Universidade Federal de Ouro Preto - UFOP
Departamento de Computação

BCC202 - Estruturas de Dados I

Trabalho Prático I: Solução de 3-CNF SAT

Aluno: [Seu Nome Completo]
Aluno: [Nome da Dupla, se houver]

28 de janeiro de 2026

1 Implementação

O objetivo deste trabalho foi desenvolver um solucionador para o problema da Satisfatibilidade Booleana na forma 3-CNF, utilizando a linguagem C, Tipos Abstratos de Dados (TAD) e recursão com *backtracking*.

1.1 Estruturas de Dados e Modularização

Conforme exigido, o código foi modularizado em três arquivos: `formula.h` (interface), `formula.c` (implementação) e `tp.c` (cliente)[cite: 81].

Para representar o problema, foram criadas as seguintes estruturas:

- **Literal:** Armazena o ID da variável e uma *flag* indicando se ela é negada.
- **Clausula:** Um vetor estático de 3 Literais.
- **Formula:** O TAD principal, contendo o número de variáveis (N), o número de cláusulas (M) e ponteiros para a alocação dinâmica das cláusulas e do vetor de valoração.

1.2 Algoritmo de Resolução (Backtracking)

A função principal, `solucaoFormula`, utiliza uma abordagem recursiva. O algoritmo tenta atribuir valores (`TRUE` e `FALSE`) para cada variável sequencialmente.

A otimização crucial implementada foi a função `verificaConflito`. Antes de avançar para a próxima variável, o algoritmo verifica se a atribuição atual tornou alguma cláusula falsa. Isso realiza a "poda" da árvore de busca, evitando processar ramos inviáveis.

Abaixo, o trecho do código que implementa a recursão com poda, conforme solicitado nas instruções de documentação:

```
int resolveRecursivamente(Formula *f, int idx_var) {
    // 1. CASO BASE: Sucesso se passamos da ultima variavel
    if (idx_var > f->num_variaveis) {
        return TRUE;
    }

    // 2. Loop de Tentativas: TRUE (1) e FALSE (0)
    int opcoes[] = {TRUE, FALSE};

    for (int i = 0; i < 2; i++) {
        f->valoracao_variaveis[idx_var] = opcoes[i];

        // PODA: So avanca se nao houver conflito imediato
        if (verificaConflito(f) == FALSE) {
            // Passo Recursivo
```

```
            if (resolveRecursivamente(f, idx_var + 1) == TRUE) {
                return TRUE;
            }
        }
    }

    // 3. BACKTRACK: Reseta valor antes de retornar erro
    f->valoracao_variaveis[idx_var] = NAO_VALORADA;
    return FALSE;
}
```

Listing 1: Função recursiva de resolução do SAT

2 Impressões Gerais

O desenvolvimento deste Trabalho Prático I proporcionou uma experiência prática robusta no gerenciamento manual de memória e na aplicação de algoritmos recursivos.

Aspectos Positivos:

- A obrigatoriedade da modularização reforçou a importância de separar a interface da implementação, facilitando a organização do código.
- O uso da ferramenta `valgrind` foi fundamental para garantir que não houvesse vazamentos de memória (*memory leaks*), especialmente ao lidar com a destruição da estrutura `Formula` em casos de erro[cite: 49].

Dificuldades Encontradas:

- A implementação da função de poda (`verificaConflito`) exigiu atenção aos detalhes da lógica booleana, para diferenciar corretamente uma cláusula "Falsa" (conflito) de uma cláusula "Indeterminada" (ainda válida).
- Garantir que o formato de saída fosse exato ao do enunciado [cite: 98] exigiu testes manuais repetidos.

3 Análise de Resultados

O problema 3-CNF SAT pertence à classe NP-Completo[cite: 38]. Teoricamente, o pior caso do algoritmo de força bruta seria $O(2^N)$. No entanto, a implementação do *backtracking* com poda demonstrou ser significativamente mais eficiente na prática.

3.1 Análise de Complexidade Assintótica

A tabela a seguir detalha a complexidade de tempo (Big-O) e de espaço auxiliar para as principais funções implementadas no Trabalho Prático. Considera-se N como o número de variáveis e M como o número de cláusulas.

Tabela 1: Análise de Complexidade das Funções

Função	**Tempo**	**Espaço (Aux)**	**Justificativa**
Gerenciamento de Memória e Estrutura			
`criaLiteral`	$O(1)$	$O(1)$	Executa apenas operações aritméticas e atribuições simples, sem laços de repetição.
`criaFormula`	$O(N)$	$O(1)$	Realiza a alocação dinâmica e um laço para inicializar o vetor de valoração de tamanho N. O espaço alocado no Heap é $O(N + M)$.
`destroiFormula`	$O(1)$	$O(1)$	Executa chamadas diretas de `free` para liberar a memória alocada, independente do conteúdo.
`adicionaClausula`	$O(1)$	$O(1)$	Acessa o vetor de cláusulas por índice direto e preenche 3 literais.
Entrada e Saída (I/O)			
`imprimeFormula`	$O(M)$	$O(1)$	Percorre o vetor de cláusulas (tamanho M) uma vez para imprimir os literais.
`imprimeValoracao`	$O(N)$	$O(1)$	Percorre o vetor de valoração (tamanho N) linearmente para imprimir os estados.
Algoritmo de Resolução (Backtracking)			
`verificaConflito`	$O(M)$	$O(1)$	No pior caso, verifica todas as M cláusulas para garantir a consistência. O custo por cláusula é constante (3 literais).
`resolveRecursivamente`	$O(M \cdot 2^N)$	$O(N)$	**Tempo:** No pior caso (força bruta sem poda efetiva), explora 2^N estados, executando a verificação ($O(M)$) em cada um. **Espaço:** Profundidade máxima da pilha de recursão é igual ao número de variáveis (N).
`solucaoFormula`	$O(M \cdot 2^N)$	$O(N)$	Wrapper que inicia a recursão, herdando a complexidade da função recursiva.

Como observado na Tabela 1, a complexidade exponencial $O(2^N)$ é inerente à natureza

NP-Completa do problema SAT. No entanto, o espaço auxiliar mantém-se linear $O(N)$ na pilha de execução, o que é seguro para os limites especificados ($N \leq 26$).

3.2 Comportamento do Algoritmo

Observou-se nos testes que, para fórmulas insatisfatíveis que geram contradições logo nas primeiras variáveis (ex: a e $\neg a$ na mesma cláusula), a função `verificaConflito` corta a execução imediatamente, impedindo o crescimento exponencial da árvore de recursão.

Para fórmulas satisfatíveis, o algoritmo segue a heurística de testar `TRUE` primeiro, encontrando rapidamente a solução quando ela reside nos primeiros ramos da árvore de decisão.

4 Conclusão

O trabalho atingiu seus objetivos ao implementar corretamente um solucionador SAT modularizado e livre de erros de memória. A solução final utiliza estruturas dinâmicas e recursão para navegar pelo espaço de busca de forma controlada.

A principal lição aprendida foi como a técnica de *backtracking*, aliada a uma boa estrutura de dados (TAD), pode tornar tratável a resolução de problemas combinatórios complexos, desde que haja mecanismos eficientes de poda para conter a complexidade exponencial. O código entregue respeita todas as especificações de entrada e saída, garantindo sua correção automática[cite: 98].