Universidade do Estado de Minas Gerais
Fundação Educacional de Ituiutaba
Curso de Engenharia de Computação
Análise de Algoritmos

APOSTILA DE

MÉTODO GULOSO

Prof. Walteno Martins Parreira Júnior
www.waltenomartins.com.br
waltenomartins@yahoo.com

2011

Método Guloso

O nome do método pode ser traduzido como guloso ou ganancioso, pois provem da tradução de "greedy". É um método simples, mas que pode ser aplicado a vários problemas, do tipo calcular um subconjunto S de um conjunto de **n** objetos. Este S deve satisfazer determinadas condições e é denominado solução viável para o problema. Associado a cada solução viável, tem-se um valor f(S), onde f é uma função chamada função-objetivo cujo significado é dado na definição do problema. Entre todas as variáveis S, deseja-se aquela que tem valor f(S) mínimo (ou máximo); e esta solução é denominada *solução ótima*.

Pelo método guloso, pode-se obter um algoritmo iterativo que construa uma solução ótima S em etapas. Em cada etapa, o algoritmo possui uma solução viável parcial P que é um subconjunto (menor do que a solução ótima) satisfazendo as condições do problema; e em cada etapa, o algoritmo escolhe um novo objeto **o**: se $\mathbf{P \cup \{o\}}$ é viável, então **o** é incluído em P, senão **o** é rejeitado. Assim, o algoritmo constrói uma solução viável parcial cada vez maior, até obter uma solução final que seja ótima. A escolha de um novo objeto **o** em cada etapa é orientada por alguma medida de otimização que forneça uma garantia de que a solução final seja ótima; esta medida de otimização se relaciona, direta ou indiretamente com a função objetivo.

1 – CÓDIGOS DE HUFFMAN PARA COMPRESSÃO DE DADOS

Resolução de um problema de comunicação de dados pelo método guloso, conforme solução de D. Huffman.

A árvore binária apresentada é denominada de árvore de codificação de Huffman. A cada folha da árvore é associada uma mensagem M_i; a seqüência de dígitos binários que ocorrem da raiz até uma folha M_i é o código de Huffman c_i correspondente a M_i. Por exemplo, o código da mensagem M_2 é 001. A seqüência de dígitos binários num código c_i corresponde à seqüência de escolhas entre filho esquerdo e filho direito no caminho da raiz até a folha M_i. Assim, se um transmissor deseja enviar uma mensagem M_i, ele só envia o código c_i, e o receptor, ao conhecer c_i, utiliza uma árvore de decodificação como a da figura para reconstituir M_i.

Dado um conjunto de mensagens $\{M_1, M_2, ..., M_n\}$ vamos supor agora que conhecemos a freqüência relativa $\mathbf{f_i}$ com a qual a mensagem M_i é transmitida. Nestas condições, queremos calcular os códigos c_i, e a árvore de decodificação correspondente, que minimize tanto o tempo de transmissão como o de decodificação. Mais especificamente, se p_i é a profundidade da folha correspondente a M_i, queremos minimizar:

$$\sum_{i=1}^{n} p_i \cdot f_i$$

que é o tempo esperado de transmissão e decodificação considerando-se os códigos c_i. A árvore correspondente é chamada *mínima*.

O valor na expressão acima é chamado de *comprimento ponderado de caminhos*, pois o fator p_i é o comprimento do caminho da raiz da árvore de decodificação até a folha M_i, e o fator f_i é uma ponderação que representa o número de vezes que tal caminho é percorrido durante as decodificações.

O método guloso aplicado ao problema de determinar os c_i que minimizem a expressão nos dá a seguinte solução: do conjunto C de freqüências relativas $\{f_1, f_2, ...,f_n\}$, escolhe-se o par de valores mínimos f e f'; substitui-se este par, em C, pela soma f+f', e repete-se este processo de escolha e substituição, até que apenas um valor reste no conjunto C. Em termos de árvore de

decodificação, os valores f_1, f_2, ..., f_n constituem as folhas, e a cada estágio de escolha de f e f'. Assim, o único valor que resta no conjunto, após todas as escolhas e substituições, corresponde à raiz da árvore de decodificação.

1.1 - Funcionamento do código de Huffman

Basicamente, o algoritmo monta uma tabela de freqüências da ocorrência de cada caracter, onde cada caracter é representado por um conjunto de bits de tamanho variado. Se utilizarmos um conjunto de bits fixo, dependendo do número de caracteres que tivermos, o espaço ocupado pelo arquivo pode ser bem maior. Segue abaixo um exemplo para se entender melhor:

Ex.: Temos 9 caracteres diferentes, portanto, se utilizarmos um comprimento fixo, serão necessários 4 bits para representar cada letra.

	a	b	c	d	e	f	g	h	i
Freqüência	50	25	15	40	35	25	10	25	25
comprimento fixo	0000	0001	0010	0011	0100	0101	0110	0111	1000
comprimento variável	11	0100	0001	011	001	0101	0000	100	101

Conforme esta na tabela, o texto tem 250 caracteres. Se estes estiverem codificados com tamanho fixo de bits, o arquivo ocuparia 250x4 bits = 1000 bits. Porém, se for utilizado o código de Huffman para se fazer uma nova codificação, o arquivo terá (50*2 + 25*4 + 15*4 + 40*3 + 35*3 + 25*4 + 10*4 + 25*3 + 25*3)=775 bits, uma economia de 22,5%. Este índice não corresponde a economia verdadeira que o código de Huffman pode proporcionar. Deve-se levar em conta que quanto mais equilibrada forem as ocorrências de cada caracter, menor será a economia.

A idéia do algoritmo de Huffman é a utilização de uma árvore binária, onde as folhas são os respectivos caracteres. Montaremos passo a passo essa árvore para o exemplo demonstrado acima:

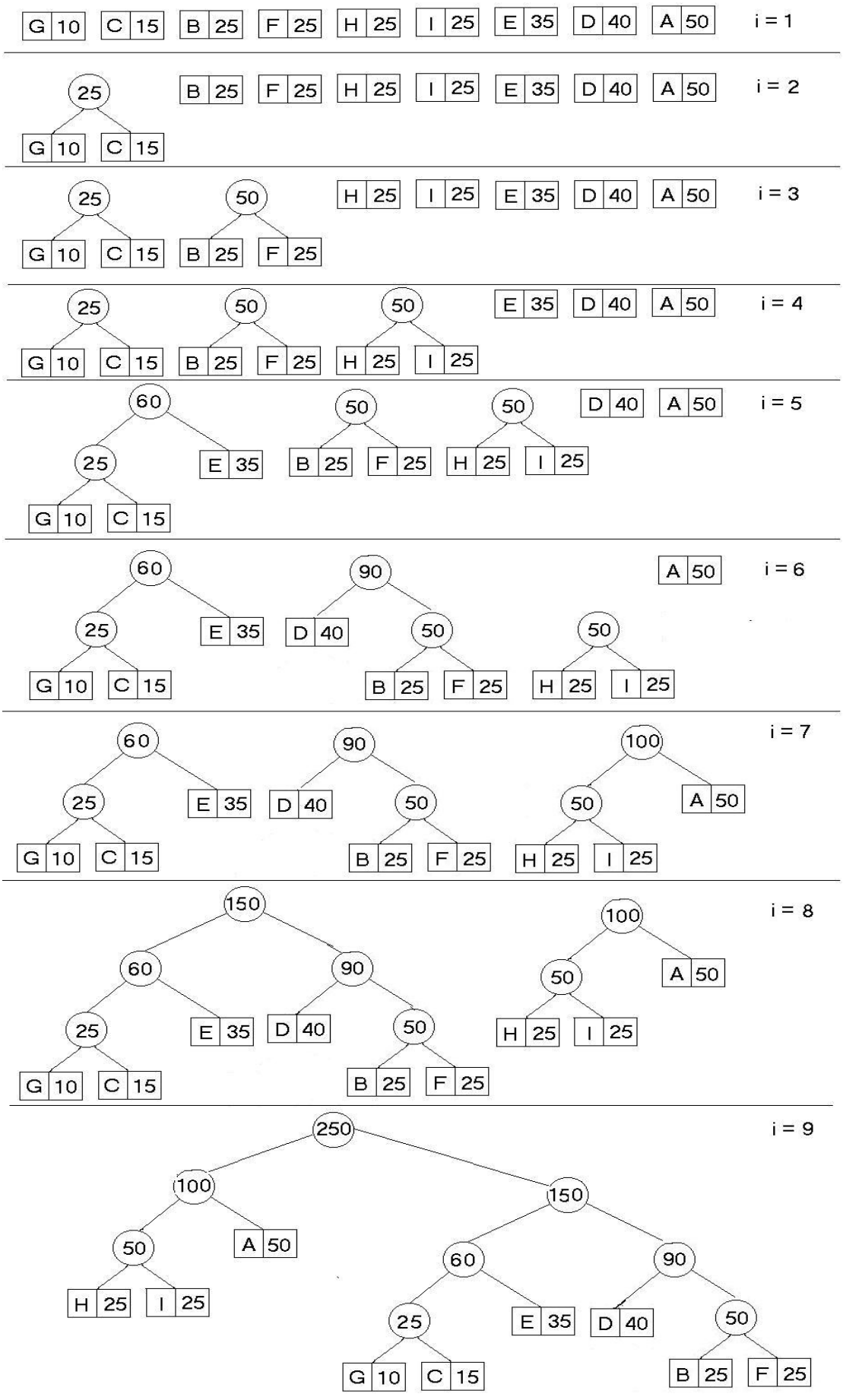
a) Primeiro, é preciso fazer uma fila de espera ordenada crescentemente pela ordem de ocorrência de cada caracter:

	g	c	b	f	h	i	e	d	a
Freqüência	10	15	25	25	25	25	35	40	50

b) Depois disso, soma-se, a cada iteração, a soma das duas freqüências menores, o que dá origem a 2 nodos folhas. Os passos são demonstrados na figura na próxima página.

c) Para se montar o código, pega-se "0" cada vez que for para a esquerda e "1" cada vez que for para a direita. Assim o código formado fica o seguinte:

	a	b	c	d	e	f	g	h	i
comprimento variável	01	1110	1001	110	101	1111	1000	000	001


G 10 C 15 B 25 F 25 H 25 I 25 E 35 D 40 A 50
i = 1
25
G 10 C 15
B 25 F 25 H 25 I 25 E 35 D 40 A 50
i = 2
25 50
G 10 C 15 B 25 F 25
H 25 I 25 E 35 D 40 A 50
i = 3
25 50 50
G 10 C 15 B 25 F 25 H 25 I 25
E 35 D 40 A 50
i = 4
60 50 50
25 E 35 B 25 F 25 H 25 I 25
G 10 C 15
D 40 A 50
i = 5
60 90
25 E 35 D 40 50 50
G 10 C 15 B 25 F 25 H 25 I 25
A 50
i = 6
60 90 100
25 E 35 D 40 50 50 A 50
G 10 C 15 B 25 F 25 H 25 I 25
i = 7
150
60 90 100
25 E 35 D 40 50 50 A 50
G 10 C 15 B 25 F 25 H 25 I 25
i = 8
250
100 150
50 A 50 60 90
H 25 I 25 25 E 35 D 40 50
G 10 C 15 B 25 F 25
i = 9

Observando que esta tabela será de grande importância para a descompressão do arquivo.

1.2 – Análise da Complexidade

A complexidade de tempo do algoritmo de Huffman é **O(n logn)**. Podemos observar que usando um heap para armazenar o peso de cada árvore, cada iteração requer **O(logn)** para determinar o peso menor e inserir o novo peso. Existem **O(n)** iterações, uma para cada item. Portanto a complexidade é **O(n logn)**.

1.3 – Um algoritmo

O código de Huffman implementado a seguir está no livro: Introduction to Algorithms de Thomas H. Cormen et alli, McGraw-Hill, 1997.

Onde, as entradas são (C,n) que é o conjunto de n freqüências relativas f_1, f_2, ..., f_n; e a saída é (r) que é a raiz da árvore binária desejada.

```
HUFFMAN(C)
    n recebe |C|
    Q recebe C
    Para i de 1 até (n – 1)faça
        alocar um novo nó z
        x recebe EXTRACT-MIN(Q)
        y recebe EXTRACT-MIN(Q)
        esquerda[z] recebe x
        direita[z] recebe y
        f[z] recebe f[x] + f[y]
        INSERT(Q,z)
    Pare com EXTRACT-MIN(Q)
```

1.4 - Atividade

1.4.1 - Usando a teoria dos códigos de Huffman para compressão de dados faça:

- Inicialmente selecionar um texto qualquer com aproximadamente uma página.
- Contar cada um dos caracteres do texto e formar uma tabela.
- Desenvolver a tabela de freqüência dos caracteres.
- Desenvolver a árvore binária.
- Criar a tabela de comprimento variável.
- Fazer a conclusão sobre a compressão de dados e responda qual a economia de bytes com a utilização da tabela de compressão variável em relação a uma tabela fixa.

1.4.2 - Dada a tabela de freqüência abaixo,

Caracter	J	B	M	H	G	R	F	S	I	E	O	A
Freqüência	10	14	18	20	22	25	28	30	33	35	40	50

- Montar a tabela de códigos de tamanho fixo.
- Montar a árvore binária para gerar o código de tamanho variável.
- Montar a tabela de códigos de tamanho variável.
- Considerando as informações apresentadas na tabela acima e nos cálculos .desenvolvidos, qual a economia em bits para o código variável?

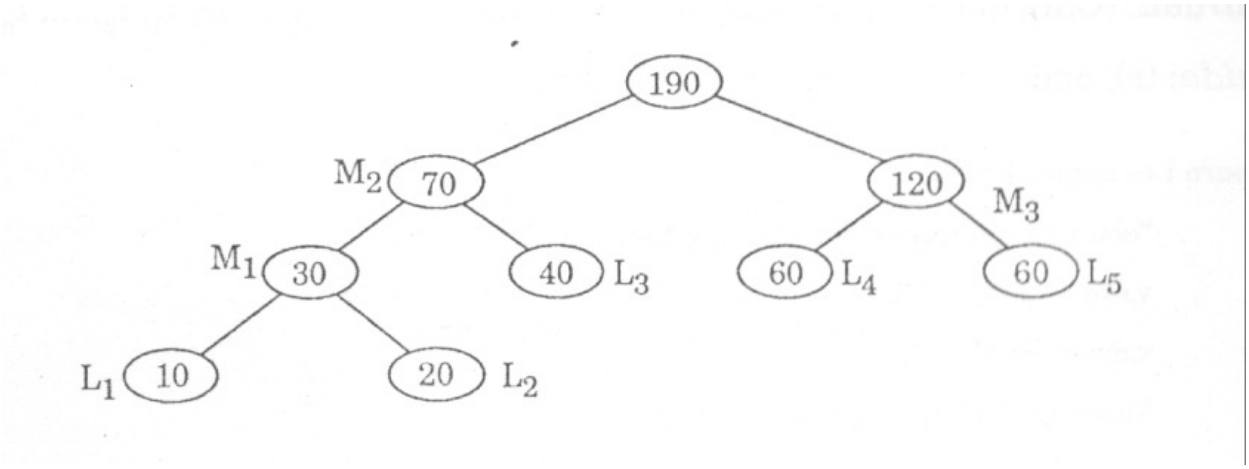
2 – SEQÜÊNCIA ÓTIMA DE INTERCALAÇÕES

Supondo que temos várias listas ordenadas para serem intercaladas, a lista ordenada final pode ser obtida através de intercalações sucessivas de pares de listas. Por exemplo, se as listas são L_1, L_2 e L_3, pode-se intercalar L_1 com L_2 resultando M e depois intercalar M com L_3, mas existem outras combinações possíveis. Para cada combinação de intercalação acarretam diferentes tempos de execução, pois exigem diferentes números de comparações.

Por exemplo, se as listas tem respectivamente 15, 10 e 5 elementos, a intercalação de L_1 com L_2 requer no máximo 25 comparações e a intercalação da lista resultante com L_3 requer 30 comparações no máximo, totalizando 55 comparações. Usando outra seqüência de intercalação com L_2 com L_3, tem-se um máximo de 15 comparações e em seguida o resultado com L_1, o número máximo será de 45 comparações.

O método guloso aplicado ao problema de determinar a seqüência ótima de emparelhamento de listas, apresenta a solução de que a cada estágio, escolhe-se o par de listas com menor número de elementos.

Um exemplo é ilustrado pela figura, onde a seqüência de emparelhamento é apresentada em forma de árvore binária. Cada folha representa uma lista, e cada vértice interno (que não é folha) representa a lista resultante da intercalação de duas listas representadas pelos filhos; cada vértice contém o número de elementos na lista correspondente. Esta árvore é denominada de **árvore binária de intercalação**.



Estas seqüências são chamadas de **seqüências de intercalações 2 a 2**, pois as intercalações são feitas aos pares. Uma generalização pode ser obtida considerando-se $k \geq 3$ listas sendo intercaladas simultaneamente, e deste modo, as seqüências de intercalações seriam representadas por árvores "k-árias" em que cada vértice interno tem k filhos.

2.1 - Descrição do Algoritmo

O conjunto L na entrada do algoritmo contém inicialmente elementos com a informação do comprimento de cada lista a ser intercalada: Compr(E) denota esta informação contida no elemento E. Estes elementos serão folhas na árvore binária de intercalação que se quer construir. Durante a execução do algoritmo, outros elementos que serão vértices internos N da árvore binária vão sendo criados, com o comprimento total das linhas representadas pelas folhas descendentes de N. Se o conjunto L é representado por uma árvore hierárquica, as operações EXTRACT-MIN e INSERT podem ser efetuadas eficientemente.

O algoritmo que é apresentado reflete a aplicação do método guloso para satisfazer o problema para a intercalação 2 a 2, este algoritmo está descrito no livro Desenvolvimento de Algoritmos e Estruturas de Dados de Routo Terada, editora Makron Books. Onde, as entradas são (L,n) que

são conjunto de n elementos que representam as n listas a serem intercaladas; e a saída é (r) que é a raiz da árvore binária desejada.

```
INTERCALA(L, n, r)
    Para i de 1 até (n – 1) faça
        "obtenha um novo elemento apontado por n";
        x recebe EXTRACT-MIN(L);
        y recebe EXTRACT-MIN(L);
        esquerda[z] recebe x ;
        direita[z] recebe y;
        Compr[z] recebe Compr[x] + Compr[y];
        INSERT(N,L);
    * o elemento remanescente em L é a raiz da árvore binária desejada *
Pare com EXTRACT-MIN(L)
```

Note que este algoritmo é estruturalmente idêntico ao da seção anterior, pois o problema é semelhante.

Considerando o exemplo apresentado, o vértice L_1 tem profundidade 3 e portanto, cada elemento de L_1 é comparado, no máximo 3 vezes: uma vez quando L_1 e L_2 são intercaladas, outra vez quando M_1 e L_3 são intercaladas e finalmente quando M_2 e M_3 são intercaladas. De um modo geral, se cada lista L_i contém c_i elementos, e se o vértice correspondente a L_i na árvore binária tem profundidade p_i, o número total máximo de comparações correspondente à árvore binária de intercalação é

$$\sum_{i=1}^{n} p_i \, . \, c_i$$

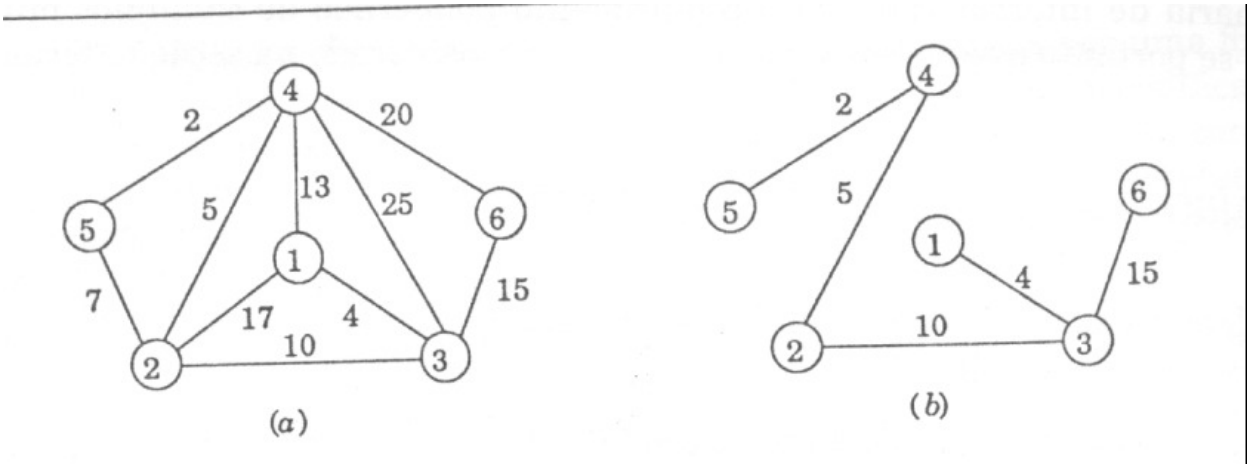
onde n é o número de linhas consideradas. A soma é chamada **comprimento ponderado de caminhos**. Uma seqüência de intercalação 2 a 2 ótima corresponde a uma árvore binária de intercalação com o comprimento ponderado de caminho mínimo.

2.2 – Análise da Complexidade

A complexidade de tempo do algoritmo é **O(n logn)**. Podemos observar isto na seção 1.2, onde está descrito o cálculo.

3 – ÁRVORE ESPALHADA MÍNIMA

Seja (VG, aG) um grafo não-orientado com um valor real, chamado custo, associado a cada aresta de G. Na figura vê-se um exemplo de grafo. Uma árvore espalhada é uma árvore não-orientada que conecta todos os vértices VG; mais formalmente, é uma árvore da forma E=(VG, aE) onde aE $\subseteq$ aG. O custo de uma árvore espalhada é simplesmente a soma dos custos das suas arestas. O problema é a determinação de uma árvore espalhada de custo mínimo para G, ou simplesmente, uma **árvore espalhada mínima** para G. Portanto, as árvores espalhadas são as soluções viáveis deste problema, e o custo da árvore espalhada é o valor da função objetivo que queremos minimizar.



Aplicações mínimas são frequentes, por exemplo em redes de circuitos elétricos. Se os vértices de G representam, por exemplo, n cidades e as arestas representam linhas de comunicação ou de transporte ligando duas cidades, então o número mínimo de linhas representam todas as soluções viáveis. Associadas às linhas temos, em geral, custos que representam, por exemplo, custos de transportes ou outros custos. Deseja-se então um conjunto de linhas conectando todas as n cidades e com o custo total (ou comprimento) mínimo. Então deseja-se determinar uma árvore espalhada de custo mínimo.

3.1 – Algoritmo

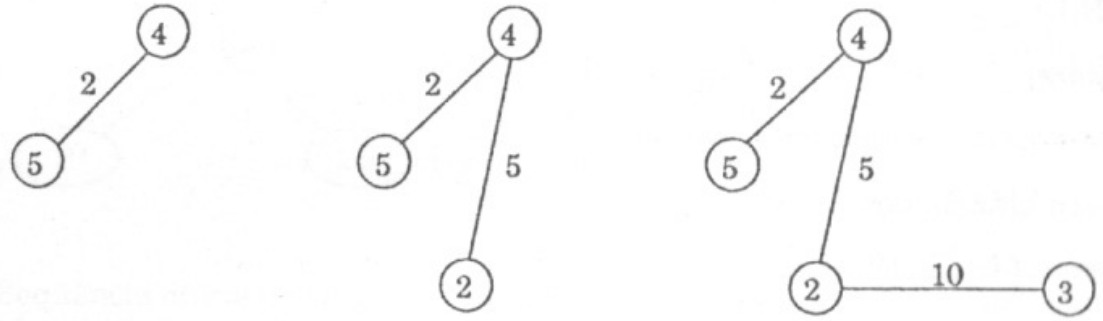
O método guloso aplicado a este problema de obtenção de uma árvore espalhada mínima sugere que a árvore seja construída aresta por aresta. A próxima aresta a ser escolhida deve satisfazer algum critério de otimização. O critério mais simples é o de escolher uma aresta que acarrete um incremento mínimo da soma de custos das arestas já escolhidas. Existem no entanto duas maneiras de se interpretar este critério.

A primeira diz que as arestas que vão sendo selecionadas devem sempre formar uma árvore. Assim, se A é o conjunto das arestas já selecionadas, A deve formar uma árvore. A próxima aresta (u,v) a ser incluída em A deve satisfazer as condições:

a) $A \cup \{(u, v)\}$ também é árvore;

b) (u, v) tem o menor custo entre as arestas não pertencentes a A.

O algoritmo correspondente a esta primeira interpretação é conhecido como algoritmo de Prim. A figura (na próxima página) ilustra a escolha das três primeiras arestas do grafo da figura anterior (letra a).



A segunda interpretação do critério de otimização que descrevemos acima dá origem ao algoritmo de Kruskal. As arestas do grafo são consideradas em ordem não decrescente dos custos. O conjunto A das arestas já selecionadas é uma floresta de árvores que poderá ser transformada em uma árvore quando novas arestas forem incluídas. Apresentamos a escolha das arestas pelo algoritmo, quando o grafo é o considerado no item anterior.

Ordem de escolha	aresta	custo
1	(4, 5)	2
2	(1, 3)	4
3	(2, 4)	5
rejeita-se	(2, 5)	7
4	(2, 3)	10
rejeita-se	(1, 4)	13
5	(3, 6)	15

Note-se que durante o processo de escolha das arestas em ordem não-decrescente de custo, algumas arestas são rejeitadas porque a sua inclusão em A acarretaria a formação de um circuito, e isto não é desejável pois A deve permanecer sendo uma floresta.

3.2 – O Algoritmo

Há uma solução genérica para este problema, apresentada abaixo, que serve de base para dois algoritmos bastante conhecidos para árvores espalhadas mínimas: os algoritmos de Kruskal e Prim. O algoritmo genérico mantém um conjunto de arestas *A* e a seguinte invariante:

A cada iteração, *A* é subconjunto do conjunto de arestas de uma árvore espalhada mínima.

O algoritmo mantém esta invariante adicionando ao conjunto *A*, a cada iteração, uma aresta *(u,v)* de maneira que a união de *A* com *(u,v)* seja um subconjunto de uma árvore espalhada mínima. Pelo fato de esta aresta manter a propriedade de *A* de ser um subconjunto de arestas de uma árvore espalhada mínima, ela é chamada de **aresta segura**.

O algoritmo genérico para Árvores Espalhadas Mínimas é apresentado logo abaixo:

```
GENERIC-MST(G,w)
    A := {}
    while A não forma uma árvore espalhada
        do encontre uma aresta (u,v) que é segura para A
           A := A união {(u,v)}
    return A
```

Dois outros conceitos relacionados a árvores espalhadas mínimas são os conceitos de **corte** em um grafo e de **arestas leves**.

Um **corte** em um grafo não direcionado *G=(V, E)* é uma partição de *V* em dois conjuntos (*S* e *V\S*). Uma aresta *(u,v)* pertencente a *E* ***cruza*** o corte *(S, V\S)* se um dos vértices adjacentes a ela está em *S* e o outro em *V\S*. Dizemos que um corte ***respeita*** um conjunto *A* de arestas se nenhuma aresta pertencente a *A* cruza o corte.

Uma aresta cruzando um corte é considerada uma **aresta leve** se seu peso for o menor dentre os pesos das arestas que cruzam o mesmo corte. Note que, embora o peso seja mínimo, várias arestas podem ter pesos iguais, o que significa que várias arestas podem ser uma aresta leve em um determinado corte.

A implementação do algoritmo de Kruskal, tal como mostrada no livro *Introduction to Algorithms* utiliza a estrutura de dados de conjuntos disjuntos para testar se dois vértices pertencem a uma mesma componente conexa ou não. A princípio, o conjunto de arestas é ordenado em ordem crescente de peso. As arestas são pegas uma a uma, em ordem, e o algoritmo executa uma chamada da função FIND-SET(*u*) para cada vértice adjacente a elas. Se os vértices não pertencerem ao mesmo conjunto, o que significa que a aresta une vértices de componentes distintas, a aresta é inserida em *A* e os conjuntos de *u* e *v* são unidos com uma chamada do procedimento UNION. Veja uma versão do algoritmo de Kruskal abaixo:

```
MST-KRUSKAL(G,w)
    A := { }
    for cada vértice v em V[G]
       do MAKE-SET(v)
    Ordene as arestas de E em ordem crescente de peso (w)
    for cada aresta (u,v), tomadas em ordem crescente de peso (w)
       do if FIND-SET(u) != FIND-SET(v)
            then A := A união {(u,v)}
                 UNION(u,v)
    return A
```

Assim como o algoritmo de Kruskal, o algoritmo de Prim também é um caso especial do algoritmo genérico. Neste caso, porém, ao invés de conectar árvores de uma floresta, o algoritmo mantém uma única árvore que cresce a cada iteração com a inserção de um vértice. Veja uma versão do algoritmo abaixo:

```
MST-PRIM(G,w,r)
      for cada u em V[G]
         do chave[u] := infinito
            pai[u] := NIL
      chave[r] := 0
      Q := V[G]
      while Q != { }
          do u := EXTRACT-MIN(Q)
             for cada v em Adj[u]
                 do if v pertence a Q e w(u,v) < chave[v]
                       then pai[v] := u
                            chave[v] := w(u,v)
```

Os parâmetros do procedimento MST-PRIM são um grafo conexo *G*, um vetor *w* de pesos associados às arestas de *G* e um vértice *r*, que será a raiz da árvore espalhada. Ao final da linha 5, o algoritmo se encontra na seguinte situação: todos os vértices de *G* estão em uma fila de prioridade mínima *Q*, e todos os vértices, com exceção de *r*, têm chave igual a infinito, enquanto *r* tem chave igual a 0.

No loop das linhas 6-11, os vértices de *G* são extraídos da fila de prioridades um de cada vez. Não é difícil ver que o primeiro vértice a sair é a raiz (*r*), uma vez que ele é o único elemento com uma chave menor que infinito até a linha 5. Toda vez que um vértice *v* sai da fila, todos os seus vizinhos são visitados e, se a chave de um vizinho for maior que o peso da aresta que liga *v* a este vizinho, tanto a chave quanto o pai do vizinho são alterados: a chave passa a ser o peso da aresta que liga este vizinho a *v* e o pai passa a ser o próprio vértice *v*.

Enquanto um vértice *v* está na fila, *pai[v]* aponta para o vértice vizinho de *v* que já está na árvore espalhada mínima e que é ligado a *v* pela aresta de menor peso entre todas as arestas que conectam *v* a um vértice atualmente na árvore. O peso da aresta (*v*, *pai[v]*) é o valor de *chave[v]* e, em qualquer iteração do comando *while* na linha 6, qualquer vértice *v* que está na fila *Q* com valor de *chave[v]* menor que infinito é vizinho de algum vértice da árvore espalhada mínima pela aresta *pai[v]*. Quando um vértice *v* sai da fila, tanto ele quanto a aresta que o liga a seu pai passam a pertencer à árvore espalhada mínima e seu pai nunca mais será alterado.

Ao final do algoritmo, o conjunto *A* retornado pelo algoritmo genérico corresponde ao conjunto formado pelas arestas *(v, pai[v])* de todos os vértices em *G* diferentes de *r*. O peso total da árvore é a soma dos valores *chave[v]*.

3.3 – Análise da Complexidade

A complexidade de tempo do algoritmo é **O(n log n)**. Podemos observar isto no algoritmo, onde para cada iteração uma única aresta é examinada e estas linhas são executadas no máximo n vezes.

4 – CAMINHO DE CUSTO MÍNIMO

Dados dois pontos A e B , várias vezes perguntamos:

i) existe um caminho de A para B ?

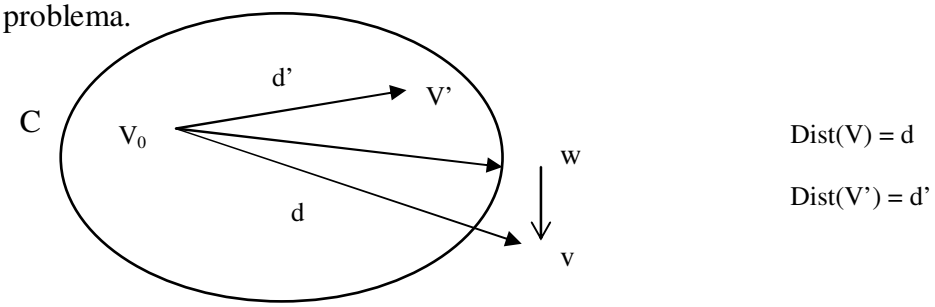
ii) se existe mais de um caminho de A para B, qual deles é mais curto?

Na prática estamos supondo que vários pontos são representados por vértices em um grafo orientado, e as arestas representam segmentos de caminhos cujos comprimentos, ou outra medida qualquer, são custos associados às arestas. E o custo de um caminho é a soma dos custos das arestas no caminho.

Cada pergunta do tipo (i) acima é trivial e pode ser respondida em tempo O(n), onde n é o número de arestas no grafo analisado. Pode-se determinar todos os pontos X tais que existe pelo menos um caminho de um ponto A para X.

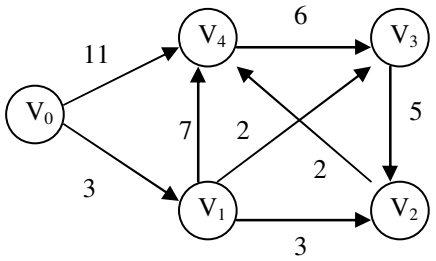
Para perguntas do tipo (ii) são casos especiais de um problema que resolveremos pelo método guloso. Queremos determinar os caminhos de custo mínimo que começam em um vértice fixo, denominado fonte, e terminam em outros vértices. Um algoritmo para resolver o problema de determinação dos caminhos de custos mínimos com a fonte fixa, tem a rapidez $O(n^2)$, onde n é o número de vértices. Não há explicação para que a rapidez não seja menor, tal como O(n).

O algoritmo constrói incrementalmente um conjunto C de vértices, que inicialmente só contém a fonte V_0. Para cada vértice V deve-se conhecer a distância mínima Dist(V), isto é o custo do caminho mínimo, da fonte V_0 até V, dentro de C, isto é considerando apenas caminhos que passam por vértices em C, exceto o próprio V. Portanto C é uma solução viável parcial do problema.



Em cada iteração, inclui-se em C o vértice w (ainda não pertencente a C) cujo Dist(w) é o mínimo em relação a todos os vértices não pertencentes a C. Portanto w é o novo objeto escolhido para incrementar a solução viável parcial, de acordo com a medida de otimização Dist.

Dada a figura abaixo, vê-se um exemplo de grafo de busca do caminho mínimo:



Para cada iteração, os incrementos feitos em C, o w escolhido é apresentado e os valores de Dist são calculados:

iteração	C	w escolhido	Dist				
			w	V_1	V_2	V_3	V_4
-	V_0	-	-	3	$+\infty$	$+\infty$	11
1	V_0, V_1	V_1	3	3	6	5	10
2	V_0, V_1, V_3	V_3	5	3	6	5	10
3	V_0, V_1, V_2, V_3	V_2	6	3	6	5	8
4	VG	V_4	8	3	6	5	8

Exercícios:

1) Dada a tabela de freqüência abaixo,

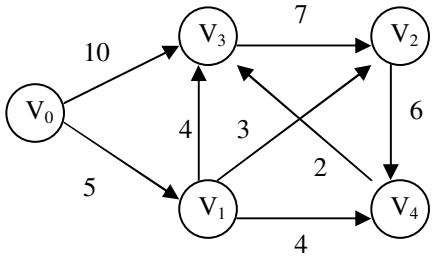
Caracter	J	B	D	C	M	H	G	R	F	S	I	E	O	A
Freqüência	10	13	14	18	20	21	22	25	28	30	33	35	40	45

a) Montar a tabela de códigos de tamanho variável.

b) Qual a complexidade do algoritmo da codificação de Huffman.

2) Montar a tabela de freqüência dos números arábicos que existem na pagina 2 desta apostila. Montar a tabela de códigos de tamanho variável, utilizando a codificação de Huffman.

3) Dado o grafo orientado com os custos, calcule o caminho mínimo de V_0 para V_4.



4) Dada a matriz abaixo que representa um grafo orientado com os seus respectivos custos, calcule o caminho mínimo de V_1 para V_6.

A)

Vértices	1	2	3	4	5	6
1	0	21	14	-	-	-
2	3	0	-	-	8	28
3	-	5	0	-	-	8
4	-	-	-	0	9	12
5	-	7	-	14	0	-
6	-	-	-	5	9	0

B)

Vértices	1	2	3	4	5	6
1	0	15	12	-	-	-
2	5	0	-	8	8	-
3	-	6	0	10	9	10
4	-	-	-	0	4	3
5	-	-	-	-	0	10
6	-	-	-	14	-	0