Trabalho Prático 0
Expressões lógicas e satisfabilidade

Raissa Gonçalves Diniz
Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG – Brasil
raissagdiniz@gmail.com

1. Introdução:

Esta documentação lida com dois assuntos: a avaliação de expressões lógicas, e o problema da satisfabilidade. O primeiro, recebe uma expressão (envolvendo os operadores lógicos de negação, conjunção, disjunção, e possíveis parêntesis) com n variáveis, e uma valoração para cada uma delas, e retorna se, para tais variáveis, aquela expressão é verdadeira ou não. Já o segundo, não trata de valores específicos para cada variável, mas sim se existe uma valoração que faz com que a expressão resulte em 1. Nele, no máximo cinco variáveis são quantificadas, sendo os quantificadores disponíveis: “e” (que simboliza ∃, ou “existe”) e “a” (∀, que representa o “para todo”).

2. Método

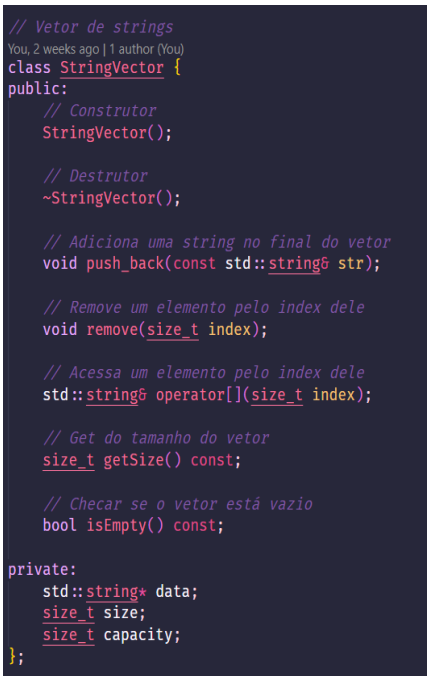
- Programa desenvolvido em C++
- Compilador g++ versão 9.4.0
- Windows 10 Home x64
- Linux Ubuntu 20.04

Especificações da máquina utilizada:
- Processador: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz 2.40 GHz
- RAM: 8 GB

2.1. Classes e funções

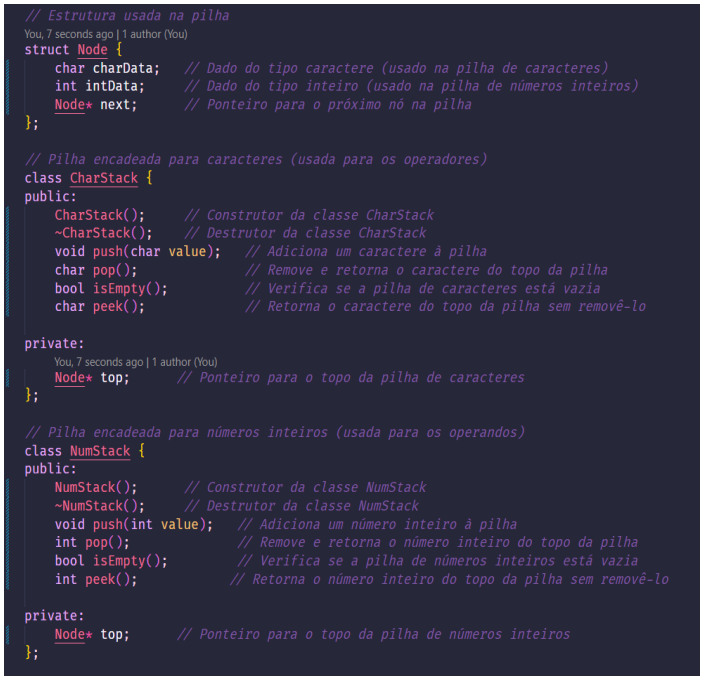
No total, o trabalho contou com cinco classes: duas classes para implementação das pilhas (dada a preferência por não utilização de templates, dois tipos de pilhas foram criados: uma para lidar com caracteres, e outra com números inteiros), uma classe para implementação de um vetor de strings, uma para lidar com a lógica para avaliar expressões, e outra para resolver o problema de satisfabilidade com quantificadores. Assim, três dessas classes são usadas para implementar já citadas estruturas de dados, e duas lidam com as operações necessárias para resolução de problemas.

As classes que lidam com as estruturas de dados necessárias para execução do trabalho possuem alguns métodos básicos para operações, e são relativamente semelhantes entre si.



```
// Vetor de strings
You, 2 weeks ago | 1 author (You)
class StringVector {
public:
    // Construtor
    StringVector();

    // Destrutor
    ~StringVector();

    // Adiciona uma string no final do vetor
    void push_back(const std::string& str);

    // Remove um elemento pelo index dele
    void remove(size_t index);

    // Acessa um elemento pelo index dele
    std::string& operator[](size_t index);

    // Get do tamanho do vetor
    size_t getSize() const;

    // Checar se o vetor está vazio
    bool isEmpty() const;

private:
    std::string* data;
    size_t size;
    size_t capacity;
};
```

Figura 1: Classe vetor de strings



```
// Estrutura usada na pilha
You, 7 seconds ago | 1 author (You)
struct Node {
    char charData;   // Dado do tipo caractere (usado na pilha de caracteres)
    int intData;     // Dado do tipo inteiro (usado na pilha de números inteiros)
    Node* next;      // Ponteiro para o próximo nó na pilha
};

// Pilha encadeada para caracteres (usada para os operadores)
class CharStack {
public:
    CharStack();     // Construtor da classe CharStack
    ~CharStack();    // Destrutor da classe CharStack
    void push(char value);   // Adiciona um caractere à pilha
    char pop();              // Remove e retorna o caractere do topo da pilha
    bool isEmpty();          // Verifica se a pilha de caracteres está vazia
    char peek();             // Retorna o caractere do topo da pilha sem removê-lo

private:
    You, 7 seconds ago | 1 author (You)
    Node* top;       // Ponteiro para o topo da pilha de caracteres
};

// Pilha encadeada para números inteiros (usada para os operandos)
class NumStack {
public:
    NumStack();      // Construtor da classe NumStack
    ~NumStack();     // Destrutor da classe NumStack
    void push(int value);    // Adiciona um número inteiro à pilha
    int pop();               // Remove e retorna o número inteiro do topo da pilha
    bool isEmpty();          // Verifica se a pilha de números inteiros está vazia
    int peek();              // Retorna o número inteiro do topo da pilha sem removê-lo

private:
    Node* top;       // Ponteiro para o topo da pilha de números inteiros
};
```

Figura 2: Classe de pilha de caractere e classe de pilha de inteiros

Já em relação às classes que contém a lógica por trás da resolução dos problemas, a classe `expEvaluator` avalia expressões lógicas que envolvem operadores lógicos (como "&", "|", "~") e parênteses, utilizando duas pilhas para realizar essa avaliação de maneira adequada. Ela processa a expressão caractere por caractere e determina se a expressão é verdadeira ou falsa com base nas valorações das variáveis lógicas e nas regras de precedência dos operadores.
Para isso, ela utilizada as duas pilhas supracitadas, e os métodos:

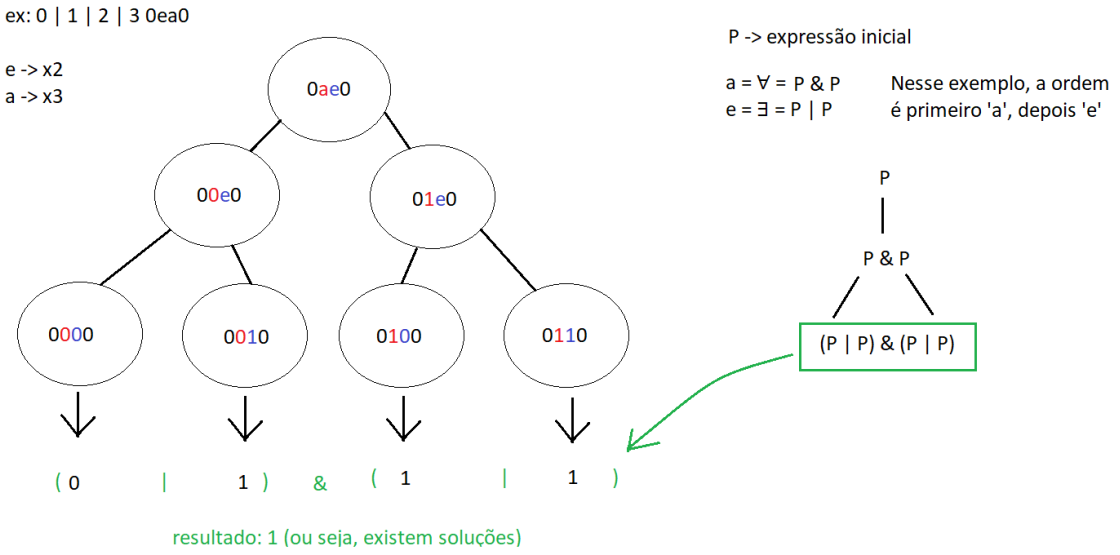
- *evaluate(std::string option, std::string exp, std::string values):* função principal para arvaliar uma expressão lógica. Inicializa variáveis e operadores em pilhas e realiza a avaliação passo a passo;
- *countVariables(const std::string& exp):* Conta o número de variáveis na expressão lógica;
- *initializeVariableValues(const std::string& values, int variableValues[]):* Inicializa os valores das variáveis com base na valoração fornecida;
- *handleMultiDigitNumber(const std::string& exp, size_t i, NumStack& operands, int variableValues[]):* Lida com números inteiros com mais de um dígito na expressão.
- *handleClosingParenthesis(NumStack& operands, CharStack& operations):* Lida com o caractere ')' e realiza operações até encontrar um '('.
- *handleOperator(char c, NumStack& operands, CharStack& operations):* Lida com operadores lógicos e realiza operações com base na precedência;
- *performOperation(NumStack& operands, CharStack& operations):* Realiza a operação especificada entre dois operandos;
- *precedence(char c):* Retorna a precedência do operador lógico.

Exemplo de funcionamento e aplicação da lógica:
Entrada: “~ (0 | 1)” 01

Caractere	Ação	Pilha de operandos	Pilha de operadores
~	Push pilha de operadores		~
(	Push pilha de operadores		(~

0	Push pilha de operandos	0	(~
|	Push pilha de operadores	0	| (~
1	Push pilha de operandos	1 0	| (~
)	Tira o 1 e o 0		| (~
	Tira o |		(~
	0 | 1 = 1, push pilha de operandos	1	(~
	Tira (	1	~
	Tira o 1		~
	Tira o ~		
	~1 = 0, push pilha de operandos	0 (resultado)	

No que tange a classe `quantifierEvaluator.cpp`, ela implementa a lógica para avaliar expressões lógicas que envolvem quantificadores, tal qual a imagem abaixo.



Seus métodos consistem em:

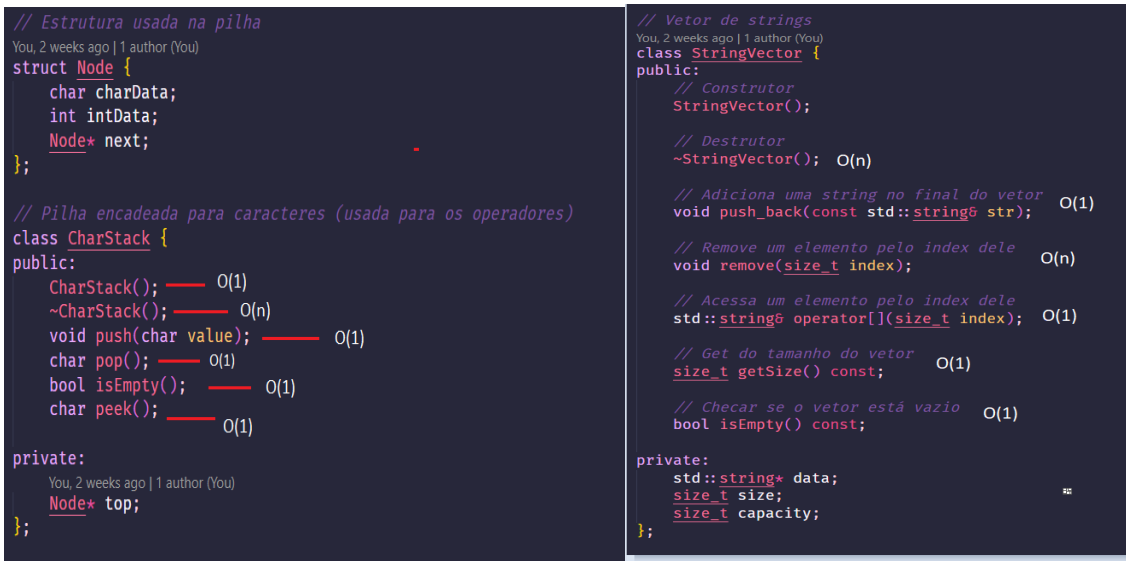
1. *generateOutcomes(std::string formula, std::string& valuation):*
   - Esta função gera todos os possíveis resultados binários com base na valoração fornecida, onde "e" e "a" são substituídos por valores binários (0 ou 1).
   - Calcula a quantidade de possíveis saídas (2^numLetters).
   - Gera todos os resultados possíveis, substituindo 'e' e/ou 'a' por valores binários.
2. *evaluateOutcomes(std::string formula, std::string valuation):*
   - Utiliza a função generateOutcomes para gerar os resultados possíveis.
   - Avalia cada uma das possibilidades na expressão e retorna os resultados em um vetor (1 se aquela possibilidade satisfaz a expressão, 0 se não).
3. *generateExpressionP(std::string formula, std::string valuation):*
   - Gera a expressão lógica baseada nos quantificadores "e" e "a" na valoração.
   - Utiliza as sequências "P & P" para "e" e "P | P" para "a" e substitui "P" na expressão inicial pela sequência apropriada para cada quantificador.
   - Retorna a expressão resultante.

4. *generateExpression(std::string formulaP):*
    - Gera a expressão com números em vez de 'P's.
    - Substitui cada ocorrência de 'P' pelo número atual (0, 1, 2, ...) na expressão.
5. *switchToA(std::string valuation):*
    - Caso o valor de uma variável não implique em alteração no resultado, substitui tal variável pela saída ‘a’ (para que a saída fique no formato especificado no enunciado do trabalho).
6. *calculateResult(std::string formula, std::string valuation):*
    - Esta função calcula o resultado final da expressão quantificada.
    - Utiliza as funções anteriores para gerar a expressão quantificada, avaliar os resultados e substituir "e" por "a" quando todos os resultados forem verdadeiros.
    - Retorna o resultado final como uma string.

Em resumo, este código permite a avaliação de expressões quantificadas, considerando todos os possíveis resultados das variáveis quantificadas e aplicando a lógica apropriada para os quantificadores "e" (existencial) e "a" (universal). Ele utiliza funções auxiliares para realizar essa tarefa complexa, garantindo que as expressões sejam avaliadas de acordo com as regras de quantificação especificadas.

3. Análise de complexidade

1. Complexidade de tempo



Aqui está a análise de complexidade de tempo para as funções das suas pilhas e do vetor:

Pilhas (CharStack e NumStack):

1. Construtor:
    - Complexidade de tempo: O(1)
    - Explicação: Ambos os construtores apenas inicializam a pilha vazia criando o nó superior (topo). Isso é uma operação de tempo constante.
2. Destrutor:
    - Complexidade de tempo: O(n)

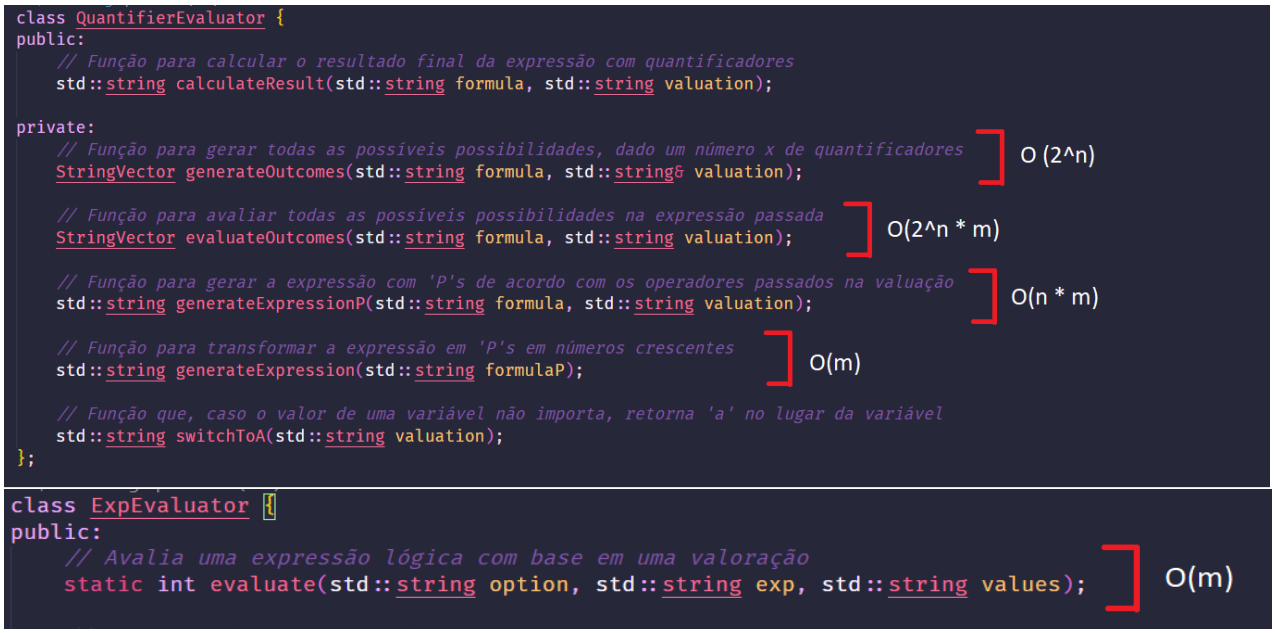
- Explicação: Os destrutores percorrem a pilha e liberam a memória de todos os nós. Isso envolve percorrer todos os elementos da pilha, portanto, é linear em relação ao número de elementos na pilha.

3. push():
   - Complexidade de tempo: O(1)
   - Explicação: A função push() insere um novo elemento no topo da pilha, o que é uma operação de tempo constante, independentemente do tamanho da pilha.
4. pop():
   - Complexidade de tempo: O(1)
   - Explicação: A função pop() remove o elemento do topo da pilha, o que também é uma operação de tempo constante, independentemente do tamanho da pilha.
5. isEmpty():
   - Complexidade de tempo: O(1)
   - Explicação: Verificar se a pilha está vazia envolve apenas uma verificação do ponteiro de topo, o que é uma operação de tempo constante.
6. peek():
   - Complexidade de tempo: O(1)
   - Explicação: A função peek() retorna o elemento no topo da pilha sem removê-lo, o que é uma operação de tempo constante.

Vetor (StringVector):

1. *Construtor:*
   - Complexidade de tempo: O(1)
   - Explicação: O construtor inicializa o vetor vazio com uma capacidade inicial predefinida, o que é uma operação de tempo constante.
2. *Destrutor:*
   - Complexidade de tempo: O(n)
   - Explicação: O destrutor libera a memória alocada para o array de strings. Isso envolve a eliminação de cada string no array, o que é linear em relação ao número de elementos no vetor.
3. *push_back():*
   - Complexidade de tempo: O(1) em média, O(n) em casos de realocação.
   - Explicação: A função push_back() insere uma string no final do vetor. Em média, a inserção é uma operação de tempo constante devido à alocação dinâmica. Nesse programa, o tamanho do vetor não é realocado.
4. remove():
   - Complexidade de tempo: O(n)
   - Explicação: A função remove() remove um elemento pelo índice e, em seguida, desloca os elementos à direita do elemento removido para preencher o espaço vazio. Isso envolve a cópia de elementos e, portanto, é linear em relação ao número de elementos no vetor.
5. Operador [] (StringVector::operator[]):
   - Complexidade de tempo: O(1)
   - Explicação: Acesso direto a um elemento do vetor pelo índice é uma operação de tempo constante, pois não requer percorrer o vetor.
6. getSize():
   - Complexidade de tempo: O(1)
   - Explicação: Obter o tamanho atual do vetor é uma operação de tempo constante, pois o tamanho é mantido internamente.
7. isEmpty():
   - Complexidade de tempo: O(1)
   - Explicação: Verificar se o vetor está vazio envolve apenas uma verificação do tamanho interno, o que é uma operação de tempo constante.

Em resumo, as pilhas e o vetor têm principalmente operações de tempo constante, mas é importante notar que o vetor pode ter uma complexidade linear em casos raros de realocaçã



1. *generateOutcomes:*
   - Complexidade de tempo: O(2^n), onde n é o número de quantificadores na valoração.
   - Explicação: A função gera todas as possíveis combinações binárias de valores para os quantificadores na valoração. Para cada quantificador, há duas opções (0 ou 1), e existem 5 quantificadores no máximo. Portanto, o número total de combinações geradas é 2^n.
2. *evaluateOutcomes:*
   - Complexidade de tempo: O(2^n * m), onde n é o número de quantificadores na valoração e m é o número de operações executadas na avaliação da expressão.
   - Explicação: A função chama `generateOutcomes`, que tem uma complexidade de O(2^n), e, em seguida, avalia cada uma das possíveis combinações geradas. A avaliação da expressão envolve a análise de cada operador e operando na expressão, o que pode levar a m operações por cada combinação.
3. *generateExpressionP:*
   - Complexidade de tempo: O(n * m), onde n é o número de quantificadores na valoração e m é o comprimento da expressão final gerada.
   - Explicação: A função itera sobre os quantificadores na valoração e cria uma expressão lógica com base em cada quantificador. A criação da expressão envolve a concatenação de sequências de caracteres, que depende do número de quantificadores.
4. *generateExpression:*
   - Complexidade de tempo: O(m), onde m é o comprimento da expressão fornecida.
   - Explicação: A função itera sobre a expressão e substitui os caracteres 'P' por números. Isso é feito em tempo linear em relação ao comprimento da expressão.
5. *calculateResult:*
   - Complexidade de tempo: O(2^n * m), onde n é o número de quantificadores na valoração e m é o número de operações executadas na avaliação da expressão.

- Explicação: A função chama `generateExpressionP` e `generateExpression`, que têm complexidades de O(n * m) e O(m) respectivamente. Em seguida, avalia a expressão modificada, que envolve a análise de cada operador e operando na expressão, levando a m operações por cada combinação gerada.

6. *ExpEvaluator::evaluate:*
   - Complexidade de tempo: O(m), onde m é o número de caracteres na expressão a ser avaliada.
   - Explicação: A função avalia a expressão passada como argumento. Ela percorre a expressão uma vez e executa operações com os operadores lógicos encontrados. A complexidade é linear em relação ao número de caracteres na expressão.

2. Complexidade de espaço

CharStack e NumStack (Pilhas Encadeadas para Caracteres e Números Inteiros):

- Complexidade de espaço: O(n), onde n é o número de elementos na pilha.
- Explicação: As pilhas foram implementadas como estruturas encadeadas, e o espaço utilizado é proporcional ao número de elementos na pilha. Cada elemento na pilha é representado por um nó, que contém um caractere ou um número inteiro, dependendo do tipo da pilha. Portanto, o espaço é linear em relação ao número de elementos na pilha.

StringVector (Vetor de Strings):

- Complexidade de espaço: O(n), onde n é o número de elementos no vetor.
- Explicação: O vetor de strings é implementado usando um array dinâmico. O espaço utilizado é proporcional ao número de elementos no vetor. Como esse vetor é usado para armazenar possibilidades e seus resultados, e levando em consideração que só podem existir até 5 quantificadores na expressão, esse vetor deve ser de tamanho até, no máximo, 2^5, o que mal gasta memória e não precisa de realocação.

Em resumo, a complexidade de espaço das suas estruturas de dados é principalmente linear em relação ao número de elementos armazenados nas pilhas e no vetor de strings.

4. Estratégias de robustez

O código utiliza exceções (std::runtime_error) para notificar erros ou exceções durante a execução. As exceções são lançadas nas seguintes ocasiões:

1. **Limite de Quantificadores**: Para evitar cenários de grande complexidade (e tal qual esclarecido no enunciado do trabalho), o código impõe um limite de até 5 quantificadores em uma expressão.
2. **Verificação de Correspondência de Variáveis**: O código verifica se o número de variáveis na expressão corresponde ao número de variáveis na valoração.
3. **Verificação de Presença de Quantificadores**: Para garantir que o problema de satisfabilidade tenha quantificadores, o código verifica se pelo menos um quantificador (existencial ou universal) está presente na valoração.
4. **Leitura de Valores**: No código da classe expEvaluator, quando os valores das variáveis são inicializados com base na valoração, é feita uma verificação para garantir que não haja quantificadores presentes na valoração.
5. **Verificação de Operador Desconhecido**: O código verifica se um operador desconhecido é usado na expressão (isto é, qualquer operador diferente de |, & ou ~).
6. **Limitação de Variáveis**: O código assume um limite de até 100 variáveis (de 0 a 99).

Essas estratégias tornam o código mais robusto, ajudando a detectar e lidar com problemas potenciais e garantindo que a lógica da avaliação de expressões, com ou sem quantificadores, seja executada corretamente.

5. Análise Experimental

6. Conclusão

O presente trabalho abordou o desafio de criar um avaliador de expressões lógicas e resolver o problema da satisfabilidade, utilizando estruturas de dados como pilhas, vetores e aplicando diversas lógicas computacionais. Ao desenvolver as funções necessárias em primeira pessoa, pude adquirir um entendimento mais profundo e pessoal das análises de complexidade e tempo, reforçando a importância desses conceitos não apenas na teoria, mas também na prática do desenvolvimento de algoritmos. Isso contribuiu significativamente para o meu aprendizado e para a melhoria das minhas habilidades na área de estrutura de dados e algoritmos.

7. Referências

Expression evaluation using stack. Disponível em: <https://www.codingninjas.com/studio/library/expression-evaluation-using-stack>. Acesso em: 18 set. 2023.

Stack implementation in cpp. Disponível em: <https://www.techiedelight.com/stack-implementation-in-cpp/>. Acesso em: 19 set. 2023.

Stack implementation in cpp. Disponível em: <https://www.techiedelight.com/stack-implementation-in-cpp/>. Acesso em: 19 set. 2023.

Conjunctive normal form. Disponível em: <https://www.wikiwand.com/en/Conjunctive_normal_form>. Acesso em: 25 set. 2023.

8. Instruções de compilação e execução

- Descompacte o arquivo com extensão .zip;

- Ache a pasta 'TP0' no terminal;

- Digite "*make*" para compilar o programa;

- digite *./bin/tp1.out -(opção)* <*expressão*> <*valoração*> no terminal para executar o programa.

As opções são:

- -a: para o avaliador de expressões;
- -s: para o problema de satisfabilidade;

Exemplo 1: para testar o avaliador de expressões: *./bin/ tp1.out -a "0 | 1 & 2" 010*
Exemplo 2: para testar a satisfabilidade da expressão: *./bin/ tp1.out -s "0 | 1" a0*