UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Campus de Presidente Prudente

RESOLUÇÃO DE PROBLEMAS UTILIZANDO TÉCNICAS DE TENTATIVA E ERRO, ALGORITMOS GULOSOS E PROGRAMAÇÃO DINÂMICA

PROJETO E ANALISE DE ALGORITMOS
PROFº. DRº. DANILO MEDEIROS ELER

BRUNO SANTOS DE LIMA
LEANDRO UNGARI CAYRES

PRESIDENTE PRUDENTE
DEZEMBRO - 2015

1. Associação de Tarefas (*Assignment Problem)*

1.1. Definição

O problema de associação de tarefas é um problema clássico que consiste em determinar relações entre dois agentes, o primeiro agente é aquele de realizara determinada tarefa ou função, o segundo agente é a tarefa em si, sendo assim vamos considerar o primeiro agente como um funcionário e o segundo agente uma tarefa, cada funcionário pode realizar qualquer uma das tarefas contudo cada um realiza uma tarefa gerando um determinado custo para a realização da mesma, custo este que é variável de acordo com o funcionário e com a tarefa que será realizada.

O problema de Associação de Tarefas visa estabelecer uma relação em que todas as tarefas são realizadas produzindo um menor custo possível, entretanto para o caso todos os funcionários devem realizar uma tarefa e todas as tarefas devem ser realizadas uma única vez, assim uma tarefa especifica não pode ser realizada mais de uma vez seja por um mesmo funcionário ou por funcionários diferentes.

Complexidade de tempo:

- Em seu pior caso a complexidade é a mesma que um algoritmo de força bruta.
- No caso médio e no melhor caso essa complexidade é melhorada.

1.2. Algoritmo

Para a resolução deste problema foi utilizada uma abordagem de tentativa e erro utilizando como base o algoritmo de Branch-And-Bound, observe abaixo os trechos de código principais utilizados na implementação, o algoritmo é executado até que todas as permutações forem realizadas:

```
public int branchAndBound(int custos [][]){

    this.tamanho = custos.length;
    this.matrizCusto = custos;

    int vetor [] = new int[this.tamanho];

    for(int i = 0; i < this.tamanho; i++){

        vetor[i] = i;
    }

    this.melhorCusto = custo(vetor);
    this.melhorSolucao = new int[this.tamanho];
    copia(melhorSolucao, vetor);
    permutacao(vetor, 0, this.tamanho);

    return(this.melhorCusto);
}

public void permutacao(int vetor [], int pos, int dim){

    if(pos == dim) custo(vetor);
    else{

        if(custo(vetor, pos) <= this.melhorCusto){

            for(int i = pos; i < dim; i++){

                troca(vetor,pos,i);
                permutacao(vetor, pos+1, dim);
                troca(vetor,pos,i);
            }

        }

    }//fim do else
}
```

1.3. Exemplo

Observe a seguir o conjunto de agentes de funcionários e de tarefas e suas respectivas soluções:

	Tarefa 1	Tarefa 2	Tarefa 3	Tarefa 4
Funcionário 1	13	4	7	6
Funcionário 2	1	11	5	4
Funcionário 3	6	7	2	8
Funcionário 4	1	3	5	9

Tabela 1- Conjunto exemplo 1

Para o conjunto exemplo 1 temos como a melhor solução que resulta em um menor custo:

- Funcionário 1 realiza tarefa 2 com custo 4
- Funcionário 2 realiza tarefa 4 com custo 4
- Funcionário 3 realiza tarefa 3 com custo 2
- Funcionário 4 realiza tarefa 1 com custo 1

Assim temos o conjunto solução: [4,4,2,1] resultando em um custo 11 no total.

	Tarefa 1	**Tarefa 2**	**Tarefa 3**	**Tarefa 4**
Funcionário 1	9	2	7	8
Funcionário 2	6	4	3	7
Funcionário 3	5	8	1	8
Funcionário 4	7	6	9	4

Tabela 2- Conjunto exemplo 2

Para o conjunto exemplo 2 temos como a melhor solução que resulta em um menor custo:

- Funcionário 1 realiza tarefa 2 com custo 2
- Funcionário 2 realiza tarefa 1 com custo 6
- Funcionário 3 realiza tarefa 3 com custo 1
- Funcionário 4 realiza tarefa 4 com custo 4

Assim temos o conjunto solução: [2,6,1,4] resultando em um custo 13 no total.

2. Codificação de Huffman

2.1. Definição

A codificação de Huffman é um método de compactação de um conjunto de símbolos atribuindo um código a cada um deles de acordo com a frequência em que eles aparecem sendo que esses códigos por sua vez tem um tamanho variável para cada símbolo.

No caso utilizamos um texto de entrada fornecido pelo usuário, esse texto pode ser uma frase ou uma palavra, onde cada caractere deste texto de entrada representa um símbolo e posteriormente terá a atribuição de um código único para o mesmo, essa

codificação visa utilizar um número menor de bits para representar o texto de entrada, é utilizado mais bits para representar caracteres que são utilizados com menos frequência e menos bits para representar caracteres que aparecem com maior frequência, com essa ideia conseguimos reduzir o número de bits necessários para representar no caso o texto de entrada.

Para a implementação é utilizada uma estrutura de árvore binária onde as folhas da árvore armazena os caracteres e o caminho da raiz até uma determinada folha representa a sequência de bits, ou seja, a codificação do caractere folha.

Complexidade de tempo: O(nlogn).

2.2. Algoritmo

Abaixo temos um pseudocódigo da função principal de Huffman, que dada um String de entrada realiza a codificação da mesma:

```
String Comprimir(String texto){
    if(String texto não é valida) return(null);
     EstruturaArmazena CaracterFrequencia = obterCaracteresFrequencia(texto);
     No raiz = criarArvoreHuffmanRetornaRaiz(CaracterFrequencia);
     EstruturaArmazena CodigoCaracter = obterCodigoCaracter(raiz);
     String codificacao = obterCodificacaoFinal(CodigoCaracter);
     return(codificação);
}
```

A função de Comprimir foi dividida em outras funções visando um melhor entendimento da mesma e maior modularidade.

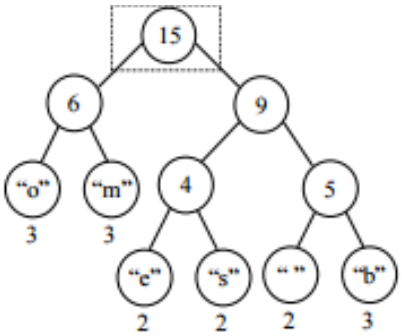
2.3. Exemplo

Para exemplificar considere o texto de entrada sendo: “bom esse bombom”, agora observe a tabela a seguir que contém os caracteres presentes no texto de entrada, a frequência em que eles aparecem no texto e sua codificação inicial que será otimizada.

Caractere	Frequência	Codificação inicial
B	3	000
O	3	001
M	3	010
E	2	011
S	2	100
Espaço	2	101

O próximo passo é organizara os caracteres de acordo com a ordem de frequência em que eles aparecem:

Depois é montada a árvore de Huffman juntando os caracteres de acordo com a sua frequência formando os nós da árvore, para formar o nó o valor do nó é a soma das frequências de cada caractere que contém o nó e o filho esquerdo e direito armazena os caracteres, assim temos no final a estrutura da árvore expressa abaixo:



Tendo a estrutura da árvore binaria montada é possível obter a codificação final do texto de entrada realizando um percurso na árvore binaria assim toda vez que percorremos para a esquerda armazenamos 0 e quando percorremos para a direita é armazenado 1. Logo temos a codificação de cada caractere segundo Huffman, observe a tabela abaixo:

Caractere	Codificação final
B	111
O	00
M	01

E	100
S	101
Espaço	110

Assim temos a codificação final expressa por "110001110100101101100110111000111100001", assim conseguimos reduzir a codificação inicial que utilizava 48 bits para representar o texto de entrada para uma representação que utiliza 39 bits.

3. Mochila Booleana (*Knapsack Problem*)

3.1. Definição

Inicialmente, temos uma mochila vazia com capacidade para *W* quilos, e temos um conjunto de *n* itens, cada qual com seu respectivo peso *w* e valor *v*. O objetivo deste algoritmo consiste em encher ao máximo esta mochila de forma que esta seja a mais valiosa possível.

Dado este conjunto, a abordagem mais simples para tal problema consiste em determinar todos os subconjuntos possíveis e analisar cada um destes de forma a encontrar a solução ideal. Porém, usando esta estratégia serão analisados 2^n subconjuntos, sendo que muitos destes são soluções improváveis, como arranjos com todos os itens ou nenhum destes.

Uma alternativa para este problema consiste no uso de divisão e conquista, mas para o emprego desta estratégia os subproblemas devem ser independentes, o que na maioria dos casos não ocorre promovendo esforço computacional repetitivo.

Por fim, a melhor técnica para a solução deste problema trata-se do uso de programação dinâmica, em que soluções anteriores são armazenadas em uma tabela sendo indicada a presença ou não do item e valor atual da mochila, sendo reutilizadas posteriormente, através de um uso maior de espaço em detrimento a redução do tempo.

Complexidade de tempo: $O(nW)$.

3.2. Algoritmo

Segue o pseudocódigo abaixo:

```
KnapSack(v, w, n, W)
{
  for (w = 0 to W) V[0, w] = 0;
  for (i = 1 to n)
    for (w = 0 to W)
      if (w[i] ≤ w)
        V[i, w] = max{V[i − 1, w], v[i] + V[i − 1, w − w[i]]};
      else
        V[i, w] = V[i − 1, w];
  return V[n, W];
}
```

3.3.Exemplo

Dado o seguinte conjunto de entrada, no modelo valor/peso: (10, 5), (40, 4), (30, 6) e (50, 3), sendo a mochila com capacidade para 10. Podemos formar a seguinte tabela, os valores em negrito representam a presença do item da respectiva linha na mochila.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	**10**	**10**	**10**	**10**	**10**	**10**
2	0	0	0	0	**40**	**40**	**40**	**40**	**40**	**50**	**50**
3	0	0	0	0	40	40	40	40	40	50	**70**
4	0	0	0	**50**	**50**	**50**	**50**	**90**	**90**	**90**	**90**

Para este exemplo, temos como uma solução a escolha dos itens 2 e 4, ou seja, (40, 4) e (50, 3).

4. Mochila Fracionária (*Fractional Knapsack Problem*)

4.1. Definição

Como no algoritmo anterior, temos uma mochila, inicialmente vazia, com uma capacidade de *W* quilos, e cada item possui um dado valor *v* e seu respectivo peso *w*, objetivando preencher a mochila de forma que esta seja a mais completa e valiosa possível. O grande diferencial trata-se na possibilidade de ocupar a mochila não somente com os itens por completo, mas sim partes desses, de forma que os pesos e seus valores sejam diretamente proporcionais as duas quantias utilizadas.

A estratégia para solução também consiste no uso de programação dinâmica, através da tabela é indicada quanto, em porcentagem, do item é utilizado e consequentemente o valor da mochila.

Complexidade de tempo:

- Caso os itens estejam em ordem decrescente de valor/peso, temos $O(n)$.
- Caso contrário, é necessária a ordenação, requerendo $O(n\ log\ n)$.
- Também é possível o uso da estrutura de dados Heap, sendo necessária para a sua construção $O(n)$, e para a solução $O(\log n)$.

4.2. Algoritmo

Abaixo temos o pseudocódigo que preenche a mochila com os itens com maior índice valor/peso de forma que esta seja a mais valiosa possível.

Greedy-fractional-knapsack (*w, v, W*)

```
FOR i =1 to n
   do x[i] =0
weight = 0
while weight < W
   do i = best remaining item
      IF weight + w[i] ≤ W
         then x[i] = 1
            weight = weight + w[i]
         else
            x[i] = (w - weight) / w[i]
```

weight = *W*
return *x*

4.3. Exemplo

Dada uma mochila com capacidade para dez e os seguintes itens: (2,2), (10, 1), (9, 7), (3,1), (5, 2) e (6,4).

Este conjunto de itens deve ser ordenado de forma decrescente para a seguinte razão (valor/peso), desta forma estarão dispostos do seguinte modo: (10, 1), (3,1), (5,2), (6,4), (9, 7) e (2,2).

A partir desta etapa, são colocados os itens que cabem por completo no respectivo espaço disponível. Então os seguintes itens são colocados sem ser necessária quebra alguma: (10, 1, 100%), (3,1, 100%), (5,2, 100%) e (6,4, 100%).

Neste ponto, temos um espaço de tamanho 2 disponível, porém o próximo item necessita de 7, devemos quebra-lo para que ocupe todo o espaço remanescente, logo a lista final de itens na bolsa será: (10, 1, 100%), (3,1, 100%), (5,2, 100%), (6,4, 100%) e (9 ,7, 28%).

5. Subsequência Máxima Comum (*Longest Common Subsequence*)

5.1. Definição

Dadas duas sequências de caracteres, busca-se determinar o comprimento da maior subsequência presente em ambas cadeias, sendo entendido como subsequência como uma cadeia de caracteres na mesma ordem, porém não necessariamente contíguos. Através do uso da álgebra elementar, é possível deduzir que a partir de uma cadeia de n caracteres são possíveis 2^n subsequências.

A solução deste algoritmo pode ser implementada através do uso de uma árvore de recursão para as subsequências a serem analisadas, possibilitando que uma mesmo subproblema seja resolvido muitas vezes. Desta forma, a melhor alternativa consiste no uso de programação dinâmica, armazenando em uma tabela de forma a evitar a repetição de cálculos.

Complexidade de tempo:

- Para a implementação com árvore recursiva, temos $O(2^n)$.
- Com o uso de programação dinâmica, temos um significativo ganho de complexidade, reduzindo para $O(mn)$.

5.2. Algoritmo

Logo abaixo, temos o código em Java responsável pela criação da tabela que determina as semelhanças entre as cadeias de caracteres:

```
int[][] tabela = new int[palavra01.length()+1][palavra02.length()+1];
int i, j;

for(i = 0; i <= palavra01.length(); i++){

    for(j = 0; j <= palavra02.length(); j++){

        if(i == 0 || j == 0) tabela[i][j] = 0;
        else if(palavra01.charAt(i-1) == palavra02.charAt(j-1)) tabela[i][j] = tabela[i-1][j-1] + 1;
        else tabela[i][j] = Integer.max(tabela[i-1][j], tabela[i][j-1]);
    }
}

tamanho = tabela[palavra01.length()][palavra02.length()];
```

5.3. Exemplo

Dada as seguintes cadeias “bola” e “corda”, temos:

	0	B	O	L	A
0	0	0	0	0	0
C	0	0	0	0	0
O	0	0	1	1	1
R	0	0	1	1	1
D	0	0	1	1	1
A	0	0	1	1	2

	0	B	O	L	A
0	0	0	0	0	0
C	0	↑	↑	↑	↑
O	0	↑	↖	←	←
R	0	↑	↑	↑	↑
D	0	↑	↑	↑	↑
A	0	↑	↑	↑	↖

A cadeia máxima comum deste exemplo é “oa”, cujo tamanho é 2.

6. Multiplicação de Cadeia de Matrizes (*Matrix Chain Multiplication*)

6.1. Definição

Entre as diversas aplicações da programação dinâmica tem-se um problema clássico e popular que utiliza desta técnica de programação para encontrar sua solução, este problema em questão é conhecido como Multiplicação de Cadeia de Matrizes.

Este problema visa a otimização, no contexto inicial considere um conjunto de matrizes, onde é necessário que as dimensões dessas matrizes sejam previamente conhecidas, assim quando é necessitamos multiplicar esses conjuntos de matrizes podemos efetuar a multiplicação por diversas ordens diferentes, entretanto existem uma sequência especifica que minimiza o número de operações necessárias para a realização dessa multiplicação e é exatamente essa sequência especifica que é o objetivo e o conjunto solução deste problema.

Complexidade de tempo:

- A complexidade do procedimento é O(n^3).

6.2. Algoritmo

Abaixo temos dois pseudocódigos que demonstram o comportamento do algoritmos utilizado para mostrar as matrizes soluções e o segundo responsável por obter a melhor sequência especifica da matriz solução e mostra-la de forma textual.

```
MATRIX-CHAIN-ORDER(p)
 1  n ← length[p] − 1                    O(n³), Ω (n³) ➔Θ(n³) running time
 2  for i ← 1 to n                       Θ(n²) space
 3      do m[i, i] ← 0
 4  for l ← 2 to n          ▷ l is the chain length.
 5      do for i ← 1 to n − l + 1
 6             do j ← i + l − 1
 7                m[i, j] ← ∞
 8                for k ← i to j − 1
 9                    do q ← m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
10                       if q < m[i, j]
11                          then m[i, j] ← q
12                               s[i, j] ← k
13  return m and s
```

```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i = j
2     then print "A"_i
3     else print "("
4          PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5          PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6          print ")"
```

6.3. Exemplo

Para exemplificar o problema considere o seguinte conjunto de matrizes que serão multiplicadas:

Matriz	Linha	Coluna
A[0]	10	100
A[1]	100	5
A[2]	5	50
A[3]	50	1

Assim para multiplicarmos as matrizes acima temos as seguintes ordem descritas abaixo:

- (A[0](A[1](A[2]A[3]))) = 1750 operações no total.
  A[2]A[3] = 250 operações, resultando em uma 5 por 1.
  A[1]A[23] = 500 operações, resultando em uma 100 por 1

A[0]A[13] = 1000 operações, resultando em uma 10 por 1.

- ((A[0]A[1])(A[2]A[3])) = 5300 operações no total.
  A[0]A[1] = 5000 operações, resultando em uma 10 por 5.
  A[2]A[3] = 250 operações, resultando em uma 5 por 1.
  A[01]A[23] = 50 operações, resultando em uma 10 por 1.

- (((A[0]A[1])A[2])A[3]) = 8000 operações no total.
  A[0]A[1] = 5000 operações, resultando em uma 10 por 5.
  A[01]A[2] = 2500 operações, resultando em uma 10 por 50.
  A[02]A[3] = 500 operações, resultando em uma 10 por 1.

- ((A[0](A[1]A[2]))A[3]) = 75500 operações no total.
  A[1]A[2] = 25000 operações, resultando em uma 100 por 50.
  A[0]A[12] = 50000 operações, resultando em um 10 por 50.
  A[02]A[3] = 500 operações, resultando em uma 10 por 1.

- (A[0]((A[1]A[2])A[3])) = 31000 operações no total.
  A[1]A[2] = 25000 operações, resultando em uma 100 por 50.
  A[12]A[3] = 5000 operações, resultando em um 100 por 1.
  A[0]A[13] = 1000 operações, resultando em uma 10 por 1.

Assim com base no número mínimo de operações necessárias que causa uma grande otimização temos que a melhor ordem para multiplicar as matrizes acima é expressa pela sequência: (A[0](A[1](A[2]A[3]))).

7. Referências Bibliográficas

[1] LEVITIN, Anany. Introduction to the design & analysis of algorithms – 3rd ed. Pearson.

[2] SOARES, Henrique Carvalho de Almeida. Um estudo sobre o problema de alocação. UNIFESP. Disponível em: < http://www.ft.unicamp.br/docentes/meira/publicacoes/2011henrique.pdf > Acesso em: 21 Dez. 2015.

[3] COSTA, Patrícia Dockhorn. A codificação de Huffman. Disponível em < http://www.inf.ufes.br/~pdcosta/ensino/2009-1-estruturas-de-dados/material/CodificacaoHuffman.pdf > Acesso em: 22 Dez. 2015.

[4] Otten, Ralph. The Knapsack Problem. Disponível em: < http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf > Acesso em: 21 Dez. 2015.

[5] GeeksforGeeks,. Dynamic Programming | Set 4 (Longest Common Subsequence) - GeeksforGeeks. Disponível em: <http://www.geeksforgeeks.org/dynamic-programming-set-4-longest-common-subsequence/>. Acesso em: 21 Dez. 2015

[6] Personal.kent.edu,. Chain Matrix Multiplication. Disponível em: <http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Dynamic/chainMatrixMult.htm>. Acesso em: 5 jan. 2016.