UNIOESTE – Universidade Estadual do Oeste do Paraná

Ciência da Computação
Projeto e Análise de Algoritmos

Carlos Pedroso
Eduardo Cembranel

RELATÓRIO 1: ANÁLISE DE ALGORITMOS

Foz do Iguaçu

2019

RESUMO

Este relatório traz resultados da análise de complexidade de espaço e tempo para os algoritmos de multiplicação de matrizes (algoritmo convencional e algoritmo de Strassen) e dois algoritmos que resolvem o problema da mochila (Knapsack) da forma binária, um utilizando a técnica de tentativa e erro e outro utilizando programação dinâmica. Os quatro algoritmos foram implementados em C++ e foi definida uma sintaxe para a entrada de dados através de arquivos. Foram executadas 3 instâncias diferentes para cada algoritmo e por fim foi comparado os resultados obtidos.

Palavras-chave: Complexidade, Matrizes, Knapsack.

SUMÁRIO

RELATÓRIO 1: ANÁLISE DE ALGORITMOS

1 INTRODUÇÃO

Para o devido trabalho foi realizado a implementação dos algoritmos convencional de multiplicação de matrizes e , posteriormente, uma adaptação baseado na técnica de projeto por divisão e conquista, o algoritmo de Strassen, foi implementado que processa a multiplicação de matrizes quadradas cuja ordem das matrizes seja da potência de 2 [3]. O programa foi desenvolvido para sistemas Unix, porém o código fonte é parcialmente compatível com o Windows. A sintaxe de arquivos para a entrada das matrizes é da forma:

N = <ordem A>
<a11> <a12> ⋯ <a1N>

⋮ ⋮ ⋮ ⋮

<aN1> <aN2> ⋯ <aNN>

N = <ordem B>
<b11> <b12> ⋯ <b1N>

⋮ ⋮ ⋮ ⋮

<bN1> <bN2> ⋯ <bNN>

O algoritmo de Strassen foi implementado baseado na leitura de [3]. Ele funciona de modo que as matrizes de entrada se dividem sempre em 4 matrizes de ordem n/2, são feitas algumas operações de adição e subtração com essas submatrizes e o próprio algoritmo é chamado recursivamente 7 vezes, de modo a obter 7 matrizes (m1...m7) que são combinadas com o objetivo de obter a matriz C; resultante da multiplicação.

Para o algoritmo de Strassen definimos que uma entrada é pequena o suficiente para ser resolvida quando a ordem das matrizes é menor ou igual a 64. O número 64 é o ponto de crossover. O ponto de crossover define para que ordem as matrizes deixam de ser divididas recursivamente e então podem ser resolvidas pelo método convencional [1]. O número 64 foi escolhido pois parece ser a melhor escolha para o hardware em que desenvolvemos o algoritmo. Foi feito um teste para o ponto de crossover com números da potência de 2, desde o 1 até o 128. Como o algoritmo apresentou tempo de

execução maior para a escolha de 32 e 128 em relação à escolha de 64, consideramos que 64 é a melhor escolha.

Apesar da escolha, definimos um parâmetro para o algoritmo em que o usuário pode escolher o ponto de crossover que desejar e tem o número 64 como padrão. Já o algoritmo de mutiplicação convencional foi implementado de forma iterativa.

Para a implementação por tentativa e erro para o problema da mochila, foi utilizada uma espécie de busca em profundidade que checa todas as possibilidades para o vetor de objetos de entradas e “quebra galhos” na árvore recursiva caso a capacidade da mochila tenha excedido. No final, obtemos o subconjunto binário solução contendo os objetos que devem entrar na mochila e também o valor máximo para a mochila.

Para a implementação do problema da mochila utilizando a técnica de programação dinâmica, foi desenvolvida uma nova solução para o problema. O algoritmo funciona como a implementação tabular (n x W) de valores vista em sala de aula que posteriormente pode ser usada para a obtenção do conjunto solução e o valor máximo. Porém o algoritmo foi modificado de modo que é mantido em memória apenas a linha atual e a linha anterior dessa tabela com o objetivo de economizar espaço, uma vez que podemos fazer todo o processamento apenas analizando a linha anterior da tabela.

O problema dessa otimização de espaço é o fato de impossibilitar, no final do procedimento, a obtenção do subconjunto de objetos que entram na mochila, pois a tabela (n x W) é descartada aos poucos durante a execução. Para solucionar isso, utilizamos uma tabela auxiliar, também (n x W), porém de tal forma que os valores que são armazenados são apenas: 0 ou 1 [4]. Isso possibilitou que a tabela seja do tipo “char”, que tem tamanho de 1 byte, reduzindo o espaço em 4 vezes em relação à implementação que utiliza a tabela completa com inteiros.

Como cada “char” terá apenas 2 valores possíveis: 0 ou 1, fizemos uma otimização de espaço ainda maior em que agora cada “char” irá representar 8 valores 0s ou 1s. Através da manipulação bit a bit de cada char, reduzimos o tamanho da tabela (n x W) para (n x W / 8) e trabalhamos com ela na forma de uma array de 1 dimensão para facilitar o acesso às posições da array e às posições de cada bit. No final reduzimos o espaço em 4 * 8 = 32 vezes. Isso possibilitou a entrada de instâncias de grande magnitude.

A sintaxe de arquivo de entrada definida para o problema da mochila é da forma:

W = <capacidade Mochila>
n = <quantidade Objetos>
w = {<peso 1>, <peso 2>, ⋯ , <peso n>}
c = {<valor 1>, <valor 2>, ⋯ , <valor n>}

Sendo ‘W’ a capacidade máxima da mochila, ‘n’ a quantidade de objetos, ‘w’ o vetor com o peso de cada objeto e ‘c’ o vetor com os valores de cada objeto; e considerando que os símbolos < e > não são representados. Todos os valores são inteiros.

Para o problema da mochila, os algoritmos resolvem o problema e retornam uma vetor na forma binária (0 ou 1), que dirá quais objetos entram na solução.

2 ANÁLISE DE COMPLEXIDADE

2.1 Complexidade de Tempo

2.1.1 Multiplicação de Matrizes Convencional

O algoritmo convencional de multiplicação de matriz é iterativo, cujo tempo é expresso em:

$$T(n)=\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left(1+\sum_{k=0}^{n-1}1\right)$$

O somatório mais interno é resolvido obtendo: n x 1 = n. O segundo somatório é resolvido obtendo n x (n + 1) = n^2 + n. O somatório mais externo é resolvido e temos n x (n^2 + n) = n^3 + n^2. O resultado completo: $T(n)=n^3+n^2$. O que leva a: $T(n)\in O(n^3)$.

2.1.2 Algoritmo de Strassen

O algoritmo de Strassen é recursivo e, enquanto “n” não é pequeno o suficiente para ser resolvido diretamente, a cada chamada recursiva há operações de complexidade O(n^2) responsáveis por alocar as matrizes auxiliares, desalocar e fazer adição e subtração de matrizes. Além disso, cada chamada recursiva chama o mesmo algoritmo 7 vezes de modo a obter as 7 matrizes (m1..m7) que serão usadas posteriormente. Cada chamada recursiva reduz o tamanho da entrada na metade, uma vez que cada matriz de entrada é dividida em 4 pedaços de ordem n/2. Caso a ordem das matrizes de entrada seja menor ou igual ao ponto de crossover (padrão = 64) definido, o algoritmo resolve pelo

método convencional, nesse caso T(n) = 64 * 64 * 64, ou seja, constante. Portanto temos a complexidade de tempo expressa por:

$T(n)=n^2+7T(\frac{n}{2})$, se n > ponto crossover.

$T(n)=1$, se n <= ponto crossover.

Com o método mestre resolvemos a recorrência:

a = 7; b = 2; f(n) = n².

Como $n^2 \in O(n^{\log_2(7-e)})$, com e > 0, temos $T(n) \in O(n^{\log_2 7})$.

2.1.3 Problema da mochila com Tentativa e Erro

O algoritmo de tentativa e erro é dividido em duas funções. A primeira apenas aloca um vetor de tamanho n com a operação new e inicializa tudo com 0. Já temos pelo menos O(n). Em seguida é chamada o procedimento da busca em profundidade. A cada nó da DFS (Depth First Search), caso o valor atual seja maior que o valor ótimo, é copiado o vetor de objetos para o vetor solução através do método auxiliar “copiaArray” que possui O(n).

Em seguida o procedimento é chamado recursivamente duas vezes para incluir e não incluir o objeto atual. A inclusão dos objetos possui O(1). E a remoção de objetos que ocorre no backtracking (retrocesso) também O(1). Como o procedimento deve considerar as duas possibilidades (dentro e fora) para cada objeto a recursão é chamada para n - 1, duas vezes. Logo, a recursão é da forma:

$T(n)=n+2T(n-1),\ se\ \ n>0$.

$T(n)=n,\ se\ \ n=0$.

Temos uma forma de recursão idêntica à uma árvore binária cheia, cuja profundidade é n + 1. Pelo método da árvore de recorrência [5] temos:

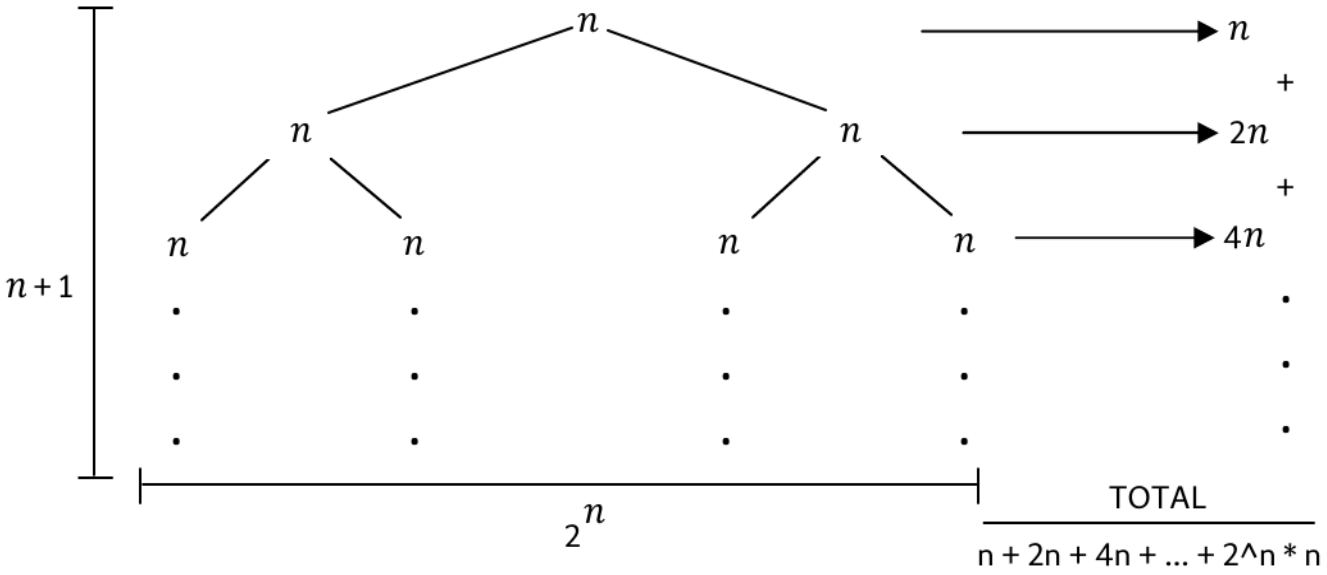

Figura 1: Árvore de Recorrência do Algoritmo de Tentativa e Erro do problema da Mochila

Total: $\sum_{i=0}^{n}(2^i n)$, ou ainda: $n\sum_{i=0}^{n}(2^i)$. Temos uma PG (Progressão Geométrica) de q = 2. Sabemos que a soma finita de termos é: $Sn = a_1 \frac{q^{n_{termos}} - 1}{q-1}$ e que a profundidade da árvore é n + 1. Portanto temos a soma de termos: $Sn = 2(\frac{2^{n+1}-1}{2-1}) = 2 * 2^{n+1} - 2$.

Precisamos ainda multiplicar o resultado por n devido à $n\sum_{i=0}^{n}(2^i)$, chegando em: $T(n) = 2n * 2^{n+1} - 2n$. Finalmente, conclui-se: $T(n) \in O(n2^n)$.

2.1.4 Problema da Mochila com Programação Dinâmica

A função “resolveComDP” aloca um vetor de tamanho 2 x (W + 1) e inicializa metade dos elementos (primeira linha) com zero. Temos O(W) nesse processo, sendo W a capacidade da mochila. Em seguida é alocado um vetor de tamanho (n + 1) x (W + 1). Temos O(nW) utilizado pelo operador

“new”, responsável alocar espaço na memória. Em seguida temos dois loops indo de 1 até n e 1 até W, respectivamete. Portanto, temos a forma: $T(n)=nW+W+\sum_{i=1}^{n} 1 \sum_{j=1}^{W} 1$.

Resolvendo o somatório interno: W x 1 = W. Resolvendo o somatório externo: n (1 + W) = n + nW. Chegando em: $T(n)=nW+W+n+nW=2nW+W+n$. O tempo para desalocação do array 2 x (W + 1) foi representado em W também. A atribuição e o return que ocorrem após os loops não estão representadas na função pois são constantes. Ainda temos a função “obtemSubconjunto” que é chamada logo em seguida. Sua complexidade é O(n), pois consiste em uma laço de repetição de n até 1 com operações constantes. Logo temos o resultado: $T(n)=2nW+W+n$, com $T(n) \in O(nW)$.

A seguir observamos a complexidade de tempo dos 4 algoritmos implementados:

Algoritmo	Complexidade de Tempo
Multiplicação de Matrizes (Convencional)	$O(n^3)$
Algoritmo de Strassen	$O(n^{\log_2 7})$
Problema da Mochila (Tentativa e Erro)	$O(n2^n)$
Problema da Mochila (Programação Dinâmica)	$O(nW)$

Tabela 1: Complexidade de Tempo dos Algoritmos Implementados

2.2 Complexidade de Espaço

Para a análise de espaço foi considerado o espaço necessário para rodar o algoritmo.

2.2.1 Multiplicação de Matrizes Convencional

O espaço que o algoritmo convencional de multiplicação de matrizes usa é constante. O algoritmo apenas usa uma variável de 4 bytes para armazenar o somatório das multiplicações antes de inserir na matriz resultado. O algoritmo recebe 3 matrizes como entrada, as matrizes A, B e a resultado C, ambas passadas como referência, portanto não há criação de cópias. Por conta disso, **O(1)** para espaço auxiliar.

2.2.2 Algoritmo de Strassen

Como o algoritmo de Strassen é recursivo, precisamos ficar atentos a algumas situações. Este recebe como argumento 3 matrizes, de ordem n, passadas por **referência**. A terceira matriz é a matriz que armazenará o resultado da multiplicação.

Uma vez que a entrada não é pequena o suficiente para ser resolvida diretamente, ocorre a alocação de 21 matrizes auxiliares: 4 para as sub-matrizes A, 4 para as sub-matrizes B, 4 para as sub-matrizes C e 7 para as matrizes m (m1..m7). Todas as 21 matrizes são alocadas de forma dinâmico-heap e cada matriz tem ordem n/2. No final (depois das 7 chamadas recursivas), as 21 matrizes são desalocadas.

Por ser um algoritmo recursivo precisamos conhecer a profundidade da pilha [2]. Sabemos que a cada chamada o tamanho reduz pela metade. Ou seja, teremos uma árvore recursiva com cada nó derivando em 7 filhos e a profundidade máxima dessa árvore será **lg(n)**.

Consideramos o espaço máximo usado em algum momento do processamento. Assim, não consideramos todos os nós da árvore, mas todos os nós de apenas um galho, da raiz até a folha. Como a cada chamada são alocadas $21(\frac{n}{2})^2$ unidades de espaço e sabendo que a cada chamada recursiva o n é reduzido na metade temos: $T(n)=21\frac{n^2}{4}+21\frac{n^2}{16}+21\frac{n^2}{64}+\cdots+21\frac{n^2}{4^i}$. Podemos ainda simplificar: $T(n)=21n^2\sum_{i=1}^{lg(n)}(\frac{1}{4^i})$.

Esse somatório configura uma PG de q = a1 = ¼. Como 0 < q, a1 < 1, logicamente, o resultado da soma de termos dessa PG estará sempre no intervalo [0, 1], para qualquer valor de “n”. Sabendo disso, podemos dizer que $T(n)\in O(n^2)$. Nota-se que independende da escolha para o ponto de crossover, a complexidade de espaço para o algoritmo de Strassen será a mesma, pois o método convencional usa O(1) para espaço auxiliar.

2.2.3 Problema da Mochila com Tentativa e Erro

Para o devido problema, o algoritmo implementado primeiramente aloca um vetor de tamanho “n” que estará sempre sendo modificado de maneira que tenha o estado binário da mochila. Esse vetor é alocado no inicio e usado durante toda a execução do algoritmo. Após a alocação desse vetor, é chamada a busca em profundidade que checa todas as possibilidades. A DFS não cria variáveis auxiliares, porém devemos considerar o tamanho da pilha com todas as chamadas recursivas. A quantidade de chamadas recursivas ativas na pilha, destacadas em negrito, para uma instância de 3 objetos é mostrada no exemplo a seguir em que a função primeiramente busca para à esquerda:

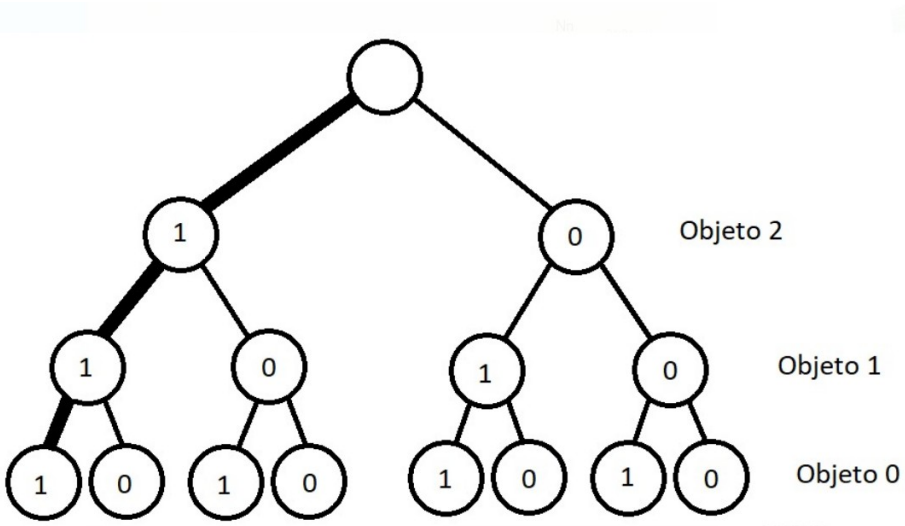

Figura 2: Representação das recursões ativas na busca em profundidade

Assim como no algoritmo de Strassen, consideramos apenas a quantidade de chamadas recursivas que estarão ativas na pilha ao mesmo tempo em algum ponto do algoritmo. Nesse caso, temos que a altura máxima da árvore recursiva será sempre n + 1, como visto anteriormente na análise de tempo. A cada chamada, a pilha cresce de forma constante, pois o que é passado por parâmetro são apenas 3 inteiros e o vetor de objetos passado por **referência.** Portanto temos: $T(n)=n+(n+1)*1$, logo: $T(n)\in O(n)$.

2.2.4 Problema da Mochila com Programação Dinâmica

O algoritmo que usa programação dinâmica usa duas estruturas auxiliares. A primeira ocupa 2 * (W + 1), ou seja O(W), sendo W a capacidade da mochila. A segunda usa ((W + 1) * (n + 1) / 8), ou

seja $O(\frac{Wn}{8})$. Temos o resultado total: $T(n)=W+\frac{Wn}{8}$. Apesar da otimização de espaço, o algoritmo ainda é: $T(n) \in O(Wn)$.

A seguir observamos a análise de espaço para os 4 algoritmos:

Algoritmo	Complexidade de Espaço
Multiplicação de Matrizes (Convencional)	$O(1)$
Algoritmo de Strassen	$O(n^2)$
Problema da Mochila (Tentativa e Erro)	$O(n)$
Problema da Mochila (Programação Dinâmica)	$O(nW)$

Tabela 2: Complexidade de Espaço dos Algoritmos Implementados

3 RESULTADOS OBTIDOS E DISCUSSÃO

Para os testes, foi implementado um gerador de arquivos para a entrada de matrizes e outro gerador de arquivos para o problema da mochila. Assim, foi possível testar instâncias com tamanhos até a quarta ordem de magnitude. Os resultados obtidos são os seguintes:

Algoritmo	Tempo de execução (min:s:ms)		
Multiplicação de Matrizes Convencional	**n = 500**	**n = 2700**	**n = 3500**
	00:01:146	06:05:157	10:54:540
Algoritmo de Strassen	**n = 512**	**n = 4096**	**n = 8192**
	00:01:243	06:19:593	44:10:850
Knapsack (Tentativa e Erro)	**n = 30 W = 3000**	**n = 35 W = 3500**	**n = 38 W = 3800**
	00:09:859	04:36:600	39:24:666
Knapsack (Prog. Dinâmica)	**n = 10000 W = 1000000**	**n = 13500 W = 1350000**	**n = 20000 W = 2000000**
	00:01:036	03:02:354	06:45:678

Tabela 3: Tempos de Execução dos Algoritmos Implementados

3.1 Multiplicação de Matrizes

Para os algoritmos de multiplicação de matrizes, comparamos o resultado obtido a seguir:

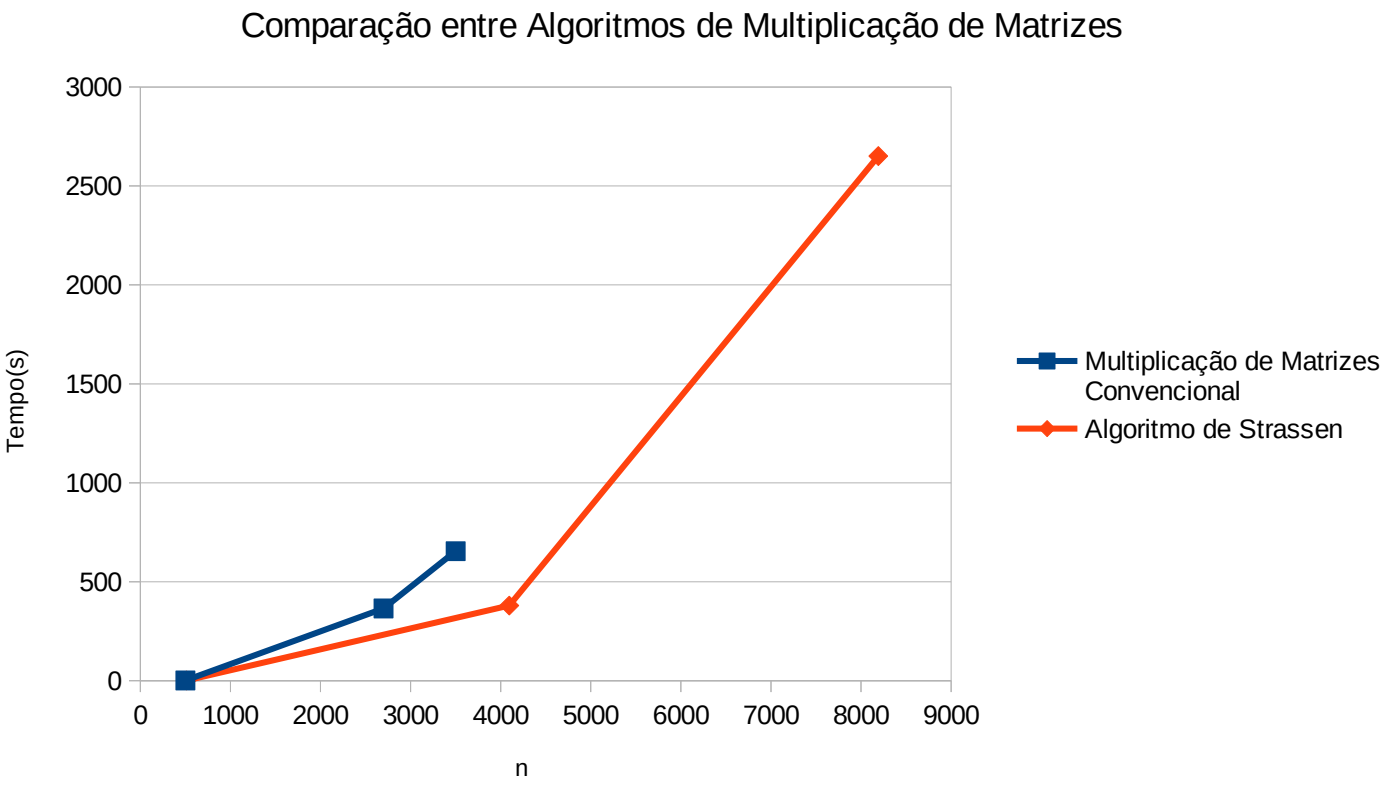

Figura 3: Comparação entre Algoritmos de Multiplicação de Matrizes para as instâncias testadas

A partir do gráfico, observa-se claramente a característica polinomial dos dois algoritmos. O algoritmo convencional cresce mais rapidamente devido à $O(n^3)$, comparado ao algoritmo de Strassen O(n^lg7). Comparando o resultado das intâncias n = 500 para o algoritmo convencional com a instância n = 512 para o algoritmo de Strassen, é notado uma semelhança muito forte no tempo de execução dos dois algoritmos. Entretanto, conforme o tamanho das matrizes cresce, a diferença entre o tempo de execução torna-se mais evidente.

Para n = 4096 com o algoritmo de Strassen, comparado com n = 3500 com o algoritmo convencional, o algoritmo de Strassen mostrou ser 73% mais rápido mesmo com uma instância maior que a do algoritmo convencional. Considerando ainda a complexidade assintótica dos dois algoritmos, podemos deduzir quantas vezes mais rápido o algoritmo de Strassen mostra ser quando n = 8192:

$\frac{8192^3}{8192^{\log_2 7}} = 5.67$, ou seja, em teoria, o algoritmo de Strassen é 5.67 vezes mais rápido que o algoritmo convencional quando n = 8192.

Apesar da diferença de tempo, devemos lembrar que o algoritmo de Strassen usa mais memória auxiliar, considerando ainda n = 8192, por exemplo, esse algoritmo usa:

$$\frac{21 * 8192^2 \left(\frac{1}{4} + \frac{1}{16} + \cdots + \frac{1}{16384}\right) * 4}{1024 * 1024 * 1024} \approx 1.75\,GB$$ de espaço auxiliar.

Em resumo, o algoritmo de Strassen mostra ser uma boa escolha quando as matrizes são de ordem muito grande, porém tem a desvantagem de usar grande bastante espaço auxiliar. Outra desvantagem do algoritmo de Strassen é o fato de apenas aceitar matrizes cuja ordem seja potência de 2, embora existem maneiras de contornar esse problema preechendo a matriz com zeros até ficar do tamanho necessário.

3.2 Problema da Mochila

A grande vantagem do algoritmo de tentativa e erro para o problema da mochila é o fato de não depender diretamente da capacidade da mochila e o fato de poder ser implementado considerando que a capacidade da mochila e pesos sejam números reais. Esse algoritmo pode ser usado caso existam poucos objetos, de modo que o número de operações feitas não seja muito grande no pior dos casos. Todavia, é um algoritmo cuja complexidade é exponencial e é totalmente inviável para instâncias maiores.

Como observado na tabela 3, para um n = 38 e W = 3800, o algoritmo de tentativa e erro levou 39 minutos em tempo de execução, enquanto o algoritmo que usa programação dinâmica levou menos de 7 minutos para um n = 20000 e W = 2000000. Porém devemos observar que a solução implementada com programação dinâmica usa uma quantidade grande de espaço auxiliar e pode variar bastante nesse quesito, pois depende de W e não apenas de n. Na instância citada acima, foi usado:

$$\frac{2(W+1) * 4\,bytes + \frac{(n+1)(W+1)}{8} * 1\,byte}{1024 * 1024 * 1024} \approx 4.67\,GB$$ de espaço auxiliar.

Portanto, o algoritmo com programação dinâmica, apesar de apresentar o formato pseudo-polinomial, mostra executar instâncias grandes em um tempo muito menor comparado com o algoritmo de tentativa e erro que é exponencial. Entretanto, é mais custoso em espaço necessário para a execução e depende do valor de W. Por exemplo, se existir uma instância em que n = 15 e W = 100000, o algoritmo que usa tentativa e erro é mais eficiente tanto em espaço, quanto em tempo, porém para uma instância em que n = 1000 e W = 100000, o algoritmo que usa programação dinâmica é, com certeza, melhor, mesmo que utilize mais espaço.

4 CONCLUSÃO

Existem diferentes algoritmos que podem ser implementados para resolver um mesmo problema computacional. Por conta disso, cabe a nós identificar a melhor técnica que podemos utilizar para a sua implementação considerando as demandas de projeto.

Nota-se que muitas vezes não é possível obter um algoritmo que será o melhor para todos os casos. Dependendo da situação é necessário escolher aquela implementação que irá fornecer um melhor resultado considerando o tempo computacional ou, talvez, aquela que usa menos espaço.

Com os quatro algoritmos implementados, observamos como uma função exponencial pode se tornar inviável para problemas muito grandes e como detalhes na implementação influenciam no espaço utilizado por um algoritmo. Por fim, devemos sempre tentar solucionar os problemas levando em consideração as complexidades de espaço e tempo de maneira a obter soluções eficientes.

5 BIBLIOGRAFIA

[1] Cormen T. H; Leiserson C. E; Leiserson, Ronald L. Rivest; Clifford Stein. Introduction to Algorithms, Terceira Edição. Capítulo 4, p. 112

[2] Cormen T. H; Leiserson C. E; Leiserson, Ronald L. Rivest; Clifford Stein. Introduction to Algorithms, Terceira Edição. Capítulo 7, p. 188

[3] Stanford, 2016. Distributed Algorithms and Optimization. Disponível em: https://stanford.edu/~rezab/classes/cme323/S16/notes/Lecture03/cme323_lec3.pdf

[4] Eletronic Systems Group, 2009. Lecture 13: The Knapsack Problem. Disponível em: http://www.es.ele.tue.nl/education/5MC10/Solutions/knapsack.pdf

[5] Silva, Rômulo César, 2010. Resolução de Recorrências, p. 5