



**UNIVERSIDADE ESTADUAL PAULISTA
“JÚLIO DE MESQUITA FILHO”**

Ciência da Computação

**Projeto e Análise de Algoritmos
Relatório Trabalho Prático II**

Conceitos e solução prática de problemas envolvendo diferentes paradigmas e técnicas de projetos de algoritmos.

Profº Danilo Medeiros Eller
Darlan Murilo Nakamura de Araújo RA:151251207
Matheus Palmeira Gonçalves dos Santos RA:151256349

Presidente Prudente - SP

Introdução

O Trabalho Prático II tinha como objetivo solucionar os seguintes problemas utilizando as técnicas de algoritmo indicadas:

1. Problema de Associação de Tarefas (Assignment Problem)

-Utilizando a técnica de **Tentativa e Erro** com **Branch and Bound**

2. Codificação de Huffman

-Utilizando um **Algoritmo Guloso** que codificaria um texto fornecido pelo usuário

3. Problema da Mochila Fracionária (Fractional Knapsack Problem)

- Utilizando uma técnica de **Algoritmo Guloso** para solucionar o problema

4. Problema da Mochila Booleana (Knapsack Problem)

-Utilizando o método de **Programação Dinâmica**

5. Problema da Subsequência Comum Máxima (Longest Common Subsequence)

-Utilizando o método de **Programação Dinâmica**

6. Problema da Multiplicação de Cadeia de Matrizes (Matrix Chain Multiplication)

-Utilizando o método de **Programação Dinâmica**

Desenvolveu-se então um programa na linguagem Java, seguindo os requisitos da proposta de desenvolvimento:

- Interface intuitiva e de fácil uso
- Interfaces gráficas

Além disso, outro importante objetivo era promover o conhecimento das características dos diferentes métodos de algoritmos na implementação prática.

Desenvolvimento

1. Tela Principal

A tela inicial do programa apresenta botões intuitivos que dão acesso às interfaces de cada problema. Ao lado dos botões disponibilizamos uma área de exibição de texto com uma breve explicação sobre o que se trata cada problema do programa. Ao clicar em um dos botões dos problemas a área de texto é modificada para explicação do respectivo problema escolhido. Em seguida, abre-se uma caixa de instrução para utilização da interface do algoritmo selecionado e por fim abre-se a nova janela. Veja:

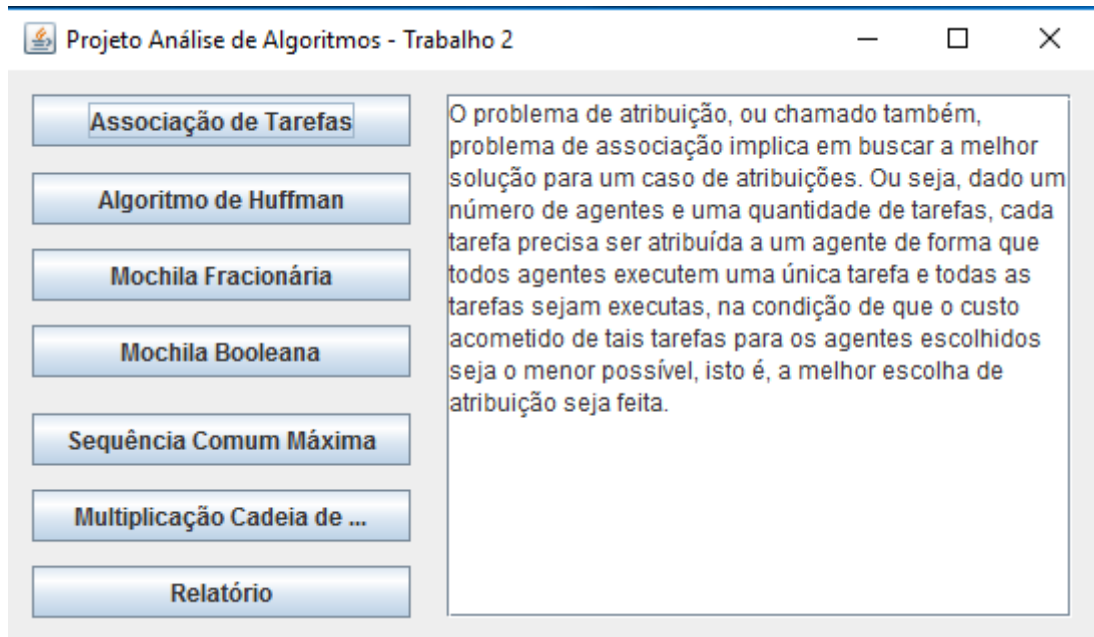


Figura 1. – Tela Inicial do programa

2. Associação de Tarefas

O problema de atribuição, ou chamado também, problema de associação implica em buscar a melhor solução para um caso de atribuições. Ou seja, dado um número de agentes e uma quantidade de tarefas, cada tarefa precisa ser atribuída a um agente de forma que todos agentes executem uma única tarefa e todas as tarefas sejam executadas, na condição de que o custo acometido de tais tarefas para os agentes escolhidos seja o menor possível, isto é, a melhor escolha de atribuição seja feita.

A técnica algorítmica que melhor soluciona essa problemática é o Branch and Bound, técnica baseada na construção de uma árvore de estados, na qual nos reflete uma escolha feita em direção a solução. O método enumera de forma inteligente as possíveis soluções, no entanto apenas uma fração das soluções candidatas, são examinadas de fato. O termo branch se refere as partições efetuadas pelo método, no espaço das soluções e o termo bound retrata o limite da otimização, calculados ao longo da enumeração, este que no desenvolvimento do algoritmo, foi descrito pela função de limitação ("PermutaMatriz();"). Ela possui grande importância para levar a melhor solução do problema em tempo de execução ótimo, pois elimina possíveis caminhos que não levarão a solução ótima, definindo um custo máximo baseado numa solução já encontrada ou um custo máximo inicial aplicado.

No programa, ao clicar no botão “Associação de Tarefas” o usuário vai receber uma caixa de instrução para utilizar a tela que se abrirá. Veja:

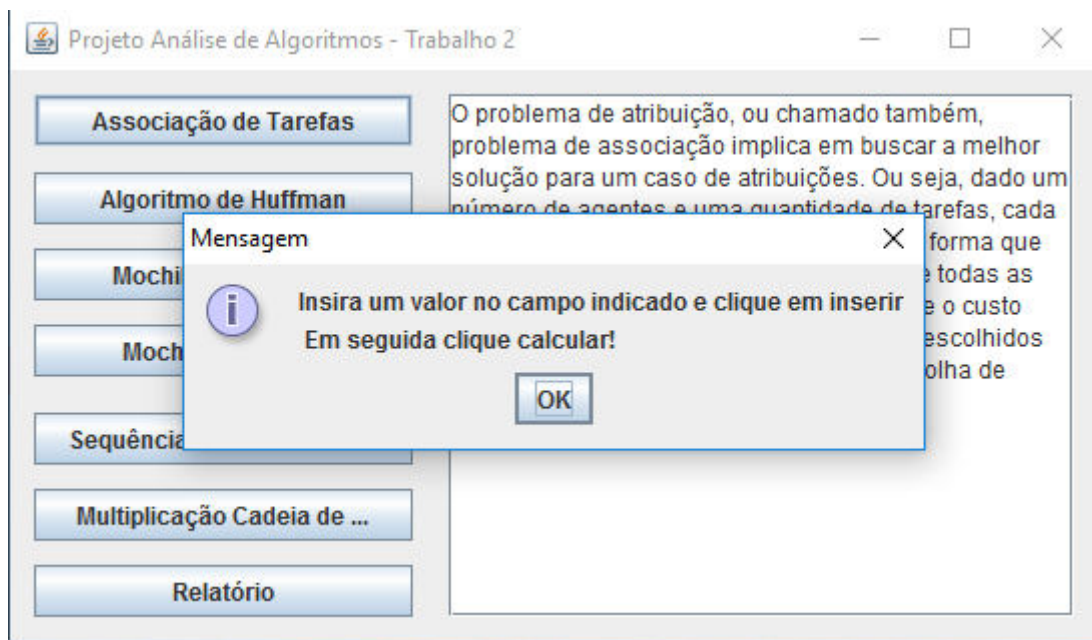


Figura 2. – Caixa de Instrução para Tela Associação de Tarefas

Em seguida, a Tela Associação de Tarefas se abrirá, bastando então o usuário seguir as instruções. Para exemplificar o funcionamento do programa iremos aplicar 6 tarefas para 6 pessoas. Assim que inserido, o programa solicitará o custo de cada Tarefa(k) para um agente(j), como na figura abaixo:

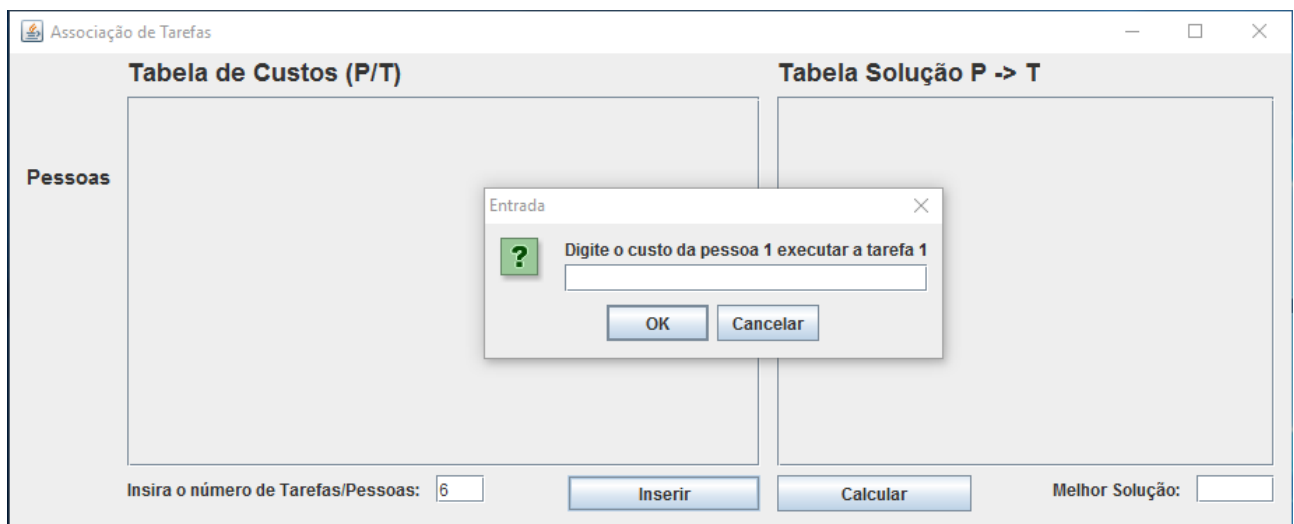


Figura 3. – Caixa de custo Tarefa(k)/Pessoa(j)

Após inserir os 6 custos de cada tarefa para as 6 pessoas e clicar em Calcular, teremos:

The screenshot shows a window titled 'Associação de Tarefas'. It contains two tables and some input fields.

Tabela de Custos (P/T)

| | Tarefa 1 | Tarefa 2 | Tarefa 3 | Tarefa 4 | Tarefa 5 | Tarefa 6 |
|----|----------|----------|----------|----------|----------|----------|
| 5 | 10 | 30 | 8 | 2 | 25 | |
| 10 | 20 | 7 | 3 | 40 | 12 | |
| 30 | 12 | 9 | 28 | 15 | 60 | |
| 16 | 4 | 2 | 36 | 40 | 8 | |
| 50 | 25 | 5 | 15 | 100 | 1 | |
| 6 | 1 | 60 | 30 | 49 | 80 | |

Tabela Solução P -> T

| Pessoa 1 | Pessoa 2 | Pessoa 3 | Pessoa 4 | Pessoa 5 | Pessoa 6 |
|----------|----------|----------|----------|----------|----------|
| Tarefa 5 | Tarefa 4 | Tarefa 3 | Tarefa 2 | Tarefa 6 | Tarefa 1 |

At the bottom, there is a label 'Insira o número de Tarefas/Pessoas:' with a text box containing '6', an 'Inserir' button, a 'Calcular' button, and a label 'Melhor Solução:' with a text box containing '25'.

Figura 4. – Tela Associação de Tarefas com a solução final

Pode-se observar que a solução foi conquistada e obteve-se a tarefa com menor custo associada ao respectivo agente, de forma que a solução geral também fosse a melhor. Para encontrar a solução, complexidade da execução do algoritmo no pior caso é de ordem exponencial.

3. Codificação de Huffman

A codificação de Huffman soluciona por meio da compressão de dados, a problemática de espaço para armazenamento. A ideia de Huffman é utilizar probabilidade de ocorrências para determinar códigos de tamanho variável e compactar o conjunto de dados. Na prática, dado um conjunto de dados o algoritmo verifica a ocorrência de símbolos para atribuir um código mínimo para o símbolo que possui maior ocorrência. Para isso, Huffman utiliza filas de prioridade e a construção de uma árvore que vai auxiliar tanto na compactação quanto na descompactação.

O código de Huffman é considerado um Algoritmo Guloso, ou seja, para resolver um problema ele escolhe o objeto mais “apetitoso” que vê pela frente (símbolo de maior ocorrência). O objeto escolhido passa a fazer parte da solução que o algoritmo constrói.

No programa desenvolvido, quando o usuário clica no botão “Algoritmo de Huffman” uma caixa de instrução aparecerá, auxiliando para a utilização do algoritmo na tela seguinte. Observe:

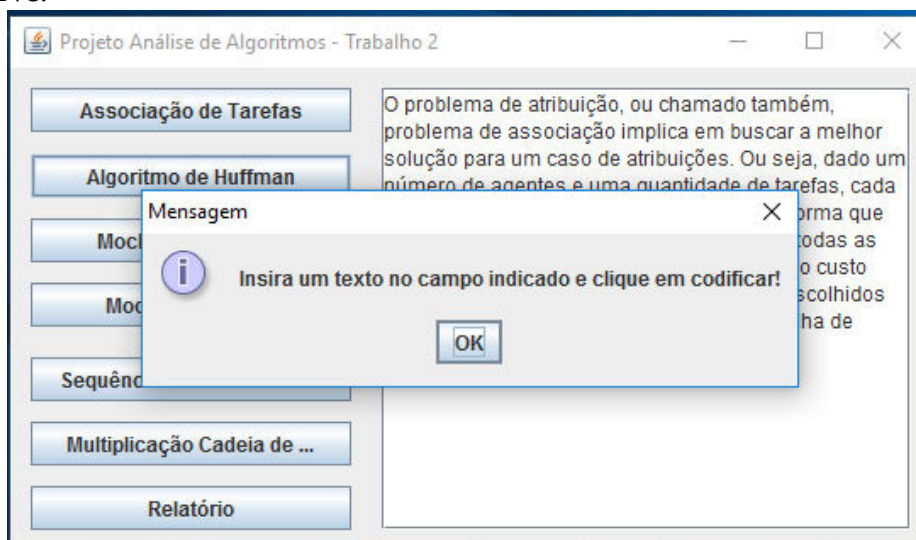
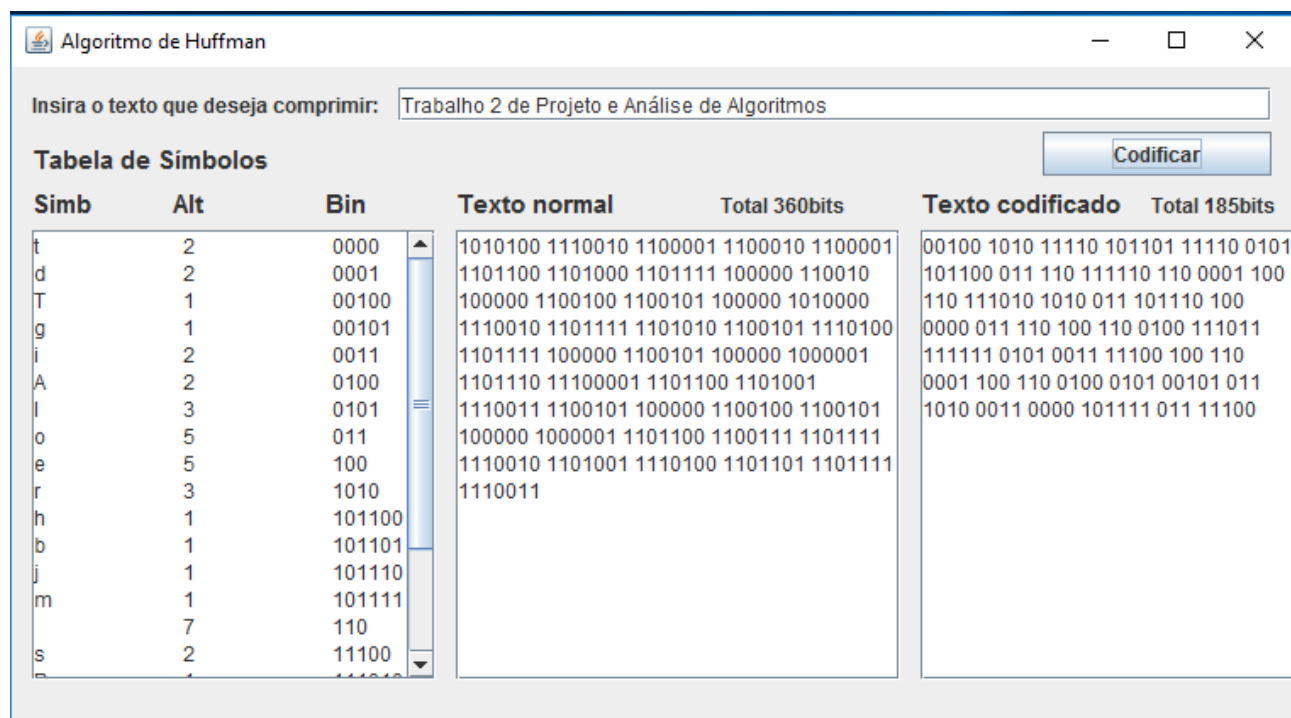


Figura 5. – Caixa de instrução Codificação de Huffman

Utilizaremos como exemplo o texto “Trabalho 2 de Projeto e Análise de Algoritmos”. Ao inserir o texto basta clicar em codificar e teremos a seguinte tela:



The screenshot shows a window titled "Algoritmo de Huffman". At the top, there is a text input field containing "Trabalho 2 de Projeto e Análise de Algoritmos" and a "Codificar" button. Below this is a table titled "Tabela de Símbolos" with columns: Simb, Alt, Bin, Texto normal, Total 360bits, Texto codificado, and Total 185bits. The table lists characters and their frequencies, along with their binary representations and the resulting Huffman codes.

| Simb | Alt | Bin | Texto normal | Total 360bits | Texto codificado | Total 185bits |
|------|-----|--------|---|---------------|------------------------------------|---------------|
| t | 2 | 0000 | 1010100 1110010 1100001 1100010 1100001 | | 00100 1010 11110 101101 11110 0101 | |
| d | 2 | 0001 | 1101100 1101000 1101111 100000 110010 | | 101100 011 110 111110 110 0001 100 | |
| T | 1 | 00100 | 100000 1100100 1100101 100000 1010000 | | 110 111010 1010 011 101110 100 | |
| g | 1 | 00101 | 1110010 1101111 1101010 1100101 1110100 | | 0000 011 110 100 110 0100 111011 | |
| i | 2 | 0011 | 1101111 100000 1100101 100000 1000001 | | 111111 0101 0011 11100 100 110 | |
| A | 2 | 0100 | 1101110 11100001 1101100 1101001 | | 0001 100 110 0100 0101 00101 011 | |
| l | 3 | 0101 | 1110011 1100101 100000 1100100 1100101 | | 1010 0011 0000 101111 011 11100 | |
| o | 5 | 011 | 100000 1000001 1101100 1100111 1101111 | | | |
| e | 5 | 100 | 1110010 1101001 1110100 1101101 1101111 | | | |
| r | 3 | 1010 | 1110011 | | | |
| h | 1 | 101100 | | | | |
| b | 1 | 101101 | | | | |
| j | 1 | 101110 | | | | |
| m | 1 | 101111 | | | | |
| s | 7 | 110 | | | | |
| . | 2 | 11100 | | | | |

Figura 6. – Tela Algoritmo de Huffman com a solução

O funcionamento do algoritmo começa a partir da análise do texto (String) inserido pelo usuário, ou seja, coleta-se o texto e se analisa a ocorrência dos símbolos que o texto contém. A partir das ocorrências conta-se as frequências, que serão importantes para definir o objeto mais “apetitoso”. Em seguida, com as frequências armazenadas é possível atribuir o prefixo (código binário) para cada símbolo e então montarmos a árvore de solução. Observe que a redução de bits é explícita!

A complexidade de tempo do algoritmo de Huffman é $O(n \log n)$. Pode-se observar que usando um “heap” para armazenar o peso de cada árvore, cada iteração requer $O(\log n)$ para determinar o peso menor e inserir o novo peso. Existem $O(n)$ iterações, uma para cada item.

4. Mochila Fracionária

Imaginando que, dada uma mochila e diversos objetos de pesos/valores diferentes deseja-se preencher a mochila com de diferentes objetos, de forma que a mochila armazene o máximo de valores possíveis sem ultrapassar o peso máximo. Tal problemática pode ser solucionada por um algoritmo guloso, neste caso, em cada iteração, o algoritmo “abocanha” o objeto de maior valor específico dentre os disponíveis, sem se preocupar com o que vai acontecer depois. O algoritmo jamais se arrepende do valor atribuído a um componente da mochila. Assim, o algoritmo produz uma mochila fracionária viável, ainda assim é necessário verificar se o resultado produzido é o valor máximo. O consumo de tempo do algoritmo é $O(n)$.

No programa desenvolvido, após clicar no botão “Mochila Fracionária” a partir da tela inicial, apresenta-se uma caixa de instrução:

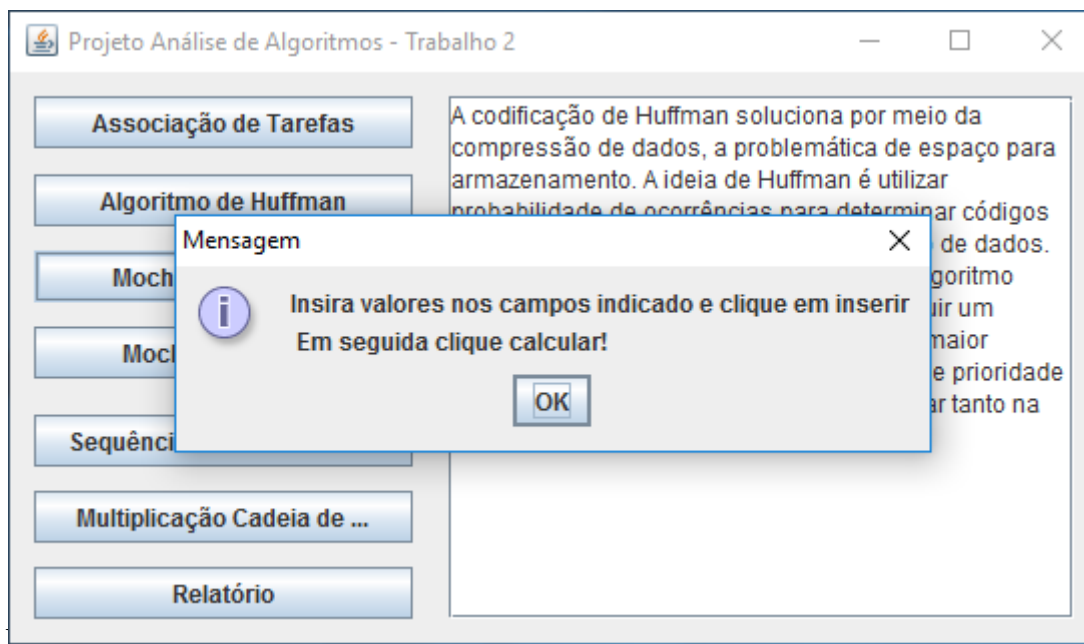


Figura 7. – Caixa de Instrução Mochila Fracionária

Utilizou-se o valor 20 como exemplo de capacidade da mochila e um total de 5 objetos. Preenchendo os campos e clicando em INSERIR têm-se:

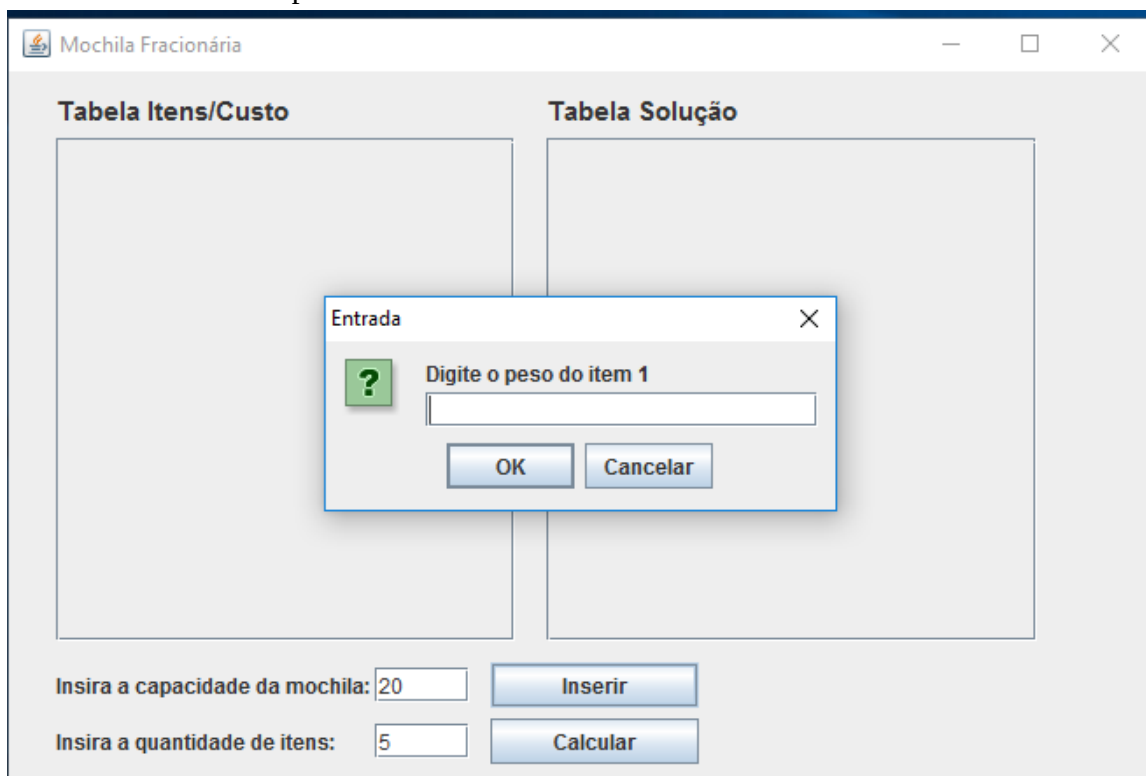
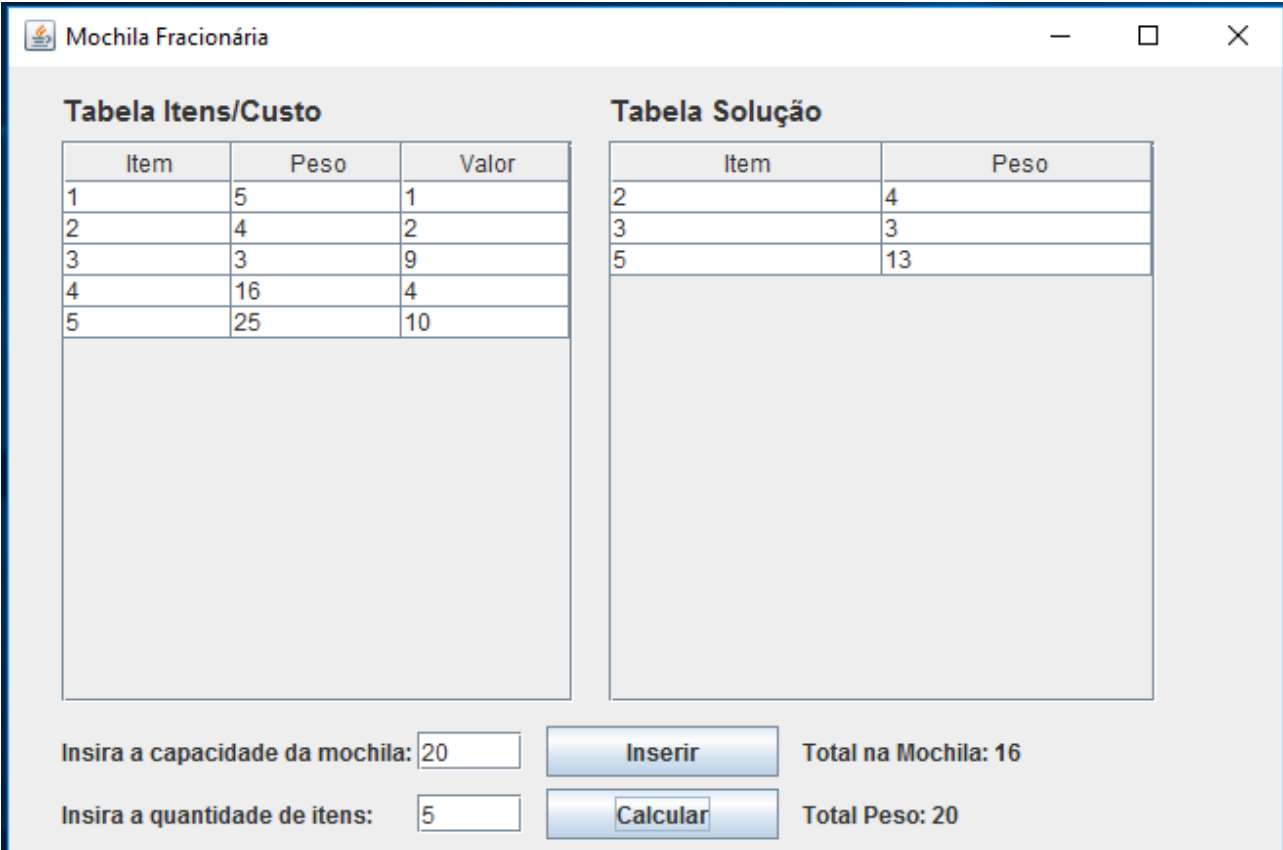


Figura 8. – Tela Mochila Fracionária preenchimento de campos

Em seguida, é necessário inserir o peso e custo do respectivo item[i] assim sucessivamente. Ao terminar basta clicar em Calcular, obtendo-se:



The screenshot shows a window titled "Mochila Fracionária" with two tables and input fields. The "Tabela Itens/Custo" table lists 5 items with their weights and values. The "Tabela Solução" table shows the selected items for the solution. Below the tables are input fields for capacity and item count, and buttons for "Inserir" and "Calcular".

| Item | Peso | Valor |
|------|------|-------|
| 1 | 5 | 1 |
| 2 | 4 | 2 |
| 3 | 3 | 9 |
| 4 | 16 | 4 |
| 5 | 25 | 10 |

| Item | Peso |
|------|------|
| 2 | 4 |
| 3 | 3 |
| 5 | 13 |

Insira a capacidade da mochila: Total na Mochila: 16

Insira a quantidade de itens: Total Peso: 20

Figura 9. – Tela Mochila Fracionária com a solução

5. Mochila Booleana

A problemática se baseia que, dada uma mochila com capacidade c , existem n itens com um peso e valor específicos, e deve-se ocupar a mochila da maneira que seja possível pegar os itens de maior valor e que a soma dos pesos dos itens seja inferior ou igual à capacidade suportada pela mochila. O algoritmo se baseia em uma tabela de valor e uma tabela booleana, a tabela de valor servirá para obtermos a resposta de qual é o maior valor que podemos carregar na mochila com capacidade c e a tabela booleana serve para recuperarmos os itens. O algoritmo utilizado é do tipo programação dinâmica (utilização de tabelas) e é simples: pegamos o item i , e verificamos se o seu valor é melhor que a solução anterior. Se sim, adicionamos a mochila e marcamos que ele pertence na tabela booleana. Após preenchermos toda a tabela, para recuperarmos os itens, vamos a ultima posição da tabela booleana, buscando encontrar 1, caso não seja possível, decrementamos a coluna, até encontrarmos 1. Ao encontrar, decrementamos do peso total da mochila o peso do item, e esse será nosso índice da coluna para onde devemos ir. Ao final, teremos os itens pertencentes à mochila. O algoritmo consome uma fatia de tempo da ordem $O(nC)$.

No programa desenvolvido, após clicar no botão “Mochila Booleana” a partir da tela inicial, aparecerá uma caixa de instrução:

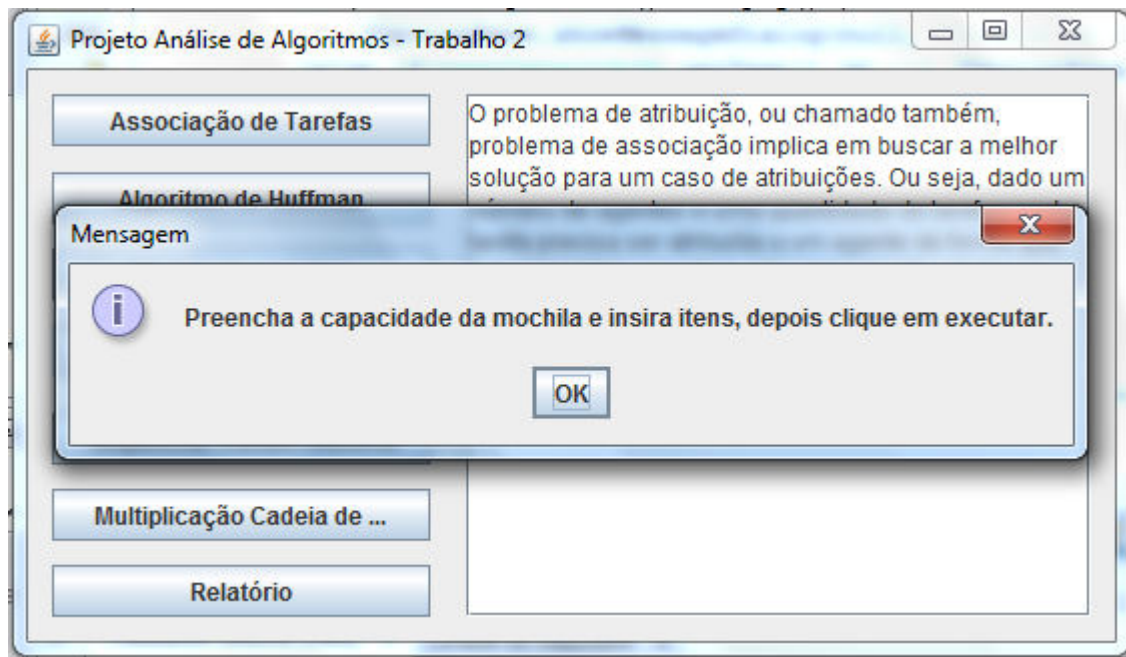


Figura 10. – Caixa de Instrução Mochila Booleana

Utilizou-se o valor 20 como exemplo de capacidade da mochila e um total de 5 objetos. Preenchendo o campo capacidade, é necessário inserir os itens clicando em “Adicionar Item”.

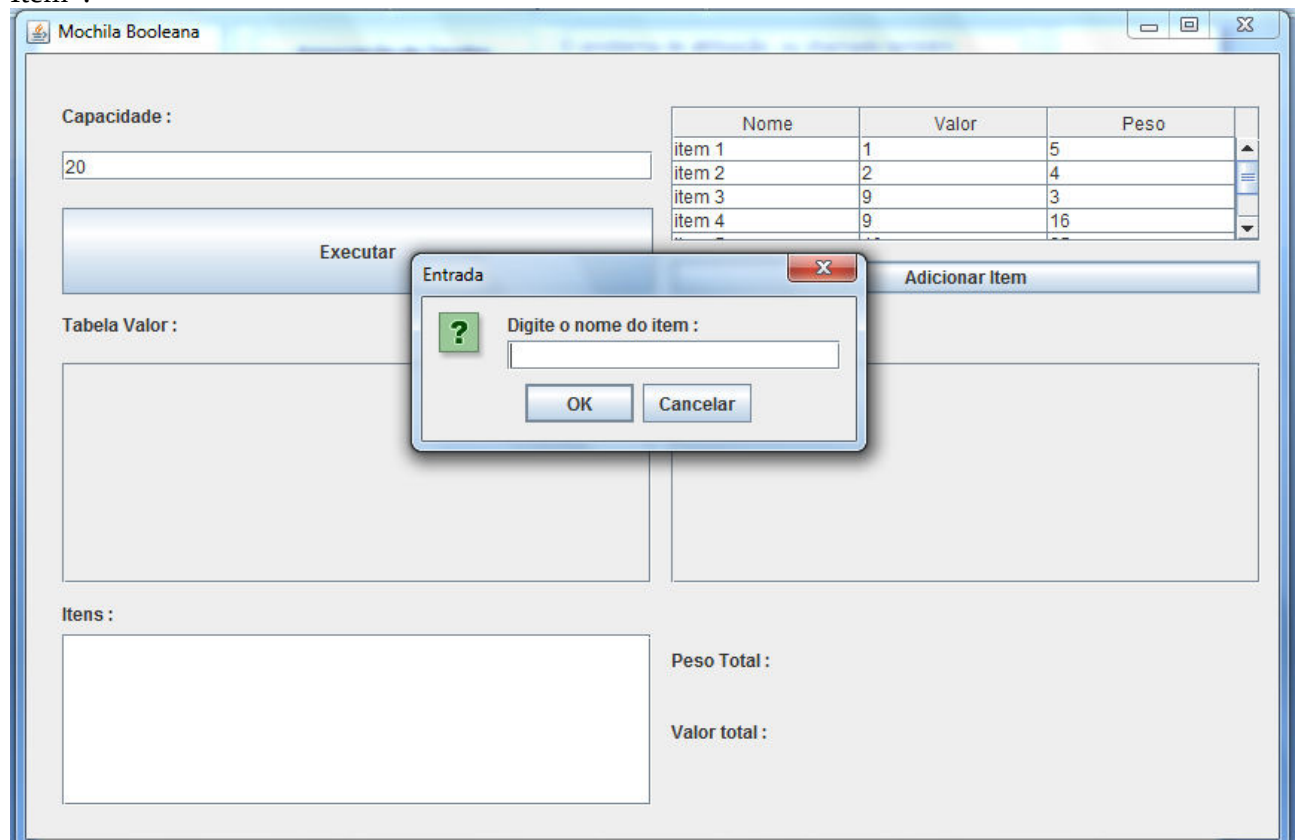


Figura 11. – Tela Mochila Booleana preenchimento de campos

Em seguida, é necessário inserir os respectivos campos relacionados ao item : nome, valor, peso. Após inserir os itens necessários, basta clicar em “Executar” para que as tabelas valor, booleana e itens seja preenchido, juntamente com os campos peso e valor total.

Mochila Booleana

Capacidade : 20

Executar

| Nome | Valor | Peso |
|--------|-------|------|
| item 1 | 1 | 5 |
| item 2 | 2 | 4 |
| item 3 | 9 | 3 |
| item 4 | 9 | 16 |

Adicionar Item

Tabela Valor :

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|------|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| i... | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| i... | 0 | 0 | 0 | 0 | 2 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 |
| i... | 0 | 0 | 0 | 9 | 9 | 9 | 9 | 11 | 11 | 11 | 11 | 11 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 |
| i... | 0 | 0 | 0 | 9 | 9 | 9 | 9 | 11 | 11 | 11 | 11 | 11 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 18 |
| i... | 0 | 0 | 0 | 9 | 9 | 9 | 9 | 11 | 11 | 11 | 11 | 11 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 18 |
| i... | 0 | 0 | 0 | 9 | 9 | 9 | 9 | 11 | 11 | 11 | 11 | 11 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 12 | 18 |

Tabela Booleana :

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
|--------|---|---|---|---|---|---|---|---|---|---|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| item 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| item 2 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| item 3 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| item 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |
| item 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| item 6 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| item 7 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

Itens :

- item 7
- item 3
- item 2

Peso Total : 19

Valor total : 18

Figura 12. – Tela Mochila Booleana com a solução

6. Sub Sequência Comum Máxima

A problemática se baseia que, dada duas cadeias de caracteres, encontrar a maior subsequência de caracteres que seja comum nas duas cadeias, ou seja, os caracteres que se repete em sua ordem em ambas as cadeias (os caracteres não precisam estar contíguos). O algoritmo utilizado para solucionar o problema é do tipo Programação Dinâmica. Utilizamos duas tabelas, uma tabela numérica que armazena o tamanho da subsequência comum máxima e uma tabela direção que serve para recuperarmos a subsequência comum máxima. Após recuperarmos pela tabela direção, ela deve ser invertida, pois o processo de recuperação é realizado de trás para frente (da última posição inicialmente). O algoritmo no pior caso leva $O(n^2)$ unidades de tempo no pior caso e $O(n)$ unidades de tempo no caso médio e melhor caso.

No programa desenvolvido, após clicar no botão “Sequência Comum Máxima” a partir da tela inicial, aparecerá uma caixa de instrução:

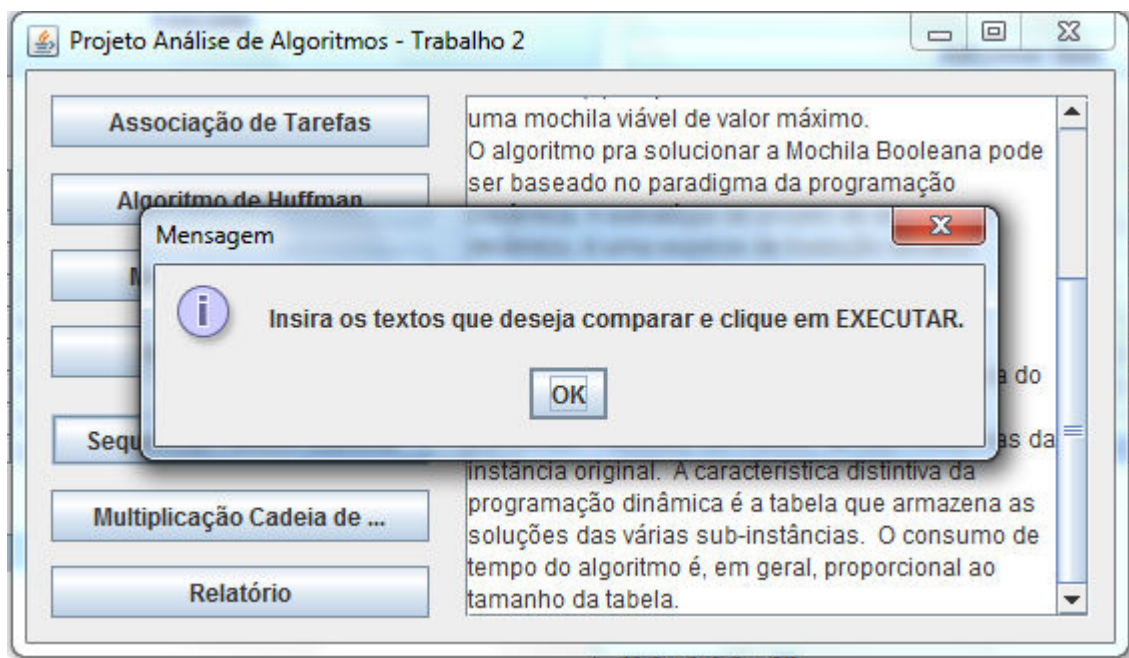


Figura 13. – Caixa de instrução Sub sequência comum máxima

Em seguida, insira as cadeias de caracteres que deseja achar a sub sequência comum máxima e clique em “Executar”.

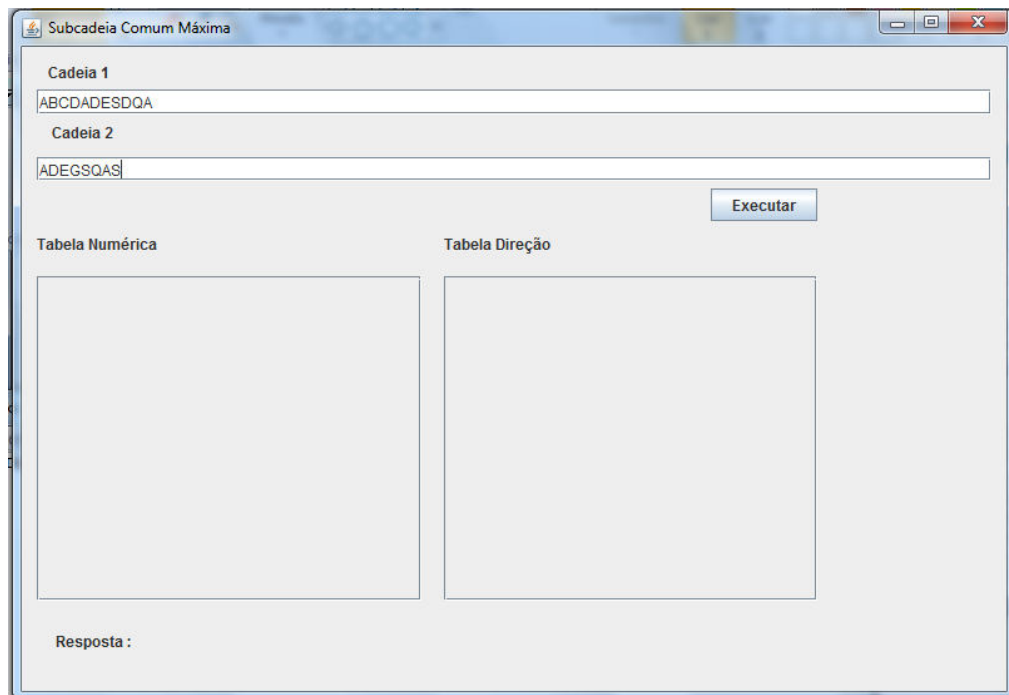


Figura 14. – Tela Sub sequência comum máxima preenchimento de campos

Após clicar em executar, a tabela numérica e tabela direção serão preenchidas, juntamente com a resposta. A tabela numérica é responsável por armazenar em sua última posição o tamanho da substring e a tabela direção é responsável por nos indicar como encontrar a resposta.

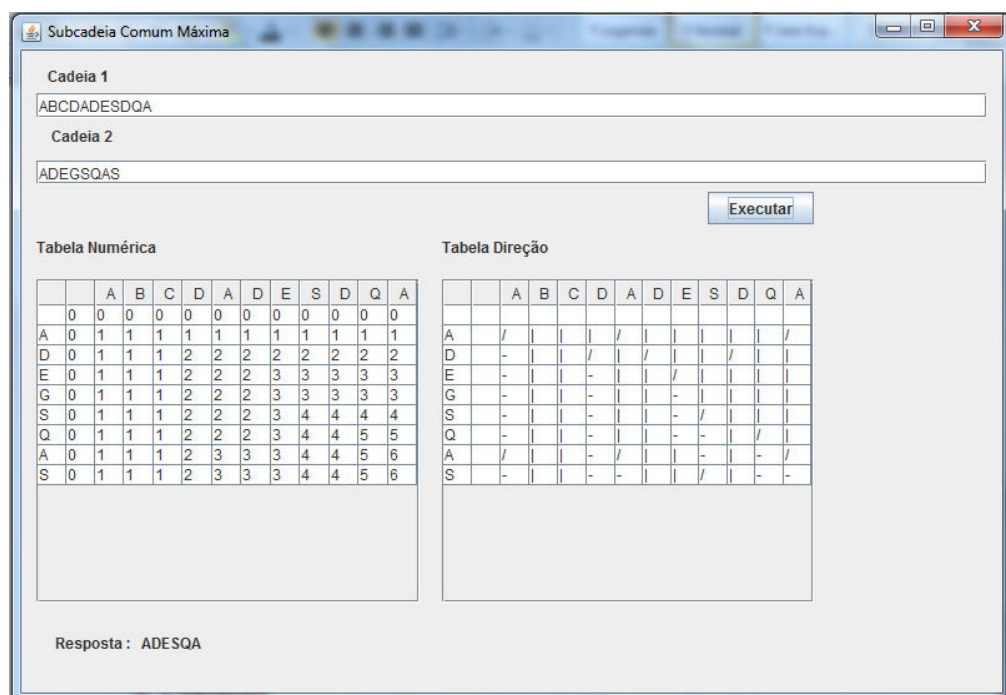


Tabela Numérica

| | | A | B | C | D | A | D | E | S | D | Q | A |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| A | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| D | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| E | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
| G | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 3 |
| S | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 4 | 4 |
| Q | 0 | 1 | 1 | 1 | 2 | 2 | 2 | 3 | 4 | 4 | 5 | 5 |
| A | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 6 |
| S | 0 | 1 | 1 | 1 | 2 | 3 | 3 | 3 | 4 | 4 | 5 | 6 |

Tabela Direção

| | | A | B | C | D | A | D | E | S | D | Q | A |
|---|--|---|---|---|---|---|---|---|---|---|---|---|
| A | | / | | | | / | | | | | | / |
| D | | - | | | / | | / | | | / | | |
| E | | - | | | - | | / | | | | | |
| G | | - | | | - | | | - | | | | |
| S | | - | | | - | | | - | / | | | |
| Q | | - | | | - | | | - | - | / | | |
| A | | / | | | - | / | | | - | | - | / |
| S | | - | | | - | - | | / | | / | - | - |

Resposta : ADESQA

Figura 15. – Tela Sub sequência comum máxima com a solução

7. Multiplicação de Cadeia de Matriz

O problema de Multiplicação de Cadeia de Matriz envolve o uso da Programação Dinâmica como solução. Dada uma sequência de matrizes, o objetivo é encontrar a forma mais eficiente de se multiplicar as matrizes. Como a multiplicação de matrizes é associativa, isso quer dizer que não importa a ordem em que serão multiplicadas o resultado será o mesmo. Isso não vale para a quantidade de operações aritméticas necessárias. Por exemplo, dada uma matriz A (10x30), B(30x5) e C(5x60). Então :

$$(A*B)*C = (10 \times 30 \times 5) + (10 \times 5 \times 60) = 1500 + 3000 = 4500 \text{ operações}$$

$$A*(B*C) = (30 \times 5 \times 60) + (10 \times 30 \times 60) = 9000 + 18000 = 27000 \text{ operações}$$

O primeiro método nos economiza 22.500 operações. Logo, podemos perceber que dada uma diferença grande entre as dimensões das matrizes e uma grande quantidade de matrizes, as diferenças entre as escolhas dos métodos fará grande diferença no tempo de execução e processamento, portanto, escolher o melhor método é extremamente importante.

O algoritmo de Programação Dinâmica, resolve o problema de multiplicação de matrizes na ordem de $O(n^3)$.

No programa desenvolvido, após clicar no botão “Multiplicação de Cadeia de Matriz” a partir da tela inicial, aparecerá uma caixa de instrução:

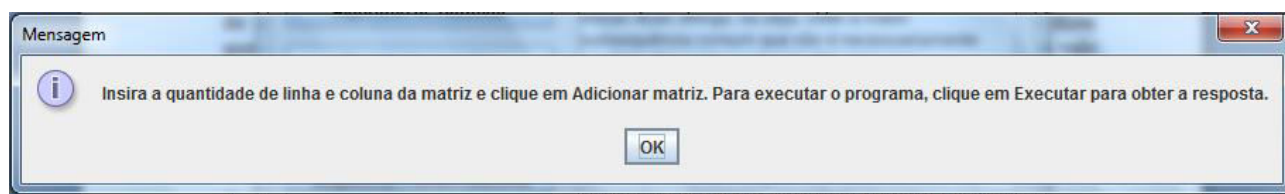


Figura 16. – Caixa de instrução Multiplicação de Cadeia de Matriz

Será aberto uma interface onde é possível inserir matrizes com dimensões m x n. Para inserir as matrizes, basta inserir o tamanho da linha da matriz e o tamanho da coluna e clicar em “Adicionar Matriz”. A matriz será adicionada a lista de matrizes.

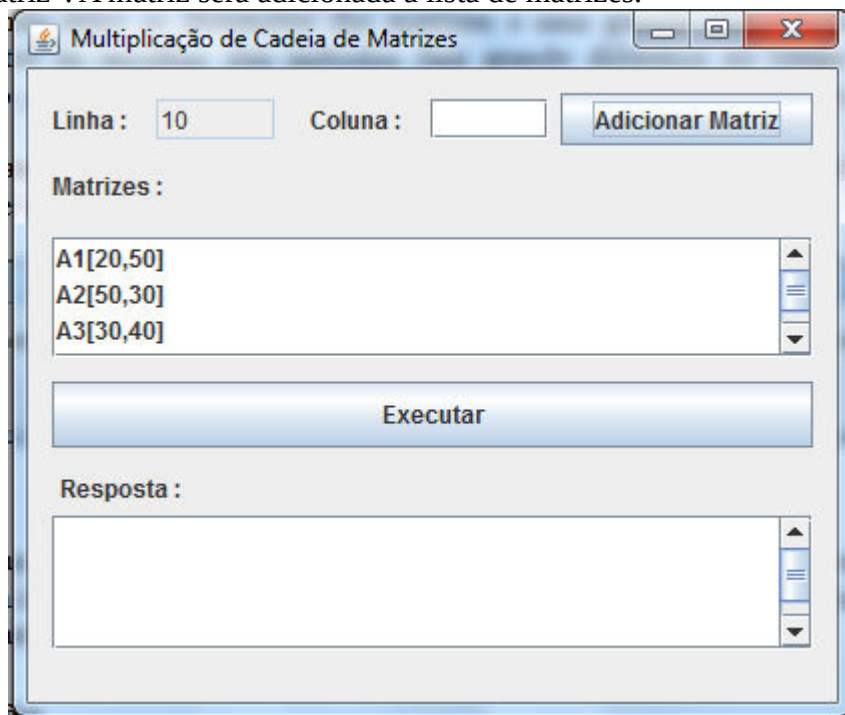


Figura 17. – Tela Multiplicação de Cadeia de Matriz, adicionando matrizes

Após inserir as matrizes com suas respectivas dimensões, basta clicar em “Executar” para que a Resposta seja visível com a melhor combinação, tal que a multiplicação de matrizes necessite da menor quantidade de operações possíveis.

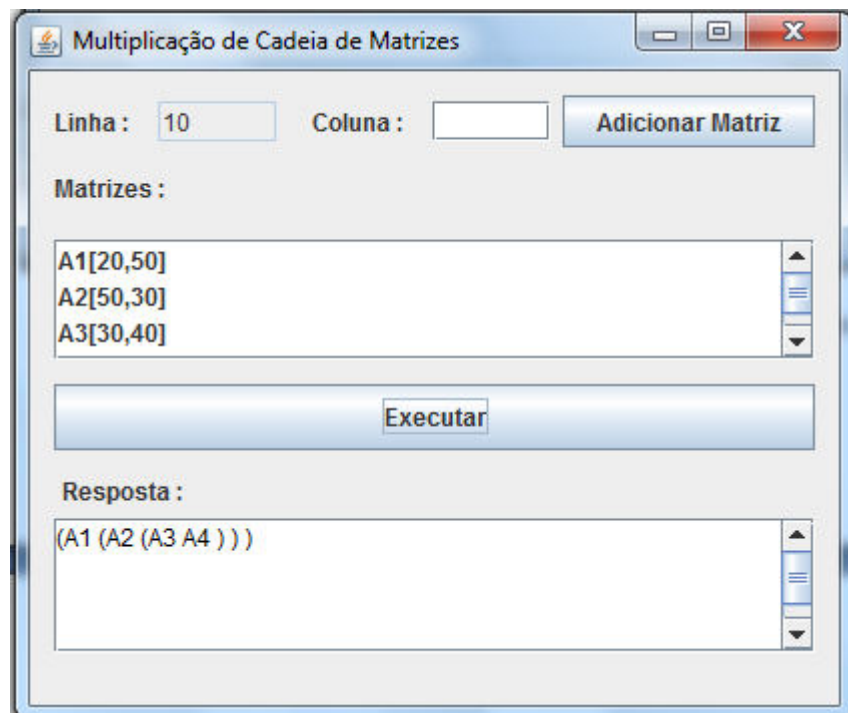


Figura 18. – Tela Multiplicação de Cadeia de Matriz solução do problema

8. Conclusão

Podemos observar que há uma grande importância em buscar as melhores soluções para os programas apresentados. Portanto, a ideia do programa é possuir uma interface de fácil uso e solucionando da melhor forma os problemas apresentados. As técnicas de algoritmos gulosos, branch-and-bound e programação dinâmica nos fornecem maneiras eficientes de solucionar os problemas. Cada problema possui sua tela específica e sua classe, com seus determinados métodos para solucionar o problema.