Trabalho de Projeto de Análise de Algoritmo

Gabriel H. Montoanelli; Kame Haung Zhu
Curso de Graduação de Ciência da Computação - Universidade Estadual do Oeste do Paraná (UNIOESTE)
Foz do Iguaçu - PR - Brasil

1 Multiplicação de Matrizes

Para a representação de dados as matrizes possuem uma grande importância, sendo uma das mais utilizadas para a representação de dados, tanto na matemática como na computação. Assim, dentre as operações fundamentais mais utilizadas, destaca-se o produto de matrizes que possuem diferentes tipos de abordagens para o seu cálculo (LUSA, 2014).

1.1 Materiais

Matrizes de ordens: 500, 2800, 3000

1.2 Matriz O(n3)

A abordagem mais tradicional, comumente possui uma complexidade de O(n3):

```
//Multiplicação de matrizes complexidade normal
//Entrada: matriz a, matriz b, matriz c, ordem das matrizes
//Retorno: nenhum
//Pré-Condições: matrizes com números setados, menos a matriz c
//Pós-Condições: matrizes multiplicadas
void mult_matrizes(int **ma, int **mb, int **mc, int n){
	int i, j, k;
	for (i = 0; i < n; i++){
		for (j = 0; j < n; j++){
			mc[i][j] = 0;
			for (k = 0; k < n; k++){
				mc[i][j] += ma[i][k]*mb[k][j];
			}
		}
	}
}
```

Onde, ao calcular sua complexidade de tempo temos:

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

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

$$T(n) = \sum_{i=0}^{n-1} \frac{n(n-1)}{2}$$

$$T(n) = \frac{n(n-1)(n-2)}{6}$$

$$T(n) = n^3 - 3n^2 + 2 \quad \therefore \; T(n) \in \Theta(n^3)$$

1.2.1 Testes

Instância 1 (500x500): 0,515 seg ~ 0,008 min
Instância 2 (2800x2800): 235,845 ~ 3,930 min
Instância 3 (3000x3000): 448,673 seg ~ 7,477 min

1.3 Algoritmo de Strassen

Diferente do produto de matrizes tradicional, o método de Strassen possui uma complexidade de tempo de O($n^{2,807}$):

```
//Multiplicação de matrizes com o algoritmo de Strassen
//Entrada: matriz a, matriz b, matriz c, ordem das matrizes
//Retorno: nenhum
//Pré-Condições: matrizes com números setados, menos a matriz c
//Pós-Condições: matrizes multiplicadas
void alg_Strassen(int **ma, int **mb, int **mc, int n){
	//caso base, quando a matriz for de ordem 1
	if (n == 1){
		mc[0][0] = ma[0][0]*mb[0][0];
	}
	else{
		int size = n/2;

		//aloca memória para todas as subdivisões
		int **a11 = alocaMatriz(size);
		int **a12 = alocaMatriz(size);
		int **a21 = alocaMatriz(size);
		int **a22 = alocaMatriz(size);
		int **b11 = alocaMatriz(size);
```

```
                int **b12 = alocaMatriz(size);
                int **b21 = alocaMatriz(size);
                int **b22 = alocaMatriz(size);

                //aloca memória para matrizes utilizadas nas cadeias de 7
multiplicações de matrizes a serem realizadas
                int **m1 = alocaMatriz(size);
                int **m2 = alocaMatriz(size);
                int **m3 = alocaMatriz(size);
                int **m4 = alocaMatriz(size);
                int **m5 = alocaMatriz(size);
                int **m6 = alocaMatriz(size);
                int **m7 = alocaMatriz(size);

                //divide a matriz a em partes iguais
                dividir(ma, a11, 0, 0, size);
                dividir(ma, a12, 0, size, size);
                dividir(ma, a21, size, 0, size);
                dividir(ma, a22, size, size, size);

                //divide a matriz b em partes iguais
                dividir(mb, b11, 0, 0, size);
                dividir(mb, b12, 0, size, size);
                dividir(mb, b21, size, 0, size);
                dividir(mb, b22, size, size, size);

                //m1 = (a11+a22)(b11+b22)
                int **temp1 = adicao(a11, a22, size);
                int **temp2 = adicao(b11, b22, size);
                alg_Strassen(temp1, temp2, m1, size);
                desalocaMatriz(temp1, size);
                desalocaMatriz(temp2, size);

                //m2 = (a21+a22)b11
                int **temp3 = adicao(a21, a22, size);
                alg_Strassen(temp3, b11, m2, size);
                desalocaMatriz(temp3, size);

                //m3 = a11(b12-b22)
                int **temp4 = subtr(b12,b22, size);
                alg_Strassen(a11, temp4, m3, size);
                desalocaMatriz(temp4, size);

                //m4 = a22(b21-b11)
                int **temp5 = subtr(b21, b11, size);
                alg_Strassen(a22, temp5, m4, size);
                desalocaMatriz(temp5, size);
```

```
//m5 = (a11+a12)b22
int **temp6 = adicao(a11, a12, size);
alg_Strassen(temp6, b22, m5, size);
desalocaMatriz(temp6, size);

//m6 = (a21-a11)(b11-b12)
int **temp7 = subtr(a21, a11, size);
int **temp8 = adicao(b11, b12, size);
alg_Strassen(temp7, temp8, m6, size);
desalocaMatriz(temp7, size);
desalocaMatriz(temp8, size);

//m7 = (a12-a22)(a21-a22)
int **temp9 = subtr(a12, a22, size);
int **temp10 = adicao(b21, b22, size);
alg_Strassen(temp9, temp10, m7, size);
desalocaMatriz(temp9, size);
desalocaMatriz(temp10, size);

//Primeira parte da matriz produto de duas matrizes
int **temp11 = adicao(m1, m4, size);
int **temp12 = subtr(temp11, m5, size);
int **c11 = adicao(temp12, m7, size);
desalocaMatriz(temp11, size);
desalocaMatriz(temp12, size);

//Segundo parte
int **c12 = adicao(m3, m5, size);

//Terceiro parte
int **c21 = adicao(m2, m4, size);

//Ultima parte
int **temp13 = adicao(m1, m3, size);
int **temp14 = subtr(temp13, m2, size);
int **c22 = adicao(temp14, m6, size);
desalocaMatriz(temp13, size);
desalocaMatriz(temp14, size);

//Junção de todas as partes em uma matriz
juntar(c11, mc, 0, 0, size);
juntar(c12, mc, 0, size, size);
juntar(c21, mc, size, 0, size);
juntar(c22, mc, size, size, size);

desalocaMatriz(m1, size);
desalocaMatriz(m2, size);
desalocaMatriz(m3, size);
```

```
                desalocaMatriz(m4, size);
                desalocaMatriz(m5, size);
                desalocaMatriz(m6, size);
                desalocaMatriz(m7, size);

                desalocaMatriz(c11, size);
                desalocaMatriz(c12, size);
                desalocaMatriz(c21, size);
                desalocaMatriz(c22, size);

                desalocaMatriz(a11, size);
                desalocaMatriz(a12, size);
                desalocaMatriz(a21, size);
                desalocaMatriz(a22, size);

                desalocaMatriz(b11, size);
                desalocaMatriz(b12, size);
                desalocaMatriz(b21, size);
                desalocaMatriz(b22, size);
        }
}
```

Onde, ao calcular a complexidade temos:

$$T(n) = \begin{cases} 1 & se\ n = 1 \\ 7T(n/2) + n^2 & se\ n > 1 \end{cases}$$

$$T(n) = 7T(n/2) + 30O(n^2)$$

Por método mestre:

$$F(n) = a\,F(n/b) + cnk$$

a = 7

b = 2

$f(n) = O\left(n^{\log_b a}\right)\ \ se\ \ \log_b a > k$

$f(n) = O\left(n^{\log_2 7}\right)$

$\log_2 7 \cong 2.807$

F(n) = n^2 e k = 2

$T(n) \in O(n^{2.807})$

1.3.1 Testes

Instância 1 (500x500): 12,014 seg ~ 0,20 min
Instância 2 (2800x2800): 2348,127 ~ 39,135 min

Instância 3 (3000x3000): 2468,434 seg ~ 41,140 min

1.4 Comparações

Teoricamente, o algoritmo de Strassen possui uma complexidade de tempo menos que o algoritmo convencional. Mas, como vemos nos testes, pelo menos em matrizes com ordem até 3000, o método convencional ainda possui um tempo de execução menor.

Contudo, isso pode ser devido a constante necessidade de alocação de memória para matrizes, assim como sua liberação ou ao fato da grande quantidade de adições (12) e subtrações (6) somados as divisões (8) e junções na matriz resultado (4) a cada chamada recursiva.

2 Problema da Mochila

2.1 Materiais

Matriz de ordem Nx2 com valores aleatórios entre 0 e 1000 como arquivo de entrada, onde cada coluna representa o valor e o peso, respectivamente.

Matriz de ordem 100x2 como arquivo de teste com 100 objetos.

Matriz de ordem 500x2 como arquivo de teste com 500 objetos.

Matriz de ordem 1000x2 como arquivo de teste com 1000 objetos.

Matriz de ordem 10000x2 como arquivo de teste com 10000 objetos.

Matriz de ordem 50000x2 como arquivo de teste com 50000 objetos.

2.2 Algoritmo por Programação Dinâmica

O algoritmo por Programação Dinâmica utiliza uma matriz que é utilizada para guardar os valores calculados para ser utilizados nas operações seguintes, evitando assim repetição de operações (BARBOSA, 1971).

O algoritmo com programação dinâmica implementado em C pode ser visto a seguir:

```
//Faz as operações do problema da mochila por programação dinâmica
//Entrada: capacidade da mochila e o vetor de solução
//Retorno: valor máximo
//Pré-Condições: nenhuma
//Pós-Condições: conjunto solução de objetos escolhidos
int knapSackPD (int w, int x[]){
   int i, j;
   int a, b;
  int **K = (int**)malloc((tamanho+1)*sizeof(int*));

   for(i = 0; i < tamanho+1; i++){
```

```
    K[i] = (int*)malloc((w+1)*sizeof(int));
  }


  //inicia a primeira linha com 0
  for(a = 0; a <= w; a++) {
    K[0][a] = 0;
  }

  //inicia a primeira coluna com 0
  for(b = 0; b <= tamanho; b++) {
    K[b][0] = 0;
  }

  //opera e compara sobre a matriz calculando as possibilidades de solução
  for (i = 1; i <= tamanho; i++){
    for(j = 1; j <= w; j++){
      // compara o peso com a capacidade da mochila e o valor com o valor da
posição anterior
      if((peso[i-1] <= j) && (valor[i-1]+K[i-1][j-peso[i-1]] > K[i-1][j])){
        K[i][j] = valor[i-1] + K[i-1][j-peso[i-1]];
      }else{
        K[i][j] = K[i-1][j];
      }
    }
  }
  //inicializa o conjunto solução com 0
  for(i = 0; i < tamanho; i++) x[i] = 0;

  i = tamanho;
  j = w;
  //enquanto não for posição vazia na matriz
  while(K[i][j] != 0){
    //se a posição for igual a posição anterior continua
    if(K[i][j] == K[i-1][j]){
      i--;
    }else{
      //senão a posição anterior fará parte do conjunto solução
      x[i-1] = 1;
      j -= peso[i-1];
      i--;
    }
  }

  //retorna a última posição da matriz que é o maior valor
  return K[tamanho][w];
}
```

O cálculo da complexidade de tempo pode ser resolvido através da recorrência:

$$T(n) = \sum_{a=0}^{W}(1) + \sum_{b=0}^{W}(1) + \sum_{i=1}^{tamanho}\sum_{j=1}^{W}(1) + \sum_{i=0}^{tamanho}(1) + \sum_{i=tamanho, j=W}^{K}(1)$$

$$T(n) = [\sum_{a=0}^{W}(1)] + [\sum_{b=0}^{W}(1)] + (W\,tamanho) + [\sum_{i=0}^{tamanho}(1)] + [\sum_{i=tamanho, j=W}^{K}(1)]$$

$$T(n) = (1 + W) + (1 + W) + (W\,tamanho) + (1 + tamanho) + (K)$$

$$T(n) = 3 + 2W + (W\,tamanho) + (tamanho) + (K)$$

$$T(n) \in O(W\,tamanho)$$

A complexidade de memória pode ser expressada por: $T(n) \in O(4n)$.

2.2.1 Testes

Instância 1 (1000x1000): 0.001000 segundos
Instância 2 (10000x5000): 0.386000 segundos
Instância 3 (50000x1000): 0.487000 segundos

2.3 Algoritmo por Backtracking

O algoritmo por Tentativa e Erro, ou Backtracking, procura a solução calculando todas as possibilidades possíveis, sem reaproveitar os cálculos anteriores.

O algoritmo com programação dinâmica implementado em C pode ser visto a seguir:

```
//Faz busca por profundidade de todas as possibilidades possíveis
//Entrada: peso e o valor atual, os dois conjuntos de soluções,
//a quantidade de objetos e a capacidade da mochila
//Retorno: nenhum
//Pré-Condições: nenhuma
//Pós-Condições: conjunto solução de objetos escolhidos
void dfs(int pesoAtual, int valorAtual, int x[], int solução[], int tam, int W){

   int i;

   //se o valor atual for maior que o valor máximo
   //ele se tornara o novo maior valor
   //e o conjunto solução receberá novos valores
   if(valorAtual > valorMaximo){
      valorMaximo = valorAtual;
```

```
      for( i = 0; i < tamanho; i++){
         solução[i] = x[i];
      }
   }

   //se ainda possuir comparações possíveis
   if(tam > 0){
      // se o peso atual mais o peso da posição atual for menor que a capacidade
      if(pesoAtual + peso[tam] <= W){
         //essa posicao fara parte do conjunto solucao
         x[tam] = 1;
         //analisa as outras possibilidades recursivamente
         dfs((pesoAtual+=peso[tam]), (valorAtual+=valor[tam]), x, solução, tam - 1,
W);
         //desfaz as operações para possibilitar a comparação as outras
possibilidades
         pesoAtual-=peso[tam], valorAtual-=valor[tam];
         x[tam] = 0;
      }
      //avança na comparação sem o objeto
      dfs(pesoAtual,valorAtual, x, solução, tam-1, W);
   }
}
```

O cálculo da complexidade de tempo pode ser resolvido através da recorrência:

$$T(n) = \sum_{i=0}^{tamanho} + \sum_{i=0}^{tamanho} (1) + T(n-1)$$

$$T(n) = 2(1 + tamanho) + T(n-1)$$

$$T(n) = 2(1 + tamanho) + (2(1 + tamanho) + T(n-2) - 1)$$

$$T(n) = 2(1 + tamanho) + (2(1 + tamanho) + 2(1 + tamanho) + T(n-3) - 2) - 1)$$

Arrumando essa expressão, chegamos que:

$$T(n) \in O(2^n)$$

A complexidade de memória pode ser expressada por: $T(n) \in O(4n)$.

2.3.1 Testes

Instância 1 (100x1700): 255.54600 segundos ~ 4m15s
Instância 2 (500x300): 787.343000 segundos ~ 13m7s
Instância 3 (1000x165):1040.024000 ~ 17m

2.4 Comparações

Teoricamente, o algoritmo por programação dinâmica possui uma complexidade de tempo menor que o algoritmo por backtracking. Pode-se perceber isso através dos testes, pois valores muito grandes são resolvidos em segundos, enquanto, por backtracking demora bem mais tempo.

Contudo, percebe-se que os testes usando programação dinâmica não possui uma diferença maior que 3 minutos, isso se dá pelo fato de sua rapidez, pois precisaria de um valor absurdamente elevado, e isso acaba levando a um estouro da pilha.

Os testes foram feitos usando um $W > n$, $n > W$ e $W \simeq n$.

3 Referências

CLRS 4.2 e KT 5.5. Disponível em: <https://www.ime.usp.br/~cris/aulas/13_1_338/slides/aula5.pdf>. Acesso em 22 de julho de 2019.

BARBOSA, F. R. S. Trabalho de Análise de Algoritmos Problema de Clique. **Leonardo**, 1971.

LUSA, D. A. Multiplicação de Matrizes Do O(n3) ao O(n2). **Revista Brasileira de Computação Aplicada**, Passo Fundo, v.5, n.2, 2014.