UNIVERSIDADE FEDERAL DE VIÇOSA - *CAMPUS* FLORESTAL

Murillo Santhiago Souza Jacob - 4243

Victória Caroline Silva Rodrigues - 3584

Willyan Guilherme Lima e Silva - 4233

TRABALHO PRÁTICO 02

SUMÁRIO

Introdução **3**
2. Implementação **3**
2.1 Estrutura de Dados 3
2.1.1 Tabela Verdade 3
2.1.2 Modos de Funcionamento 5
2.3 Programa Principal 5
2.4 Organização do Código e Decisões de Implementação 5
3. Análise de Complexidade **6**
3.1 Programa principal 6
3.2 Tabela Verdade 6
3.2 Modos de Funcionamento 9
4. Testes **13**
Conclusão **14**
Referências **14**

Introdução

Neste trabalho prático iremos abordar o tema de análise de complexidade, ou seja, iremos considerar os aspectos de tempo de execução e espaço ocupado durante a implementação do código.

O objetivo do trabalho é realizar a análise e avaliação do impacto do desempenho do algoritmo durante a execução, para isso iremos fazer a implementação do problema conhecido por 3-FNC-SAT, que é uma versão de SAT. Em seguida, devemos calcular o tempo gasto, de acordo com os diferentes valores de entrada N.

2. Implementação

2.1 Estrutura de Dados

Para a implementação do trabalho foram criados os seguintes de arquivos de cabeçalho:

2.1.1 Tabela Verdade

void GerarTabela(int numVariaveis,int numClausulas,int **matrizPTR,int opcao):

Essa função tem a finalidade de gerar a tabela da verdade a partir do número de variáveis informada pelo usuário. Ela também recebe outra função chamada Expressão Logica, que de acordo com o resultado dessa função imprime os valores verdadeiros da expressão booleana.

int ExpressaoLogica(bool tabelaVerdade[ ],int numVariaveis,int numClausulas,int **matrizLogicaPTR)

Busca retornar o valor 0 quando a expressão booleana é falsa e 1 quando é verdadeira.

Para realizar essa verificação iremos utilizar um vetor de tamanho 3 que irá armazenar os possíveis valores de cada linha da tabela da verdade de negado ou não negado. Depois disso, pegamos os 3 valores e iremos fazer uma OR (ou) para cada valor do vetor, e por fim, armazenamos esse valor em um vetor que usaremos para verificar se a expressão é verdadeira ou falsa. Ou seja,

caso cada posição do vetor seja 1, expressão verdadeira e retorna 1, caso contrário, retorna 0.

int ExpressaoLogicaIterativa(bool tabelaVerdade[ ],int numVariaveis,int numClausulas,int **matrizLogicaPTR)

É a função que gera a saída lógica quando se usa o modo Iterativo, retorna 0 para as saídas lógicas Falsas, e 1 para as saídas lógicas Verdadeiras.

Para conseguir gerar essa saída, utilizamos de 2 vetores, um vetor com 3 posições, do tipo bool, chamado LinhaLógica que será responsável por armazenar os valores, cláusula por cláusula, e também outro vetor do tipo bool, chamado de LogicaBool, que tem como tamanho, o número de cláusulas, e será responsável por armazenar a OR(ou) entre os 3 elementos de cada cláusula.

Para fazer isso, pegamos o valor salvo dentro da variável matrizLogicaPTR, passada como parâmetro da função, que funciona como uma matriz com alocação dinâmica, e verificamos seus valores, sabendo que cada cláusula está alocada em uma linha da matriz. Então, percorremos cada linha da matriz, sendo que em cada vez que fazemos isso, checamos qual é o valor armazenado na posição, se ele for negativo, pegamos então a negação do valor da coluna do módulo do elemento da matrizLogicaPTR menos um (pois os valores gerados para a variável 1 na tabela verdade, começa na coluna 0), do vetor tabelaVerdade,, ou seja, supondo que o valor salvo na matriz seja -4, vamos pegar no código !(tabelaVerdade[abs(4)-1], em que abs corresponde ao módulo, e armazena-se esse valor dentro da posição x (em que x corresponde ao contador for) do vetor LinhaLogica, e após esse for percorrer toda a linha do vetor, armazenamos na posição y (em que y se refere ao contador de outro for) do vetor LogicaBool, a expressão LinhaLogica[0] | LinhaLogica[1] | LinhaLogica[2], que é a or entre todos os valores de LinhaLogica, que correspondem a uma cláusula. Esse processo é feito até que todas as cláusulas sejam percorridas.

Após isso, percorre-se o vetor Lógica Ball, e se qualquer variável dele for 0, a função retorna o valor 0, pois em uma and, qualquer valor de 0 na entrada, resulta em 0 na saída. Porém, se todos os valores da entrada forem 1, o programa retorna 1, dizendo que a saída lógica da expressão é 1.

2.1.2 Modos de Funcionamento

void ModoIterativo();

No modo interativo todos os dados serão inseridos pelo usuário permitindo a inserção da quantidade de variáveis e podendo definir a expressão booleana, ou seja, definir se a variável será negada ou não. Este modo gera uma matriz alocada dinamicamente na memória, em que nela, são armazenados todos os valores digitados pelo usuário, sendo que em cada linha, tem uma cláusula armazenada, e o número de linhas é o número de cláusulas. Quando as variáveis são negadas, a variável é multiplicada por -1, então, por exemplo, se a variável for 4 e ela tiver negada, ela será armazenada como -4 na matriz.

void ModoAutomatico():

Nessa função fizemos a geração de expressão booleana aleatoriamente de acordo com a quantidade de variáveis e usamos a equação (qtdvariaveis/3)*2 para definir a quantidade de cápsulas, e logo em seguida, imprimimos essa tabela.

E por fim, chamamos a função Gerar Tabela para criar a tabela da verdade e fazer as comparações, como já foi especificado acima.

2.3 Programa Principal

O programa implementado tem a finalidade de imprimir as saídas verdadeiras de acordo com a tabela da verdade e com a expressão booleana passada pelo usuário ou criada de maneira automática de acordo com a quantidade de variáveis informadas. Dessa forma, desenvolvemos um menu com as opções modo interativo, em que o usuário poderá inserir a quantidade de variáveis e cápsulas e informar a expressão booleana. E o modo automático que irá inserir somente quantidade de variáveis.

2.4 Organização do Código e Decisões de Implementação

Para realizar a implementação do código decidimos optar por criar dois arquivos cabeçalhos separados do arquivo principal, onde se encontraria o menu de funcionalidades do programa, com a interação automática ou mecânica. Dessa forma, teríamos um segmento com as especificações, que o usuário poderia interagir e outras com as implementações, onde as operações seriam

realizadas. Assim, temos um código mais organizado e mais prático de encontrar alguma falha no sistema.

Outro fato que decidimos implementar foi alocar dinamicamente blocos de memória utilizando o conceito de ponteiro para ponteiros para os valores inseridos pelos usuários para as expressões booleanas, ou seja, utilizamos o processo de alocação dinâmica. Também utilizamos a tática ponteiro para ponteiro, a cada nível do ponteiro criamos uma nova dimensão no vetor. E por fim, a memória foi liberada na ordem inversa da alocação. Escolhemos essa forma, pois à medida que a lista fosse crescendo os novos valores eram armazenados no espaço de memória durante o tempo de execução, fazendo com que não estourasse o espaço de memória.

3. Análise de Complexidade

3.1 Programa principal

No programa principal temos 4 opções que o usuário pode escolher. A primeira, o modo interativo, a segunda, modo automático, terceira, sair do programa e a quarta, alternativa incorreta, tente novamente. Dessa forma, a complexidade do programa para o melhor caso é a função f(n) = 1, caso ele escolha a primeira, segunda ou terceira opção. O pior caso seria se ele errasse todas as opções de entrada, sendo a função f(n) = n, n sendo o número de vezes que repetiu o comando de repetição while.

3.2 Tabela Verdade

Nesse cabeçalho temos 3 funções que têm o papel de realizar a geração da tabela da verdade e fazer a verificação e impressão dos valores verdadeiros da expressão booleana a partir da tabela da verdade. Agora iremos realizar os cálculos de complexidade de cada uma dessas funções.

void GerarTabela(int numVariaveis,int numClausulas,int **matrizPTR,int opcao):

Como podemos observar pelo código abaixo para realizar a geração da tabela da verdade passamos por um comando de condição, ou seja, se a condição inicial for verdadeira, ignoramos a seguinte, caso contrário, realizamos as duas operações.

```
for(i=0;i<pow(2,numVariaveis);i++){
    contador = i;
    //conta em binario
    for (j = numVariaveis-1; j >= 0; j--){
        if (contador % 2 == 0)
            contadorBinario[j] = 0;
        else
            contadorBinario[j] = 1;
        contador = contador / 2;
    }
```

Esse algoritmo possui 2 loops aninhados: o externo faz 2^n iterações for(i=0;i<pow(2,n);i++). Dentro desse loop há outro loop fazendo a uma atribuição, passando por um comando de condição (if), verdadeiro ou falso. Que realiza as operações n-1 + n-2 + ... + 0 = n iterações for(j=n-1;j>=0;j++), ou seja, a função f(n) = (n*(2^n)).

Dentro dessa função ainda iremos fazer a impressão das saídas verdadeiras. E como podemos observar que temos dois loops separados que variam pela quantidade de variáveis, que denominamos como n, portanto, temos a função f(n) = 2n.

```
if(flagMensagem==0){
    printf("Variaveis\n");
    for(k=1;k<=numVariaveis;k++){
        printf("%d ",k);
    }
    printf("\n");
    printf("\n");
    flagMensagem =1;
}
for(k=0;k<numVariaveis;k++){
    if(contadorBinario[k]==1)
        printf("V ");
    else
        printf("F ");
}
printf("\n");
}
break;
```

f(n) = n

f(n) = n

Juntando todos os f(n) encontrados temos que f(n) = (n*(2^n)) + 2n.

int ExpressaoLogica(bool tabelaVerdade[],int numVariaveis,int numClausulas,int **matrizLogicaPTR);

Esse algoritmo também possui 2 loops aninhados: o externo faz (n/3)*2 iterações, vamos definir n como número de variáveis, logo temos que for(i=0;i<(n/3)*2;i++), a função f(n) = (n/3)*2.Dentro desse loop há outro loop fazendo a uma atribuição, passando por um comando de condição (if e else), verdadeiro ou falso. Que faz n iterações, for (j=0;j<n;j++), ou seja, f(n) = n.

Dessa forma, f(n) = (n*(n/3)*2), simplificando temos, f(n) = (⅔)*n^2.

```
for(i=0;i<numClausulas;i++){
    for(j=0;j<numVariaveis;j++){
        if(matrizLogicaPTR[i][j]==1){
            LinhaLogica[posicaoVetor] = !(tabelaVerdade[j]);
            posicaoVetor++;
        }
        else if(matrizLogicaPTR[i][j]==2){
            LinhaLogica[posicaoVetor] = tabelaVerdade[j];
            posicaoVetor++;
        }
    }
    posicaoVetor=0;
    LogicaBool[i] = LinhaLogica[0] || LinhaLogica[1] || LinhaLogica[2];
}
```

Por fim, temos um último comando de repetição, que iremos assumir o pior caso, ou seja, percorrer todo o for(i=0;i<(n/3)*2;i++), onde temos (n/3)*2 iterações, f(n) = (n/3)*2.

```
for(i=0;i<numClausulas;i++){
    if(LogicaBool[i]==0)
        return 0;
}
return 1;
```

Dessa forma, temos a função f(n) = (n/3)*2 + (⅔)*n^2.

int ExpressaoLogicaIterativa(bool tabelaVerdade[],int numVariaveis,int numClausulas,int **matrizLogicaPTR);

Utilizando o mesmo raciocínio da função anterior temos o seguinte:2 loops aninhados: o externo faz m iterações, onde m é o número de cápsulas, for(i=0;i<m;i++), f(m) = m. Dentro desse loop há outro loop fazendo a uma

atribuição, passando por um comando de condição (if e else if), verdadeiro ou falso. Que faz 3 iterações, for(j=0;j<3;j++), f(m) = 3. Portanto, temos f(m) = 3*m.

```
for(i=0;i<numClausulas;i++){
    for(j=0;j<3;j++){
        if(matrizLogicaPTR[i][j]<0){
            LinhaLogica[j] = !(tabelaVerdade[abs(matrizLogicaPTR[i][j])-1]);
        }
        else if(matrizLogicaPTR[i][j]>0){
            LinhaLogica[j] = tabelaVerdade[abs(matrizLogicaPTR[i][j])-1];
        }
    }
    LogicaBool[i] = LinhaLogica[0] || LinhaLogica[1] || LinhaLogica[2];
}
```

E temos outro for que vamos supor o pior caso, em que teremos que percorrer todo for. Logo, temos que f(m) = m.

```
for(i=0;i<numClausulas;i++){
    if(LogicaBool[i]==0)
        return 0;
}
return 1;
```

Dessa forma, temos a função f(m) = 3m^2.

3.2 Modos de Funcionamento

void ModoIterativo();

Nesse algoritmo temos primeiramente um for simples. Vamos definir m, como número de cápsulas e n como número de variáveis, igual estávamos fazendo anteriormente nas outras funções.

```
for (i=0; i < numClausulas; i++)
    ValoresBool[i] = malloc (3 * sizeof (int));
for(i=0;i<numClausulas;i++){
    for(j=0;j<3;j++){
        ValoresBool[i][j] = 0;
    }
}
for(i=0;i<numClausulas;i++){
    for(j=0;j<3;j++){
        printf("Insira o numero da variavel\n");
        scanf("%d",&numVariavel);
        printf("Insira se ela esta negada(1) ou nao(2)\n");
        scanf("%d",&Negacao);
        if(Negacao==1)
            ValoresBool[i][j] = numVariavel * -1;
        else if(Negacao==2)
            ValoresBool[i][j] = numVariavel;
    }
}
```

Como podemos observar, temos o seguinte:

O primeiro for, com a f(m) = m. Logo em seguida, temos 2 loops aninhados: o loop externo faz m iterações, logo, f(m) = m. E loop interno faz 3 iterações, logo temos, f(m) = 3. Dessa forma, temos a função f(m) = m + 3*m.

O próximo 2 loops aninhados temos: o loop externo faz m iterações, logo, f(m) = m. E loop interno faz 3 iterações, logo temos, f(m) = 3. Dessa forma, temos a função f(m) = 3*m.

Juntando todos os f(n) temos: f(n) = 3*m + m + 3*m = 6m+m

void ModoAutomatico():

Nesse algoritmo temos primeiramente um for simples. Vamos definir (n/3)*2, como número de cápsulas e n como número de variáveis.

```
numClausulas = (qntVariaveis/3)*2;
//MATRIZ COM ZERO EM TODAS AS POSIÇOES
int **ValoresBool;
ValoresBool = malloc (numClausulas * sizeof (int*));
for (i=0; i < numClausulas; i++)
    ValoresBool[i] = malloc (qntVariaveis * sizeof (int));
for (i=0; i<numClausulas;i++){
    for (j=0; j<qntVariaveis;j++){
        ValoresBool[i][j] = 0;
    }
}
```

Como podemos observar, temos o seguinte:

O primeiro for, com a f(n) = (n/3)*2.Logo em seguida temos 2 loops aninhados: o loop externo faz (n/3)*2 iterações, logo, f(n) = (n/3)*2. E loop interno faz n iterações, logo temos, f(n) = n. Dessa forma, temos f(n) = (n/3)*2 + (n/3)*2*n

Depois disso vamos fazer a geração dos valores aleatórios da expressão booleana.

Vamos começar analisando por partes.

```
//Gerando expressoes booleanas aleatoria
for (i=0; i< numClausulas;i++){
    for(k=0;k<3;k++){
        valoresPos[k] = qntVariaveis+1;
    }
    for(j=0;j<3;j++){
        flag = 0;
        v = 1+(rand()%2) ;
        pos = rand()%qntVariaveis;
        for(k=0;k<3;k++){
            if(valoresPos[k]==pos){
                flag=1;
                j--;
            }
        }
        if(flag!=1){
            ValoresBool[i][pos] = v; //mat
            valoresPos[j] = pos;
        }
    }
}
```

f(n) = (n/3)*2

f(n) = 3

f(n) = 3

f(n) = 3

Dessa forma, temos f(m) = (n/3)*2*3*3*3 = (n/3)*54 = n*18

Logo em seguida, imprimimos essa matriz gerada com as expressões booleanas, de forma que o usuário consiga verificar se está ocorrendo da maneira correta.

```
printf("Matriz Gerada:\n");
for (i=0; i < numClausulas; i++){
    for(j=0;j<qntVariaveis;j++)
        printf("%d\t",ValoresBool[i][j]);
    printf("\n");
}
```

Esse algoritmo possui 2 loops aninhados: o externo faz (n/3)*2 iterações for(i=0;i< (n/3)*2;i++). Dentro desse loop há outro loop fazendo a impressão. Que realiza n iterações for (j=0;j<n; j++), f(n) = n. Portanto, temos f(n) = (n/3)*2*n

Juntando todas as f(n) forma temos a função f(n) = (n/3)*2*n + n*18 + (n/3)*2 + (n/3)*2*n

	f(n) / f(n, m)
Programa principal	1
void GerarTabela	(n*(2^n)) + 2n
int ExpressaoLogica	(n/3)*2 + (⅔)*n^2
Int ExpressaoLogicaIterativa	3m^2
void ModoIterativo	6*m + m
void ModoAutomatico	(n/3)*2*n + n*18 + (n/3)*2 + n*(n/3)*2

Para o modo Interativo, teremos o seguinte:

F(n, m) = 1+ (n*(2^n)) + 2n + 6*m + m + 3m^2

Substituindo todas as constantes por c, temos:

F(n, m) = c + (n*(2^n)) + 2n + c*m + m + c*m^2

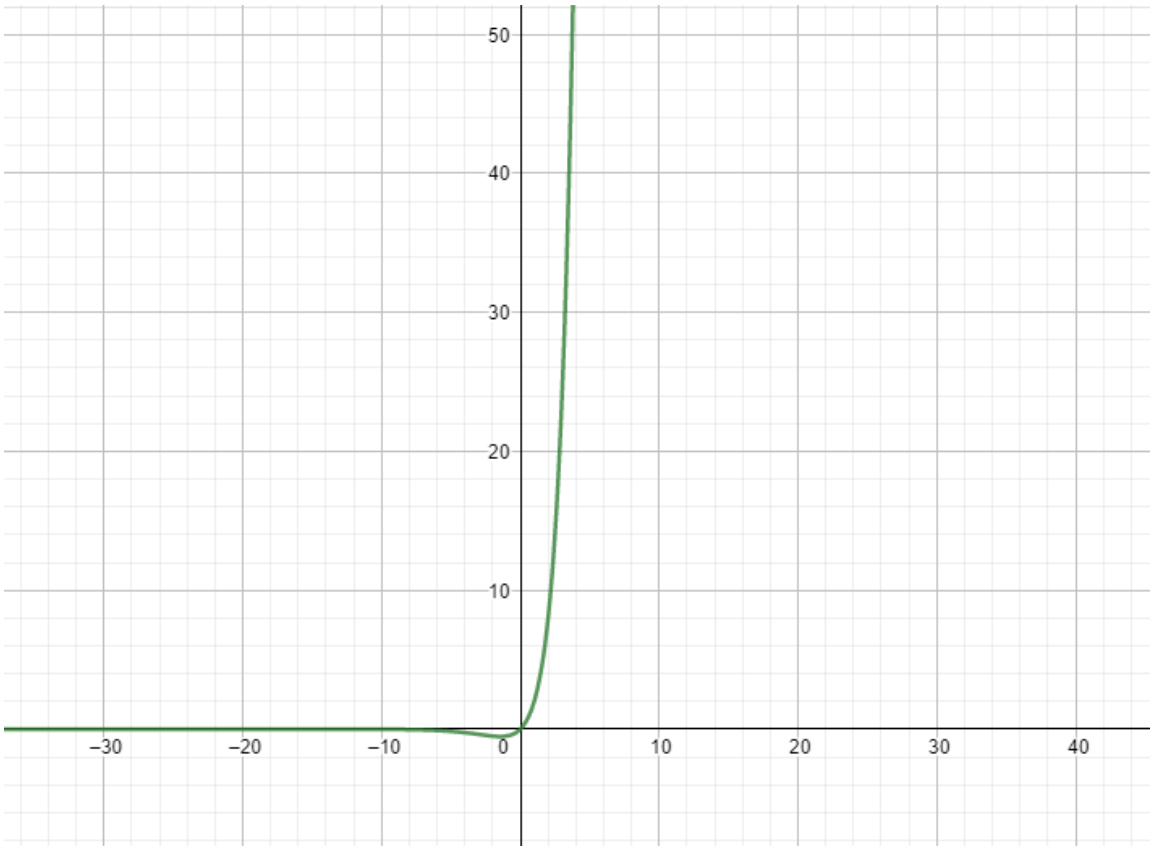
Para o modo automático, teremos o seguinte:

F(n) = 1 + (n*(2^n)) + 2n + (n/3)*2 + (⅔)*n^2 + (n/3)*2*n + n*18 + (n/3)*2 + n*(n/3)*2

Substituindo todas as constantes por c, temos:

F(n) = c + (n*(2^n)) + 2n + n*c+ c*n^2 + c*n^2 + n*c+ n*c + c*n^2

Como podemos observa nas duas tempo de execução f(n) do programa é O(n*(2^n)), significa o programa é da ordem de no máximo n*(2^n).



4. Testes

Nos testes executados utilizamos a máquina com as seguintes especificações:

- Memória ram com 16G
- Placa de video Gtx 1050 ti DDR4
- Processador i7 7700k
- Windows 10

No teste automático de 15 variáveis demorou em torno de 7998ms, ou seja, em torno de 8 segundos.

No teste automático de 20 variáveis demorou em torno de 243102ms, ou seja, em torno de 4 minutos.

Nos testes de o número de variáveis superior a 30 conseguimos executar o código, porém não conseguimos informar o tempo de execução, pois demoraria muito e ficou inviável fazer essa contagem.

Seria razoável executar o seu algoritmo para valores de N maiores do que 45? Justifique a resposta. Aparentemente pelo desenvolvimento do trabalho, se tornaria inviável calcular com n maior 45, visto que com 30 para cima o programa já teve dificuldades de realizar as operações.

Conclusão

A implementação do trabalho transcorreu sem maiores problemas, conseguimos realizar a implementação e a análise de complexidade de desempenho solicitados.

Entretanto, ao decorrer do trabalho surgiram algumas dificuldades a partir do aumento do número de variáveis, ou seja, quanto maior o número, maior eram as chances do código não conseguir compilar ou ter um tempo não viável. Dessa forma, podemos concluir que usar a força bruta para resolver um problema pode trazer sérios problemas como tempo de execução ou espaço de memória.

Referências

N. Ziviani, Projeto de Algoritmos, Editora Cengage, 2006