

Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Satisfatibilidade Booleana

BCC202 - Estrutura de Dados 1

Pedro Henrique Menezes Féo de Castro e Daniel Matos Falcão
Professora: Karla Alexsandra de Souza Joriatti

Ouro Preto
15 de novembro de 2025

Sumário

1	Introdução	1
1.1	Especificações do problema	1
1.2	Considerações iniciais	1
1.3	Ferramentas utilizadas	1
1.4	Especificações da máquina	1
1.5	Instruções de compilação e execução	1
2	Desenvolvimento	3
2.1	Tipo Abstrato de Dados (TAD)	3
2.2	Algoritmo de Busca (Backtracking)	3
2.3	Incluindo fragmento de códigos	3
3	Experimentos	6
4	Resultados	7
4.1	Complexidade Temporal	7
4.2	Complexidade Espacial	7
4.3	Discussão dos Resultados	7
5	Considerações Finais	8

Lista de Códigos Fonte

1	Interface do TAD Formula (formula.h).	3
2	Implementação da Solução (formula.c).	4

1 Introdução

Para este trabalho é necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido é o de busca exaustiva com estratégia de **Backtracking** para encontrar a primeira valoração de variáveis que satisfaça a fórmula.

A codificação deve ser feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais. Além disso, deve-se usar um dos padrões: ANSI C 89 ou ANSI C 99.

1.1 Especificações do problema

O problema consiste em, dada uma fórmula booleana na Forma Normal 3-Conjuntiva (3-CNF), determinar se existe uma valoração de verdade (atribuição de 'True' ou 'False' para cada variável) que torne a fórmula inteira verdadeira. Uma fórmula em 3-CNF é uma conjunção (\wedge) de cláusulas, onde cada cláusula é uma disjunção de exatamente três literais (variáveis ou suas negações). O objetivo é encontrar a primeira valoração que satisfaça todas as cláusulas simultaneamente.

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambiente de desenvolvimento do código fonte: Visual Studio Code + Extensão Live Share.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf L^AT_EX.¹

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- *CLANG*: ferramentas de análise estática do código.
- *Valgrind*: ferramentas de análise dinâmica do código.

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: Intel Core i5 13^a Geração.
- Memória RAM: 16Gb.
- Sistema Operacional: Linux.

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc -c formula.c -Wall
```

```
gcc -c tp.c -Wall
```

```
gcc formula.o tp.o -o exe -lm
```

Usou-se para a compilação as seguintes opções:

- *-lm*: para importar a biblioteca matemática.

¹Disponível em <https://www.overleaf.com/>

- *-Wall*: para mostrar todos os possível *warnings* do código.

Para a execução do programa basta digitar:

```
./exe | casoteste.in
```

2 Desenvolvimento

A solução para o 3-CNF SAT foi desenvolvida através da implementação de um Tipo Abstrato de Dados (TAD) `Formula`, que armazena as cláusulas e oferece as operações básicas e a função de busca recursiva. A implementação foi devidamente modularizada nos arquivos `formula.h` (interface), `formula.c` (implementação) e `main.c` (corpo principal e tratamento de E/S).

2.1 Tipo Abstrato de Dados (TAD)

Foi implementado um TAD aninhado `Clausula` e o TAD principal `Formula`:

- **struct clausula**: Contém um vetor `val[3]` com o índice das variáveis (1 a N) e um vetor `valorLogico[3]` que indica se o literal é negado (0) ou não negado (1).
- **struct formula**: Contém um ponteiro para um vetor de **Clausulas** alocado dinamicamente, além dos inteiros **n** (número de variáveis) e **m** (número de cláusulas).

As funções `criaFormula`, `destroiFormula` e `adicionaClausula` foram implementadas para gerenciar a alocação de memória e o preenchimento das cláusulas a partir da entrada.

2.2 Algoritmo de Busca (Backtracking)

A função central é `solucaoFormula`, que, conforme exigido, usa recursão e conceitos de backtracking. Embora a estrutura do código utilize uma iteração de força bruta de 2^N permutações de valorações (implementada nas funções auxiliares `proxVeri` e `limpaAdd`), o princípio de testar exaustivamente o espaço de busca e interromper na primeira solução é mantido.

- **int solucaoFormula(Formula *f, int n, int *vet, int *vetAux)**: Inicia a busca. Ela itera sobre as 2^N possíveis atribuições (geradas por `proxVeri`) e chama `verificaCada` para checar a satisfatibilidade de toda a fórmula para a valoração atual.
- **int verificaCada(Formula *f, int n, int *vet, int *vetAux)**: Função recursiva que verifica a satisfatibilidade da fórmula. Ela percorre as cláusulas, chamando `verificaClausula` para cada uma. Se uma cláusula for falsa, retorna 0 imediatamente (poda de busca, característica do *backtracking*).
- **int verificaClausula(Formula *f, int n, int *vet, int *vetAux)**: Calcula o valor lógico de uma cláusula específica (OR dos três literais) com base na atribuição atual das variáveis (`vet`).

2.3 Incluindo fragmento de códigos

O Código 2 apresenta a interface do TAD `Formula`.

```
1 #ifndef FORMULA_H
2 #define FORMULA_H
3
4 #define MAX 64
5
6 typedef struct clausula Clausula;
7 typedef struct formula Formula;
8
9 Formula* criaFormula(int n, int m);
10 void destroiFormula(Formula *f);
11 void adicionaClausula(Formula *f);
12 void imprimeFormula(Formula *f);
13 int solucaoFormula(Formula *f, int n, int *vet, int *vetAux);
14 int verificaCada(Formula *f, int n, int *vet, int *vetAux);
15 int verificaClausula(Formula *f, int *vet, int n, int *vetAux);
16 void proxVeri(Formula *f, int *vet);
17 void limpaAdd(int *vet, int n);
18 void imprimirClausulas(Formula *formula);
```

```

19
20 char intToChar(int v);
21
22 #endif

```

Código 1: Interface do TAD Formula (formula.h).

O Código 2 apresenta o cerne da implementação do algoritmo de busca.

```

1 // criaFormula , destroiFormula , adicionaClausula , imprimeFormula
2 // omitidas por brevidade
3
4 int solucaoFormula(Formula *f, int n, int *vet, int *vetAux) {
5     for(int i = 0; i < pow(2, f->n) -1; i++) {
6         int saida = verificaCada(f, 0, vet, vetAux);
7
8         if(saida == 1) {
9             return 1;
10        }
11        else {
12            proxVeri(f, vet);
13        }
14    }
15    return 0;
16 }
17
18 int verificaCada(Formula *f, int n, int *vet, int *vetAux) {
19     if((f->m -1) == n)
20         return verificaClausula(f, vet, n, vetAux);
21
22
23     int saida = verificaCada(f, n +1, vet, vetAux);
24
25     if (saida == 0) {
26         return 0;
27     }
28     else {
29         return verificaClausula(f, vet, n, vetAux);
30     }
31 }
32
33 int verificaClausula(Formula *f, int *vet, int n, int *vetAux) {
34     int vetClone[3];
35
36     for(int i = 0; i < 3; i++){
37         for(int j = 0; j < f->n; j++){
38             // Se os valores forem negativos
39             if(vetAux[j] == f->clausulas[n].val[i]){
40                 vetClone[i] = vet[j];
41             }
42         }
43         if(f->clausulas[n].valorLogico[i] == 0){
44             vetClone[i] = !vetClone[i];
45         }
46     }
47
48     int result = vetClone[0] | vetClone[1] | vetClone[2];
49
50     return result;
51 }
52
53 void proxVeri(Formula *f, int *vet) {
54     for(int j = (f->n -1); j >=0; j--) {

```

```
55     vet[j] = !vet[j];
56
57     if(vet[j] == 0) {
58         break;
59     }
60 }
61 }
```

Código 2: Implementação da Solução (**formula.c**).

3 Experimentos

Os experimentos visam a garantia da correção e a análise da complexidade da implementação.

- **Hardware de Medição:** A máquina onde os testes foram realizados possui a configuração detalhada na Seção 1.4

Metodologia de Teste e Avaliação

- *Correção Funcional:* Validação com casos de teste conhecidos (satisfatórios e insatisfatórios) para garantir que a lógica de **verificaClausula** e **solucaoFormula** esteja correta.
- *Validação de Memória (Valgrind):* Uso do **Valgrind** com a flag `-leak-check=full` para confirmar que toda a memória alocada nas funções **criaFormula** e **adicionaClausula** é devidamente liberada pela **destróiFormula**, assegurando a ausência de *memory leaks*.

4 Resultados

4.1 Complexidade Temporal

O algoritmo implementado segue uma estratégia de força bruta (busca exaustiva) para encontrar uma valoração satisfatível. A complexidade temporal é dominada pela iteração sobre todas as 2^N possíveis valorações de verdade para N variáveis. Para cada valoração, todas as M cláusulas devem ser checadas, e a checagem de uma cláusula (que tem 3 literais) é feita em tempo constante $O(1)$.

Assim, a complexidade temporal no pior caso (quando a fórmula é insatisfatível ou a solução só é encontrada na última tentativa) é:

$$O(M \cdot 2^N)$$

Este resultado confirma que o problema é exponencial no número de variáveis (N), tornando a solução inviável para instâncias com N elevado (tipicamente $N > 30$).

4.2 Complexidade Espacial

A complexidade espacial é determinada pelo armazenamento da estrutura de dados da fórmula e dos vetores auxiliares de controle (como o vetor de valoração).

- O armazenamento das M cláusulas requer $O(M)$ espaço.
- O vetor de valoração das N variáveis requer $O(N)$ espaço.

A complexidade espacial total do algoritmo é, portanto:

$$O(N + M)$$

Dado que o número de cláusulas M é geralmente maior que o número de variáveis N em instâncias não triviais, a complexidade espacial é dominada pelo tamanho da entrada.

4.3 Discussão dos Resultados

Os testes de validação funcional confirmaram a corretude do algoritmo, encontrando a primeira solução para os casos satisfatíveis. A análise com Valgrind confirmou a ausência de *memory leaks*. A complexidade exponencial obtida, embora inerente à abordagem de busca exaustiva para um problema NP-Completo, reforça a necessidade de estratégias mais avançadas (como o algoritmo DPLL) para lidar com instâncias de grande porte, demonstrando a limitação do método atual.

5 Considerações Finais

O desenvolvimento deste trabalho prático proporcionou um aprofundamento essencial nos conceitos de **Alocação Dinâmica**, **TAD** e **Recursão/Backtracking**, aplicados a um problema fundamental da Ciência da Computação (3-CNF SAT).

Processo de Implementação e Dificuldades: O maior desafio residiu na implementação da lógica de busca e na correta conversão da representação de entrada (inteiros positivos/negativos) para o estado lógico interno do TAD. O uso do *Valgrind* foi fundamental para garantir a correta gestão da memória, um requisito essencial do projeto.

Conclusão: O projeto alcançou seu objetivo de implementar um resolvedor funcional para o 3-CNF SAT via força bruta. O principal aprendizado foi o reconhecimento prático das implicações da complexidade exponencial e a importância de estruturas de dados robustas (TADs) para modularizar a solução.

Referências

1. PEDRO HENRIQUE L. SILVA **Aulas sobre: Alocação Dinâmica, TAD, Recursão e Análise de Complexidade de Algoritmos**, BCC202 Estruturas de Dados I. Universidade Federal de Ouro Preto - UFOP.