamadas *matróides*. Este tipo de estrutura não cobre todo tipo de problema que pode ser resolvido por um algoritmo guloso, como o algoritmo de *Huffman* (este será abordado posteriormente), mas cobre muitos casos de interesse prático.

1.3.1. Definição

De acordo com [3], um matróide é um par ordenado $M = (S, I)$, satisfazendo às seguintes condições:

1. S é um conjunto finito não vazio.
2. I é uma família não vazia de subconjuntos de S, chamada de subconjuntos independentes de S, tal que se $B \in I$ e $A \subseteq B$ então $A \in I$. Observe que o conjunto vazio deve ser necessariamente um membro de I. Neste caso, dizemos que I é hereditário.
3. Se $A \in I$, $B \in I$ e $|A| < |B|$, então existe algum elemento $x \in (B - A)$ tal que $A \cup \{x\} \in I$. Dizemos que M satisfez a propriedade da troca.

Lema: todo conjunto independente maximal em um matróide tem o mesmo tamanho. **Prova.** Por contradição: suponha que existam conjuntos independentes maximais A e B, e $|A| < |B|$. Pela propriedade de troca deve existir $x \in (B - A)$ tal que $(A + x) \in I$. Isto contradiz o fato de A ser maximal.

Exemplo: considere o *problema da mochila 0-1* (o qual também será apresentado) onde todos os itens (digamos S) têm peso unitário e a mochila tem peso K. Podemos definir o conjunto de independência I como sendo o conjunto $I = \{I \subseteq S : |I| \leq K\}$. Então $M = (S, I)$ é um matróide.

1.3.2. Algoritmos Gulosos e Matróides

Muitos problemas que podem ser resolvidos através de um algoritmo guloso recaem no problema de encontrar um conjunto independente maximal em um matróide. Neste caso, vamos considerar que cada elemento $x \in S$ de um matróide $M = (S, I)$ tem um peso estritamente positivo $w(x)$. Neste caso, dizemos que o matróide M é um matróide com pesos [3].

Dado um conjunto $A \subseteq S$, vamos denotar por $w(A)$ a soma $w(A) = \sum_{e \in A} w(e)$. O resultado mais importante relacionando matróides com pesos e algoritmos gulosos, é que sempre é possível encontrar um conjunto independente maximal através de um algoritmo guloso.

Seja $M = (S, I)$ um matróide com pesos $w : S \to \Re^+$. Considere o algoritmo 3 que tem a propriedade gulosa.

Algoritmo 3 Guloso

```
1:  função GULOSO(M, w)                          ▷ Onde M = (S, I) e w : S → ℜ+
2:      A ← 0
3:      Seja {e1, e2, ..., en} o conjunto S em ordem não crescente de peso
4:      para i ← 1, n faça
5:          se A ∪ {ei} ∈ I então
6:              A ← A ∪ {ei}
7:          fim se
8:      fim para
9:      retorne A
10: fim função
```

O conjunto A retornado pelo algoritmo *Guloso* é um conjunto independente. A complexidade deste algoritmo pode ser dada por $O(n \log n + nf(n))$, onde $f(n)$ é o tempo para verificar se um conjunto $A \cup \{x\}$ está em I.

Lema 1: seja $M = (S, I)$ um matróide com pesos $w : S \to \Re^+$ e S está ordenado em ordem não decrescente de peso, $|S| \geq 1$. Seja x o primeiro elemento de S tal que $\{x\}$ é independente, se tal x existe. Se x existe, então existe um subconjunto ótimo $A \subseteq S$ que contém x.

Lema 2: Seja x o primeiro elemento de S escolhido pelo algoritmo *Guloso* para o matróide $M = (S, I)$. O problema de achar um subconjunto maximal contendo x se restringe a achar um subconjunto maximal no matróide $M' = (S', I')$ onde,

$$\begin{aligned} S' &= \{y \in S : \{x, y\} \in I\}, \\ T' &= \{B \subseteq S - \{x\} : B \cup \{x\} \in I\}. \end{aligned}$$

Chamamos o matróide M' de contração de M pelo elemento x.

Teorema: se $M = (S, I)$ é um matróide com pesos $w : S \to \Re^+$, então $Guloso(M, w)$ retorna um subconjunto independente maximal. **Prova**: por indução em $|S|$. Para um matróide com um elemento, o algoritmo *Guloso* está correto. Agora suponha que o algoritmo esteja correto para matróides com $|S| = n - 1$.

Seja $M = (S, I)$ um matróide com $|S| = n$. Uma vez que o primeiro elemento x foi escolhido de forma gulosa pelo *Lema 1* nos garante que podemos estendê-lo a uma solução ótima. Por outro lado,

o *Lema 2* nos diz que a solução ótima contendo x é a união da solução ótima do matróide $M' = (S', I')$ (contração de M pelo elemento x) com x. Note que as próximas escolhas feitas pelo algoritmo *Guloso* podem ser interpretadas como se atuando sobre o matróide M', já que B é independente em M' se e somente se $B \cup \{x\}$ é independente em M. Pela hipótese de indução, o algoritmo *Guloso* encontra um conjunto maximal em M'. Assim, o algoritmo encontra a solução ótima em M.

Capítulo 2

Relacionamento com programação dinâmica

Embora tanto a programação dinâmica quanto os algoritmos gulosos trabalhem com problemas de otimização e com uma seqüência de passos, onde em cada passo é feita uma escolha, eles diferem claramente na *forma* em que esta escolha é realizada.

Enquanto na programação dinâmica a escolha pode depender das soluções para subproblemas, em algoritmos gulosos a escolha feita é a que parece ser a melhor no momento, não dependendo das soluções para os subproblemas (embora possa depender das escolhas até o momento).

Com isso, na programação dinâmica os subproblemas são resolvidos de baixo para cima, ou seja, parte-se de subproblemas menores para maiores. Por outro lado, os algoritmos gulosos progridem de cima para baixo, isto é, a instância de um problema é reduzida a cada iteração, através da realização de escolhas gulosas.

Tendo em vista que o tema de um dos seminários apresentados foi “programação dinâmica”, e que este tema está fortemente relacionado a algoritmos gulosos, julgou-se pertinente fazer uma abordagem envolvendo os dois temas conjuntamente.

Desta forma, essa seção irá apresentar um problema, sua solução utilizando programação dinâmica e a conversão desta solução para uma solução gulosa. Embora de maneira geral nem todas as etapas que serão abordadas a seguir são necessariamente utilizadas para desenvolver um algoritmo guloso, elas ilustram de forma clara o relacionamento entre programação dinâmica e algoritmos gulosos [3].

Como a programação dinâmica e os algoritmos gulosos possuem algumas características em comum (propriedade de subestrutura ótima, por exemplo), algumas vezes pode-se pensar em utilizar uma solução gulosa quando na verdade seria necessária uma solução de programação dinâmica, ou então a desenvolver uma solução de programação dinâmica quando uma solução mais simples utilizando algoritmos gulosos seria suficiente. Apresentaremos duas variações do problema da mochila para tentar ilustrar um caso onde esta “dúvida” acontece.

2.1. Um problema de seleção de atividade

Este problema consiste em programar um recurso entre diversas atividades concorrentes, mais especificamente, selecionar um subconjunto de tamanho máximo de atividades mutuamente compatíveis.

Seja $S = a_1, a_2, \ldots, a_n$ um conjunto n atividades que desejam utilizar um mesmo recurso, o qual pode ser utilizado por apenas uma atividade por vez. Cada atividade a_i terá associado um tempo de início (s_i) e um tempo de término (f_i), sendo que $0 \leq s_i < f_i < \infty$ (isto é, o tempo de início deve ser menor que o tempo de fim, e este por sua vez, deve ser finito).

Caso uma atividade a_i seja selecionada, ela irá ocorrer no intervalo de tempo $[s_i, f_i)$. Diz-se que duas atividades são compatíveis se o intervalo de tempo no qual uma delas ocorre não se sobrepõe ao intervalo de tempo da outra (considerando a restrição de que o recurso pode ser utilizado por apenas uma atividade por vez). Sendo assim, as atividades a_i e a_j são compatíveis se os intervalos $[s_i, f_i)$ e $[s_j, f_j)$ não se sobrepõem, ou seja, se $s_i \geq f_j$ (a atividade a_i inicia depois que a_j termina) ou então $s_j \geq f_i$ (a atividade a_j inicia depois que a_i termina).

Como exemplo, considere-se o conjunto S de atividades a seguir, o qual está ordenado de forma monotonicamente crescente de tempo de término:

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	8	9	10	11	12	13	14

No caso deste algoritmo específico, temos como conjunto de atividades mutuamente compatíveis, ou seja, que atendem à restrição explicitada anteriormente, o conjunto $\{a_3, a_9, a_{11}\}$. Este porém, não é um subconjunto máximo, pois pode-se obter dois conjuntos de atividades compatíveis com quatro elementos (subconjuntos máximos), sendo eles: $\{a_1, a_4, a_8, a_{11}\}$ e $\{a_2, a_4, a_9, a_{11}\}$.

2.1.1. Desenvolvendo uma solução utilizando programação dinâmica

A primeira coisa a ser feita é definir uma subestrutura ótima, e então utilizá-la para contruir uma solução ótima para o problema em questão, a partir de soluções ótimas para subproblemas (conforme foi explicado com detalhes no seminário sobre programação dinâmica).

Para tal, é necessária a definição de um espaço de subproblemas apropriado. Inicialmente, define-se conjuntos

$$S_{ij} = \{a_k \in S : f_i \leq s_k < f_k \leq s_j\} \tag{2.1}$$

sendo que S_{ij} é o conjunto de atividades que podem ser executadas entre o final da atividade a_i e o início da atividade a_j. Pode-se ver que tais atividades são compatíveis com as atividades a_i, a_j, as que não terminam depois de a_i terminar e as que não começam antes de a_j começar (esta observação é direta a partir da definição de atividades compatíveis, apresentada no início da seção 2.1 - os intervalos de tempo que as atividades estão executando não se superpõem).

Para representar o problema todo, são adicionadas "atividades fictícias" a_0 e a_{n+1}, e convenciona-se que $f_0 = 0$ e $s_{n+1} = \infty$. A partir disso, $S = s_{0,n+1}$ e os intervalos para i e j são dados por $0 \leq i, j \leq n+1$. Para que os intervalos de i e j sejam restringidos ainda mais, supõe-se que as atividades estão dispostas em ordem monotonicamente crescente de tempo de término (por isso tal suposição foi feita no exemplo dado anteriormente), isto é:

$$f_0 \leq f_1 \leq f_2 \leq \ldots \leq f_n < f_{n+1} \tag{2.2}$$

Com essa suposição, o espaço de subproblemas passa a ser selecionar um subconjunto máximo de atividades mutuamente compatíveis de S_{ij}, para $0 \leq i < j \leq n+1$, sendo que que $S_{ij} = \emptyset \quad \forall \quad i \geq j$.

Para mostrar que tal afirmação é verdadeira, vamos supor que exista uma atividade $a_k \in S_{ij}$ para algum $i \geq j$, de tal forma que na seqüência ordenada, a_j é seguido por a_i, ou seja, $f_j \leq f_i$. Entretanto, a partir da suposição que $i \geq j$ e de (2.1) tem-se que $f_i \leq s_k < f_k \leq s_j < f_j$, que contradiz a hipótese que a_i segue a_j na seqüência ordenada.

Agora, para "ver" a subestrutura do problema de seleção de atividades, considere um subproblema não vazio S_{ij} [1], e suponha que uma solução para este subproblema envolva a atividade a_k. No entanto, a utilização de a_k irá gerar dois subproblemas:

1. S_{ik} : conjunto de atividades que podem ser executadas entre o início da atividade a_i e o final da atividade a_k, isto é, que começam depois de a_i terminar e terminam antes de a_k começar.
2. S_{kj} : conjunto de atividades que podem ser executadas entre o final da atividade a_k e o início da atividade a_j.

onde cada uma das atividades é um subconjunto das atividades de S_{ij}. Sendo assim, o número de atividades da solução para S_{ij} é a soma do tamanho das soluções de S_{ik} e S_{kj}, mais uma unidade, correspondente à atividade a_k. Agora, falta (i) mostrar a subestrutura ótima e (ii) utilizá-la para mostrar que é possível construir uma solução ótima para o problema a partir de soluções ótimas para subproblemas.

(i) seja A_{ij} uma solução ótima para o problema S_{ij}, e suponha que tal solução envolva a atividade a_k. Esta suposição implica que as soluções A_{ik} e A_{kj} (para os problemas S_{ik} e S_{kj}, respectivamente)também devem ser ótimas. Aqui aplica-se o argumento de "recortar e colar" discutido no seminário sobre programação dinâmica.

(ii) subconjunto de tamanho máximo A_{ij} de atividades mutuamente compatíveis é definido como:

$$A_{ij} = A_{ik} \cup \{a_k\} \cup A_{kj} \tag{2.3}$$

Isso porque a partir das definições do item (i), pode-se afirmar que a solução geral A_{ij} (conjuntos de tamanho máximo de atividades mutuamente compatíveis em S_{ij}) pode ser obtida a partir da divisão do problema principal em dois subproblemas, e posterior busca dos conjuntos de tamanho máximo para cada um destes (A_{ik} e A_{kj}).

Uma solução recursiva

No desenvolvimento de uma solução de programação dinâmica, o segundo passo consiste na definição recursiva do valor de uma solução ótima. Seja $c[i, j]$ o número de atividades no subconjunto S_{ij} (que contém o número máximo de atividades mutuamente compatíveis com i, j).

Considerando um conjunto não vazio S_{ij} (lembrando: $S_{ij} = \emptyset \quad \forall \quad i \geq j$) e considerando que a utilização de uma atividade a_k implicará em outros dois subconjuntos S_{ik} e S_{kj}, a partir de (3) temos que:

[1] embora Sij às vezes seja tratado como conjunto de atividades e outras vezes como subproblema, procurou-se deixar claro no texto quando estamos nos referindo à cada caso.

$$c[i,j] = c[i,k] + c[k,j] + 1$$

ou seja, o número de atividades no subconjunto S_{ij} é o número de atividades em S_{ik} (denotado por $c[i,k]$) , o número de atividades em S_{kj} ($c[k,j]$) mais a atividade a_k.

O problema com essa equação recursiva é que o valor de k não é conhecido, sendo que sabe-se apenas que existem $j - i - 1$ valores possíveis para k ($k = i + 1, \ldots, j - 1$). Mas levando-se em consideração que o conjunto S_{ik} deve usar um destes valores para k, toma-se o melhor deles. Sendo assim, a definição recursiva completa é

$$c[ij] = \begin{cases} 0, & \text{se } S_{ij} = \emptyset; \\ \max_{i \leq k < j}\{\text{c[i,k]+ c[k,j] + 1}\}, & \text{se } S_{ij} \neq \emptyset. \end{cases} \quad (2.4)$$

2.1.2. Convertendo uma solução de programação dinâmica em uma solução gulosa

A partir da recorrência (2.4) é possível escrever um algoritmo de programação dinâmica (isto fica como exercício. . .). Mas com base em algumas observações, pode-se simplificar a solução. Considere o seguinte teorema:

Teorema 1: Considere qualquer problema não vazio S_{ij}, e seja a_m a atividade em S_{ij} com tempo de término mais antigo (ou seja, a primeira atividade que terminou).

$$f_m = min\ \{\ f_k\ :\ a_k \in S_{ij}\ \}$$

Com base no Teorema 1, pode-se afirmar:

(i) a atividade a_m é utilizada em algum subconjunto de tamanho máximo de atividades mutuamente compatíveis de S_{ij};
Prova: Suponha que A_{ij} é um subconjunto de tamanho máximo de atividades mutuamente compatíveis com S_{ij}, ordenadas em ordem monotonicamente crescente de tempo de término. Seja a_k a primeira atividade em A_{ij}.
   (a) se $a_m = a_k$, prova-se direto que (i) é verdade;
   (b) caso contrário, constrói-se o subconjunto $A'_{ij} = \{a_k\} \cup \{a_m\}$, cujas atividades são disjuntas. Devido a A'_{ij} ter o mesmo número de elementos que A_{ij}, pode-se afirmar que A_{ij}é subconjunto de tamanho máximo de atividades mutuamente compatíveis de S_{ij} que inclui a atividade a_m.

(ii) o subproblema S_{im} é vazio, de forma que a escolha de a_m deixa o subproblema S_{mj} como o único que pode ser não vazio.
Prova: Suponha que S_{im} não é vazio. Então existe uma atividade a_k, tal que $f_i \leq s_k < f_k \leq s_m < f_m$. Tal condição também implica que a_k está em S_{ij}. Como a_k termina antes que a_m, a escolha de a_m como atividade com o tempo de término mais antigo não está correta (chegou-se a uma contradição). Desta forma, conclui-se que S_{im} é vazio.

Antes de "mostrar" porque o Teorema 1 é tão importante, vale ressaltar que a subestrutura ótima varia na quantidade:

- de subproblemas usados em uma solução ótima para o problema original;
- de escolhas possíveis para determinar quais os subproblemas que serão utilizados.

Na solução proposta anteriormente (utilizando programação dinâmica) para o problema de seleção de atividades, utilizaram-se dois subproblemas e haviam $j - i - 1$ escolhas para se resolver o subproblema S_{ij}.

No entanto, utilizando-se o Teorema 1, é possível reduzir tais quantidades, passando-se a utilizar apenas um subproblema (pois é possível garantir que o outro subproblema é vazio) e também passando-se a ter apenas uma escolha para resolver S_{ij} (que é a atividade com tempo mais antigo em S_{ij}, a qual é de fácil determinação).

Além disso, o Teorema 1 possibilita resolver o problema através de uma abordagem *top-down*, e vez de fazê-lo *bottom-up* (abordagem esta utilizada em programação dinâmica, conforme exposto no seminário relativo a este tema). Sendo assim, para resolver o subproblema S_{ij}, primeiro escolhe-se a atividade com tempo de término mais antigo (a_m), e então resolve-se S_{mj} (anteriormente, resolvia-se S_{mj} antes).

Escolhendo-se sempre a atividade com tempo de término mais antigo, se faz uma escolha gulosa. É possível, então, propor um algoritmo guloso. Primeiro será discutida uma abordagem recursiva e depois se converterá este algoritmo para uma abordagem iterativa.

Algoritmo guloso recursivo

O algoritmo é dado como:

Algoritmo 4 Solução recursiva para o problema de seleção de atividades

```
1:  função SELECIONA_ATIVIDADE_RECURSIVO(s,f,i,j)   ▷ s e f são, respectivamente os tempos
        de início e término das atividades; e i e j são os índices iniciais do problema S_ij que deve ser
        resolvido
2:      m ← i + 1
3:      enquanto m < j e s_m < f_i faça              ▷ Encontrar a primeira atividade em S_ij
4:          m ← m + 1
5:      fim enquanto
6:      se m < j então
7:          retorne {a_m} ← SELECIONA_ATIVIDADE_RECURSIVO(s,f,m,j)
8:      senão
9:          retorne ∅
10:     fim se
11: fim função
```

Pode-se descrever o funcionamento do algoritmo como segue: o loop while é responsável por encontrar a primeira atividade de S_{ij}, isto é, a primeira atividade a_m que seja compatível com a_i (uma das condições de parada se deve ao fato que tal atividade tem $s_m \geq f_i$). O laço pode terminar por dois motivos:

1. quando $m \geq j$, não foi encontrada nenhuma atividade compatível com a_i entre as atividades que terminam antes que a_j. O retorno do procedimento é "vazio" (linha 9), pois $m < j$ e $S_{ij} = \emptyset$
2. se uma atividade compatível com a_i é encontrada, o procedimento irá retornar a união de $\{a_m\}$ com o subconjunto de tamanho máximo de S_{mj}. Este último subconjunto é retornado pela chamada recursiva SELECIONA_ATIVIDADE_RECURSIVO*(s,f,m,j)*.

Considerando que a chamada inicial é SELECIONA_ATIVIDADE_RECURSIVO$(s, f, 0, n+1)$, e supondo que as atividades estão ordenadas segundo o tempo de término de forma monotonicamente crescente, temos que o tempo de execução é $\Theta(n)$. Isto porque nas chamadas recursivas realizadas, examina-se uma vez cada atividade no loop while. A atividade a_k é examinada na última chamada feita em que $i < k$. Caso as atividades não estejam ordenadas, gasta-se tempo $O(n \log n)$ para tal.

Algoritmo guloso iterativo

Agora é apresentado um versão iterativa para o algoritmo recursivo exposto anteriormente. Este algoritmo também pressupõe que as atividades de entrada estejam ordenadas em ordem monotonicamente crescente por tempo de término.

Algoritmo 5 Solução iterativa para o problema de seleção de atividades

```
1:  função SELECIONA_ATIVIDADE_ITERATIVO(s,f)      ▷ s e f são, respectivamente os tempos de
        início e término das atividades
2:      n ← comprimento[s]
3:      A ← {1}
4:      para todo m ← 2 até n faça
5:          se s_m ≥ f_i então
6:              A ← A ∪ {a_m}
7:              i ← m
8:          fim se
9:      fim para
10:     retorne A
11: fim função
```

O fato de supor que as atividades estão ordenadas segundo seu tempo de término leva à seguinte afirmação:

$$f_i = max\ \{\ f_k\ :\ a_k \in A\ \} \tag{2.5}$$

isto é, f_i é o tempo de término máximo de qualquer atividade em A. O funcionamento do algoritmo se dá da seguinte forma:

- nas linhas 2 e 3, a atividade a_1 é selecionada e então incluída em A e indexada por i;
- o laço for (linhas 4 a 7) é responsável por selecionar a atividade mais antiga a terminar em $S_{i,n+1}$, denotada por a_m;
- na linha 5 verifica-se se a_m é compatível com as demais atividades já incluídas em A (a partir da definição (2.5)). Se sim, ela é adicionada a A e a variável i é definida como m (linhas 6 e 7).

Supondo que a entrada está ordenada como necessário, o procedimento SELECIONA_ATIVIDADE_ITERATIVO() tem complexidade $\Theta(n)$, assim como o algoritmo recursivo.

2.2. Mais um exemplo: o problema da mochila

Conforme exposto anteriormente neste trabalho a propriedade de subestrutura ótima é utilizada tanto pela programação dinâmica quanto pelos algoritmos gulosos. No entanto, este fato pode levar à tentativa de utilizar uma solução gulosa quando na verdade seria necessária uma solução de programação

dinâmica, ou então induzir a desenvolver uma solução de programação dinâmica quando uma solução mais simples utilizando algoritmos gulosos seria suficiente. O problema exposto a seguir possibilitará identificar as sutilezas de cada tipo de solução.

O problema da mochila é um problema clássico de otimização, e aqui serão abordadas duas variações: o problema da mochila 0-1 e o problema da mochila fracionária. Seja a seguinte configuração: um ladrão que rouba uma loja encontra n itens, onde cada item i vale v_i reais e w_i quilos, sendo v_i e w_i inteiros. O ladrão deseja levar a carga mais valiosa possível mas, no entanto, consegue carregar apenas W quilos na sua mochila (W também é um inteiro).

De uma maneira mais formal [2]:

$$\text{maximize} \quad \sum_{i=1}^{n} x_i \, v_i \quad \text{sujeito à} \quad \sum_{i=1}^{n} x_i \, w_i \leq W$$

onde a definição de x_i irá depender do problema sendo tratado. O próximo parágrafo definirá x_i para cada caso.

No problema da mochila 0-1, as restrições são que o ladrão tem que levar os itens inteiros (isto é, x_i assume apenas valor inteiro: 0 ou 1), e pode levar um mesmo item apenas uma vez. Já no problema da mochila fracionária, o ladrão pode levar frações de um item, e desta forma x_i pode assumir valores tal que $0 \leq x_i \leq 1$. Os dois problemas possuem uma subestrutura ótima (a prova desta afirmação fica como exercício...).

Pode-se utilizar uma estratégia gulosa para resolver o problema da mochila fracionária, mas não para resolver o problema da mochila 0-1. Seja o exemplo ilustrado na Figura 2.2 e com a seguinte configuração:

Item	Valor (R$)	Peso (Kg)	Valor por Kg
1	60	10	6
2	100	20	5
3	120	30	4

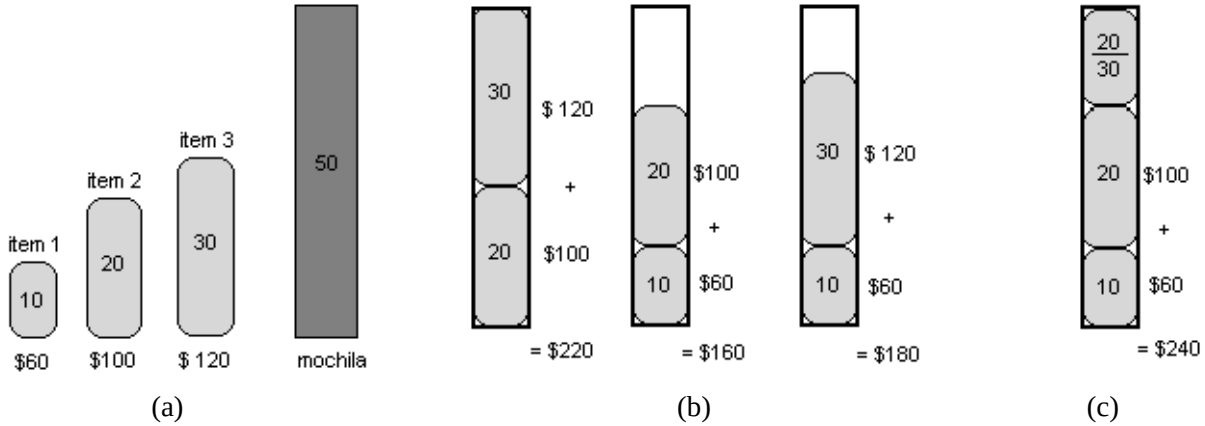

Figura 2.1: Problema da mochila

O peso que a mochila pode suportar é 50 Kg. A estratégia gulosa selecionaria primeiro o item 1 (considerando como critério de seleção escolher o item que possui o maior valor por quilo).

No entanto, conforme pode-se observar na Figura 2.2(b) a seguir, a escolha do item 1 não leva à uma solução ótima, independente dos itens escolhidos posteriormente (se o item 2 ou o item 3 fossem escolhidos a mochila não poderia ser preenchida totalmente). O problema da mochila 0-1 pode ser resolvido através de programação dinâmica.

No exemplo anterior, vale a pena observar que, com a mesma configuração, se a estratégia utilizada fosse a do problema da mochila fracionária o problema chegaria a uma solução ótima (Figura2.2(c)). No entanto, vamos nos aprofundar um pouco no problema da mochila fracionária, relacionando sua solução às características gerais dos algoritmos gulosos apresentadas na seção 1.1 e construindo um algoritmo guloso que resolva este problema.

Conforme exposto anteriormente, o problema consiste em

$$\text{maximize} \quad \sum_{i=1}^{n} x_i \, v_i \quad \text{sujeito à} \quad \sum_{i=1}^{n} x_i \, w_i \leq W$$

onde $v_i > 0$, $w_i > 0$ e $0 \leq x_i \leq 1$, para $1 \leq i \leq n$. As condições em v_i e w_i são restrições na instância e as condições em x_i são restrições na solução. Como candidatos temos os diferentes objetos e a solução é dada por um vetor $\{x_1, \ldots, x_n\}$ que indica que fração de cada objeto deve ser incluída. Uma solução é considerada viável se ela respeita as restrições anteriores. A função objetivo é o valor total dos objetos na mochila, e a função de seleção será definida posteriormente.

Em uma solução ótima, $\sum_{i=1}^{n} x_i w_i = W$. A estratégia consiste então em selecionar convenientemente objetos de tal forma a colocar a maior fração possível do objeto selecionado na mochila, e parar quando a mochila estiver totalmente cheia.

O algoritmo é dado a seguir.

Algoritmo 6 Problema da Mochila

```
1:  função MOCHILA(w[1..n], v[1..n], W)      ▷ w é o vetor de pesos, v é o vetor de valores e W é o peso máximo suportado pela mochila
2:      para todo i = 1 até n faça
3:          x[i] ← 0                          ▷ x é vetor contendo a fração de cada objeto que deve-se selecionar
4:      fim para
5:      peso ← 0                              ▷ peso é a soma dos pesos dos objetos selecionados até o momento
6:      enquanto peso < W faça                                              ▷ loop guloso
7:          i ←o melhor objeto restante       ▷ veja no texto que segue este algoritmo
8:          se peso +w[i] ≤ W então
9:              x[i] ← 1
10:             peso ← peso + w[i]
11:         senão
12:             x[i] ← (W − peso)/w[i]
13:             peso ← W
14:         fim se
15:     fim enquanto
16:     retorne x
17: fim função
```

Existem três funções de seleção que podem ser utilizadas:

1. escolher a cada estágio o objeto restante que possui o maior valor. O argumento utilizado é que desta forma o valor da carga aumenta tão rápido quanto possível;
2. escolher a cada estágio o objeto restante que possui o menor peso. O argumento utilizado é que desta forma a capacidade é utilizada tão lentamente quanto possível;
3. escolher o item cujo valor por peso seja o maior possível.

As duas primeiras alternativas não chegam necessariamente a soluções ótimas. Um exemplo de configuração de problema em que isto acontece é o seguinte:

w	10	20	30	40	50
v	20	30	66	40	60
v/w	2.0	1.5	2.2	1.0	1.2

Selecione	x_i					Valor
Max v_i	0	0	1	0.5	1	146
Min w_i	1	1	1	1	0	156
Max v_i / w_i	1	1	1	0	0.8	164

Neste exemplo, a terceira abordagem produziu uma solução ótima. De fato, isso é sempre verdade, ou seja, a terceira abordagem sempre conduz a uma solução ótima para o problema da mochila fracionária, conforme provado a seguir.

2.2.1. Prova de corretude

Suponha que os objetos estão ordenados em ordem decrescente de acordo com o valor por peso, isto é:

$$\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$$

Seja $X = \{x_1, \ldots, x_n\}$ a solução encontrada pelo algoritmo guloso. Caso todos os elementos de X possuam valor 1, a solução é ótima. Caso contrário, considere j como sendo o menor índice tal que o elemento x_j é menor que 1 (neste caso, $x_i = 1$ quando $i < j$; $x_i = 0$ quando $i > j$; e $\sum_{i=1}^{n} x_1 wi = W$). Seja $V(X) = \sum_{i=1}^{n} x_1 v_i$ o valor da solução de X.

Seja também $Y = \{y_1, \ldots, y_n\}$ uma solução viável qualquer, e desta forma, $\sum_{i=1}^{n} y_1 wi \leq W$) e portanto, $\sum_{i=1}^{n}(x_i - y_1)wi \geq 0$. Considerando $V(Y) = \sum_{i=1}^{n} y_1 v_i$ o valor da solução de Y.

Tem-se então que:

$$V(X) - V(Y) = \sum_{i=1}^{n}(x_i - y_i)v_i = \sum_{i=1}^{n}(x_i - y_i)w_i\frac{v_i}{w_i}$$

Quando $i < j$, $x_i = 1$ e então $x_i - y_i$ é maior ou igual a zero, enquanto $v_i/w_i \geq v_j/w_j$. Quando $i > j$, $x_i = 0$ e então $x_i - y_i$ é menor ou igual a zero, enquanto $v_i/w_i \leq v_j/w_j$. Por fim, quando $i = j$, $v_i/w_i = v_j/w_j$. Assim, em todo caso $(x_i - y_i)(v_i/w_i) \geq (x_j - y_j)(v_j/w_j)$. Portanto,

$$V(X) - V(Y) \geq (v_j/w_j)\sum_{i=1}^{n}(x_i - y_i)wi \geq 0$$

Com isso, provou-se que nenhuma solução viável possui um valor maior que $V(X)$, e sendo assim, a solução X é ótima.

2.2.2. Complexidade

A implementação do algoritmo pode ser considerada simples. Considerando que o custo mínimo para ordenar os objetos em ordem decrescente segundo seu valor por peso é $O(n \log n)$, e que o loop guloso executa n vezes (o que é facilmente observado no algoritmo), temos que o tempo total de execução é, segundo a regra da soma da notação O, $O(max(n, n \log n)) = O(n \log n)$.

Uma abordagem que seria mais rápida quando fossem necessários poucos objetos para preencher a mochila seria a de manter os objetos em um *heap* com o maior valor de v_i/w_i na raiz. O custo para criar o heap é $O(n)$, enquanto cada ciclo do loop guloso custa $O(\log n)$. Vale ressaltar que a análise no pior caso não se altera.

Capítulo 3

Exemplos de problemas que utilizam uma estratégia gulosa

Neste capítulo apresentaremos mais dois exemplos de problemas, utilizando a estratégia gulosa. São eles os Códigos de Huffman e o Problema do Planejamento(*scheduling*). O primeiro está relacionado à compressão de dados e o segundo busca organizar tarefas de tal forma que o tempo médio que cada tarefa fica no sistema é minimizado (a definição de "tarefas" e "sistema" pode variar de um caso para outro. Mas você pode imaginar tarefas como pessoas em uma fila de banco e o sistema como o operador de caixa, por exemplo).

Optou-se por apresentar os problemas envolvendo grafos separadamente no capítulo seguinte.

3.1. Códigos de Huffman

A importância da compressão de dados é indiscutível. De forma simplificada pode-se dizer que ela está relacionada com maneiras de representar os dados originais em menos espaço, o que leva à vantagens tais como: menor espaço necessário para armazenamento, acesso e transmissão mais rápidos, entre outros [7]. Existe, claro um custo para tais vantagens, que é o custo de codificação e decodificação dos dados.

Um dos métodos de codificação mais conhecidos são os Códigos de Huffman, cuja idéia consiste em atribuir códigos de representação menores aos símbolos com maior freqüência. A seguir será apresentada a "idéia principal" do problema de compressão, bem como os Códigos de Huffman.

Suponha que deseje-se armazenar de forma compacta um arquivo de dados que possua 100.000 caracteres, cujas distribuições são dadas na primeira linha da Tabela 1 a seguir:

Tabela1

	a	b	c	d	e	f
Freqüência (em milhares)	45	13	12	16	9	5
Palavra de código de comprimento fixo	000	001	010	011	100	101
Palavra de código de tamanho variável	0	101	100	111	1101	1100

Existem diferentes formas de representar este arquivo. Considere o problema de projetar um código de caracteres binários (que iremos denominar código, por simplificação), no qual utiliza-se uma cadeia binária única para representar um caractere. Neste caso, são duas as abordagens:

- utilizar um código de comprimento fixo, isto é, utiliza-se uma quantidade fixa de bits para representar cada caractere. No exemplo em questão, seriam utilizados 3 bits para representar cada um dos seis caracteres (veja a segunda linha da Tabela 1). Como o arquivo possui 100.000 caracteres, seriam necessários 300.000 bits para codificá-lo.
- utilizar um código de comprimento variável, o qual possui um desempenho melhor que o item anterior. Nesta abordagem, caracteres que aparecem com maior freqüência são representados por uma cadeia de bits menor do que caracteres que aparecem pouco. No caso do exemplo, o caractere a que possui a maior freqüência (45) é representada por um único bit, enquanto o caractere f, que aparece apenas 5 vezes no texto é representado por 4 bits (veja a terceira linha da Tabela 1). Para representar o arquivo inicial seriam necessários 224.000 bits (somatório da freqüência de cada caractere "vezes" número de bits necessários para codificá-lo, no caso do exemplo, 45*1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4). Com isso, há uma redução de aproximadamente 25% do espaço necessário para armazenamento.

3.1.1. Códigos de prefixo

Com o objetivo de facilitar a codificação, serão considerados apenas códigos de prefixo, que são aqueles onde nenhum código é prefixo de alguma outra palavra de código (desta forma, não teremos casos onde existem no mesmo problema os códigos 010 e **010**1). É importante ressaltar que esta restrição não leva à uma perda de generalidade, pois a compressão de dados ótima que pode ser obtida por meio de um código de caracteres sempre pode ser alcançada com um código de prefixo.

Um código de caracteres binários é obtido através da concatenação das palavras de código que representam cada caractere. Exemplo: ao codificar os caracteres *abc* utilizando código de prefixo de tamanho variável, conforme dados da Tabela 1, teríamos $0 \cdot 101 \cdot 100 = 0101100$.

Na decodificação, como estamos utilizando códigos de prefixo de tamanho variável, sabemos que a palavra que começa um arquivo codificado não é ambígua (por exemplo: temos que o caractere *b* é representado por 101, e devido à utilização de códigos de prefixo, sabemos que nenhuma outra palavra começa com 101. Sendo assim, se um arquivo possui como primeiros bits 1, 0 e 1, tem-se que o primeiro caractere do arquivo só pode ser *b*). Desta forma, na decodificação pega-se a palavra inicial do arquivo, e então a decodifica, tira do arquivo e repete-se o processo com o arquivo resultante.

Entretanto, neste esquema de decodificação é preciso uma representação conveniente para o código de prefixo, a qual pode ser obtida através de uma árvore binária, cujas folhas representam os caracteres.

Uma palavra de código para um determinado caractere pode ser interpretado como o cami nho percorrido da raiz até o caractere, sendo que o caminho indo para o filho da esquerda é representado por 0 e para o filho da direita por 1.

Considere as árvores da Figura 3.1.1, onde a árvore (a) representa o código de comprimento fixo e (b) a árvore do código de comprimento variável, ambas para a configuração de problema da Tabela 1.

Na árvore (b) para chegar ao caractere 'd' percorremos o caminho 1, 1, 1, isto é, na árvore "caminha-se" para o filho da direita 3 vezes para alcançar a folha que representa o caractere 'd' (isso é facilmente observado na Figura anterior).

Um fato importante a ser considerado é que uma árvore binária completa (uma árvore onde cada no que não é folha tem dois filhos) corresponde a um código de prefixo ótimo. Observando novamente a Figura 3.1.1, vemos que a árvore (a) não é completa. Sendo assim, o código de prefixo

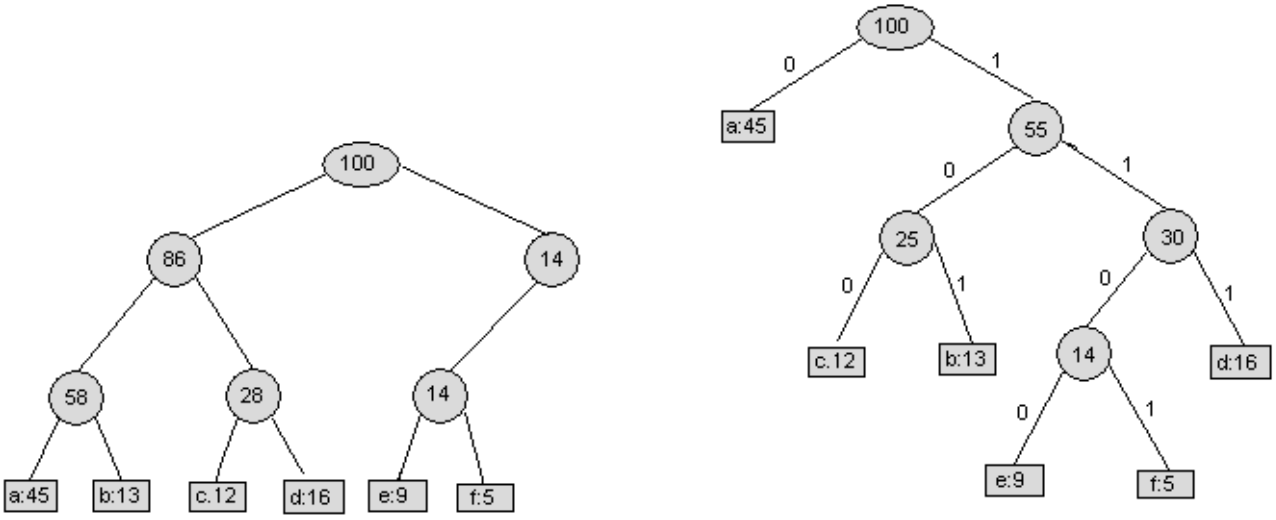

Figura 3.1: Árvores correspondentes aos esquemas de codificação da Tabela 1

de tamanho fixo do nosso problema não é ótimo. A árvore B por sua vez, é completa, e desta forma podemos afirmar que o código de prefixo de tamanho variável do nosso problema é ótimo.

Como estamos interessados em um código ótimo, podemos restringir a atenção somente às árvores binárias completas. Neste caso, considere C o alfabeto de onde os caracteres são obtidos e também considere que as freqüências de caracteres são positivas. Temos então que a árvore para um código de prefixo ótimo terá exatamente $|C|$ folhas (ou seja, uma folha para cada item do alfabeto), e $|C| - 1$ nós internos (sendo que o número de nós de grau 2 em qualquer árvore binária não vazia é uma unidade menor que o número de folhas. Isso pode ser provado por indução).

Pode ser observado que o tamanho da palavra de código para um determinado caractere é igual à profundidade deste caractere na árvore. Observe, pela Figura 3.1.1, que a profundidade da folha que representa o caracter *d* na árvore é três, que é exatamente o tamanho da palavra de código que representa este caractere (111). A freqüência que cada caractere aparece no arquivo também é conhecida. Com estes dados, podemos definir o custo de uma árvore que representa um código de prefixo, ou seja, podemos determinar facilmente quantos bits serão necessários para codificar o arquivo.

Sendo uma árvore T correspondente a um código de prefixo, $f(c)$ a freqüência do caractere c e $d_T(c)$ a profundidade da folha de c na árvore (tem-se $f(c)$ e c para todo $c \in C$). Então o custo da árvore T é:

$$B(T) = \sum_{c \in C} f(c)\ d_T(c) \tag{3.1}$$

3.1.2. A construção de um código de Huffman

A seguir será apresentado um algoritmo do Código de Huffman, que é um guloso que produz um código de prefixo ótimo, e após será provada sua correção (baseando-se na propriedade de escolha gulosa e subestrutura ótima, que foram abordadas na seção 1.2).

Algoritmo 7 Códigos de Huffman

```
1:  função HUFFMAN(C)
2:      n ← |C|
3:      Q ← C
4:      para todo i = 1 até n − 1 faça
5:          alocar um novo nó z
6:          esquerda[z] ← x ← EXTRAI_MIN(Q)
7:          direita[z] ← y ← EXTRAI_MIN(Q)
8:          f[z] ← f[x] + f[y]
9:          INSIRA(Q, z)
10:     fim para
11:     retorne EXTRAI_MIN(Q)
12: fim função
```

O funcionamento do algoritmo se dá da seguinte forma: a fila de prioridades Q é inicializada na linha 2, recebendo os elementos de C. De forma geral, o objetivo d laço for nas linhas 4-10 é substituir, na fila de prioridades, os dois nós de mais baixa freqüência (nós x e y, no caso do algoritmo), por um novo nó (z) que representa sua intercalação.

O novo nó z possui como filhos x e y (é indiferente qual deles é filho da esquerda e qual é filho da direita, pois independentemente da ordem o código terá o mesmo custo), e como frequência a soma das freqüências de x e y. Após $n - 1$ intercalações, retorna-se a raiz da árvore na linha 9.

O funcionamento do algoritmo é descrito a seguir na Figura 3.2 (próxima página).

3.1.3. Prova de corretude

Conforme exposto, para provar a correção do algoritmo, será mostrado que o problema de determinar um código de prefixo ótimo exibe as propriedades de escolha gulosa e de subestrutura ótima. Sejam os dois lemas seguintes (o primeiro mostra que a propriedade de escolha gulosa é válida e o segundo que o problema de construir códigos de prefixo ótimos tem a propriedade de subestrutura ótima).

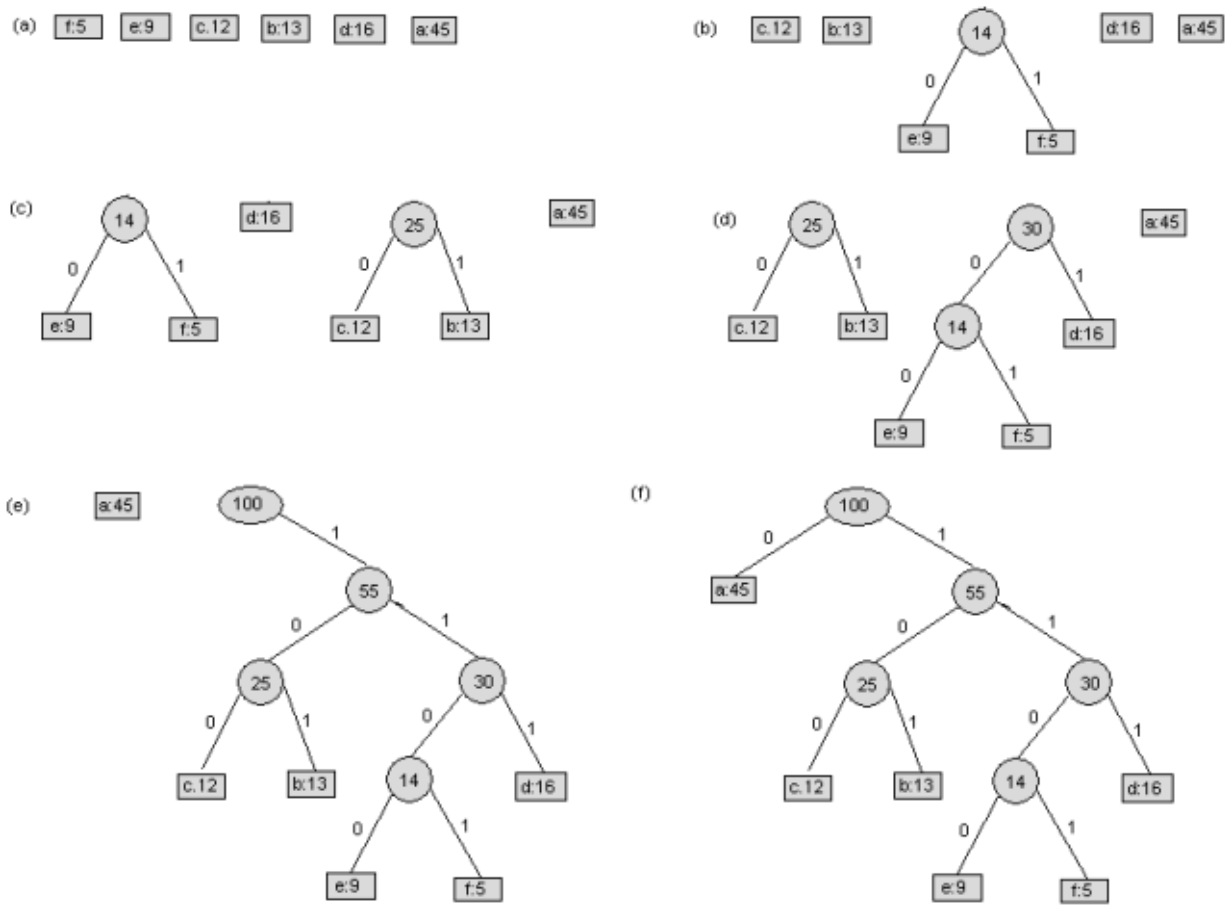

Figura 3.2: Funcionamento do algoritmo de Huffman

Lema 1: Seja C um alfabeto em que cada caractere $c \in C$ tem freqüência $f[c]$. Sejam x e y dois caracteres em C que tem as freqüências mais baixas. Então, existe um código de prefixo ótimo para c no qual as palavras de código para x e y têm o mesmo comprimento e diferem apenas no último bit.

Prova do lema 1: a idéia é modificar uma árvore que possua um código de prefixo ótimo em uma árvore que possua um outro código de prefixo ótimo, de tal forma que os caracteres x e y apareçam como folhas irmãs na profundidade máxima da nova árvore. Se isso for feito, as palavras de código para x e y terão o mesmo comprimento e irão diferir apenas no último bit.

Considere a Figura 3.3 a seguir.

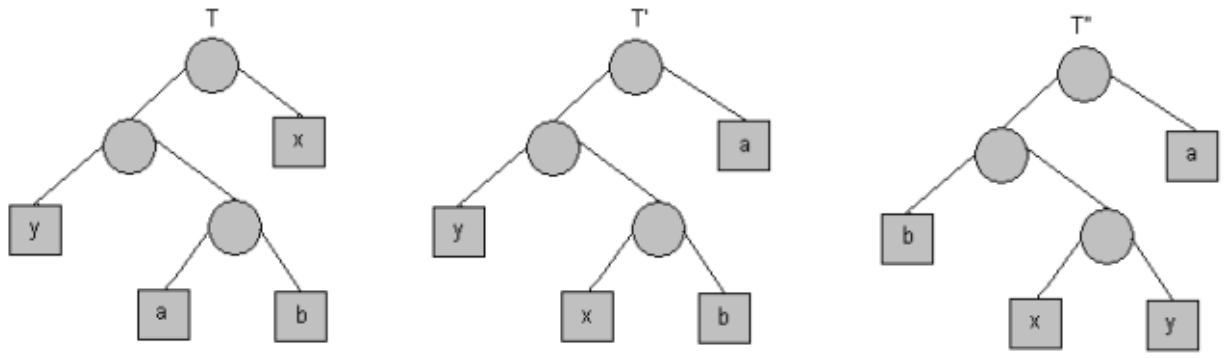

Figura 3.3: Passo chave do lema 2

Na árvore T, os caracteres a e b representam folhas irmãs de profundidade máxima. Suponha que $f[a] \leq f[b]$ e $f[x] \leq f[y]$ (isso não implica em perda de generalidade). Disso decorre que $f[x] \leq f[a]$ e $f[y] \leq f[b]$ (sendo que $f[x]$ e $f[y]$ são as duas freqüências mais baixas de folhas).

A árvore T' é resultado da permutação de a com x em T, e a árvore T'' é resultado da troca de b com y em T'. Pela equação (3.1) a diferença de custo entre T e T' é ≥ 0, assim como a diferença de custo entre T' e T'' também é ≥ 0 (isso pode ser facilmente verificado utilizando a equação (3.1). Dica: observe que a profundidade de x em T' é igual à profundidade de a em T, e vice-versa).

Então, temos que $B(T') \leq B(T)$ e, como T é ótima, $B(T) \leq B(T'')$, o que implica que $B(T'') = B(T)$. Desta forma, T'' é uma árvore ótima na qual os caracteres x e y aparecem como folhas de profundidade máxima, e a partir disso decorre o Lema 1.

Lema 2: Seja C um dado alfabeto com freqüência $f[c]$ definida para cada caractere $c \in C$. Sejam dois caracteres x e y em C com freqüência mínima. Seja C' o alfabeto C com os caracteres x, y removidos e o (novo) caractere z adicionado, de forma que $C' = C - \{x - y\} \cup \{z\}$; defina f para C' como para C, exceto pelo fato de que $f[z] = f[x] + f[y]$. Seja T' qualquer árvore representando um código de prefixo ótimo para o alfabeto C'. Então a árvore T, obtida a partir de T' pela substituição do nó de folha para z por um nó interno que tem x e y como filhos, representa um código de prefixo ótimo para o alfabeto C.

Prova do lema 2: o primeiro passo é mostrar que o custo $B(T)$ da árvore T pode ser expresso em termos do custo $B'(T)$ da árvore T'. Tem-se que para todo $c \in C - \{x, y\}$, $d_T(c) = d_{T'}(c)$, e desta forma $f[c]\ d_T(c) = f[c]\ d_{T'}(c)$. Com menos formalidade, para todos os caracteres c exceto para x e y, a sua profundidade na árvore T é igual à sua profundidade em T'. Em resumo, tirando-se os caracteres x e y de T e o caractere z de T', os custos das duas árvores são iguais.

A profundidade de x e y em T é igual à profundidade de $z+1$ em T', isto é: $d_T(x) = d_T(y) = d_{T'}(z) + 1$. Assim, temos que:

$$\begin{aligned} f[x]d_T(x) + f[y]d_T(y) &= (f[x] + f[y])\ (d_{T'}(z) + 1) \\ &= f[z]d_{T'}(z) + f[x] + f[y] \end{aligned}$$

Pode-se concluir então que

$$B(T) = B(T') + f[x] + f[y]$$

ou, equivalentemente,

$$B(T') = B(T) - f[x] - f[y]$$

A partir de tais fatos, podemos provar o lema por contradição. Vamos supor que a árvore T não represente um código de prefixo ótimo para C. Então existe uma árvore T'' tal que $B(T'') < B(T)$. Pelo lema 1, T'' tem x e y como irmãos. Por fim, seja T''' a árvore T'' com o pai comum de x e y substituído por uma folha z com freqüência $f[z] = f[x] + f[y]$. Assim,

$$\begin{aligned} B(T''') &= B(T'') - f[x] - f[y] \\ &< B(T) - f[x] - f[y] \\ &= B(T') \end{aligned}$$

Neste ponto chegou-se a uma contradição para a hipótese (que T' representa um código de prefixo ótimo para C'), e desta forma T deve representar um código de prefixo ótimo para C.

A partir dos lemas 1 e 2, a prova do teorema a seguir é imediata:

Teorema 2: O procedimento HUFFMAN produz um código de prefixo ótimo.

Em [7], p. 327, é apresentada a compressão de Huffman utilizando palavras, que possui as mesmas idéias básicas que o caso apresentado neste trabalho: cada palavra diferente do texto é considerado um símbolo, suas respectivas freqüências são determinadas e então um código de Huffman é gerado para representar as palavras.

3.1.4. Complexidade

Na análise da complexidade do algoritmo 7, suponha que a fila Q é implementada como um heap binário (definido como um objeto arranjo que pode ser visto como uma árvore binária praticamente completa. Mais detalhes podem ser obtidos em [3], p. 103). A inicialização da linha 3 pode ser realizada em $O(n)$ (veja a seção 6.3 em [3], p. 107).

O loop for contribui $O(n \log n)$ no tempo de execução, sendo que o loop é executado $n - 1$ vezes e cada operação de heap exige tempo $O(\log n)$. Sendo assim o tempo total de execução do algoritmo para uma entrada de tamanho n é $O(n \log n)$.

3.2. Planejamento (*scheduling*)

A idéia é organizar tarefas de tal forma que o tempo médio que cada tarefa fica no sistema é minimizado. Um problema mais complexo é aquele onde cada tarefa possui um prazo de término, e o lucro depender do término da tarefa no prazo. Neste problema, o objetivo é maximizar o lucro (este problema é apresentado em [2], p.207).

No nosso exemplo, vamos pensar em um sistema como sendo um caixa de banco, e em uma tarefa como um cliente. O objetivo é então minimizar o tempo que cada cliente espera desde quando

entra no banco até o momento em que termina de ser atendido. Formalizando um pouco, temos n clientes, sendo que cada cliente i irá exigir um tempo de serviço t_i ($1 \leq i \leq n$), e queremos minimizar o tempo médio que cada cliente i gasta no sistema.

Como n (número de consumidores) é fixo, o problema é minimizar o tempo total gasto no sistema por todos os clientes, isto é:

$$T = \sum_{i=1}^{n} (\text{tempo no sistema gasto pelo consumidor } i)$$

Agora um exemplo prático: suponha que existam três clientes, com $t_1 = 5$, $t_2 = 10$ e $t_3 = 3$. As possíveis ordens que os clientes podem ser atendidos são:

Ordem	T
123	5 + (5 + 10) + (5 + 10 + 3) = 38
132	5 + (5 + 3) + (5 + 3 + 10) = 31
213	10 + (10 + 5) + (10 + 5 + 3) = 43
231	10 + (10 + 3) + (10 + 3 + 5) = 41
312	3 + (3 + 5) + (3 + 5 + 10) = 29 (solução ótima)
321	3 + (3 + 10) + (3 + 10 + 5) = 34

A interpretação é simples: a primeira linha indica que o cliente 1 foi atendido imediatamente, o cliente 2 precisou esperar pelo cliente 1 e o cliente 3 precisou esperar os dois anteriores serem atendidos.

Pode-se observar que a configuração da solução ótima encontrada foi a seguinte: atende-se os clientes segundo o tempo que eles utilização o sistema, isto é, quem "gasta" menos tempo é atendido antes.

Para provar que esta configuração resulta em um planejamento ótimo, pense em um algoritmo guloso que constrói o planejamento ótimo item a item. Suponha que após os clientes $i_1, i_2, \ldots, i_m$ tivessem suas tarefas agendadas, fosse adicionado o cliente j. O tempo em T neste momento é o tempo necessário para os clientes $i_1, i_2, \ldots, i_m$, mais t_j, que é o tempo necessário para o novo cliente j.

Como um algoritmo guloso nunca revê as escolhas já realizadas (ou seja, ele não vai alterar a configuração feita para os clientes $i_1, i_2, \ldots, i_m$), o que podemos fazer é minimizar t_j. Desta forma, o algoritmo guloso se torna simples, pois a cada passo seleciona-se entre os clientes que restam aquele que necessita de menos tempo, e coloca-o no final da "lista de agendamentos".

3.2.1. Prova de corretude

Agora é preciso provar que este algoritmo guloso é ótimo. Considere então $P = p_1 p_2 \ldots p_n$ como sendo qualquer permutação de inteiros de 1 até n, e seja $s_i = T_{pi}$. Se os clientes são atendidos na ordem P, então o i-ésimo cliente necessitará de um tempo s_i para ser atendido. O tempo gasto por todos os clientes é então:

$$\begin{aligned} T(P) &= s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \ldots \\ &= n s_1 + (n-1) s_2 + (n-3) s_3 + \ldots \end{aligned}$$

$$= \sum_{k=1}^{n} (n-k+1)s_k$$

Suponha agora que P não esteja organizado de forma crescente segundo os tempos de serviço. Então podem existir dois inteiros a e b, tal que $a < b$ e $s_a > s_b$, isto é, o a-ésimo cliente é atendido antes mesmo exigindo mais tempo que o b-ésimo cliente. Observe a Figura 3.4.

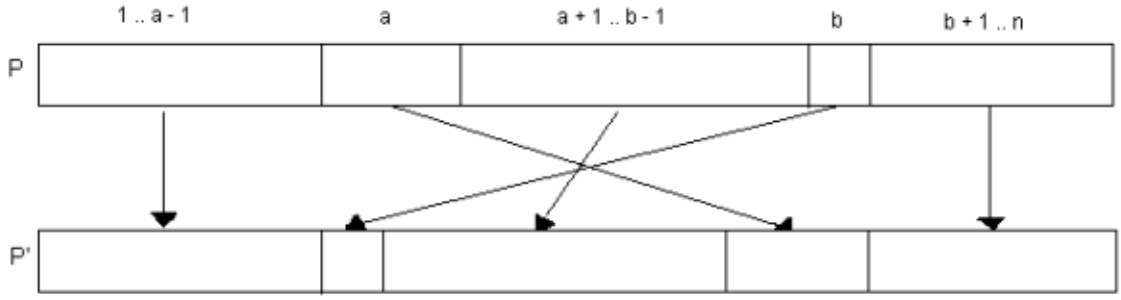

Figura 3.4: Trocando dois clientes de lugar

O que fazemos é trocar a ordem do cliente a com a do cliente b, o que leva o cliente que gasta menos tempo a ser atendido antes. Com essa troca, temos uma nova ordem de configuração de serviços, que denominaremos P'. Caso P' seja utilizado como configuração, o tempo total gasto no sistema por todos os clientes é:

$$(n-a+1)s_b + (n-b+1)s_a + \sum_{k=1;\ k\neq a,b}^{n} (n-k+1)s_k$$

Se fizermos a diferença entre P e P', temos que:

$$\begin{aligned} T(P) - T(P') &= (n-a+1)(s_a - s_b) + (n-b+1)(s_b - s_a) \\ &= (b-a)(sa - sb) > 0 \end{aligned}$$

Pode-se observar então que a configuração P' é melhor, pois possui um custo menor que P. O mesmo raciocínio pode ser realizado através da observação da Figura 3.4. Comparando as configurações P e P', vemos que os tempos de saído dos $a-1$ primeiros e os $n-b$ últimos clientes não são alterados mudando de uma configuração para outra.

Na configuração P' pode-se observar que os clientes nas posições $a+1 \ldots b-1$ deixam o sistema antes, sendo que o cliente b gasta menos tempo que o cliente a. Em resumo: como b deixa de utilizar o sistema antes, os demais clientes (neste caso, os clientes das posições a+1 ... b-1) também são atendidos antes.

De acordo com o que foi exposto, pode-se ver que é possível melhorar o funciomento de sistemas que possuam atividades que demoram mais tempo executando antes que atividades que demandam menos tempo. Com isso, os únicos planejamentos que restam são aqueles onde as tarefas

(ou no nosso exemplo, os clientes) estão organizadas de acordo com os tempos demandados para realização do serviço, e desta forma também são ótimos.

3.2.2. Complexidade

Na implementação do algoritmo para realizar tal tarefa, o essencial é ordenar os clientes em ordem crescente de acordo com os tempos de serviço, o que custa no mínimo $O(n \log n)$. Escrever o algoritmo fica como exercício.

Capítulo 4

Algoritmos Gulosos em Grafos

Em seguida, são apresentados dois algoritmos para se descobrir a árvore geradora mínima em um grafo conexo e ponderado, um problema que cobre muitos casos de interesse prático como em redes de computadores.

Finalmente, é apresentado um algoritmo para se computar o caminho mínimo em um grafo. Este problema também é de muito interesse em várias áreas da computação como a de otimização de projeto de circuitos lógicos.

O Anexo 1 apresenta algumas definições relativas a grafos. Recomendamos que você o leia antes de continuar este capítulo, pois será possível relembrar alguns conceitos.

4.1. Árvore geradora mínima

Supondo um grafo valorado, isto é, onde a cada aresta é atribuído um peso, podemos distinguir as árvores geradoras desse grafo em relação à soma total dos pesos. Temos aqui um problema interessante: como identificar a árvore que minimiza essa soma? Vamos chamar esta árvore de árvore geradora mínima de tal grafo.

Para identificar a árvore geradora mínima de um grafo, existem dois algoritmos bastante conhecidos *Kruskal* e *Prim* que apresentam a propriedade gulosa e resolvem o problema.

Esses algoritmos podem ser usados para se definir a topologia mínima e de custo mínimo de uma rede durante o projeto da mesma. Os pesos, nesse caso, poderiam ser uma função que relacionasse as distâncias entre os roteadores e o custo do cabeamento.

Um outro uso do destes algoritmos, seria usá-los em conjunto com algoritmos de roteamento dinâmicos. Tais algoritmos recalculam as rotas entre os roteadores da rede sempre que há necessidade. Por exemplo, quando o tráfego aumenta muito entres dois roteadores (modificando os pesos das arestas do grafo da rede) ou quando uma falha física elimina uma ou mais rotas (modificando o grafo da rede). Nesses casos, a árvore de menor custo indicaria a melhor rota que conectasse todos os roteadores. “Melhor”, neste caso leva em consideração a velocidade ou o *throughput* da rede.

Seja $G = (V, E)$ um grafo conexo não direcionado e valorado. O problema consiste em achar um subconjunto A de E, tal que A forme uma árvore e a soma dos pesos é a menor possível. Aplicando o algoritmo *Guloso* nesse caso, teremos as seguintes definições:

- O conjunto S de candidatos é um conjunto de arestas distintas de G.
- O conjunto S é viável se ele não contém nenhum circuito.
- O conjunto S é uma solução se ele contém todos os vértices de G.
- Um conjunto S de arestas é promissor se ele pode ser completado para produzir uma solução ótima. Em particular, o conjunto vazio é promissor.
- Uma aresta toca um conjunto S de vértices se exatamente um vértice ligado por essa aresta existe em S.

4.1.1. Algoritmo de Kruskal

Funcionamento do algoritmo

Seja $G = (V, E)$ um grafo de n vértices. No início, A é vazio, e supomos um grafo nulo composto dos n vértices (isto é, um grafo de n vértices isolados). O conjunto B é o conjunto das arestas de G. Selecionamos a aresta de B que tem o menor valor. Se ela conecta dois componentes diferentes, colocamo-la no conjunto A e juntamos os dois componentes para formar um novo componente. Se ela liga dois vértices de um mesmo componente, ela é rejeitada, dado que isso formaria um circuito. Continuamos assim até a obtenção de um único componente. Nesse caso, o conjunto A constitui uma solução.

Para ilustrar o funcionamento do algoritmo considere o grafo da Figura 4.1. O pseudocódigo de Kruskal é apresentado no algoritmo 8 e está baseado em [3]. O funcionamento passo a passo é mostrado na figura 4.2.

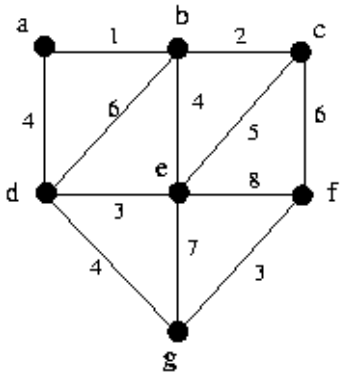

Figura 4.1: Grafo do qual se deseja a árvore geradora mínima

Algoritmo 8 AGM Kruskal

```
1:  função KRUSKAL(G)        ▷ Entrada: G = (V, E), custo : E → ℜ+. Saída: A (AGM)
2:      A ← ∅                                              ▷ Conjunto de arestas de A
3:      C ← ∅                                              ▷ Conjunto de componentes
4:      para todo v ∈ V faça
5:          C ← C + {v}
6:      fim para
7:      ordene as arestas de E em ordem não decrescente de peso
8:      para cada aresta {x, y} ∈ E, em ordem não decrescente faça
9:          seja c_x e c_y os conjuntos de x e de y em C
10:         se c_x ≠ c_y então
11:             A ← A + {x, y}
12:             C ← C − c_x − c_y + {(c_x ∪ c_y)}
13:         fim se
14:     fim para cada
15:     retorne A
16: fim função
```

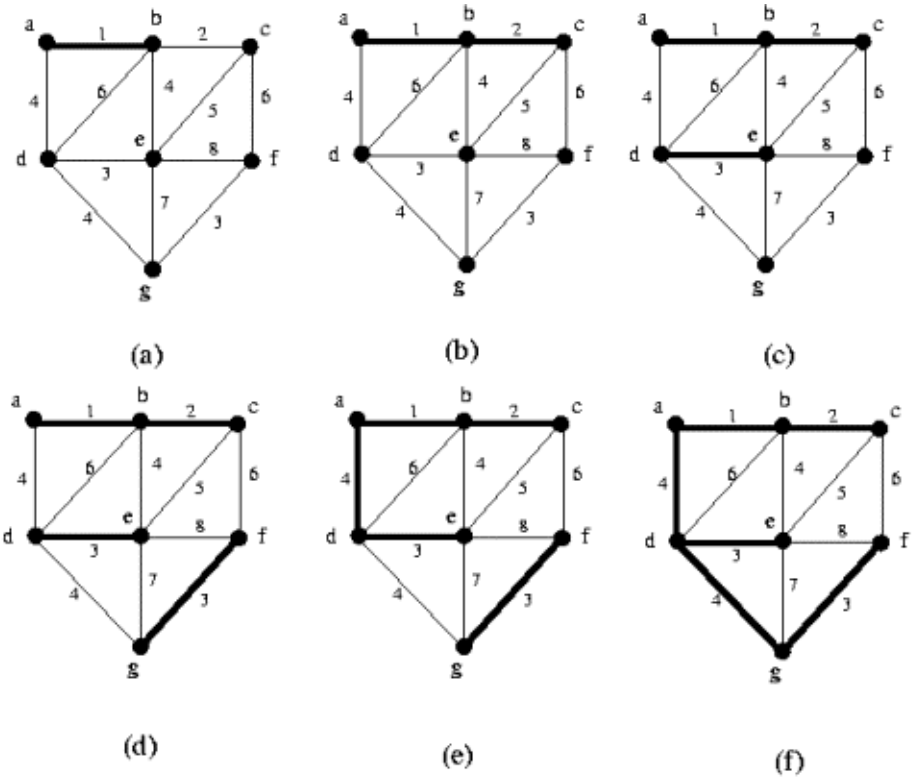

Figura 4.2: Kruskal: passo a passo

Na Figura 4.2, o conjunto de A de arestas ordenadas pelo seu peso igual é: $\{a, b\}$, $\{b, c\}$, $\{d, e\}$, $\{f, g\}$, $\{a, d\}$, $\{b, e\}$, $\{d, g\}$, $\{c, e\}$, $\{b, d\}$, $\{c, f\}$, $\{e, g\}$ e $\{e, f\}$. O algoritmo funciona como segue:

Passo	Aresta considerada	Componentes conectados
Inicialização	-	$\{a\}$ $\{b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$
1	$\{a, b\}$	$\{a, b\}$ $\{c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$
2	$\{b, c\}$	$\{a, b, c\}$ $\{d\}$ $\{e\}$ $\{f\}$ $\{g\}$
3	$\{d, e\}$	$\{a, b, c\}$ $\{d, e\}$ $\{f\}$ $\{g\}$
4	$\{f, g\}$	$\{a, b, c\}$ $\{d, e\}$ $\{f, g\}$
5	$\{a, d\}$	$\{a, b, c, d, e\}$ $\{f, g\}$
6	$\{b, e\}$	*Rejeitada*
7	$\{d, g\}$	$\{a, b, c, d, e, f, g\}$

Ao final do algoritmo, A contém as arestas escolhidas $\{a, b\}$, $\{b, c\}$, $\{d, e\}$, $\{f, g\}$ $\{a, d\}$ e $\{d, g\}$.

Prova de corretude

Teorema: o algoritmo de Kruskal encontra uma árvore geradora de peso mínimo.

Prova 1: A prova é por indução matemática no número de arestas no conjunto A. Nós precisamos mostrar que se A é promissor, i.e, pode nos levar à solução ótima, em qualquer estágio do algoritmo, então ele continuará sendo promissor quando uma aresta extra for adicionada. Quando o algoritmo pára, A é uma solução para nosso problema. Desde que A continue sendo promissor então a solução é ótima. **Base da indução**: O conjunto vazio é promissor pois o nosso grafo G é conexo e assim, por definição, existe uma solução. **Passo da indução**: Assuma que A é um conjunto promissor no exato momento em que iremos adicionar a aresta $e = \{u, v\}$. As arestas em A dividem os nós de G em dois ou mais componentes conexos; o nó u está em um desses componentes e v está em um componente diferente. Seja B o conjunto de nós no componente que inclui u. Temos,

- O conjunto B é um subconjunto estrito de nós de G. Isto é, ele não inclui v.
- A é um conjunto promissor de arestas tais que nenhuma aresta em A deixa B. Para uma aresta qualquer em A ou ela liga dois vértices em B ou ela nem está em B.
- e é a aresta de menor peso que deixa B (todas as arestas com peso menor, ou foram adicionada a A ou foram rejeitadas por estarem no mesmo componente conectado).

A partir das condições acima satisfeitas, podemos concluir que o conjunto $A \cup \{e\}$ também é promissor. Isto completa a nossa prova. Dado que A continua um conjunto promissor em qualquer estágio do algoritmo, quando o algoritmo pára A não é uma simples solução mas a solução ótima [2].

Prova 2: (por matróides). Vamos definir um conjunto de independência I de $G = (V, E)$ como $I = \{X \subseteq E : G[X]$ é um grafo acíclico$\}$. Então, provando que $M = (E, I)$ é um matróide, podemos usar o *teorema 1* para provar que o algoritmo de Kruskal devolve uma árvore de peso mínimo. Vamos provar a propriedade de troca dado que as outras duas são diretas.

Vamos provar esta propriedade por contradição. Seja A e B dois conjuntos de I tal que $|A| < |B|$. Suponha que não exista aresta em b tal que $(A + e) \in I$. Seja $G_1, \ldots, G_k$ o conjunto de componentes conexos de $G[A]$, cada grafo G_i com n_i vértices e $n_i - 1$ arestas. Então cada aresta de B liga vértices de um mesmo componente em $G[A]$. Como o número máximo de arestas de um grafo acíclico com n vértices é $n - 1$, a quantidade máxima de arestas de B em G_i é $n_i - 1$. Portanto,

$$|B| \leq (n_1 - 1) + (n_2 - 1) + \ldots + (n_k - 1) = |A|, \tag{4.1}$$

contrariando o fato de $|A| < |B|$ [3].

Prova 3: (por contradição). Suponha por absurdo que o algoritmo de Kruskal não encontra uma AGM (Árvore Geradora Mínima). Seja $K = \{e_1, e_2, \ldots, e_{n-1}\}$ as arestas selecionadas pelo algoritmo de Kruskal, nesta ordem. Seja i o maior índice tal que $\{e_1, e_2, \ldots, e_{i-1}\}$ pertence a alguma árvore geradora de custo mínimo, mas e_i não pertence, i.e, $\{e_1, e_2, \ldots, e_i\}$ não pertence a nenhuma AGM. Seja T uma AGM contendo $\{e_1, e_2, \ldots, e_{i-1}\}$. Vamos considerar o exato momento em que o algoritmo Kruskal insere a aresta e_i. Então $e_i + T$, contém um circuito, digamos C. Então deve existir uma outra aresta f de C que não está em K (pois caso todas fossem, K teria um circuito, contrariando o fato de K ser árvore). Certamente, o custo de f é pelo menos o custo de e_i, pois o algoritmo de Kruskal escolheu e_i exatamente porque é uma aresta de menor custo tal que $\{e_1, e_2, \ldots, e_{i-1}\} + e_i$ não contivesse circuito. Então, $T' = T + e_i - f$ é uma árvore geradora de custo menor ou igual a T. Como T é uma AGM, T' também deve ser. Isto contradiz o fato de $\{e_1, e_2, \ldots, e_i\}$ não pertencer a nenhuma AGM [3].

Complexidade

A complexidade do algoritmo de Kruskal depende da implementação utilizada para manipular conjuntos. Note que a ordenação dos valores de E consome $O(|E| \log |E|)$ [3].

• **Vetor de etiquetas.** Esta implementação usa um vetor, indexado pelos vértices de G, onde cada posição tem um valor (etiqueta) diferente inicialmente. Encontrar a etiqueta de um conjunto consome $O(1)$. Para fazer a união de dois conjunto podemos trocar todas as etiquetas de um dos conjuntos pelo do outro, gastando tempo $O(|V|)$. Como fazemos $(|V| - 1)$ uniões, temos uma implementação de complexidade de tempo $O(|E| \log |E| + |V|^2)$.

• **Estrutura para conjuntos disjuntos (*Union-Find*).** Considere a implementação da estrutura de dados de conjuntos disjuntos através da floresta de conjuntos disjuntos com as heurísticas de união por ordenação e compressão de caminho, pois essa é a implementação assintoticamente mais rápida conhecida. A inicialização do conjunto A pode ser feita em $O(1)$.

A inicialização dos conjuntos de componentes consome $O(|V|)$ pois cada conjunto terá 1 vértice e temos $|V|$ vértices. O *loop* da linha 8 do algoritmo executa $O(|E|)$ operações FIND e UNION sobre a floresta de conjuntos disjuntos. Juntamente com as $|V|$ operações de construção dos conjuntos de componentes, essas operações consomem o tempo $O((|V| + |E|)\alpha|V|)$ onde α é uma função de crescimento muito lento (inverso da função de *Ackermann*). Pelo fato de G ser supostamente conectada, temos $|E| \geq |V| - 1$, e assim, as operações sobre conjuntos disjuntos demoram tempo $O(|E|\alpha|V|)$. Além disso, tendo em vista que $\alpha|V| = O(\log |V|) = O(\log |E|)$, o tempo de execução total do algoritmo de Kruskal é $O(|E| \log |E|)$. Observando que $|E| < |V|^2$, tempos que $\log |E| = O(\log |V|)$, e portanto podemos redefinir o tempo de execução do algoritmo de Kruskal como $O(|E| \log |V|)$.

4.1.2. Algoritmo de Prim

A principal característica do algoritmo de Kruskal é que ele seleciona a melhor aresta sem se preocupar com a conexão das arestas selecionadas antes. O resultado é uma proliferação de árvores que eventualmente se juntam para formar uma árvore.

Já que sabemos que no final temos que produzir uma árvore só, por que não tentar fazer com que uma árvore cresça naturalmente até a obtenção da árvore geradora mínima? Assim, a próxima

aresta selecionada seria sempre uma que se conecta à arvore que já existe. Essa é a idéia do algoritmo de Prim.

A idéia do algoritmo consiste em começar uma árvore a partir de um vértice qualquer, e ir incrementando esta árvore com arestas, sempre mantendo-a acíclica e conexa. A próxima aresta a ser incluída deve ser uma de menor peso, daí porque este algoritmo também é guloso. A chave para se ter uma boa implementação para este algoritmo é fazer a busca da próxima aresta de forma eficiente.

O pseudocódigo de Prim é apresentado no algoritmo 9 e está baseado em [3] e [2].

Algoritmo 9 AGM Prim

```
1:  função PRIM(G, r)                ▷ Entrada: G = (V, E), w : E → ℜ+. Saída: A (AGM)
2:      Q ← V                ▷ O conjunto de vértices inseridos em A é V − Q. Inicialmente A é vazio
3:      para todo v ∈ Q faça
4:          peso(v) ← ∞
5:          pred(v) ← NULL
6:      fim para
7:      peso(v) ← 0
8:      enquanto Q ≠ ∅ faça
9:          u ← ExtraiMin(Q)                ▷ É inserido a aresta {u, pred(u)} em A
10:         para cada vértice v adjacente a u faça
11:             se v ∈ Q e w(u, v) < peso(v) então
12:                 pred(v) ← u
13:                 peso(v) ← w(u, v)                ▷ Decrementa o peso
14:             fim se
15:         fim para cada
16:     fim enquanto
17: fim função
```

Para entender o funcionamento do algoritmo considere novamente o grafo da Figura 4.1. O passo a passo do algoritmo é apresentado na Figura 4.3.

O algoritmo funciona como segue: arbitrariamente selecionamos o vértice a como inicial.

Passo	**Aresta considerada**	**Nós conectados**
Inicialização	-	$\{a\}$
1	$\{a, b\}$	$\{a, b\}$
2	$\{b, c\}$	$\{a, b, c\}$
3	$\{a, d\}$	$\{a, b, c, d\}$
4	$\{d, e\}$	$\{a, b, c, d, e\}$
5	$\{d, g\}$	$\{a, b, c, d, e, g\}$
6	$\{g, f\}$	$\{a, b, c, d, e, f, g\}$

O resultado A = $\{a, b\}$, $\{b, c\}$, $\{a, d\}$, $\{d, e\}$, $\{d, g\}$ e $\{g, f\}$.

Prova de corretude

Teorema: o algoritmo de Prim encontra uma árvore geradora de peso mínimo.

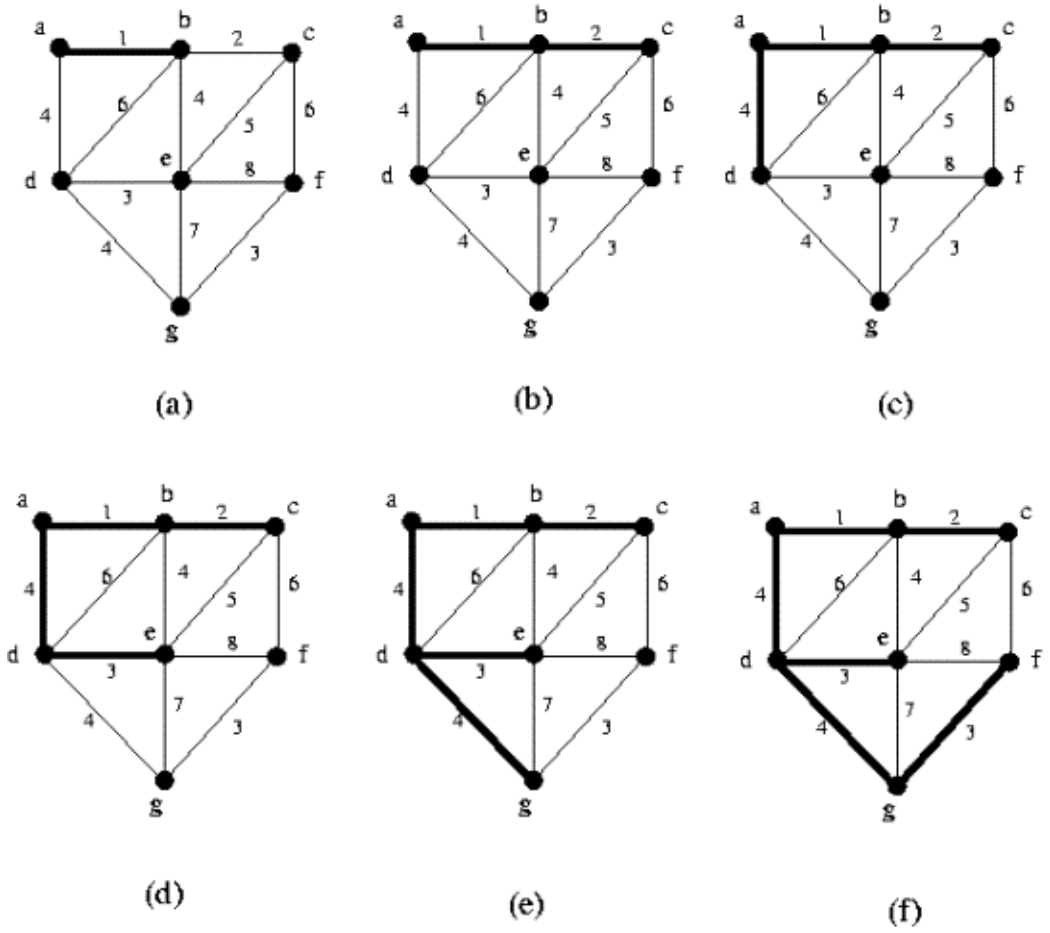

Figura 4.3: Prim: passo a passo

Prova 1: vamos supor que o teorema é válido para árvores construídas com até m arestas. Vamos provar que para árvores construídas com $m + 1$ arestas, o teorema também é verdadeiro. Vamos provar este passo por *contradição*: suponha que o teorema não seja válido para $m + 1$ arestas. Neste caso, seja $T' = (V', E')$ a árvore encontrada na iteração anterior, e a próxima aresta de menor peso ligando V' a $V - V'$, e T^* uma árvore ótima tal que T' é subgrafo de T^*. Assim, $T^* + e$ tem apenas um circuito. Certamente deve existir uma aresta $f \neq e$ ligando vértices em V' e $V - V'$ pois se ao ligar e criamos um circuito. Portanto $T^* + e - f$ também é uma árvore mínima, contrariando o fato de não existir árvore ótima englobando T' e e [3].

Prova 2: suponha que o teorema é falso. Então deve existir uma árvore T gerada ótima (note que uma árvore que é expandível para uma árvore de peso mínimo é a árvore contendo apenas o vértice r). Seja T uma menor árvore gerada que não é expandível para uma árvore ótima. Seja e a última aresta inserida em T e $T' = (T - e)$. Seja T^* uma expansão de T' para uma árvore ótima. Assim, $T^* + e$ tem um circuito. Remova a aresta f deste circuito, tal que $f \neq e$ e f tem apenas uma extremidade nos vértices de T'. Pela escolha de e, $T^* + e - f$ é uma AGM, contrariando a escolha de T [3].

Complexidade

A complexidade do algoritmo de Prim depende da implementação utilizada para manipular conjuntos [3].

• **Heap binário.** Podemos usar a rotina *Constrói-Heap* nos passos 1-5, com complexidade de tempo $O(|V|)$. A cada iteração do *loop* do passo 8, é removido um elemento de Q, e portanto o *loop* é iterado por $|V|$ vezes. Como o *ExtraiMin(Q)* é executado em tempo $O(\log |V|)$, o passo 9 é executado $O(|V| \log |V|)$. O *loop* do passo 8 juntamente com o *loop* do passo 10 percorre todas as arestas, visitando cada uma duas vezes, uma para cada extremidade. Portanto, os passos 11-13 são executados $O(|E|)$ vezes. Especificamente o passo 13 é uma operação de decrementação, que em um *heap binário* pode ser implementado em tempo $O(\log |V|)$. Finalmente, o tempo total desta implementação é $O(|V| \log |V| + |E| \log |V|)$, i.e, $O(|E| \log |V|)$.

Para exemplificar, suponha:

- V = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
- Peso = 9, 7, 10, 8, 14, 16, 4, 2, 3, 1
- PosHeap = 2, 5, 6, 4, 3, 1, 9, 8, 7, 10

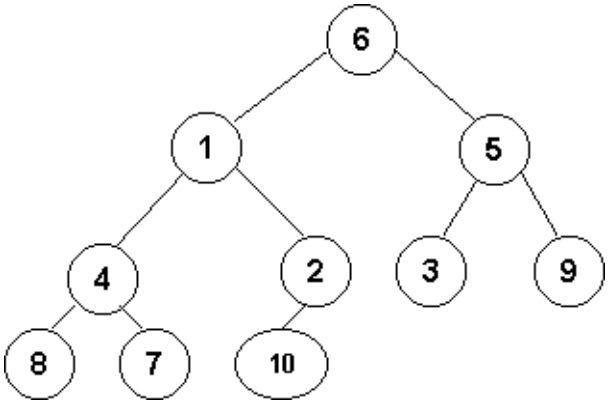

Figura 4.4: Estrutura de dados *heap*

• **Árvores balanceadas.** Outra estrutura que também pode ser utilizada é a estrutura de árvore AVL. Nesta árvore, temos a construção da árvore em $O(n \log n)$ e qualquer consulta, inserção ou remoção em $O(\log n)$. Portanto, a complexidade do algoritmo usando esta estrutura também é $O(|E| \log |V|)$.

4.2. Caminho mínimo grafo

Para representar o caminho mínimo de um vértice s até um vértice v, usaremos uma função $pred[]$. Esta função é responsável por dizer quem é o predecessor de um dado vértice. Deste modo, estando em um vértice y e tendo que $pred[x] = y$ sabemos que chegamos em x a partir de y.

O algoritmo que iremos considerar utiliza a técnica de relaxamento. Para cada vértice $u \in V$, manteremos um atributo $d[v]$ que será um limitante superior para um caminho mínimo de s a v. De fato, a cada momento, usando $pred$, teremos um caminho de peso $d[v]$. Os valores de $pred[v]$ e $d[v]$ devem ser inicializados pelo algoritmo 10. O relaxamento sobre uma aresta (u, v) é feito pelo algoritmo 11. O valor de $d[v]$ significa que há um caminho de s para v com peso $d[v]$.

Algoritmo 10 InicializeCaminhoMínimo

```
1: procedimento INICIALIZECAMINHOMÍNIMO(G, s)        ▷ Entrada: G = (V, E)
2:     para cada vértice v ∈ V faça
3:         d[v] ← ∞
4:         pred[v] ← NULL
5:     fim para cada
6:     d[s] ← 0
7: fim procedimento
```

Algoritmo 11 Relax

```
1: procedimento RELAX(u, v, w)   ▷ Entrada: vértice u e v e conjunto de pesos das arestas (w)
2:     se d[v] > d[u] + w(u, v) então
3:         d[v] > d[u] + w(u, v)
4:         pred[v] ← u
5:     fim se
6: fim procedimento
```

4.2.1. Algoritmo de Dijkstra

O algoritmo de Dijkstra é um algoritmo guloso para encontrar caminhos mínimos a partir de um vértice sobre grafos com pesos não negativos.

O algoritmo mantém um conjunto S que contém os vértices com os caminhos mínimos calculados até o momento. A cada iteração o algoritmo seleciona um novo vértice v para ser incluído em S, tal que v é escolhido entre os vértices de $V - S$ com menor valor de $d[v]$. O vértice v é incluído em S e todas as arestas que saem de v são processadas pela rotina *Relax*.

O pseudocódigo de Dijkstra é apresentado no algoritmo 12.

Algoritmo 12 Dijkstra

```
1: procedimento DIJKSTRA(G, w, s)  ▷ Entrada: G = (V, E), conjunto de pesos das arestas (w) e
       o vértice inicial s
2:     S ← ∞
3:     Q ← V
4:     enquanto ( façaQ ≠ ∅)
5:         u ← EstraiMin(Q)                                  ▷ Em relação a d[]
6:         S ← S ∪ {u}
7:         para cada vértice v ∈ Adj(u) faça        ▷ Adj(u) é a lista de arestas para o vértice u
8:             Relax(u,v,w)
9:         fim para cada
10:    fim enquanto
11: fim procedimento
```

Para entender o funcionamento do algoritmo considere as figuras 4.5 a Figura 4.14 nesta ordem.

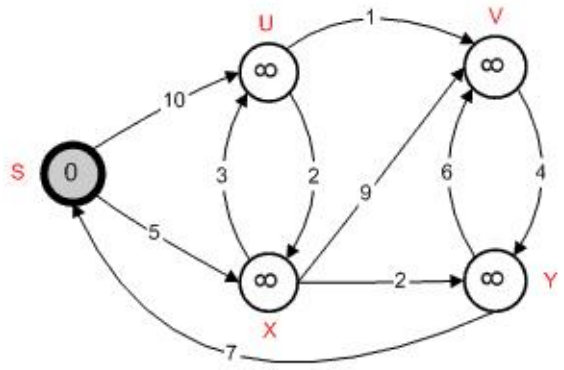

Figura 4.5: Dijkstra: passo 1

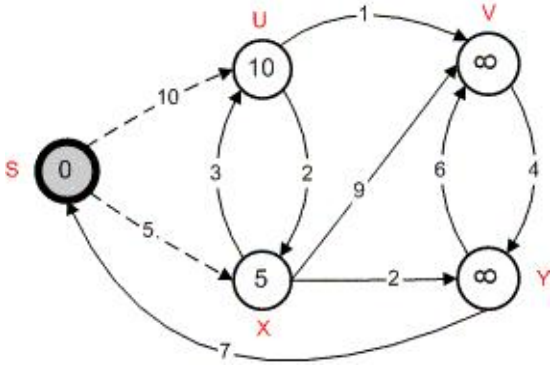

Figura 4.6: Passo 2

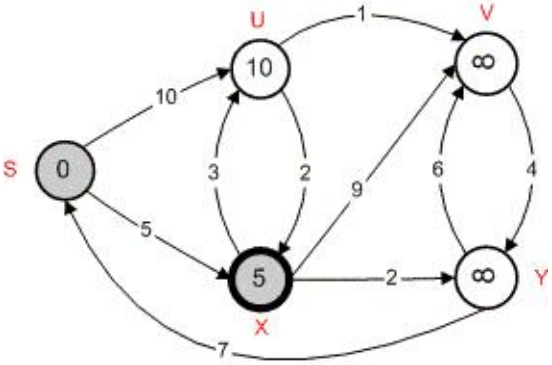

Figura 4.7: Passo 3

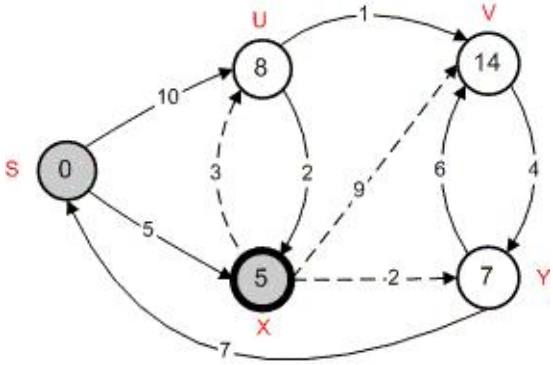

Figura 4.8: Passo 4

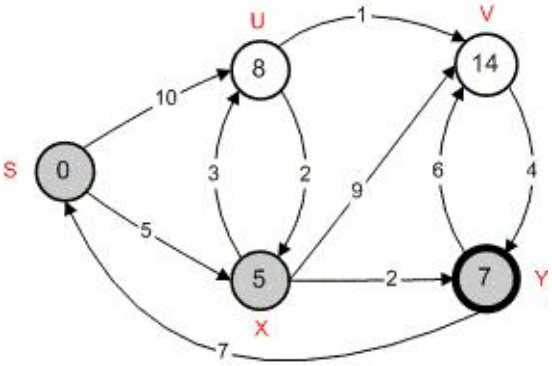

Figura 4.9: Passo 5

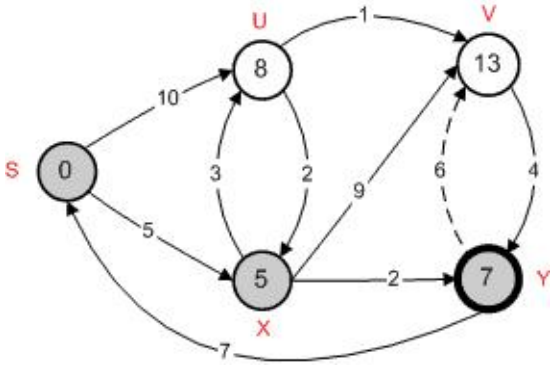

Figura 4.10: Passo 6

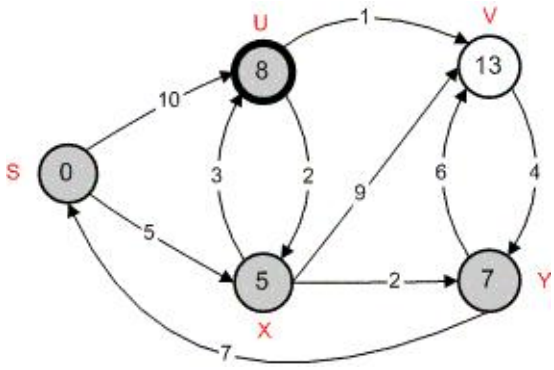

Figura 4.11: Passo 7

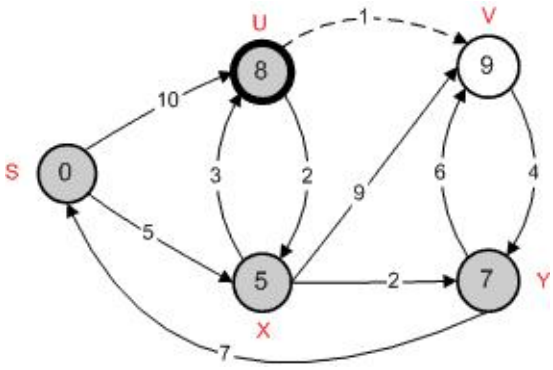

Figura 4.12: Passo 8

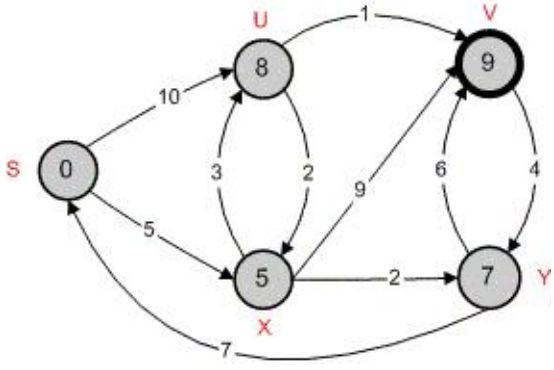

Figura 4.13: Passo 9

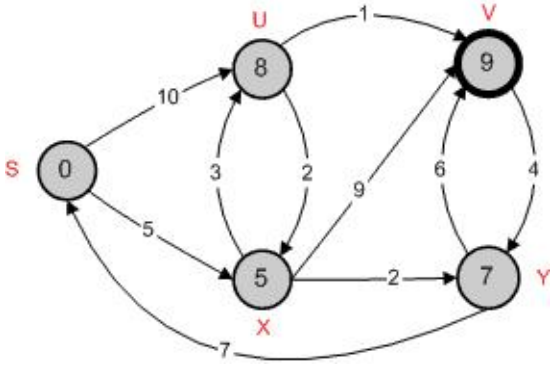

Figura 4.14: Passo 10

Prova de corretude

Após a finalização de algoritmo de Dijkstra sobre um grafo orientado, com pesos não negativos nas arestas $w : E \rightarrow \Re^+$ e vértice de início s, temos $d[u] = \delta(s, u)$, para todo vértice $u \in V$. Nota: $\delta(u, v)$ denota o valor da soma dos pesos das arestas de u até v sendo ∞ quando não há um caminho de u a v.

Prova: Vamos mostrar que no instante em que u é inserido em S, $d[u] = \delta(s, u)$. Vamos mostrar por contradição. Suponha que exista um vértice u tal que ao inserir u em S, $d[u] \neq \delta(s, u)$. Escolha u como o primeiro vértice em que isso ocorre. Claramente, $u \neq s$, já que $d[s] = \delta(s, s) = 0$;

Se o caminho associado a $d[u]$ não é de peso mínimo então deve existir um outro caminho P de s a u de peso $\delta(s, u) < d[u]$. Seja (x, y) a primeira aresta de P que liga um vértice $x \in S$ com um vértice $y \in (V - S)$. Como $x \in S$ e $y \in Adj[x]$ temos, pelos passos 7 e 8 (chamada de *Relax*), que:

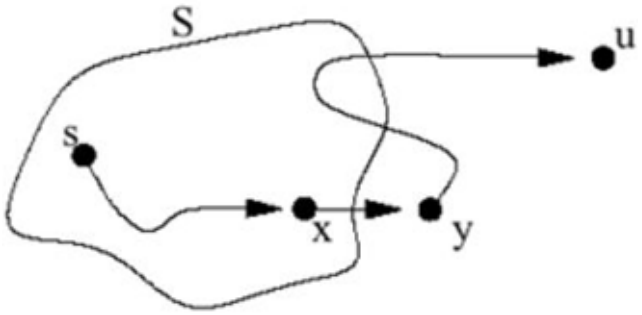

Figura 4.15: $d[y] \leq d[x] + w(x, y)$**.**

Como todos os pesos de P são não negativos e $(x, y) \in P$, temos que

$$d[y] \leq d[x] + w(x, y) \leq \delta(s, u). \tag{4.2}$$

Como x e y pertencem a $(V - S)$ e u é escolhido antes de y temos que

$$d[u] \leq d[y] \tag{4.3}$$

Das inequações 4.2 e 4.3 temos que

$$d[u] \leq d[y] \leq d[x] + w(x, y) \leq \delta(s, u). \tag{4.4}$$

Claramente, a inequação 4.4 contraria a escolha de u.

Complexidade

A complexidade de tempo de *InicializaCaminhoMínimo* é $O(|V|)$. No entanto, a complexidade total do algoritmo de Dijkstra depende da implementação de Q (a fila de prioridades).

• **Vetores lineares**. O uso de vetores lineares para representar os vetores $d[]$ e $pred[]$, indexados pelos vértices e usando a implementação de *lista de vizinhanças* para os grafos podemos realizar

o passo 5 com complexidade de tempo total $O(|V|^2)$ e o passo 8 com complexidade de tempo total $O(|E|)$ (a chamada da rotina *Relax* é executada em tempo constante). Desta forma, a complexidade final do algoritmo nesta implementação é $O(|V|^2)$.

• **Heap binário**. O uso de um *heap* binário para representar Q, nos permite construir o *heap* (passo 3) com tempo $O(|V|)$. A operação *ExtraiMin(Q)* (passo 5) é executada $|V|$ vezes e pode ser feita com complexidade de tempo $O(\log |V|)$, isto é, $O(|V| \log |V|)$. A operação de decremento de um valor do *heap* (passo 2 da rotina *Relax*) pode ser feita em tempo $O(\log |V|)$ e portanto *Relax* pode ser implementada em $O(\log |V|)$. Como pode haver até $|E|$ chamadas de *Relax* no passo 8, temos uma complexidade total de $O(|V| + |E| \log |V|)$.

Capítulo 5

Algoritmos Gulosos como heurística

Existem problemas para os quais nenhum algoritmo guloso conhecido produz uma solução ótima, mas para os quais algoritmos gulosos possuem uma alta probabilidade de produzirem soluções “boas”. Nos casos em que uma solução quase ótima, ou seja, aquela que fica uma certa porcentagem acima da solução ótima, é aceitável, os algoritmos gulosos freqüentemente fornecem a maneira mais rápida para se chegar a tal solução “boa”.

Na verdade, quando para se chegar a uma solução ótima de um problema é preciso realizar uma técnica de busca exaustiva, a escolha de um algoritmo guloso (ou outra heurística) pode ser a alternativa viável, mesmo não produzindo uma solução ótima.

A seguir, será apresentado o Problema do Caixeiro Viajante (TSP - *Traveling Salesman Problem*), no qual os únicos algoritmos conhecidos que produzem uma solução ótima são do tipo “tentar todas as “possibilidades”. Tais algoritmos podem ter tempos de execução que são exponenciais no tamanho da entrada [1].

Resumidamente, pode-se dizer que o problema consiste em encontrar em um grafo não-direcionado, com pesos nas arestas, um caminho (ciclo simples que inclua todos os vértices) de tal forma que a soma dos pesos das arestas seja mínimo. Um caminho é freqüentemente chamado de ciclo Halmitoniano.

Exemplos de aplicações práticas deste problema incluem deste a determinação de rotas até o problema do caminho do cavalo (*knight’s tour problem*). Neste último, o objetivo é encontrar uma seqüência de movimentos de tal forma que o cavalo visite cada quadro do tabuleiro de xadrez somente uma vez e retorne à sua posição inicial. Para tornar a idéia do problema mais clara, considere a Figura 5.1 a seguir.

O item (a) da Figura 5.1 mostra uma instância do problema TSP, onde são dadas as coordenadas de cada vértice (no problema TSP estes são chamados cidades), e o peso de cada aresta é dado pelo seu tamanho. Aqui assumimos que todas as arestas existem. Os demais itens da Figura 5.1 ((b) - (e)) mostram quatro diferentes caminhos entre as cidades, que possuem tamanhos 50.00, 49.73, 48.39 e 49.78, respectivamente. O caminho mais curto, portanto, é o dado em (d).

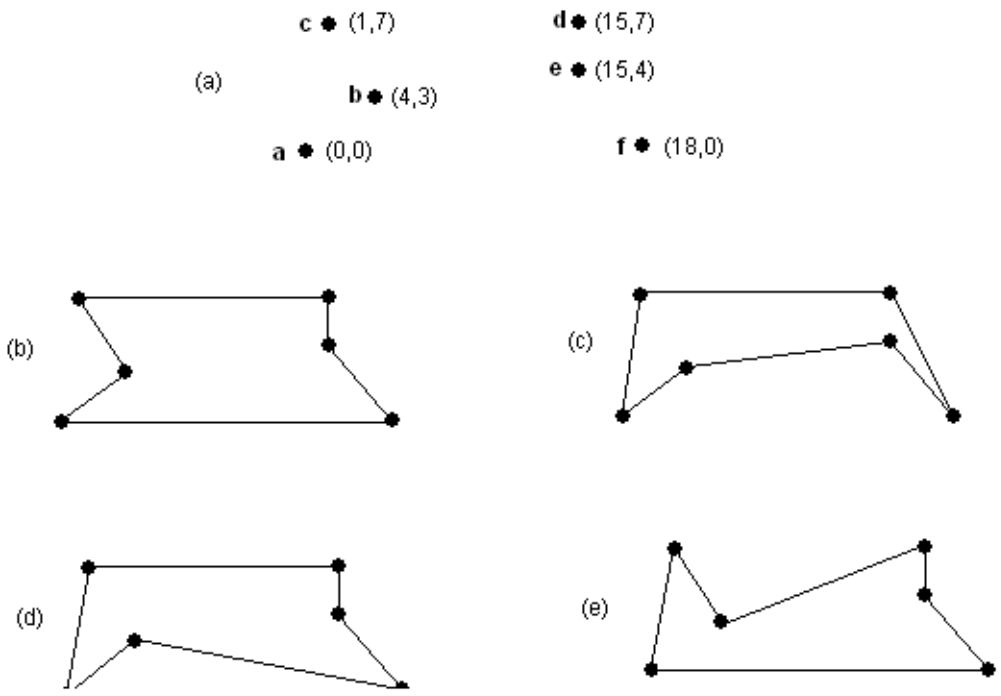

Figura 5.1: Uma instância do problema do caixeiro viajante

O algoritmo guloso idealizado para o problema TSP é uma variante do algoritmo de Kruskal's apresentado anteriormente. Como critérios de aceitação, isto é, como critérios para determinar se uma aresta sob consideração irá ou não se unir às já selecionadas, temos que a aresta sob consideração:

- não leve um vértice a ter grau três ou mais;
- não venha a formar um círculo, a menos que o número de arestas selecionadas seja igual ao número de vértices no problema.

Grupos de arestas selecionadas sobre tais critérios irão formar um grupo de caminhos não-conectados até o último passo, onde o único caminho restante é fechado para formar um caminho.

Voltando ao exemplo apresentado na Figura 5.1 o caminho escolhido é o representado pelo item (b) da figura, que é o caminho $a \rightarrow b \rightarrow c \rightarrow d \rightarrow e \rightarrow f$ (e eventualmente, $a \rightarrow f$ para completar o caminho). Embora não seja a solução ótima, é uma solução com um custo a mais de apenas 4%, que é uma diferença aceitável (considerando nosso critério inicial de aceitar soluções quase ótimas). Achar o caminho fica como exercício (Dicas: (i) a primeira aresta pega é a aresta (d,e), que possui tamanho 3; (ii) as arestas (a,c), (d,f), (b,e) e (b,d) são descartadas por não obedecerem aos critérios de aceitação relacionados anteriormente. Agora ficou fácil fazer o exercício).

Capítulo 6

Conclusões

Muitas vezes na computação tem-se a idéia de que quanto mais sofisticado um algoritmo mais preciso ele é. Esta é uma idéia errada. É possível combinar elegância e simplicidade quando se conhece bem o problema sendo tratado. Isso é o que nos prova a teoria dos algoritmos gulosos. Os algoritmos de *Kruskal*, *Prim* e *Dijkstra* são exemplos de idéias de certa forma intuitivas ao ser humano (a busca do que nos parece ótimo no momento atual) e que dão certo.

Obviamente, nem sempre essa abordagem irá funcionar. No entanto, uma vez que o problema intrinsecamente tenha a propriedade de ser resolvido por essa abordagem, muito trabalho poderá ser evitado na busca de algoritmos capazes de resolvê-lo.

Anexo 1 - Grafos

Neste anexo mostraremos alguns conceitos relacionados à teoria de grafos. Estes conceitos serão importantes em algumas das seções anteriores, principalmente quando tratarmos de algoritmos gulosos em grafos. Os conceitos foram tirados de [2], [3] e [4].

6.1. Conceitos e definições

Um grafo $G = (V, E)$ é um conjunto V de vértices e um conjunto E de arestas (***edges*** em inglês) onde cada aresta é um par de vértices (Ex.: (v, w)). Um grafo é representado graficamente usando conectores "•" para vértices e retas ou curvas para arestas. Veja um exemplo na Figura 6.1.

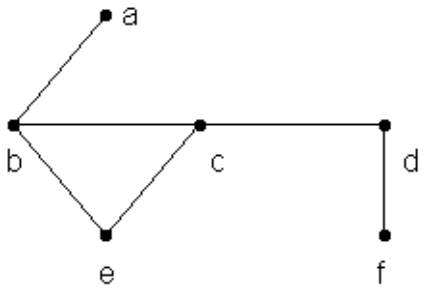

Figura 6.1: Exemplo de um grafo simples

O grafo da Figura 6.1 possui V = {a, b, c, d, e, f} e E = {(a, b), (b, c), (b, e), (c, e), (c, d), (d, f)} onde (a, b) é uma aresta entre os vértices a e b. Normalmente, arestas do tipo (a, a) não são permitidas.

Um grafo pode ser dirigido ou não dirigido. Em um grafo dirigido, a ordem entre os vértices de uma aresta (v, w) é importante. Esta aresta é diferente da aresta (w, v) e é representada com uma flecha de v para w como mostra a Figura 6.2.

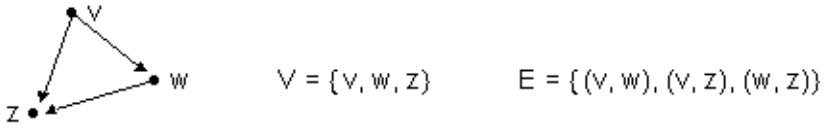

Figura 6.2: Exemplo de um grafo dirigido

Em um grafo não dirigido, $(v, w) = (w, v)$.

Um caminho (*path*) é uma seqüência de vértices $v_1, v_2, \ldots, v_n$ conectados por arestas $(v_1, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n)$. As arestas são também consideradas como parte do caminho. Veja um exemplo na Figura 6.3.

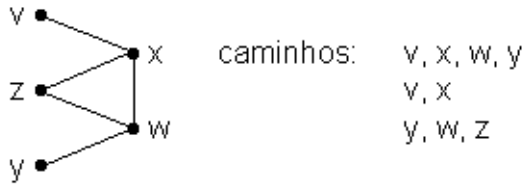

Figura 6.3: Um caminho em um grafo

Um circuito é um caminho onde $v_1 = v_n$, como b, c, d, e, f, d, b.

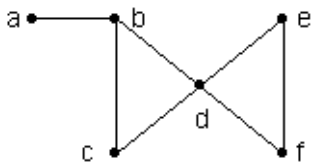

Figura 6.4: Circuito em um grafo

Um circuito será simples se nenhum vértice aparecer mais de uma vez, exceto o primeiro e o último. Um circuito simples é chamado de ciclo.

- **Definição 1:** dado um grafo, dizemos que o vértice v é adjacente ao vértice w se existe aresta (v, w) no grafo.
- **Definição 2:** um grafo é conectado se existe um caminho entre dois vértices quaisquer do grafo.
- **Definição 3:** dígrafo é um grafo dirigido.
- **Definição 4:** o grau de um vértice é o número de arestas adjacentes a ele. Em um grafo dirigido, o grau de entrada de um vértice v é o número de arestas (w, v) e o grau de saída é o número de arestas (v, w). Veja um exemplo na figura 6.5.

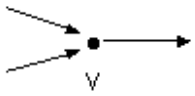

Figura 6.5: Grau de um vértice. v possui grau de entrada 2 e grau de saída 1

- **Definição 5:** uma fonte é um vértice com grau de entrada 0 e grau de saída ≥ 1. Um sumidouro é um vértice com grau de saída 0 e grau de entrada ≥ 1.
- **Definição 6:** um grafo é completo quando existe uma aresta entre dois vértices quaisquer do grafo. O grafo completo de n vértices é denotado por K_n. Veja um exemplo na figura 6.6.
- **Definição 7:** Um subgrafo $G' = (V', E')$ de um grafo $G = (V, E)$ é um grafo tal que $V' \subseteq V$ e $E' \subseteq E$. Veja exemplo na figura 6.7.

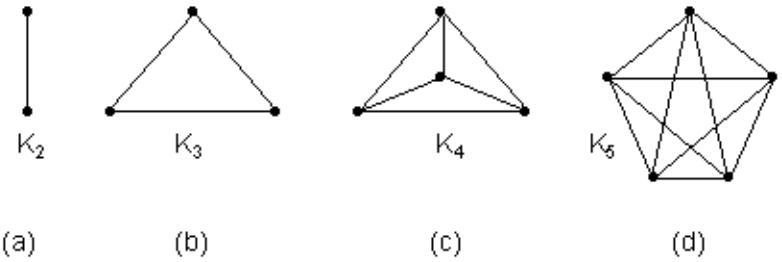

Figura 6.6: Grafos completos

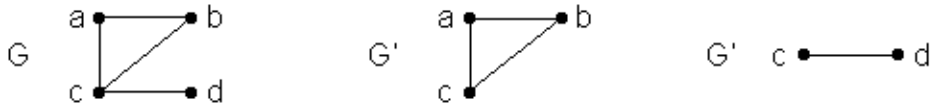

Figura 6.7: Subgrafos

- **Definição 8:** Um grafo $G = (V, E)$ é bipartido se V pode ser dividido em dois conjuntos V_1 e V_2 tal que toda aresta de G une um vértice de V_1 a outro de V_2. Veja exemplo na figura 6.8

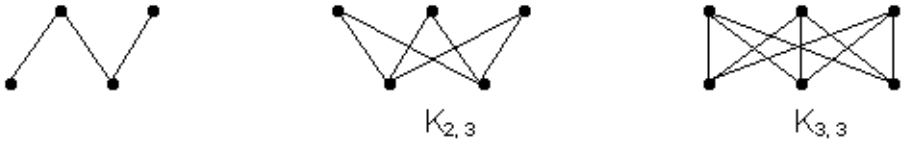

Figura 6.8: Grafos bipartidos

- **Definição 9:** Dados dois grafos $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$, dizemos que G_1 é isomorfo a G_2 se e somente se existe uma função $f : V_1 \longrightarrow V_2$ tal que $(v, w) \in E_1$ se

$$(f(v), f(w)) \in E_2 \; \forall \; v, w \in V_1. \tag{6.1}$$

  Veja exemplo na Figura 6.9.

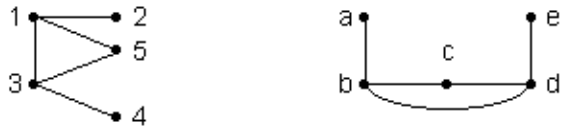

Figura 6.9: Grafos isomorfos. $f = (1, b), (5, c), (3, d), (2, a), (4, e)$

6.2. História

O primeiro problema de teoria dos grafos estudado foi o das pontes de *Könisberg*.

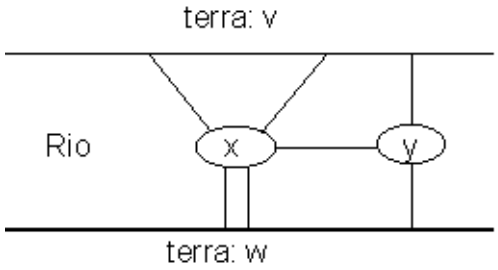

Figura 6.10: As pontes de *Könisberg*

Esta cidade possuía um rio com duas ilhas conectadas por sete pontes. A figura 6.10 apresenta uma versão simplificada. O problema é saber se é possível caminhar de um ponto qualquer da cidade e retornar a este ponto passando por cada ponte exatamente um vez.

Euler resolveu este problema criando um grafo em que terra firme é vértice e ponte é aresta como na Figura 6.11.

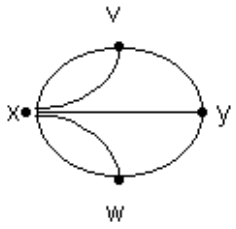

Figura 6.11: Proposta de Euler ao problema das pontes de *Könisberg*

Quando caminhamos por um vértice, temos que entrar e sair dele (ou vice-versa, no caso do ponto inicial), o que significa que usamos um número par de arestas cada vez que passamos por um vértice.

Como o grafo acima possui vértices com número ímpar de arestas, a resposta para o problema é NÃO.

6.3. Estruturas de dados para grafos

Um grafo $G = (V, E)$ é usualmente representado por uma *matriz de adjacências* ou por uma *lista de vizinhanças*.

Matriz de adjacências

Sendo n o número de vértices de G, uma matriz de adjacência para G é uma matriz $A = (a_{ij})$ nxn tal que $a_{ij} = 1$ se $(v_i, v_j) \in E$.

A desvantagem desta representação é que ela ocupa muito espaço se há poucas arestas. Neste caso, a maior parte da matriz é inútil. A vantagem é que podemos saber se uma aresta existe ou não em tempo constante.

Veja exemplo na Figura 6.12

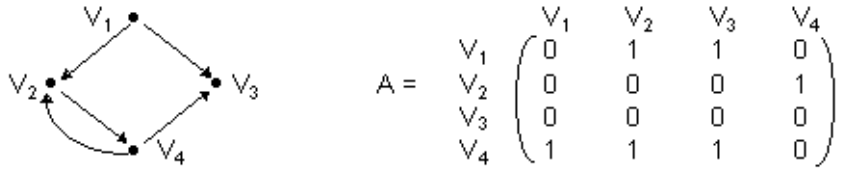

Figura 6.12: Representação de grafos por *matriz de adjacência*

Lista de vizinhanças

Há um vetor de n posições cada uma apontando para uma lista. A posição i do vetor aponta para uma lista contendo números j tal que $(v_i, v_j) \in E$. Para o grafo da Figura 6.12 temos a lista de vizinhanças da Figura 6.13.

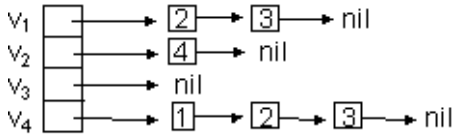

Figura 6.13: Representação de grafos por *lista de vizinhança*

6.4. Caminhos

6.4.1. Caminho Euleriano

Definição: um grafo G não dirigido é Euleriano se existe um caminho fechado (circuito) que inclui cada aresta de G. Este caminho é chamado de "caminho Euleriano".

Arestas em um circuito não se repetem. Assim, um caminho Euleriano contém cada aresta do grafo exatamente uma vez. Veja exemplo na Figura 6.14.

Figura 6.14: Caminho Euleriano em um grafo

6.4.2. Caminho Hamiltoniano

Definição: um circuito Hamiltoniano em um grafo conectado G é um circuito que inclui cada vértice de G exatamente uma vez.

Encontrar um circuito Hamiltoniano (CH) é claramente mais difícil que encontrar um caminho Euleriano (CE), pois cada vez que atingimos (num percurso) um vértice v a partir de uma aresta e, nunca mais podemos usar v e quaisquer de suas arestas. Em um CE, não poderíamos usar e, mas poderíamos usar v e suas outras arestas. Veja exemplo na Figura 6.15.

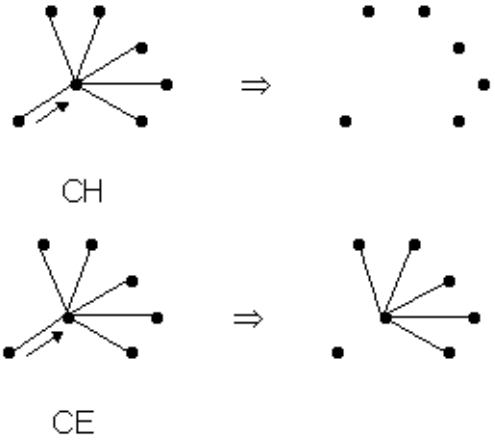

Figura 6.15: Caminho Hamiltoniano em um grafo

6.4.3. Caminho ponderado entre vértices

Definição: um grafo com comprimentos (ou pesos) associa a cada aresta um comprimento que é um número real positivo. O comprimento de um caminho é a soma dos comprimentos de suas arestas. Veja exemplo na Figura 6.16.

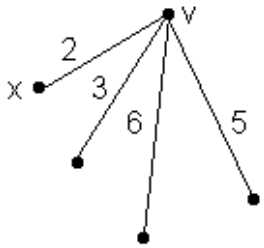

Figura 6.16: Caminhos ponderados em um grafo

Referências

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Addison-Wesley Publishing Company, California, 1987. ISBN 0-201-00023-7.

[2] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice Hall, Nova Iorque, 1995. ISBN 01-333-5068-1.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Algoritmos, teoria e prática*. Editora Campus, Rio de Janeiro, 2002. ISBN 85-352-0926-3.

[4] Udi Manber. *Introduction to Algorithms: A Creative Approach*. Pearson Education POD, Nova Iorque, 1989. ISBN 02-011-2037-2.

[5] Flávio Keidi Miyazawa. Notas de aula de complexidade de algoritmos. Disponível para *download* em `www.ic.unicamp.br/~fkm`, 2002.

[6] Pedro J. Rezende. Complexidade de algoritmos i. Disponível para *download* em `www.ic.unicamp.br/~rezende`, 2002.

[7] Nivio Ziviani. *Projeto de Algoritmos*. Pioneira Thomson Learning, São Paulo, 2004. ISBN 85-221-0390-9.

[]

Exercícios propostos

Todos os exercícios são referentes a [2] e [3].

1. em [2]:
   (a) **6.1**: Características gerais dos algoritmos gulosos;
   (b) **6.7**: Grafos;
   (c) **6.9** : Kruskal;
   (d) **6.10**: Kruskal e Prim;
   (e) **6.11**: Sobre a existência de mais de uma AGM para um mesmo grafo;
   (f) **6.13**: Implementação de Prim com *heaps*,
   (g) **6.14** e **6.15**: Dijkstra.
2. em [3]:
   (a) **16.1-4**: Seleção de atividades;
   (b) **16.2-3**: Sugestão de algoritmo guloso para uma variante do problema da mochila;

UNICAMP - Univeridade Estadual de Campinas

Instituto de Computação

MO417 - Complexidade de Algoritmos

Algoritmos Gulosos

Alunos: Leyza E. Baldo Dorini

Anderson de Rezende Rocha

Campinas, Maio de 2004