Universidade Federal de Viçosa - Campus Florestal

Disciplina: Algoritmos e Estruturas de Dados I
Professor(a): Thais Regina de Moura Braga Silva

Trabalho Prático 02

Problema da Satisfabilidade (SAT)

Icaro Gabriel dos Santos Moreira - 03856
Samuel Aparecido Delfino Rodrigues - 03476
Paula Teresa Mota Gibrim - 04234

Florestal – MG

2021

Sumário

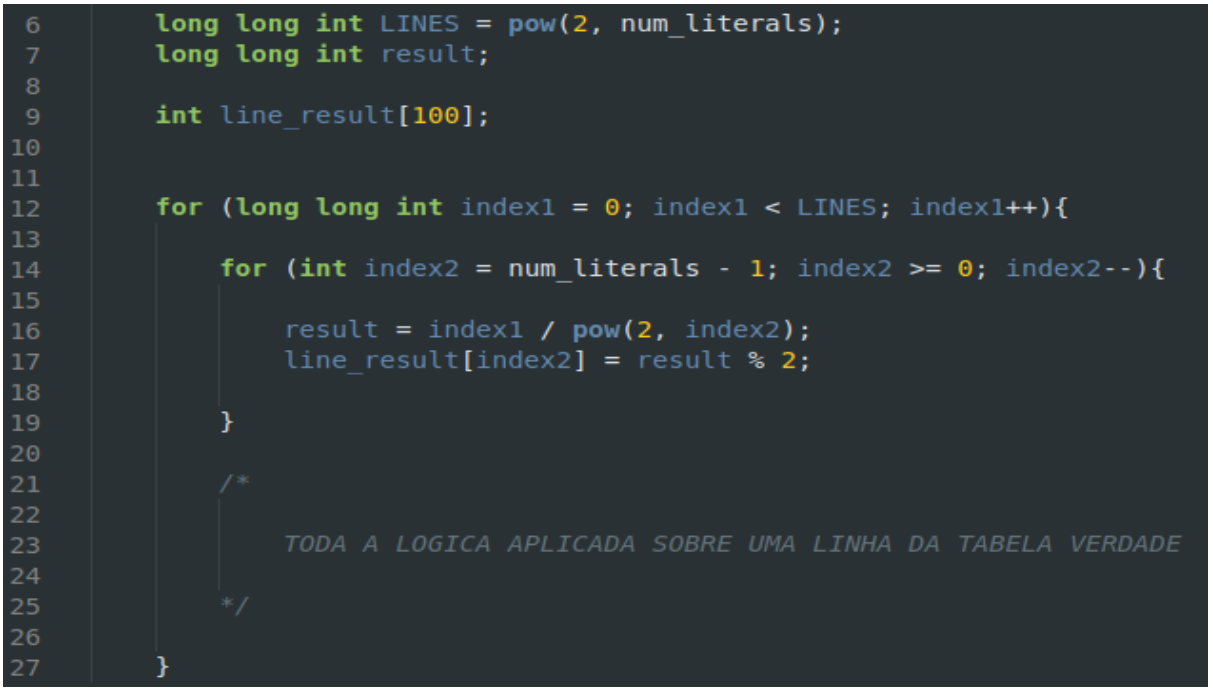
1. Introdução

O trabalho se baseou na implementação do Problema da Satisfabilidade (SAT), mais especificamente o problema “3-FNC-SAT”, onde dado uma entrada em equação booleana, o objetivo é encontrar uma configuração de valores lógicos para as variáveis envolvidas na equação de entrada, tal que a mesma seja avaliada como verdadeira, isto é, que ela seja satisfeita.

2. Decisões de projeto

A principal decisão do projeto foi desenvolver do 0 nosso próprio algoritmo que gera a tabela verdade, pois assim pudemos fazer de tal forma que o algoritmo atenda todas as nossas demandas sem necessidade de adaptação de um código de terceiros.

Outra decisão de extrema importância foi ao gerar a tabela verdade, não armazená-la na memória pois do contrário tornaria inviável em alguns casos pois excederia o limite de memória do computador, então decidimos gerar a tabela verdade linha a linha e já realizando os cálculos, assim na próxima iteração do laço de repetição que gera a próxima linha, a anterior é sobrescrevida.



```
 6      long long int LINES = pow(2, num_literals);
 7      long long int result;
 8
 9      int line_result[100];
10
11
12      for (long long int index1 = 0; index1 < LINES; index1++){
13
14          for (int index2 = num_literals - 1; index2 >= 0; index2--){
15
16              result = index1 / pow(2, index2);
17              line_result[index2] = result % 2;
18
19          }
20
21          /*
22
23              TODA A LOGICA APLICADA SOBRE UMA LINHA DA TABELA VERDADE
24
25          */
26
27      }
```

Figura 2.1 Algoritmo que gera a tabela

A repetição interna (linhas 14 a 19) é responsável por gerar uma linha inteira da tabela verdade, já fora do laço de repetição interno e dentro do externo (linhas 20 a 23) aplica toda a lógica sobre uma linha.

Essa imagem não reflete o algoritmo por si só, tratando apenas de um exemplo como esse trecho do código se comporta particularmente.

É possível notar na linha 9 que o vetor é alocado estaticamente, essa decisão foi tomada devido a um bug que não permitia que fosse alocado dinamicamente um vetor com um número de posições equivalentes ao número de literais da tabela verdade

3. Implementação

Foram criadas duas structs sendo elas “element_tuple” e “clause_tuple”, sendo que a struct “clause_tuple” armazena 3 structs “element_tuple” que armazenam o valor da tupla, possuindo os campos “num_literal” e “state” que armazenam o índice do literal e o estado(negado ou não) respectivamente.

```
6   typedef struct element_tuple element;
7   typedef struct clause_tuple clause;
8
9   struct element_tuple {
10
11        int num_literal;
12        int state;
13
14  };
15
16  struct clause_tuple{
17
18        element tuples[3];
19
20  };
```

Figura 3.1. Código da struct

Para a opção automatizada, a professora especificou como deveria ser feito o procedimento, ficando a cargo do grupo decidir a melhor forma de realizá-la. Utilizamos a função “rand()” que gera números aleatórios de 0 a N para randomizar os quais colunas da matriz seriam preenchidas, foi utilizada também a função “srand()” responsável por semear a função “rand()”. Foi passado o argumento “time(NULL)” para que os valores pseudo aleatórios não se repetissem.

```
237     srand(time(NULL));
238
239     int random;
240
241     for (int index1 = 0; index1 < num_clauses; index1++){
242
243         for (int index2 = 0; index2 < 3; index2++){
244
245             random = rand() % (num_literals - 1);
246
247             while (matrix[index1][random] != 0){
248
249                 random = rand() % (num_literals - 1);
250
251             }
252
253             if (matrix[index1][random] == 0){
254
255                 matrix[index1][random] = 1 + rand() % 2;
256
257             }
258         }
259     }
```

Figura 3.2 Código para gerar números aleatórios

Foram criadas duas funções responsáveis pela leitura dos valores da equação booleana, uma interativa e outra automática.

Essa função é responsável pela leitura das cláusulas no formato automático, alocando dinamicamente uma cláusula e ao executar os laços de repetição essas cláusulas são preenchidas com os valores recebidos da matriz recebida no parâmetro **matrix, matriz essa que foi especificada pela professora como deveria ser gerada.

```
38  clause *read_clauses_automatic(int num_clauses, int num_literals, int **matrix){
39
40      clause *clauses = (clause*) malloc(num_clauses * sizeof(clause));
41
42      // LÊ CADA CLAUSULA
43      for (int index1 = 0; index1 < num_clauses; index1++){
44
45          // LE CADA ELEMENTO DA CLAUSULA
46          for (int index2 = 0; index2 < num_literals; index2++){
47
48              //printf("%d" ,matrix[index1][index2]);
49
50              if (matrix[index1][index2] != 0){
51
52                  clauses[index1].tuples[index2].num_literal = index2;
53                  clauses[index1].tuples[index2].state = matrix[index1][index2];
54
55              }
56          }
57      }
58
59      return clauses; //RETORNA VETOR DE CLAUSULAS
60
61  }
```

Figura 3.3 Código para leitura das cláusulas automático

Por sua vez a função abaixo é responsável por receber os valores das cláusulas no modo interativo, sendo delegado ao usuário passar índice a literal desejada e o seu estado, a cada iteração do for interno é lido todas as 3 tuplas com o índice do literal e o seu estado, já o for externo é responsável por cada cláusula.

```
9   clause *read_clauses(int num_clauses, int num_literals){
10
11      clause *clauses = (clause*) malloc(num_clauses * sizeof(clause));
12
13      // LÊ CADA CLAUSULA
14      for (int i = 0; i < num_clauses; i++){
15
16          // LE CADA ELEMENTO DA CLAUSULA
17          printf(":::::::::::::::::: CLAUSULA %d ::::::::::::::::::\n", i + 1);
18
19          for (int j = 0; j < CONST_ELEMENTS_CLAUSE; j++){
20
21              printf("Digite o literal (de 0 a %d): ", num_literals - 1);
22
23              scanf("%d", &clauses[i].tuples[j].num_literal);
24
25              printf("Digite o estado (1 = negado | 2 = normal): ");
26
27              scanf("%d", &clauses[i].tuples[j].state);
28          }
29
30          printf("::::::::::::::::::::::::::::::::::::::::::::::::\n");
31
32      }
33
34      return clauses; //RETORNA VETOR DE CLAUSULAS
35
36  }
```

Figura 3.4 Código para leitura das cláusulas interativo

Para calcular as opções que satisfazem a equação, foi feito dois laços de repetição que percorre cada uma das 3 tuplas das N cláusulas realizando as comparações das colunas das linhas, sendo que cada linha é gerada e comparada por vez como foi mostrado na seção “decisões de projeto”, sem armazenamento na memória.

```
87          for (int j = 0; j < num_clauses; j++){
88
89              int clause_value = 0;
90
91              for (int k = 0; k < CONST_ELEMENTS_CLAUSE; k++){
92
93                  if (clauses[j].tuples[k].state == 1){
94
95                      clause_value = clause_value || !line_result[clauses[j].tuples[k].num_literal];
96
97                  }
98
99                  else {
100
101                     clause_value = clause_value || line_result[clauses[j].tuples[k].num_literal];
102
103                 }
104             }
105
106             if (!clause_value){
107
108                 break;
109
110             }
111
112             else {
113
114                 aux++;
115
116             }
```

Figura 3.5 Código para calculo que satisfaz a equação

4. Testes de execução

O testes foram feitos na máquina com as seguintes configurações:

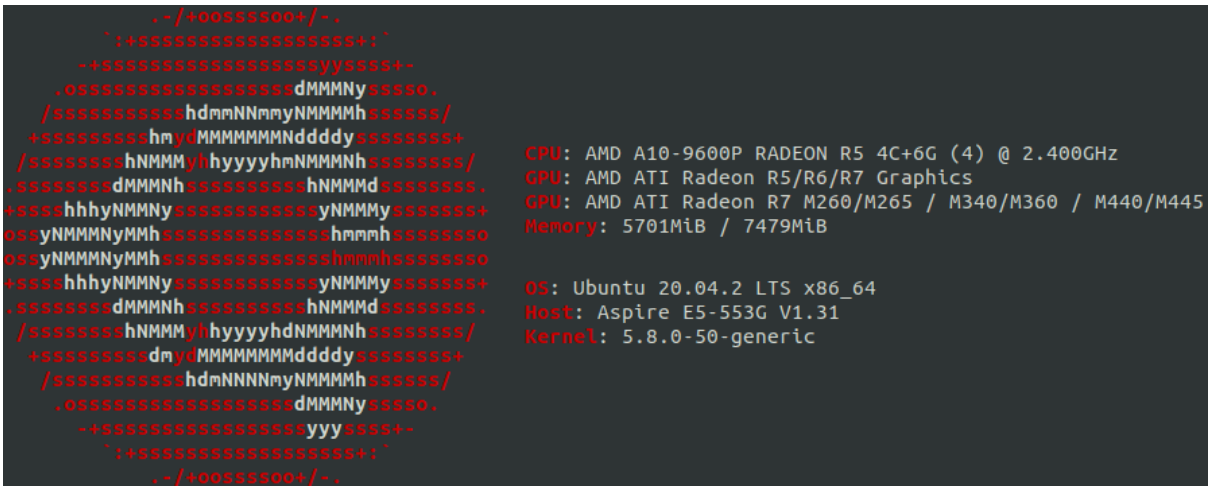


Figura 4.1 configurações do computador

Desempenho do algoritmo:

Para tal custo, cada entrada de valor N teve uma variação em tempo de relógio dado a arquitetura de software utilizada de 10 minutos a 40 minutos.

Para Tempo de execução do algoritmo em execução, segue o tempo em segundos dada cada entrada:

Entrada N = 15 - Tempo: 0s
Entrada N = 20 - Tempo: 7s
Entrada N = 30 - Tempo: 10800s
Entrada N = 45 - Tempo: 61200s

OBS: Os valores de tempo estão arredondados e a entrada de 15 obviamente não é 0 mas foi considerado pois comparado com entradas maiores o valor é desprezível.

No gráfico abaixo é nítido o aumento exponencial do tempo de processamento do algoritmo à medida que a entrada aumenta, tendo um comportamento assintótico $O(n^2)$.

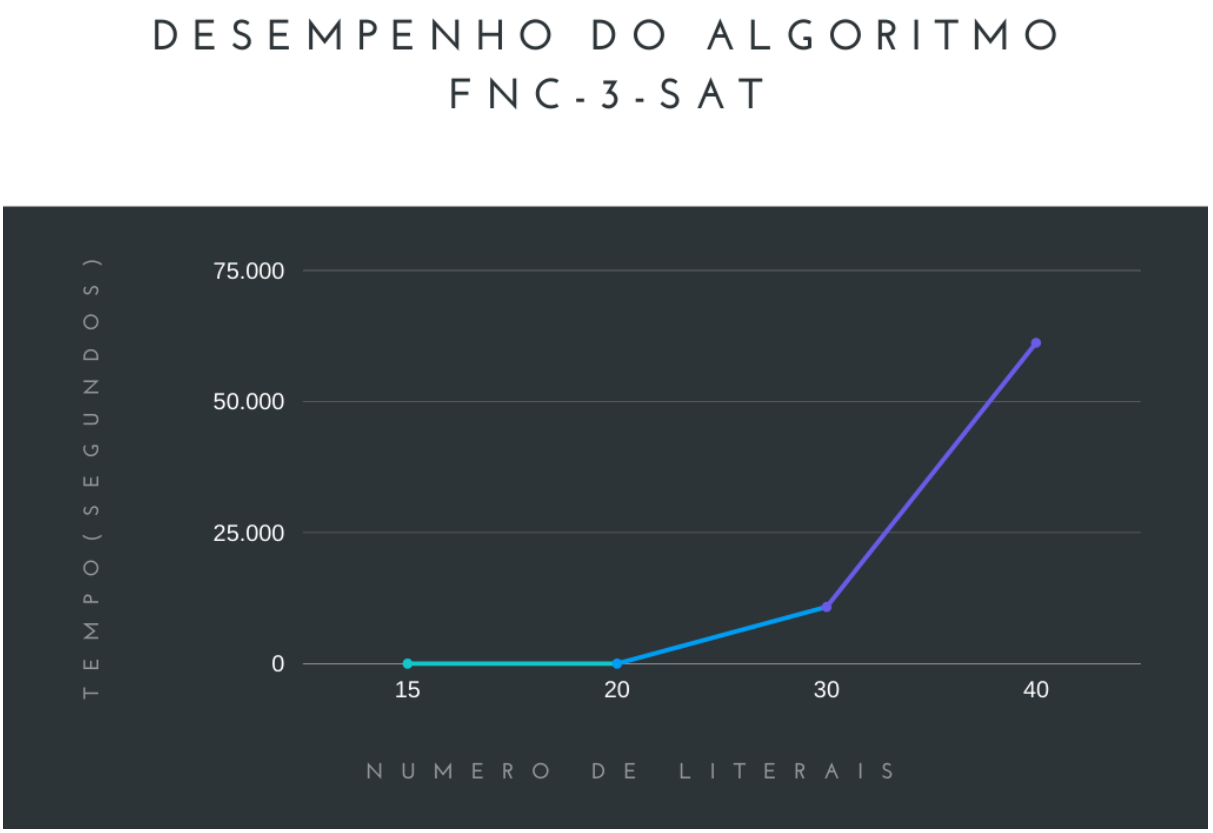


Figura 4.3. Comportamento do desempenho

Seria razoável executar o seu algoritmo para valores de N maiores do que 45? Não, pois o número de combinações seria muito elevado, dado que esse algoritmo é um algoritmo de complexidade quadrática, então cada vez que aumenta a entrada, o número de combinações aumenta.

5. Conclusão

Após a implementação do trabalho prático, notamos que uma determinada entrada N para o algoritmo 3-SAT, pode afetar no tempo de execução do mesmo, por isso o estudo e o cálculo do custo do algoritmo é importante para sempre implementar algoritmos rápidos e que tenham o seu custo ótimo .

6. Referências Bibliográficas

[1] Ziviani, N. (2010). *Projeto De Algoritmos: Com Implementações Em Pascal E C.* Cengage Learning.