CCF 211 - Algoritmos e Estruturas de Dados I Trabalho Prático 02 - SAT

Vinícius Mendes - 3881, João Roberto - 3883, Artur Papa - 3886

19 de abril de 2021

1 Introdução

O trabalho em questão tem como objetivo abordar os conteúdos trabalhados na disciplina de Algoritmos e Estrutura de Dados I, em específico, como visto nas aulas teóricas e práticas, a avaliação do impacto causado pelo desempenho dos algoritmos em sua execução real. A priori, é válido dizer que o projeto de algoritmos é fortemente influenciado pelo estudo de seus comportamentos, depois que um problema é analisado e decisões de projeto são finalizadas, o algoritmo tem de ser implementado em um computador.

Neste momento, o projetista deve estudar as várias opções de algoritmos a serem utilizados, em que os aspectos de tempo de execução e espaço ocupado são considerações importantes. Muitos desses algoritmos são encontrados em áreas tais como pesquisa operacional, otimização, teoria dos grafos, estatística, probabilidades, entre outras. [2]

2 Desenvolvimento

2.1 Principais Escolhas

Ao observarmos a proposta do trabalho, traçamos algumas estratégias importantes que determinaram o segmento do trabalho prático. Inicialmente, como forma de aprendizado, percebemos que trabalhar na parte que julgássemos mais difíceis primeiro tornaria o desenvolvimento do projeto mais fluido, visto que teríamos mais tempo para tratar dos problemas que poderiam surgir.

Dessa forma, iniciamos pelo modo automático, buscando solucionar o problema inicial utilizando diversos códigos de gerações de combinações até encontrarmos o que melhor nos adaptamos.

2.2 Dificuldades

Antes de tudo, um dos principais obstáculos que enfrentamos foi o entendimento da proposta, visto que demandou maturidade maior ao ler e entender o que foi realmente pedido. Subsequentemente, nos deparamos com uma grande quantidade de códigos que dizem ser sobre combinações, entretanto, tratam-se de permutações. O entendimento de cada código nos demandou bastante tempo, pois verificar cada um tornou-se uma obrigação, dado que, necessitaríamos do algorítmo correto.

Adiante, encontramos outra adversidade no momento de realizar as comaparações e verificações, pois no começo julgamos ser necessário a exibição apenas das colunas específicas que satisfaziam as cláusulas, o que foi corrigido no decorrer do desenvolvimento. Por fim, a exibição

dos resultados esperados no projeto gerou certos impasses, que foram solucionados ao entender melhor a proposta do TP.

2.3 Configurações de hardware e software

Outrossim, antes de falarmos da execução do programa, é necessário dizer sobre as máquinas em que foram testadas os códigos, tendo em vista que uma utiliza o sistema operacional Linux e a outra Windows, a seguir as especificações:

1. Especificação do Computador A
   (a) Intel(R) Core(TM) i5-8250U CPU @1.80 GHz
   (b) Sistema Operacional: Linux Ubuntu 18.04 (64-bit)
   (c) Chipset: Lenovo Core7U (Kabylake-U) ULV Host Bridge/DRAM Registers
   (d) Placa-mãe: LENOVO LNVNB161216
   (e) Memória Primária (RAM): 7.91 GB 665 MHz DDR4 SDRAM
2. Especificação do Computador B
   (a) Intel(R) Core(TM) i3-7100 CPU @ 3.90GHz
   (b) Sistema Operacional: Windows 10 PRO (64-bit)
   (c) Chipset: Intel H110 Kaby Lake
   (d) Placa-mãe: ASUSTeK COMPUTER INC. H110M-CS/BR
   (e) Memória Primária (RAM): 8.192 GB 1200 MHz DDR4 SDRAM

2.4 Complexidade assintótica

Antes de falar sobre as entradas, é de suma importância citar sobre a função de complexidade para gerar as cláusulas no modo automático. Ademais, analisando para o pior caso, veremos que a função possui 2 loops logo no início, sendo esses as duas repetições utilizadas para preencher a matriz com 0's, o que resulta na função $g(N) = N^2$. Terminado este processo, temos mais um loop, que será responsável para criar e gerar as cláusulas, nota-se que neste loop o índice está em função de C, analisando a fórmula passada na documentação do trabalho temos que $C = N/3 * 2$.

Além disso, vale notar que ainda temos duas repetições dentro deste ***for***, que, considerando como caso menos notável será executado N^2 vezes, assim, a equação geral da função será:

$$g(N) = 2 * N^3/3 + N^2$$

Posto isso, a seguir temos uma imagem do gráfico da função, é válido observar que o gráfico a seguir foi gerado em python, utilizando as bibliotecas ***matplotlib.pyplot*** e ***numpy***.

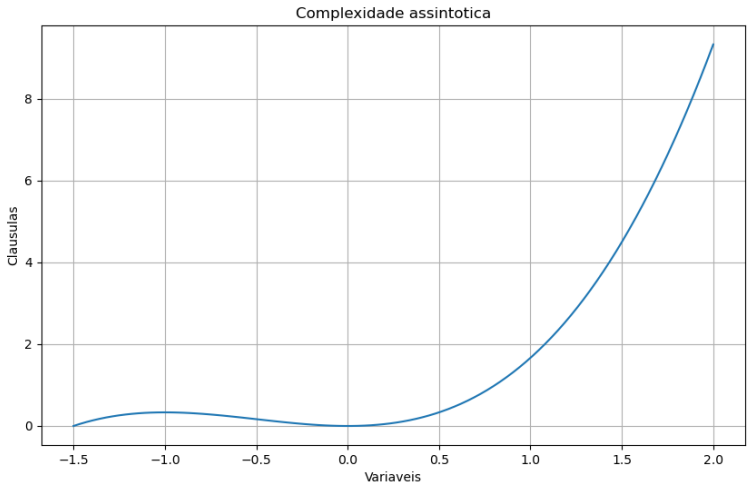


Figura 1: Complexidade assintótica

2.5 Diversidade de tempos de execução em diferentes sistemas operacionais

Ao realizarmos os testes propostos identificamos uma diferença gritante entre o tempo de execução do código em máquinas que utilizam Linux e Windows. Os testes realizados em Linux obtiveram melhores tempos comparados ao do outro sistema operacional, com uma diferença de mais de 200 vezes.

Buscamos uma resposta para o porquê isso acontece, entretanto não obtivemos um retorno adequado.

3 Opções de Entrada

3.1 Entrada Manual

Diante das decisões já descritas, o modo manual foi o último a desenvolvermos, visto que já possuíamos em mãos todas as funções necessárias para torná-lo possivel.

```
int C, N;
printf("Insira o Numero C de clausulas: ");
scanf("%d", &C);
printf("Insira o Numero N de variaveis: ");
scanf("%d", &N);

TipoLista lista2;
TItem registro2;
FLVazia(&lista2);

printf("C : %d\n", C);
printf("N : %d\n", N);
int variavel1, variavel2, variavel3;
int estado1, estado2, estado3;
int aux1, aux2, aux3, aux4, aux5, aux6;

LOOP:
for (int i = 0; i < C; i++)
{
    printf("Digite a clausula: a,b c,d e,f Sendo, a/c/e numero da variavel, e b/d/f o estado dela.\n");
    scanf("%d,%d %d,%d %d,%d", &variavel1, &estado1, &variavel2, &estado2, &variavel3, &estado3);
    aux1 = variavel1;
    aux2 = estado1;
    aux3 = variavel2;
    aux4 = estado2;
    aux5 = variavel3;
    aux6 = estado3;
```

Figura 2: Função Manual

Recebemos o número de cláusulas (C) do usuário e o número de variáveis (N) com o intuito de passar esses dados como parâmetro para a função ***Imprime***, que, por sua vez é responsável pelo desencadear dos processos necessários. Posteriormente a essa insersão de dados, pedimos novamente ao usuário que informe como cada cláusula solicitada deve ser, então verificamos se não há algum erro nesse passo, como por exemplo a inserção de uma coluna inexistente, um número inválido ou até uma repetição de coluna.

3.2 Entrada Automática

Diferentemente da entrada manual, para o modo automático as cláusulas precisam ser geradas de forma aleatória, sendo que elas seriam criadas a partir de uma matriz CxN. A priori, a matriz seria preenchida com 0's, e depois disso seriam escolhidas três colunas aleatórias em cada linha com o valor de 1 ou 2 - negado ou não negado respectivamente- e ao final do processo teríamos o valor de todas as cláusulas.

Após a criação das clásulas, ocorre uma chamada de funções contínua que gerarão e retornarão todos os valores booleanos que satisfazem cada uma.

Uma das principais funções se chama ***TruthTable***:

```
int truthTable(int n, int leng, TItem *pItem)
{
    long long int coluna[40];
    int pos = 0;
    int colum = 0;
    int i, j = 0, k = 0, z = 0;
    long long int bit = 1 << leng - 1;
    while (bit)
    {
        coluna[j] = n & bit ? 1 : 0;
        j++;
        bit >>= 1;
    }
```

Figura 3: Função TruthTable

Esta é capaz de gerar, bit a bit, uma tabela verdade na qual será realizada as comparações para obtenção do resultado satisfatório para a claúsula. Ao observarmos que a função retorna uma linha da tabela verdade para cada vez que é chamada (baseado no ***n*** passado como parâmetro), optamos por salvá-la em um vetor que nos facilitaria na hora de verificar a satisfatibilidade da cláusula. Vale fazer uma observação de que o código foi encontrado no site do Stack Overflow.[1]

```
#include <stdio.h>
#define LENGTH 3
void print_binary(int n)
{
        int bit = 1<<LENGTH - 1;
        while ( bit ) {
        printf("%d", n & bit ? 1 : 0);
        bit >>= 1;
        }
        printf("\n");
}
int main(){
    int n = 1<<LENGTH, i;
    for(i=0;i<n;i++)
        print_binary(i);
}
```

Figura 4: Função TruthTable original

3.3 Resultados obtidos

Dito isso, será mostrado os resultados para as devidas entradas - 15, 20, 30 e 40 - juntamente com o tempo em que elas demoraram pra executar.

3.3.1 Entrada de 15

Para o primeiro teste pode-se notar que a execução foi bem rápida, apesar da diferença de tempo entre Sistemas Operacionais:

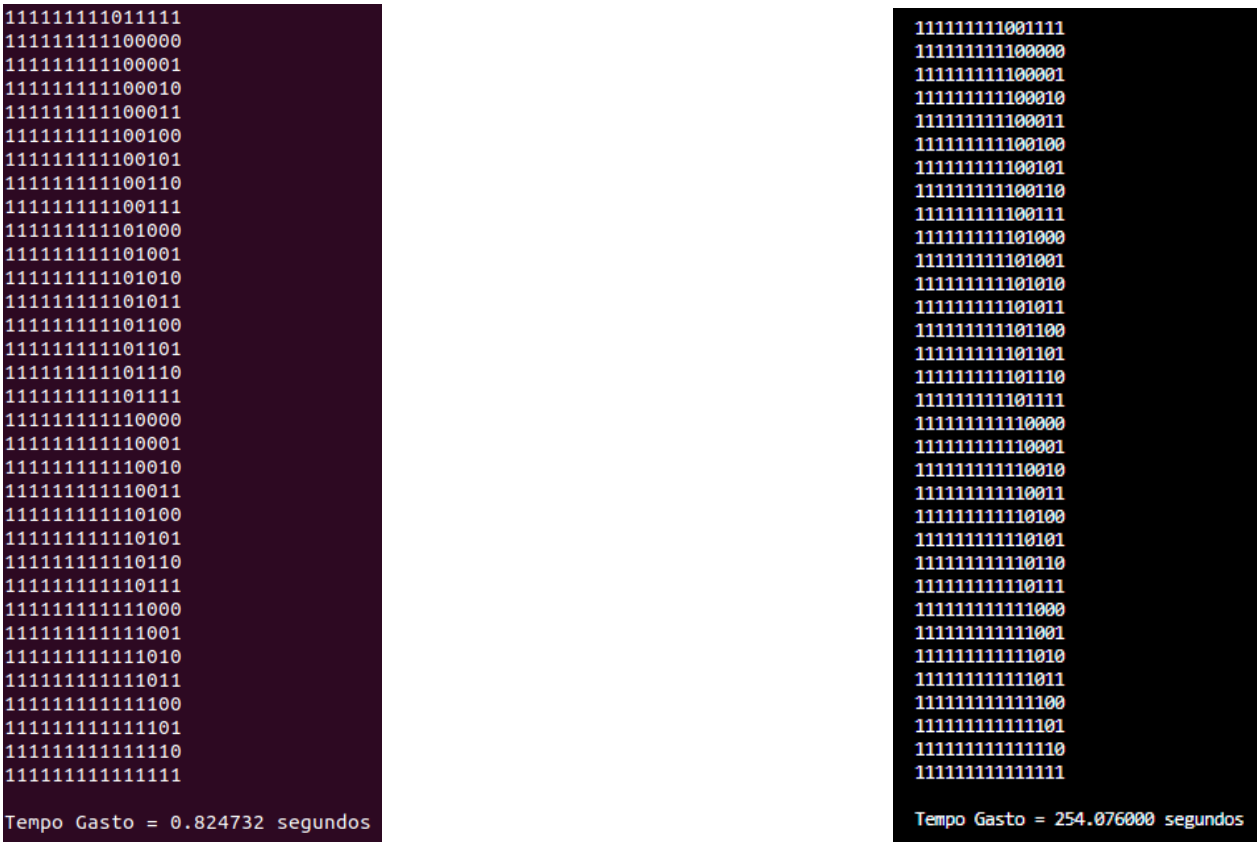
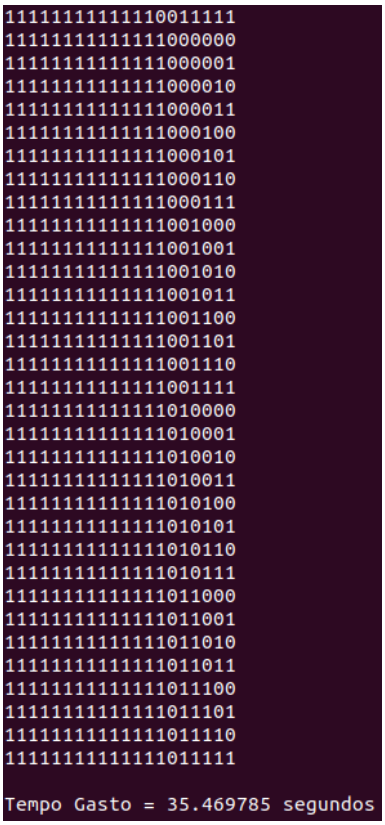


Figura 5: Entradas de 15

3.3.2 Entrada de 20

Para o segundo teste, já houve uma diferença maior de tempo, sendo que, quando comparado ao Linux, o Windows obteve um resultado 373 vezes mais lento, aproximadamente. Ademais, observa-se que para o Windows, a execução foi feita apenas para a uma tabela verdade, como sabemos que será gerada uma tabela verdade para cada cláusula, neste caso, basta multiplicarmos o resultado por 12, e, assim encontramos o tempo de aproximadamente 13069 segundos:



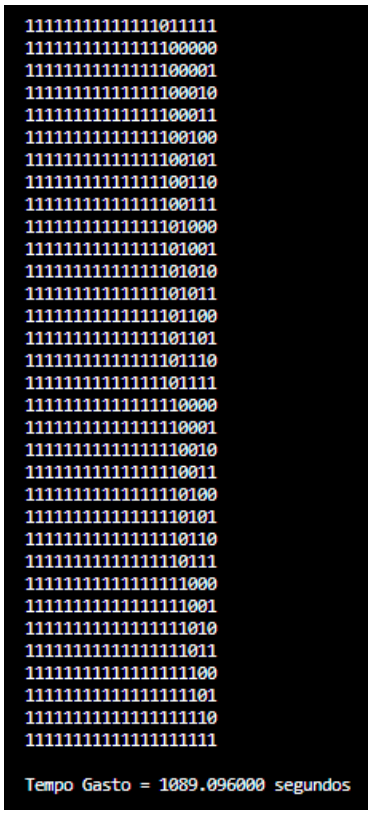


Figura 6: Entradas de 20

3.3.3 Entrada de 30

Para as entradas de 30, seguimos o mesmo raciocínio que na entrada anterior, ou seja, executamos a tabela verdade uma vez e multiplicamos o tempo de execução por 20, para o Linux, obteríamos um tempo de execução de aproximadamente 27 horas para todas as cláusulas, já para o Windows não obtivemos um resultado satisfatório, haja visto que o programa ficou executando durante 12 horas e ainda assim não estava perto do resultado:

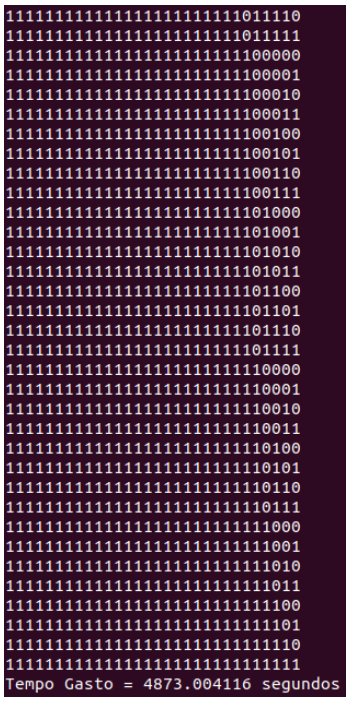


Figura 7: Entradas de 30

3.3.4 Entrada de 40

Para o último teste, não obtivemos resultados em ambos os SO's, visto que demoraria um tempo imensurável para finalizar a execução, o que seria impossível de ser verificado, visto que a função para a tabela verdade tem complexidade $O(2^n)$ o programa seria executado $1,1 * 10^{12}$ vezes, o que daria aproximadamente cerca de 5192 horas, tendo em conta que o algoritmo seria executado uma vez a cada 0,000017 segundos, multiplicando esse valor pelo número de cláusulas - 26 nesse caso - demoraria aproximadamente 134995 horas para o algoritmo ser finalizado.

3.3.5 Entrada para 45 ou mais

Logo após serem explicadas as entradas, é necessário dizer para casos em que teremos 45 variáveis ou mais, tendo como base o cálculo feito para a entrada de 40, observa-se que para gerar a tabela verdade de 2^{45} apenas uma vez, demoraríamos aproximadamente 18 anos, pegando esse resultado e multiplicando pelo número de cláusulas - 30 nesse caso - obteríamos o resultado após 596 anos aproximadamente, ou seja, não é razoável executarmo o código para valores de N maiores que 45.

4 Conclusão

Posto isso, conclui-se que o trabalho em questão foi desenvolvido conforme o esperado, atingindo todas as especificações requeridas na descrição do mesmo em implementar uma provável solução para o Problema da Satisfabilidade(SAT) 3-FNC-SAT.

Por conseguinte podemos ressaltar que o livro-texto da disciplina [2] foi de suma importância para o desenvolvimento do projeto, haja vista que as explicações dadas pelo autor nos ajudaram bastante a entender as etapas a serem executadas e na construção dos códigos. Além disso podemos ressaltar que vídeos como os disponibilizados na disciplina de Algoritmos e Estrutura de Dados I, foram de grande ajuda para o entendimento dos conceitos necessário para o desenvolvimento do projeto .

Em adição, é válido dizer que apesar das dificuldades na implementação do código, o trio foi capaz de superar e corrigir quaisquer erros no desenvolvimento do algoritmo. Por fim, verificou-se a assertiva para o objetivo do projeto em implementar uma possível solução para o 3-FNC-SAT.

Referências

[1] combinacao:https://stackoverflow.com/questions/14632555/algorithm-to-generate-all-possible-combinations-of-0s-1s-for-any-length-of-di/14635496.

[2] Nivio Ziviani. *Projeto de Algoritmos com implementações em Pascal e C*, volume 3. 2010.