Universidade Federal de Viçosa
Campus Florestal
CCF 251 - Algoritmos e Estruturas de Dados I

# Trabalho Prático 2

# Trabalho Prático 02 – AEDS 1

Gabriel Miranda - 3857
Felipe Dias - 3888
Mariana Souza - 3898

# Sumário

## 1 Introdução

Neste trabalho prático foram desenvolvidas funções a fim de achar soluções que seriam plausíveis para o “Problema da Satisfabilidade (SAT)”. Para resolver esse problema, é necessário fazer com que a equação que é composta por cláusulas seja no fim, verdadeira. Explicando melhor a equação, temos que cada cláusula é formada por 3 tuplas, essas tuplas são compostas por duas posições. A primeira demonstra a variável que vai de 1 até n, sendo n o número de combinações com repetição possíveis para $2^n$ , e a segunda representa o sinal de negado-(1) ou não negado-(2). As tuplas de cada cláusula são separadas pelo operador lógico (|) - ou, e as cláusulas separadas pelo operador lógico (&) - e. A equação é representada da seguinte forma: Para n = 3 e número de cláusulas C = 2; temos:
{(1,2) | (2,2) | (3,2)} & {(2,2) | (2,1) | (1,2)}, nesse caso a combinação que satisfaria esse exemplo seria: 1 = V, 2 = F, 3 = V, umas das possíveis soluções.

Sendo assim, passando esse problema para a linguagem de programação, é bem mais simples calcular o número de combinações possíveis para resolver essa equação. Sendo que, para achar a solução, basta verificar se uma determinada combinação satisfaz essa equação, e analisar quais seriam os melhores casos possíveis para tornar a equação verdadeira, ou seja, se pelo menos uma tupla de cada cláusula for verdadeira, a cláusula se torna verdadeira.

## 2 Desenvolvimento

Para alcançar o objetivo do trabalho prático proposto, avaliamos o impacto causado pelo desempenho dos algoritmos em sua execução. Iniciamos pela busca de um algoritmo que realizasse as combinações, posteriormente, implementamos o algoritmo no modo interativo para gerar todas as combinações possíveis de valores booleanos para o tamanho N. No modo automático, o valor N é atribuído conforme o número de cláusulas, gerando aleatoriamente a expressão booleana.

Ao final das implementações, foi utilizado a biblioteca em c para executar o programa com os diferentes valores de N e medir o tempo de execução gasto.

### 2.1 Algoritmo de Combinação

Começamos o processo de desenvolvimento procurando um algoritmo de permutação capaz de gerar todas as possíveis combinações booleanas para N valores lógicos. Foram aplicadas algumas alterações para adequar ao programa proposto. Abaixo temos o código em C para gerar as combinações com repetição para um determinado conjunto de entradas N. Onde input será a string de entrada com MAX de 250 caracteres, e a variável str irá receber cada combinação, a alocação de memória é um vetor de r+1 posições onde a última é reservada para indicar quando todos os números de tamanho r foram gerados, também fizemos algumas adaptações para adequar o código à descrição da documentação do trabalho prático proposto, sendo assim, optamos por utilizar esse algoritmo pois ele apresenta apenas estruturas simples de loops e vetores para elaborar todo o código.

```c
int main(){
    int *num;
    char input[MAX] = "VF" , str[MAX];
    int n,r, i, j, k;
    printf("Entre com o N: ");
    scanf("%d", & r);
    n = strlen(input) ;
    j = 0;
    str[0] = 0 ;
    for ( i = 0; i < n; i++ ) {
        if ( strchr(str, input[i]) == NULL ) {
            str[j] = input[i] ;
```

```
            j++ ;
            str[j] = 0 ;
        }
    }
    strcpy(input, str) ;
    n = strlen(input) ;
    num = (int *) calloc( (r+1), sizeof(int)) ;
    if ( num == NULL ) {
        perror("calloc") ;
        return -1;
    }
    while ( num[r] == 0 ) {
        for ( i = 0; i < n; i++ ) {
            for ( j = 0, k = r-1; j < r; j++ ) {
                str[k] = input[num[j]] ;
                k-- ;
            }
            str[r] = 0 ;
            printf("{%s} ", str) ;

            num[0]++ ;
        }
        for ( i = 0; i < r; i++ ) {
            if ( num[i] == n ) {
                num[i] = 0 ;
                num[i+1]++ ;}
        }
    }
    return 0;
}
```

Link algoritmo de combinação:
(https://daemoniolabs.wordpress.com/2011/02/11/gerando-permutacoes-r-com-repeticao-em-c/).

## 2.2 Estruturas

Foram criadas para a inserção efetiva dos conceitos de tuplas e cláusulas no código, respectivamente, as estruturas Ttupla e Tclausula. Ttupla representa uma única tupla e é composta de um campo inteiro var (variável entre 1 e N do literal) e um campo inteiro VF (indicando se o literal é 1 - negado ou 2 - não negado). Já a Tclausula é composta de um único campo, que é um vetor Ttupla de três posições.

## 2.3 Transforma True

A função "transforma true" analisa cada tupla de cada cláusula, e quando achado um valor verdadeiro para esse tupla ele já não verificar se a próxima tupla é verdadeira ele vai para a próxima cláusula e verificar se existe pelos menos uma tupla que seja verdadeira na cláusula, e como isso funciona, vamos aplicar em um exemplo, se n = 2 e C = 2, temos: {(1,1),(1,2),(2,2)}, {(1,2),(1,1),(2,2)}, cada variável representa um posição na string que tem as combinações, temos a string[v,v], a variável 1 seria na string a posição 0, é só observar no algoritmo abaixo, valor = tupla[i].var, e dentro do índice da string que se chama combinação, no algoritmo, temos combinacao[valor-1], nesta posição iremos verificar se essa variável da tupla é verdadeira de acordo com sua combinação, se esse valor for igual a 1, significa que é negado e se a combinação for verdadeira então naquela tupla temos uma variável falsa, se não for verdadeira, no caso falsa, então ela será verdadeira, e assim sucessivamente como mostrado no código abaixo. Para saber se aquela combinação satisfaz a equação é só fazer um contador que conte todas a vezes que o programa ache uma tupla verdadeira e depois pare a execução e vá para outra cláusula, e no fim se esse contador for igual ao número de cláusulas a combinação satisfaz a equação, por que no mínimo em cada cláusulas existe uma tupla verdadeira.

```
void TransfomarTrue(Tclausula *clausula, int nclaus, char*combinacao){
     int fazTrue = 0;
     int i,j = 0;
     int valor;
     char verifica;
     bool op = true;

     for(i = 0; i < nclaus; i++){
         op = true;              // n
         j = 0;                  // n

         /*se n = 2 -- combinacao[0,1]-2 caracteres -
         *valor vai de de 1 até N que nesse caso é 2;
         *então (valor = 1 -> combinacao[0, ou valor-1])
         */
         while((op == true) && (j<3) ){
             valor = clausula[i].tupla[j].var;    //1 ou 3n
             /*recebendo posicao se valor = 1;
             *na combinação[0] - combinacao[valor-1]
             */
               if(clausula[i].tupla[j].VF == 1){//se é negado//0 ou 3n
```

```
                    //se for negado a combi[i] = V;
                    if(combinacao[valor-1] == 'V'){   //0 ou 3n
                        verifica = 'F'; //verifica = F;
                    }else{
                        verifica = 'V'; //verifica = V //0 ou 3n
                    }
                }else{ //se nao é negado

                    //se combinacao[i] = V
                    if(combinacao[valor-1] = 'V'){ //0 ou 3n
                        verifica = 'V';             //0 ou 3n
                    }
                }
                if(verifica == 'V'){ //0 ou 3n
            //achar pelo menos um valor V na cláusula, para o programa
                    fazTrue++;   // 0 ou 1
                    op = false;   //0 ou 1
                }
            j++; // 0 ou 3n
        }
    }
    /*se a quantidade de verdadeiros
    * em cada cláusula for igual ao Nº cláusulas
    * - condição verdadeira
    */
    if(fazTrue == nclaus){ // n
        printf("Essa solucao resolve a Equacao:%s\n", combinacao);
    }
}
```

Função de complexidade da transformaTrue:

Melhor caso:
F(n) = n + n + 1 + n = 3n + 1  => O(n)

Pior caso:
F(n) = n + n + 3n + 3n + 3n + 3n + 3n + 1 + 1+ 3n + n = 21n + 2 => O(n)

## 2.4 Modo Interativo

No modo interativo, o usuário que deve digitar a quantidade de cláusulas e combinações para Combinações = $2^n$ . Nesse modo, são inicializadas as cláusulas pelo próprio usuário, que vai digitar a variável de cada tupla, indo de 1 até N, e se ela é negada ou não negada que serão representados por 1 e 2, exemplo: (1<= V<=N, 1|2), Nesse modo teremos um menu principal que inicializa uma cláusula manualmente e imprime a equação.

É importante salientar que logo após a inicialização das cláusulas, será chamada a função "transforma true" que vai mostrar ao usuário todas as combinações que resolvem a equação.

## 2.5 Modo Automático

No modo automático o usuário digita apenas o valor de N, que vai ser usado também para definir o tamanho de cláusulas que será igual a C = (N/3)*2. Nesse modo, a Matriz é preenchidas automaticamente, para fazer esse preenchimento usamos a função “rand” que retorna o resto do valor definido após a função, ou seja, para achar valores para serem colocados nas posições da Matriz as chamada var e VF no algoritmo desenvolvido pelo grupo, fizemos:

```
for(i = 0; i<nclaus; i++){
          op = true;
          // gerando números de 0 a n
          ale1 = rand() % N;

          matriz[i][ale1] = 1 + rand() % 2;
          while (op == true){
              ale2 = rand() % N;
              if (ale1 != ale2){
                   matriz[i][ale2] = 1 + rand() % 2;
                   op = false;
              }
          }
          op = true;
          while (op == true){
            ale3 = rand() % N;
            if(ale2 != ale3 && ale1 != ale3){
              matriz[i][ale3] = 1 + rand() % 2;
              op = false;
            }
         }
```

Exemplo de como ficaria a clausula depois da matriz ter sido preenchida com valores aleatórios:

Se tivermos 2 cláusulas = {(1,1),(2,2),(3,2)}, {(1,2),(2,1),(3,2)} e n = 3.

| | | | | |
|---|---|---|---|---|
| 1 | 2 | 2 | 0 | 0 |
| 2 | 1 | 2 | 0 | 0 |

Teríamos essa matriz, vale lembrar que isso é uma simples explicação do que estamos fazendo, mas essa condição nunca existiria no algoritmo.
Logo após a matriz ser preenchida chamamos a função randclausula que passa os valores de 1 e 2 da matriz para uma estrutura de cláusulas para serem avaliadas as equações que resolvem a equação de cláusulas, isto na função transformatrue.

```
int k;
      for (int i = 0; i < nclaus; i++){
         k = 0;
         for (int j = 0; j < n; j++){
             if(matriz[i][j] == 1 || matriz[i][j] == 2){
                  clausula[i].tupla[k].var = j+1;
                  clausula[i].tupla[k].VF = matriz[i][j];
                  k++;
             }
         }
```

## 2.6 Tempo de execução

Para a medição do tempo de execução, foi utilizado a biblioteca time disponível na linguagem c (`#include <time.h>`), a variável (`clock_t t;`), irá armazenar o tempo. Na compilação do código foi utilizado o compilador GCC e a IDE Visual Studio Code.

Especificações do computador utilizado para a execução:
**Notebook:** Acer Aspire 5  A515-54-59X2;
**Processador:** Intel Core i5-10210U (10ª geração) – 6 MB de cache
4 núcleos e 8 *threads* – de 1.60 GHz até 4.20 GHz;
**Memória RAM:** 8 GB - DDR4 2400 MHz;
**SSD:** 512gb;
**Sistema Operacional:** Windows - 64 bits.

À medida em que fomos executando para cada caso de N, percebemos, a partir de N=30, o longo período de execução que seria exigido para concluir, pela grande demora para o seu término até então. Portanto, foi necessário o desenvolvimento de cálculos para obtermos uma estimativa do tempo total de execução. Sabia-se que a quantidade final de combinações possíveis a serem encontradas é $2^n$, e tínhamos o tempo de execução resultante em segundos do caso N = 15, 1,80, junto, com isso, a sua quantidade de combinações possíveis, 32.768. Logo, sabíamos também que N = 30 tinha 1.073.741.824 combinações ao todo. Dessa forma, foi aplicado uma simples regra de três com o tempo de execução e o número de combinações de N = 15 e a quantidade de combinações de N = 30 a fim de se encontrar o tempo de execução em segundos de N = 30, resultando em 58.982,4 segundos. O mesmo foi feito com N = 40 no lugar de N = 30 e, assim, para esse caso, foi encontrado o surpreendentemente longo valor de 60.397.977,6 segundos, o que ultrapassa um ano e meio de execução.

| Tempo | Milissegundos | Segundos | Minutos | Horas |
|---|---:|---:|---:|---:|
| **N=15** | **1800** | **1,80** | **0,03** | **0,0005** |
| **N=20** | **57440000** | **57,44** | **0,95733333** | **0,015955556** |
| **N=30** | **58982400** | **58982,4** | **983,04** | **16,384** |
| **N=40** | **60397977600** | **60397977,6** | **1006632,96** | **16777,216** |
| **N=45** | **193273528320** | **1932735283,2** | **32212254,72** | **536870,911** |

Todos esses valores de tempo encontrados são valores estipulados, como já dito fizemos uma regra de 3 para achar algo que seria aproximado para entradas grandes, nos baseamos nas médias para N = 15, para achar os valores de tempo, calculamos a média de 10 tempos de N = 15, e estipulamos esses valores para entradas maiores, nesse cálculo respondendo a pergunta para N = 45 o algoritmos iria rodar mais de 60 anos, ou seja, nada bom, pois o processo seria muito demorado. Fizemos os cálculos da seguinte forma:

Procuramos uma média de tempos para N = 15 - que geralmente dava entre 1,70s a 1,90s, nessa média encontramos 1,80s:

Então foi feita a seguinte regra de 3:

N = 15 - Quantidade de combinações: 32768

N = 45 - Quantidade de combinações: 3,518437E13

32768 ----- 1,8s          32768x = 3,518437E13/1,8

3,518437E13 ---x          x = 1932735283,2s, ou seja, 61 anos.

Então para N = 45, o usuário demoraria 61 anos para obter as várias respostas, nesse caso, para essa entrada, não seria bom esse algoritmo.

# 3 Conclusão

Portanto, conclui-se que, nesse trabalho prático o algoritmo que foi criado para resolver as equações do problema SAT se mostrou muito eficiente para entradas menores que n = 22, e isto foi perceptível com a biblioteca time.h de c, já que a média de tempo para n = 22 foi de 90s , isto para uma equação aleatória, então concluímos que para entradas menores que 22 o algoritmo mostra ótimos resultados, já para entradas maiores que 22 ele já começa a crescer exponencialmente em relação ao seu tempo.

Também percebemos com cálculos de aproximação de entradas aleatórias, que para entradas maiores que n = 30, o algoritmo demoraria cerca de 12 horas para compilar, ou seja, nada bom. Outrossim, são entradas maiores que n = 40, já que com uma simples regras de 3 estipulamos que demoraria mais de 1 ano e meio para o algoritmo gerar todas as combinações possíveis. Nesse sentido, o grupo percebeu como é ruim se ter uma algoritmo que não é útil para entradas grandes e que como é importante se fazer o cálculo de tempo para se perceber onde o algoritmo começaria a falhar, já que para entradas muito grandes essa implementação de combinação não seria nada boa. Então, é notório que para algoritmos de alta complexidade, que iram ter entradas muito grandes ou que se tenha que fazer muitos cálculos computacionais é bom achar tanto a função de complexidade do algoritmo, como também analisar o seu custo computacional e o seu tempo de execução, para assim corroborar a eficiência do código.