TRABALHO PRÁTICO 3:
Counting Boolean Parenthesization

Bruno Maciel Peres

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

brunomperes@dcc.ufmg.br

Resumo. *Este relatório descreve uma solução proposta para o problema de contagem das formas de parentização de uma expressão booleana. Os principais objetivos deste trabalho são: exercitar o paradigma de programação Programação Dinâmica, com o intuito de analisar o seu desempenho em termos de tempo de execução e a quantidade de memória alocada.*

1. INTRODUÇÃO

Neste trabalho, descrevemos e implementamos um algoritmo para solucionar o problema de Contagem de Parentização Booleana, ou *Counting Boolean Parenthesization*. Esse problema é um problema conhecido e foco de estudo em Ciência da Computação, mais especificamente em Paradigmas de Programação.

O problema em questão consiste em contar a quantidade de formas de se posicionar parênteses em uma expressão booleana, contendo variáveis, negações (*not*) e operadores, de tal forma que essa parentização retorne verdadeiro.

Para isso são dadas várias instâncias do problema em um arquivo de entrada e o programa deve realizar a contagem de maneira recursiva e armazenar os valores já calculados para otimizar o cálculo de sub-expressões que contenham combinações já calculadas. Essa abordagem troca espaço por tempo de execução.

A abordagem proposta para solucionar o problema foi utilizar um método de contagem que divide a equação em sub-expressões e os operadores da expressão são as linhas de divisão, e então executa a recursão para a sub-expressão à esquerda e à direita do operador, até que um caso já calculado ou um caso base sejam alcançados.

Foi realizado uma análise experimental do programa, executando-o em algumas entradas geradas. As métricas para avaliar a execução do programa para cada expressão de quantidade de elementos t são: o tempo de execução e a quantidade de memória alocada que dependa do tamanho da entrada, sendo o restante considerado como *overhead*.

Neste problema, o tamanho da expressão é referenciado através do tamanho t, a quantidade de operadores por op e a quantidade de variáveis será referenciada por var nas medidas de análise da complexidade de espaço e de execução dos algoritmos.

2. SOLUÇÃO PROPOSTA

A solução proposta para o problema é utilizar um método recursivo que reduz a expressão em duas sub-expressões, uma à esquerda e outra à direita de um operador, em busca de uma sub-expressão já resolvida ou de um caso base. Para avaliar todos os casos, em cada expressão todas as sub-expressões são geradas através de um *loop*.

Para tratar o problema através do paradigma de Programação Dinâmica, utilizou-se duas matrizes de memória que armazenam os valores já calculados, uma para as combinações de parentização em que os resultados são verdadeiros e outra matriz em que os resultados são falsos.

Nas matrizes de memória o índice da coluna representa o início da expressão e o índice da linha representa o final da expressão. Exemplo: numa expressão de tamanho t, o resultado do cálculo da sub-expressão de i a j é armazenado nas matrizes nas posições: MemóriaTrue[i][j] e MemóriaFalse[i][j].

2.1. MODELAGEM

2.1.1. Estruturas de dados

O problema foi modelado na estrutura de um vetor dinâmico de tipo definido no código, criando um novo tipo de dado Expressão Booleana (BoolExp). O objetivo foi criar um Tipo Abstrato de Dado para que, caso seja necessária a mudança do tipo do vetor definido durante o desenvolvimento do algoritmo, essa mudança implicasse em alterações somente no TAD. Dessa forma, a execução do programa abstrai a implementação do tipo, operando sobre as funções do TAD.

Para cada instância do programa, o vetor de BoolExp é alocado, realiza-se a contagem das parentizações, os resultados são então escritos no arquivo de saída e por fim o vetor é liberado para a execução da próxima instância.

A complexidade de espaço para armazenamento da expressão é $O(t)$. Contudo, o armazenamento dos resultados nas matrizes de resultados é $O(2t^2)$ já que deverão ser armazenadas 2 matrizes e em cada uma todas as possibilidades de sub-expressões de dimensão $t \times t$. Assintoticamente, a complexidade de espaço final é $O(t^2)$.

2.2. Método de resolução recursivo

A complexidade assintótica dessa função é $O(n^3)$, visto que ela divide em sub-expressões à esquerda e à direita do operador testado na iteração corrente, e testa todas as sub-expressões com operadores dessa sub-expressão.

A seguir uma descrição do algoritmo recursivo utilizado.

O lado esquerdo de uma expressão é representado em i a $k-1$, k é o operador

entre os lados esquerdo e direito, e o lado direito é $k+1$ a j. Ou seja, $(i, k-1)k(k+1, j)$.

1: Conta Parentização(expressão, i, j, k)

```
//Caso base da recursão;
if i = j then
    Armazena o valor da variável nas matrizes memória (exp[i]);
else
    for k = i; k < j; k++ do
        if é um operador ou um NOT (exp[k]) then
            if não é NOT(exp[k]) then
                //lado esquerdo da expressão;
                if Não foi calculado ainda(i, k-1) then
                    ContaParentizacao(i, k-1);
                end
            end
            //lado direito da expressão;
            if NaoCalculou(k+1, j) then
                ContaParentizacao(k+1, j);
            end
            Avalia o lado esquerdo e direito com o operador k entre eles(exp[k], i, j, k);
        end
    end
end
```

3. IMPLEMENTAÇÃO

O critério de separação dos arquivos do programa objetivou a modularização do mesmo, para possível posterior utilização, evitar duplicamento de funções, além de melhorar a legibilidade e organização do mesmo.

3.1. Código

3.1.1. Arquivos .c

- **main.c:** Arquivo principal do programa, contém as chamadas de funções de entrada e saída e chamada de execução da função principal.
- **io.c:** Realiza a comunicação do programa com o ambiente, lendo o arquivo de entrada, armazenando as informações lidas na memória e escrevendo os resultados nos arquivos de saída.
- **heuristica.c:** Define o funcionamento da função principal de programa.
- **boolexp.c:** Define as funções do TAD Expressão Booleana.
- **timerlinux.c:** Contém funções para mensurar o tempo de execução do programa.

3.1.2. Arquivos .h

- **io.h:** Contém o cabeçalho das funções de entrada e saída.
- **heuristica.h:** Define o cabeçalho da principal função de execução do programa e define o tipo de dado que a memória utilizará.
- **boolexp.h:** Define o cabeçalho das funções do TAD Expressão Booleana e o tipo de dado do vetor BoolExp.
- **timerlinux.h:** Cabeçalho e instruções para cronometrar o tempo de execução do programa.

3.2. Compilação e execução

O programa deve ser compilado através do compilador GCC através de um *makefile* ou através do seguinte comando em terminal:

```
gcc main.c boolexp.c heuristica.c io.c timerlinux.c -o tp3
```

Para execução do programa, são requeridos dois parâmetros, o nome do arquivo de entrada e do arquivo de saída, em qualquer ordem. Caso não haja ao menos 2 argumentos, o programa encerra a execução. A análise de execução do programa pode ser habilitada passando $-a$ para a execução do programa, habilitando a escrita das medidas de avaliação do programa num arquivo de texto. O tamanho máximo de nome de arquivo é 255 caracteres. Um exemplo é dado a seguir.

```
./tp3 -i input.txt -o output.txt
```

3.3. Entrada e saída

3.3.1. Formato da entrada

O arquivo de entrada cujo nome é passado como argumento para o executável deve conter, na primeira linha, a quantidade de instâncias que este arquivo possui, ou seja, quantas frases estão contidas nesse arquivo e deverão ser executadas pelo programa.

Em seguida, em cada linha há as expressões a serem avaliadas pelo programa. Não podem haver espaços no final de cada linha, do contrário o programa será executado incorretamente.

Um exemplo de arquivo de entrada é dado a seguir:

```
2
not False or False and True
True and not True and True
```

3.3.2. Formato da saída

A saída do programa contém informações sobre a contagem das formas da expressão avaliada ser *True*, respectivamente de acordo com ordem do arquivo de entrada. Ou seja, a 1ª frase da entrada possui sua resposta na 1ª linha do arquivo de saída.

Um exemplo de saída é dado abaixo:

```
5
1
```

4. ANÁLISE EXPERIMENTAL

4.1. Gerador de entradas

Para avaliar as heurísticas utilizadas, criou-se um gerador de entrada, que gera entradas aleatoriamente de tamanho $0 < t \leq 50$. O gerador segue anexo a este documento.

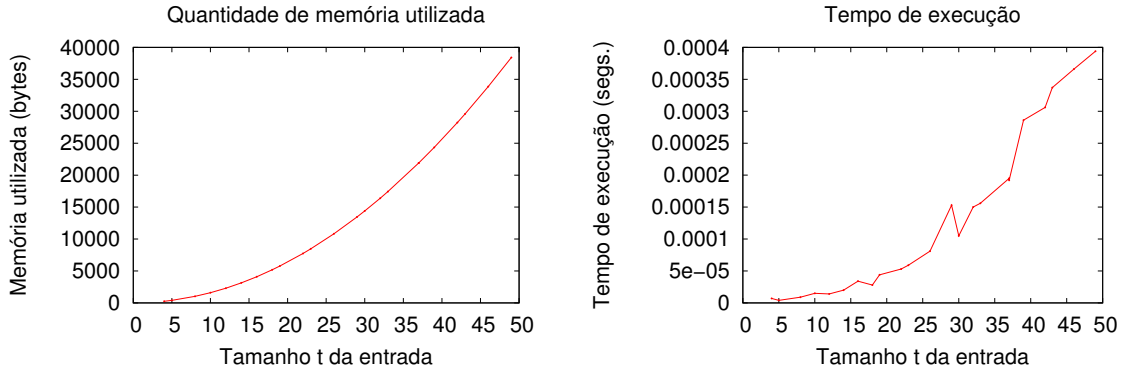
4.2. Resultados

Para analisar a qualidade dos resultados gerados, utilizou-se as métricas já citadas.

Abaixo uma tabela com os resultados obtidos.

Tamanho t entrada	Quantidade de memória alocada (bytes)	Quantidade de parentizações	Tempo execução(seg.)
4	256	1	0.000007
5	400	1	0.000004
8	1024	14	0.000009
10	1600	47	0.000015
12	2304	33	0.000014
14	3136	181	0.000020
16	4096	4004	0.000034
18	5184	1218	0.000028
19	5776	8102	0.000044
22	7744	27556	0.000053
23	8464	82885	0.000059
26	10816	313583	0.000081
29	13456	3251665	0.000153
30	14400	9275321	0.000105
32	16384	22296146	0.000150
33	17424	30030824	0.000156
37	21904	208951422	0.000195
37	21904	577662545	0.000192
39	24336	2414372285	0.000286
42	28224	2264464499	0.000306
43	29584	2962363739	0.000337
46	33856	3257643153	0.000366
49	38416	1092737478	0.000394

Tabela 1. Resultados análise experimental



5. CONCLUSÃO

A partir da análise experimental, nota-se que obviamente o tempo de execução cresceu à medida que o tamanho de entrada aumenta. Outra observação é de que o tamanho da entrada é limitado à 50, possibilitando a execução do algoritmo em pequeno espaço de tempo, mesmo sendo um algoritmo de complexidade cúbica. A memória alocada para armazenar os resultados, também cresceu, mas como pode-se notar no gráfico acima, de forma exponencial e mais acentuada que o tempo de execução.

Observa-se que o paradigma de Programação Dinâmica possibilitou o menor tempo de execução do programa, mas a preço do custo quadrático de memória nesse

caso. Dessa forma, esse paradigma é adequado para situações em que necessita-se um otimizar o tempo de execução e há possibilidade de arcar com o custo de memória.

Uma decisão tomada durante a implementação do algoritmo, foi a de se usar o tipo *unsigned long long int* para armazenar os resultados já calculados na matriz de memória. Essa decisão foi tomada pois à medida que a entrada crescia, os valores de *unsigned long int* não eram suficientes para armazenar os valores. Isso acarretou em maior espaço para armazenamento, visto que esse tipo utilizado requer 64 bits cada variável.

O trabalho atingiu seus principais objetivos: exercitar o paradigma de programação Programação Dinâmica, analisando seu desempenho em termos de tempo de execução e quantidade de memória alocada.

Dentre as adversidades encontradas durante o desenvolvimento desse algoritmo, a maior foi tratamento do caso do *not*, que apesar da solução ser aparentemente trivial o raciocínio implicou levantar diversos casos e testagem de diversas maneiras de implementá-lo.

Algumas melhorias que poderiam ser consideradas neste trabalho são:

- Reduzir o tamanho da matriz de memória à somente o necessário, pois foi constatado que alguns campos não são utilizados.

6. REFERÊNCIAS

[1] Cormen, T. (2001). Introduction to algorithms. MIT press.

[2] Ziviani, N. (2007). Projeto de Algoritmos com implementacoes em Pascal e C. Pioneira Thomson Learning.

[3] MROZEK, Michael - Disponível em: http://stackoverflow.com/questions/6701812/counting-boolean-parenthesizations-implementation Acesso em 09/05/2012

[4] DEAN, C. Brian - Disponível em: http://people.csail.mit.edu/bdean/6.046/dp/ Acesso em 09/05/2012