

Apostila Introdutória de Algoritmos

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

Versão rascunho para o curso de Análise de Algoritmos da Unirio
2009.

2 de Outubro de 2009

Método Guloso

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

6.1. Fecho convexo: Algoritmo de Jarvis

Já vimos como é conveniente trabalhar com polígonos convexos. Muitas vezes, não temos um polígono convexo, ou sequer um polígono, mas apenas um conjunto S de pontos no plano. Digamos, por exemplo, que desejamos responder uma série de consultas de pontos extremos de S para várias direções d . Neste caso, vale a pena descobrir qual o menor polígono convexo P que envolve os pontos de S . Este polígono P é chamado de fecho convexo de S e está ilustrado na figura 6.1(a). Este é o nosso problema:

PROBLEMA 15. *Dado um conjunto S de pontos no plano, determinar P o fecho convexo de S .*

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

A idéia do método guloso é adicionar um elemento de cada vez na solução e, jamais, remover algum elemento dela. Neste problema, vamos começar descobrindo um primeiro vértice de P . Isto é fácil, o ponto extremo em qualquer direção é um vértice de P . Podemos pegar o ponto de S mais a direita (de maior coordenada x) e chamar este ponto de v_1 . Não podemos usar o algoritmo da sessão 2.3 para encontrar um vértice extremo porque ainda não temos um polígono convexo. Ainda assim, podemos resolver o problema em tempo linear inspecionando todos os vértices e retornando o de maior coordenada x .

Já temos um vértice. Como fazemos para achar outro? Geometricamente, podemos inclinar aos poucos a reta vertical que passa por v_1 até que ela toque outro vértice $v_2 \in S$, ou seja, pegamos o ponto $v_2 \in S$ que minimiza a função $\odot(v_1 - (0, 1), v_1, v_2)$ (a função $\odot(p_1, p_2, p_3)$ está definida na sessão 2.3 e retorna o ângulo $\widehat{p_1 p_2 p_3}$ medido no sentido anti-horário). Para acharmos o próximo vértice, pegamos o ponto $v_3 \in S$ que minimiza $\odot(v_1, v_2, v_3)$ e assim por diante, até retornarmos ao ponto v_1 (figura 6.1(b)). Caso tenhamos mais de um ponto $v \in S$ com o mesmo valor de $\odot(v_i, v_{i+1}, v)$, devemos escolher, dentre esses, o mais distante de v_{i+1} . Este algoritmo se chama algoritmo de Jarvis ou embrulho para presente.

Vamos provar que o algoritmo de Jarvis funciona, isto é, encontra um polígono convexo cujos vértices pertencem a S e todos os pontos de S estão no interior deste polígono. Para isto, usaremos a definição de que um polígono $v = (v_1, \dots, v_n)$ é convexo se $\odot(v_{i-1}, v_i, v_{i+1}) < 180^\circ$ para i de 1 a n com os índices módulo n (ver página 17 para explicação de índices módulo n).

TEOREMA 6.1. *O polígono $P = (v_1, \dots, v_{|P|})$ gerado pelo algoritmo de Jarvis tendo como entrada um conjunto S de pontos no plano é realmente o fecho convexo de S .*

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

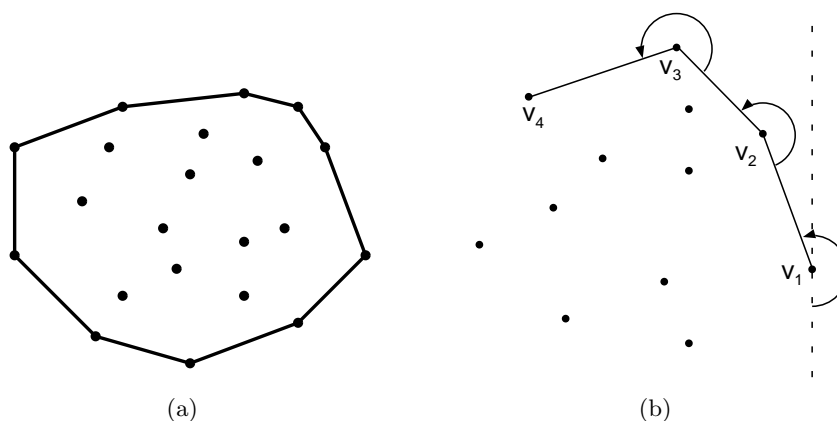


FIGURA 6.1. (a) Fecho convexo de um conjunto de pontos no plano. (b) Algoritmo de Jarvis fazendo o “embrulho para presente”.

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

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

TEOREMA 6.2. *A complexidade de tempo do pior caso do algoritmo de Jarvis é $\Theta(nh)$, onde n é o número de pontos de entrada e h é o número de vértices do fecho convexo.*

6.2. Árvore geradora mínima: Algoritmo de Prim

Um problema de grafos com diversas aplicações em telecomunicações, redes de computadores, confecção de placas de circuitos etc é o da árvore geradora mínima. Alguns termos de grafos serão necessários agora. Se você não está familiar com eles a notação básica para grafos está na sessão 11.2.

PROBLEMA 16. *Seja G um grafo com pesos reais nas arestas, obter a árvore geradora mínima de G .*

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

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

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

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

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

Como podemos obter a próxima aresta? Também existem várias respostas que funcionam para esta questão (ver exercício 6.3, para um outro exemplo). Vamos pegar uma aresta que seja adjacente em exatamente um dos extremos às arestas já escolhidas para a árvore geradora mínima e que tenha custo mínimo (figuras 6.3(b) e 6.3(c)). Este é o algoritmo de Prim (pseudocódigo na figura 6.2), que funciona devido ao teorema que veremos a seguir.

Entrada:

G : Grafo conexo com custos associados às arestas.

Saída:

T : Árvore geradora mínima de G .

Prim(G)

$V(T) \leftarrow$ um vértice qualquer de $V(G)$

Enquanto $|V(T)| \neq |V(G)|$

 Encontre aresta (u, v) com $u \in V(T)$ e $v \notin V(T)$ de custo mínimo

 Adicione v à $V(T)$ e (u, v) à $E(T)$

Retorne T

FIGURA 6.2. Solução do Problema 16

TEOREMA 6.3. *Seja T' uma árvore geradora mínima de G e T uma árvore tal que $E(T) \subset E(T')$. Seja $e = (v_1, v_2)$ uma aresta de G que tenha custo mínimo dentre as arestas (v_1, v_2) tais que $v_1 \in V(T)$ e $v_2 \notin V(T)$. Então existe uma árvore geradora mínima de G que contém $E(T) \cup \{e\}$.*

DEMONSTRAÇÃO. Suponha que T' seja uma árvore geradora mínima de G que não contém e mas contém todas as arestas de T . Caso não exista T' , o teorema já está provado. Se adicionamos e a T' obtemos o grafo $T' \cup \{e\}$ que contém um único ciclo. Como $v_1 \in V(T)$, $v_2 \notin V(T)$ e T é uma árvore, então existe uma aresta $e' = (v'_1, v'_2)$ neste ciclo tal que $v'_1 \in V(T)$ e $v'_2 \notin V(T)$, como ilustra a figura 6.3(a). O custo da árvore obtida a partir de T' pela remoção de e' e adição de e é $c(T' \cup \{e\} - \{e'\}) = c(T') + c(e) - c(e')$. Como $c(e) \leq c(e')$ então $c(T' \cup \{e\} - \{e'\}) \leq c(T')$. Portanto existe árvore geradora $T' \cup \{e\} - \{e'\}$ de custo mínimo que contém $E(T) \cup \{e\}$. \square

Existem várias maneiras de implementar este algoritmo, usando diferentes estruturas de dados. Cada implementação tem uma complexidade de tempo diferente. A alternativa mais simples não usa nenhuma estrutura de dados sofisticada, usando apenas vetores, como mostra a figura 6.4.

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

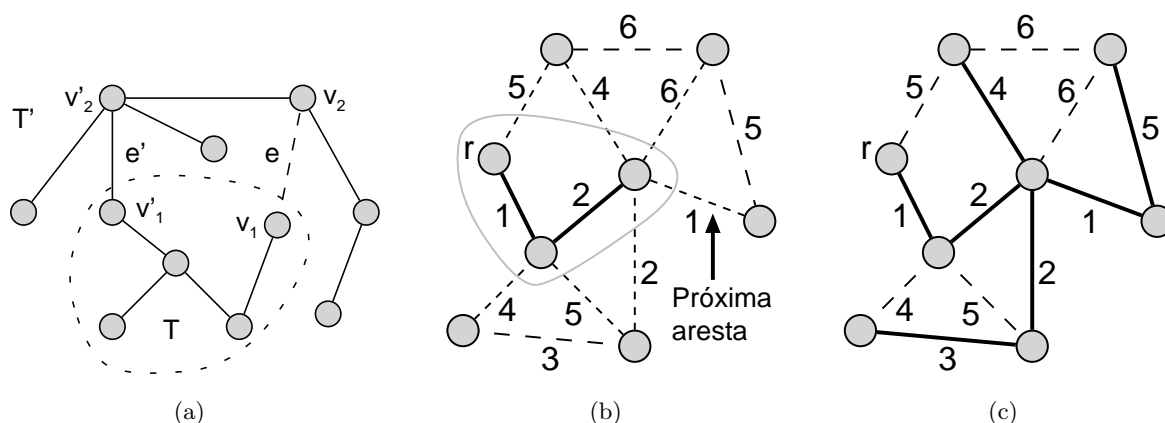


FIGURA 6.3. (a) Ilustração referente a prova do teorema 6.3. (b) Execução do algoritmo de Prim na 3ª iteração. (c) Árvore gerada pelo algoritmo de Prim.

Entrada:

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

Saída:

T : Árvore geradora mínima de G .

PrimVetor(G)

$V(T) \leftarrow v \leftarrow$ um vértice qualquer de $V(G)$

Marcar v

Para todo vértice $u \neq v$

$u.custo \leftarrow G.custo(u, v)$

$u.vizinho \leftarrow v$

Enquanto $|V(T)| \neq |V(G)|$

$v \leftarrow$ vértice não marcado com menor $custo$

Adicione v à $V(T)$ e (u, v) à $E(T)$

Marcar v

Para todo vértice u não marcado

Se $custo(u, v) < u.custo$

$u.custo \leftarrow G.custo(u, v)$

$u.vizinho \leftarrow v$

Retorne T

FIGURA 6.4. Solução do Problema 16 usando apenas vetores.

podemos fazer, pois o número de arestas do grafo pode ser $\Theta(n^2)$ e todas precisam ser examinadas. Na prática, grafos esparsos (poucas arestas) são muito mais comuns do que grafos densos. Por isso, seria bom que conseguíssemos expressar a complexidade de tempo não só em função de n , mas também em função do número de arestas m . O algoritmo de Prim pode ser facilmente modificado para ter complexidade de tempo $O(m \lg n)$, usando um *heap* binário. O novo pseudo-código está na figura 6.5.

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

Entrada:

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

Saída:

T : Árvore geradora mínima de G .

Observações:

H : Heap mínimo que armazena vértices usando como chave o campo $custo$.

PrimHeap(G)

$V(T) \leftarrow v \leftarrow$ um vértice qualquer de $V(G)$

Marcar v

Para todo vértice $u \neq v$

$u.custo \leftarrow G.custo(u, v)$

$u.vizinho \leftarrow v$

Inserir(H, u)

Enquanto $|V(T)| \neq |V(G)|$

$v \leftarrow$ ExtrairMínimo(H)

Adicione v à $V(T)$ e (u, v) à $E(T)$

Marcar v

Para todo vértice u não marcado e adjacente à v

Se $custo(u, v) < u.custo$

ReduzirCusto($H, u, G.custo(u, v)$)

$u.vizinho \leftarrow v$

Retorne T

FIGURA 6.5. Solução do Problema 16 usando um heap.

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

6.3. Compactação de dados: Árvores de Huffman

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

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

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

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

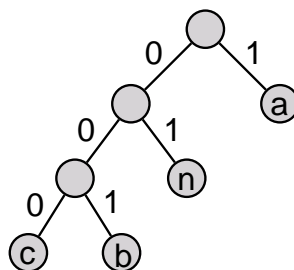


FIGURA 6.6. Árvore de Huffman da palavra “cabana”.

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

Existe uma relação direta entre árvores binárias e códigos de prefixo. A cada folha podemos associar um caractere. Cada *bit* indica se devemos seguir para a direita ou esquerda na árvore. A árvore correspondente ao código para a palavra *cabana* está na figura 6.6.

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

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

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

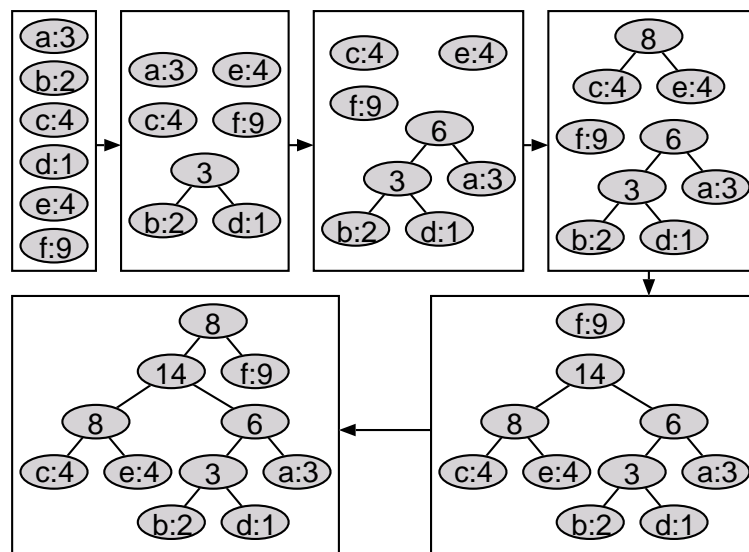
PROBLEMA 17. *Dado um conjunto de caracteres C , com suas freqüências, construir uma árvore de Huffman de C .*

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

Começamos criando um nó para cada caractere. Estas serão as folhas da árvore e têm como freqüência a própria freqüência do caractere. A cada passo do algoritmo, os dois nós de menor freqüência são colocados como filhos de um novo nó. A freqüência do novo nó é a soma das freqüências de seus filhos. Este procedimento está ilustrado no pseudo-código da figura 6.7 e exemplificado na figura 6.8.

Entrada: C : Vetor de caracteres a serem codificados. f : Vetor com as frequências dos caracteres.Saída:Vértice raiz da árvore de Huffman de C .Observações: H : Heap mínimo que armazena vértices usando como chave o campo $freq$.Huffman(C, f)Para i de 1 até $|C|$ $v \leftarrow \text{CriarVértice}()$ $v.\text{caractere} \leftarrow C[i]$ $v.\text{freq} \leftarrow f[i]$ Inserir(H, v)Para i de 1 até $|C| - 1$ $v \leftarrow \text{CriarVértice}()$ $v.\text{dir} \leftarrow \text{ExtrairMínimo}(H)$ $v.\text{esq} \leftarrow \text{ExtrairMínimo}(H)$ $v.\text{freq} \leftarrow v.\text{dir}.\text{freq} + v.\text{esq}.\text{freq}$ Inserir(H, v)Retorne ExtrairMínimo(H)

FIGURA 6.7. Solução do Problema 17

FIGURA 6.8. Algoritmo de Huffman passo a passo, para uma entrada $C = (a, b, c, d, e, f)$ e $f = (3, 2, 4, 1, 4, 9)$.

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

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

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

LEMA 6.4. *Toda árvore de Huffman é estritamente binária.*

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

LEMA 6.5. *Seja $C = \{c_1, \dots, c_n\}$ um alfabeto onde todo caractere c_i tem frequência $f(c_i)$ e $f(c_i) \leq f(c_{i+1})$. Neste caso, existe árvore de Huffman onde c_1 e c_2 são folhas irmãs.*

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

LEMA 6.6. *Se T_C é uma árvore de Huffman para o alfabeto $C = \{c_1, \dots, c_n\}$, então $T_{C'}$, obtida acrescentando dois filhos c'_1 e c'_2 a uma folha c_k de T_C onde $f(c_k) = f(c'_1) + f(c'_2)$ e $f(c'_1), f(c'_2) \leq f(c_i)$ para $1 \leq i \leq n$, é uma árvore de Huffman para o alfabeto $C - \{c_k\} \cup \{c'_1, c'_2\}$.*

DEMONSTRAÇÃO. O custo de $T_{C'}$ é $c(T_{C'}) = c(T_C) + f(c'_1) + f(c'_2)$, então $c(T_C) = c(T_{C'}) - f(c'_1) - f(c'_2)$. Para obter um absurdo, suponha que $T'_{C'}$ seja uma árvore de Huffman de C' com $c(T'_{C'}) < c(T_{C'})$. Pelo lema 6.5 podemos considerar que c'_1 e c'_2 são folhas irmãs em $T'_{C'}$. Removendo estas duas folhas c'_1 e c'_2 e atribuindo o caractere c_k ao pai delas obtemos uma árvore T'_C com $c(T'_C) = c(T'_{C'}) - f(c'_1) - f(c'_2) < c(T_C)$, o que contradiz o fato de $c(T_C)$ ser uma árvore de Huffman para C . \square

TEOREMA 6.7. *O algoritmo gera uma árvore de Huffman para C .*

DEMONSTRAÇÃO. O lema 6.5 é a base da indução. O lema 6.6 fornece o passo indutivo. Note que provamos que a árvore é uma árvore de Huffman na ordem contrária a que ela é construída. Começamos provando que a raiz com seus dois filhos é uma árvore de Huffman e vamos descendo na árvore, como mostra a figura 6.8. \square

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

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

6.4. Compactação de dados: LZSS

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

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

Descompactado1: método gulosogulosométodo

Compactado1: método gulo(6,6)(18,6)

Descompactado2: bananadadebanana

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

Note que em um símbolo (p, l) , p nunca pode referenciar um caractere que ainda não foi codificado, portanto $p \geq 1$. Podemos também forçar $l \geq 2$, pois preferimos escrever um único caractere explicitamente a referenciá-lo. O descompactador para este arquivo é bem simples e seu pseudo código está na figura 6.9. Neste exemplo trabalhamos com vetores e não arquivos.

Entrada:

C : Vetor compactado.

D : Vetor onde será escrito o arquivo descompactado.

Saída:

Retorna o número de caracteres do arquivo descompactado.

DescompactadorLZSS1(C, D)

$Di \leftarrow 1$

Para Ci de 1 até $|C|$

Se $C[Ci]$ é um caractere

$D[Di] \leftarrow C[Ci]$

$Di \leftarrow Di + 1$

Senão

Para i de 0 até $C[Ci].l - 1$

$D[Di] \leftarrow D[C[Ci].l + i]$

$Di \leftarrow Di + 1$

Retorne $Di - 1$

FIGURA 6.9. Descompactador LZSS em vetor.

Temos alguns problemas no método de compactação que definimos. Um deles é como representarmos no arquivo um par (p, l) já que tanto p quanto l podem ser tão grandes quanto o tamanho do arquivo descompactado. Outro problema é que o descompactador teria que voltar no arquivo várias vezes para encontrar as referências, o que tornaria a descompactação lenta. A solução para estes dois problemas é limitarmos o valor de p e de l e usarmos um *buffer* circular em memória. Assim nos limitamos a armazenar em memória as últimas posições escritas no arquivo descompactado. Limitaremos p ao valor p^* ($1 \leq p \leq p^*$) e l ao valor l^* ($2 \leq l \leq l^*$). O nosso *buffer* precisa armazenar apenas p^* caracteres. O descompactador em arquivo usando um *buffer* circular está ilustrado no pseudo-código da figura 6.10.

Entrada: C : Arquivo compactado. D : Arquivo onde será escrito o arquivo descompactado. p^* : Valor máximo de p em uma símbolo (p, l) .Observações:

A operação $i \bmod n$ é definida como: $i \bmod n$ é o resto da divisão de i por n se i não é divisível por n e $i \bmod n = n$ se i é divisível por n .

DescompactadorLZSS2(C, D, p^*) $B \leftarrow \text{AlocarVetor}(p^*)$ $Bi \leftarrow 1$ Enquanto o arquivo C não tiver chegado ao fim $c \leftarrow \text{LerSímbolo}(C)$ Se c é um caractereEscrever(D, c) $B[Bi] \leftarrow c$ $Bi \leftarrow Bi + 1 \bmod p^*$

Senão

Para i de 1 até $c.l$ Escrever($D, B[Bi - c.p] \bmod p^*$) $B[Bi] \leftarrow c$ $Bi \leftarrow Bi + 1 \bmod p^*$

FIGURA 6.10. Descompactador LZSS em arquivo.

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

PROBLEMA 18. *Dada uma seqüência de caracteres D e dois valores p^* e l^* , encontrar a seqüência de símbolos C correspondente a D que contém o menor número de símbolos com a restrição de que todo símbolo (p, l) satisfaz $1 \leq p \leq p^*$ e $1 \leq l \leq l^*$*

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

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

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

TEOREMA 6.8. *O algoritmo guloso gera uma seqüência de símbolos correspondente a entrada com o número mínimo de símbolos possível.*

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

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

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

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

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

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

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

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

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

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

6.5. Resumo e Observações Finais

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

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

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

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

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

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

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

Exercícios

- 6.1) Escreva o pseudo-código da função $\circlearrowleft(p_1, p_2, p_3)$.
- 6.2) O envelope superior de um conjunto de retas S no plano cartesiano é a seqüência de segmentos de retas de S com valor y máximo para x variando de $-\infty$ à $+\infty$ (figura 6.11). Escreva um algoritmo guloso que determina o envelope superior de um conjunto de retas no plano. Calcule a complexidade de tempo deste algoritmo em função do tamanho da entrada e da saída.

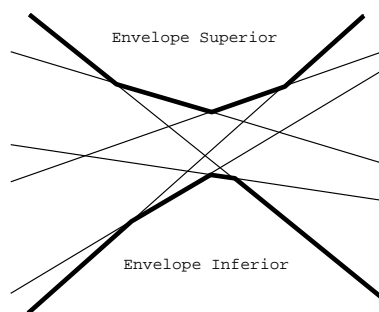


FIGURA 6.11. Envelope superior e envelope inferior de um conjunto de retas.

- 6.3) O algoritmo de Kruskal resolve o problema da árvore geradora mínima escolhendo a cada passo a aresta de menor custo que não adiciona ciclos ao grafo. Ao longo do algoritmo não temos uma árvore crescendo, mas várias árvores surgindo e crescendo até se unirem em uma única árvore. Prove que este algoritmo resolve corretamente o problema.
- 6.4) Forneça um único conjunto de caracteres, todos com frequências distintas, para o qual existem duas árvores de Huffman não isomorfas.
- 6.5) Dado um grafo direcionado G , escreva um algoritmo para colorir suas arestas de modo que não haja duas arestas com a mesma cor entrando em um vértice nem duas arestas com a mesma cor saindo de um vértice. A solução obtida deve minimizar o número de cores diferentes usadas. Prove que seu algoritmo está correto. A mesma abordagem funciona para o caso do grafo não direcionado?
- 6.6) Digrafos acíclicos têm diversas aplicações. Eles podem ser usados, por exemplo, para representar o sistema de matérias com pré-requisitos de uma faculdade. Uma informação extremamente útil sobre os digrafos acíclicos é sua ordenação topológica. Dado um digrafo acíclico, uma ordenação topológica dele é uma ordenação de seus vértices onde todas as arestas partam de um vértice anterior para um vértice posterior na ordenação. Escreva um algoritmo guloso que encontre a ordenação topológica de digrafos acíclicos, prove que seu algoritmo está correto e analise sua complexidade de tempo.

- 6.7) Um problema natural em grafos é encontrar o caminho mais curto entre pares de vértices. Na versão com pesos nas arestas, o comprimento de um caminho é a soma dos pesos das arestas pertencentes a ele. Escreva um algoritmo guloso que, dados um grafo com pesos nas arestas e um vértice v deste grafo, encontre a distância de v a todos os demais vértices do grafo. Prove que seu algoritmo está correto. Sugestão: seu algoritmo deve ser bastante semelhante ao algoritmo de Prim para árvore geradora mínima, mas deve construir a árvore formada pela união dos caminhos mais curtos que partem de v . Este algoritmo é chamado de algoritmo de Dijkstra.
- 6.8) Um país possui moedas de 1, 5, 10, 25 e 50 centavos. Você deve programar uma máquina capaz de dar troco com essas moedas de modo a fornecer sempre o número mínimo de moedas, qualquer que seja a quantia. Considere que a máquina possui quantidades ilimitadas de todas essas moedas. Prove que seu algoritmo funciona. O algoritmo continuaria funcionando se a máquina tivesse apenas moedas de 1, 10 e 25 centavos?
- 6.9) Escreva um algoritmo guloso onde: A entrada é um conjunto de palavras (cadeias de caracteres quaisquer) que formam um dicionário D e uma frase f (outra cadeia de caracteres), onde todo caractere de f está em D . A saída é uma seqüência de segmentos de palavras que concatenados formam f . Um segmento de palavra é representado por uma tripla (p, ini, fim) onde p é o número da palavra no dicionário D , ini é o número do primeiro caractere do segmento e fim é o número do último caractere do segmento. Por exemplo: $D = (\text{camelo}, \text{aguia}, \text{sapo})$, $f = \text{guloso}$; saída: $(2, 2, 3), (1, 5, 6), (3, 1, 1), (1, 6, 6)$. Claro que o seu algoritmo deve gerar o mínimo de triplas possível. Prove que o seu algoritmo resolve o problema com este número mínimo de triplas.
- 6.10) Deseja-se realizar o máximo possível de tarefas de um conjunto de tarefas onde cada tarefa tem um horário de início e um horário de término. Duas tarefas não podem estar sendo realizadas simultaneamente. Toda tarefa que for realizada deverá iniciar exatamente no seu horário de início e terminar exatamente no seu horário de término. Escreva um algoritmo guloso que fornece o maior número possível de tarefas que podem ser realizadas. Prove que seu algoritmo funciona.
- *6.11) O fecho convexo de um conjunto de pontos no espaço é o menor poliedro convexo que contém todos estes pontos. Dado um conjunto de n pontos no espaço tridimensional, escreva um algoritmo para determinar seu fecho convexo em tempo $O(nh)$, onde h é o número de vértices do poliedro do fecho convexo.