**Trabalho Prático 0**

**Expressões lógicas e satisfabilidade**

Raissa Gonçalves Diniz

Universidade Federal de Minas Gerais (UFMG)

Belo Horizonte - MG – Brasil

[raissagdiniz@gmail.com](mailto: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 com *n* variáveis (envolvendo os operadores lógicos de negação, conjunção/interseção, disjunção/união, representados, respectivamente, pelos caracteres “~”, “&” e “|”) e uma valoração para cada uma delas, retornando 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”).

1. **Método**

- Programa desenvolvido em C++

- Compilador g++ versão 9.4.0

- Windows 10 Home x64

- Linux Ubuntu 20.04

Especificações da máquina utilizada:

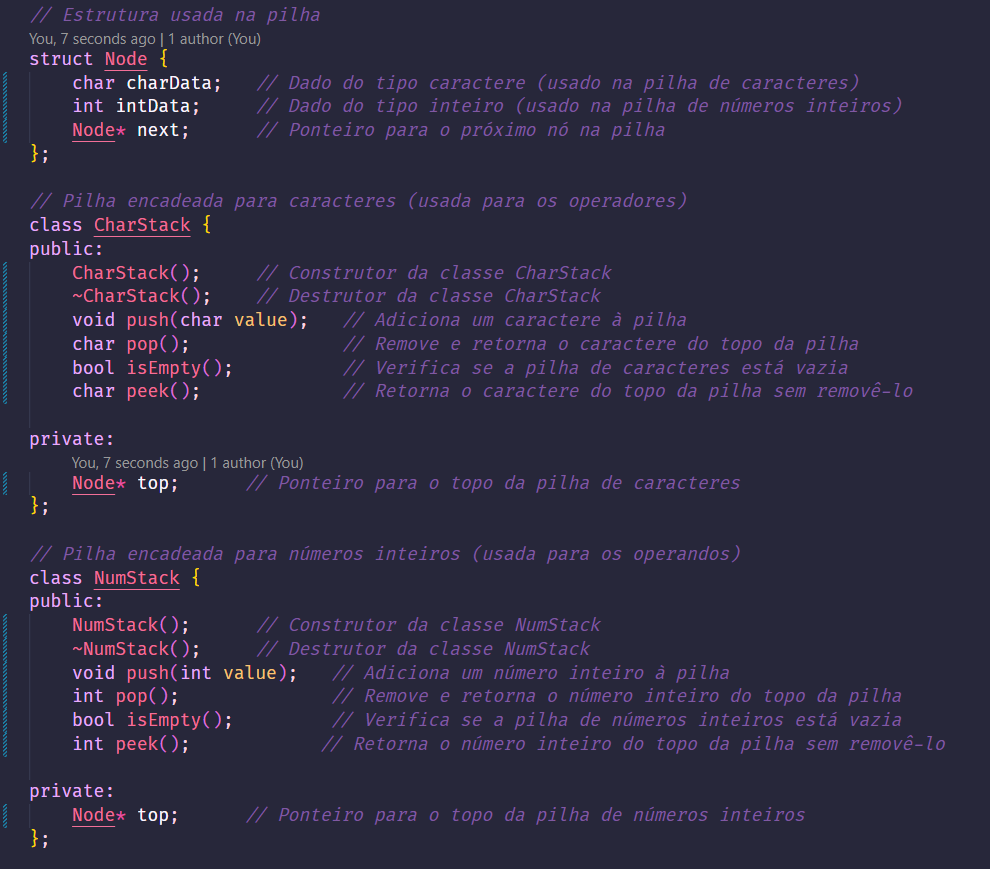
- Processador: Intel(R) Core(TM) i5-9300H CPU @ 2.40GHz   2.40 GHz

- RAM: 8 GB

* 1. **Classes e funções**

No total, o trabalho contou com quatro 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 uma árvore binária, e outra para lidar com a lógica de avaliação de expressões (com ou sem quantificadores). Assim, três dessas classes são usadas para implementar já citadas estruturas de dados, e uma lida com as operações necessárias para resolução de problemas.

A classe expEvaluator avalia expressões lógicas que envolvem operadores lógicos (sendo eles "&", "|", "~") 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:



Métodos das classes CharStack (pilha de caracteres) e NumStack (pilha de números inteiros)

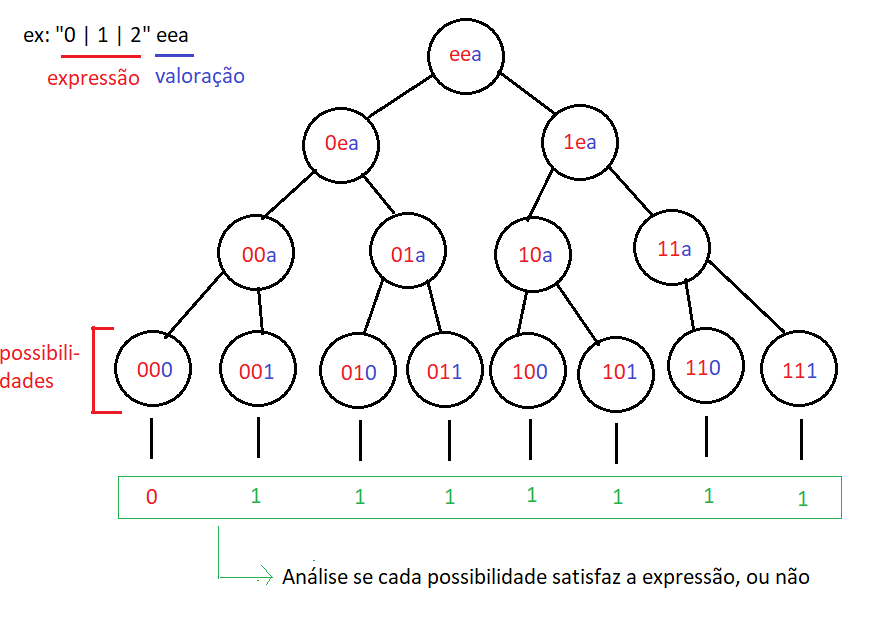
* ***Int evaluate(std::string option, std::string exp, std::string values):*** função principal para avaliar uma expressão lógica. Inicializa variáveis e operadores em pilhas e realiza a avaliação passo a passo;
* ***Int countVariables(const std::string& exp):***Conta o número de variáveis na expressão lógica;
* ***Void initializeVariableValues(const std::string& values, int variableValues[]):*** Inicializa os valores das variáveis com base na valoração fornecida;
* ***Size\_t handleMultiDigitNumber(const std::string& exp, size\_t i, NumStack& operands, int variableValues[]):*** Lida com números inteiros com mais de um dígito na expressão.
* ***Void vhandleClosingParenthesis(NumStack& operands, CharStack& operations):*** Lida com o caractere ')' e realiza operações até encontrar um '(';
* ***Void handleOperator(char c, NumStack& operands, CharStack& operations):*** Lida com operadores lógicos e realiza operações com base na precedência;
* ***Int performOperation(NumStack& operands, CharStack& operations):***Realiza a operação especificada entre dois operandos;
* ***Int precedence(char c):*** Retorna a precedência do operador lógico;
* ***Void satisfabilityCheck(std::string expression., std::string evaluation):*** Responsável por chamar os métodos necessários de BinaryTree na ordem necessária para avaliar a expressão com quantificadores corretamente.

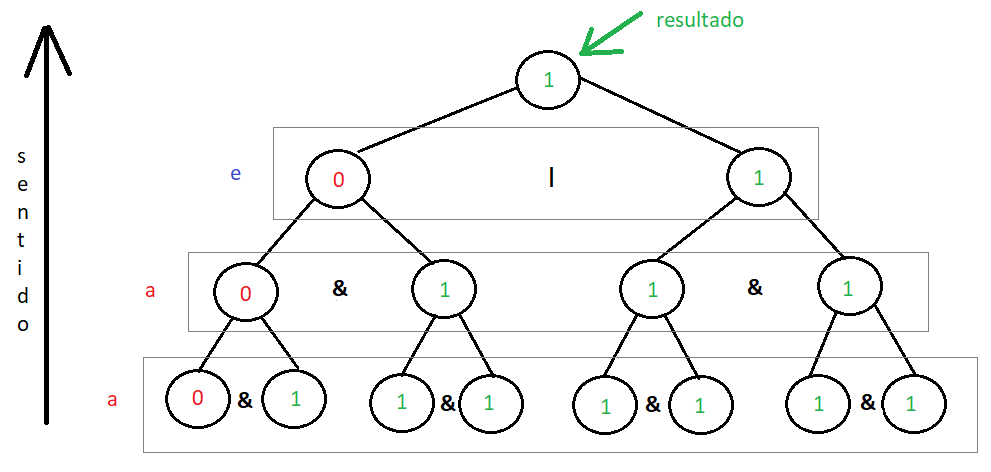
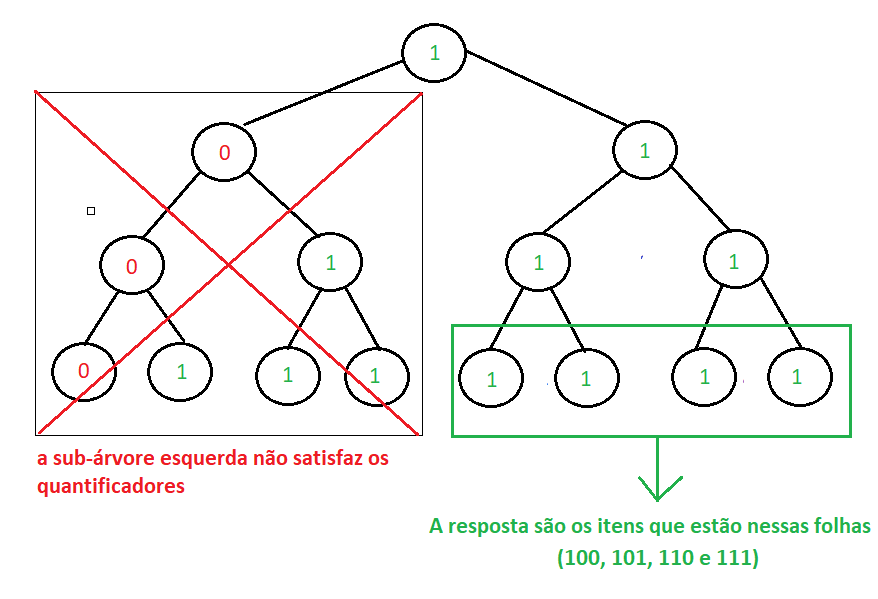
Exemplo de funcionamento e aplicação da lógica:

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 **BinaryTree,** ela é majoritariamente responsável por implementar a estrutura e lógicas necessárias para avaliar expressões lógicas que envolvem quantificadores. Um exemplo de seu funcionamento foi anexado nas imagens abaixo.

.

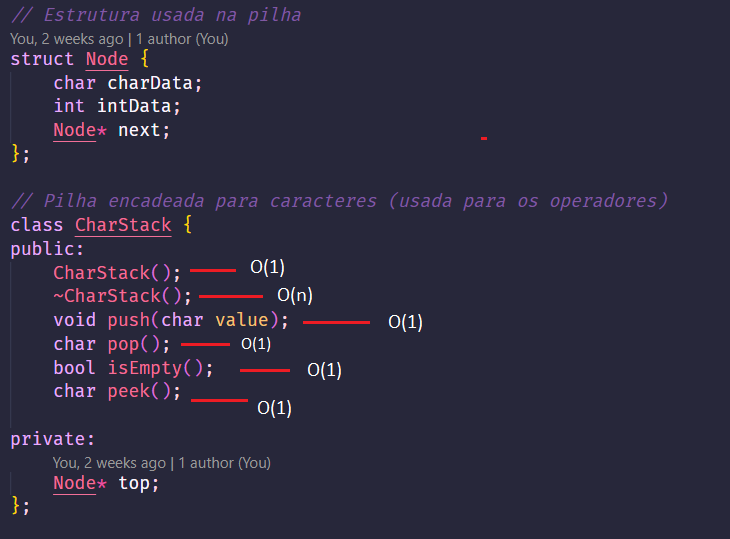


Seus métodos consistem em:

1. ***BinaryTree()****:*
   * Construtor da classe BinaryTree.
   * Inicializa a raiz da árvore binária como nula.
2. ***~BinaryTree:***
   * Destrutor da classe BinaryTree.
   * Limpa toda a árvore binária, chamando a função clear().
3. ***Void BuildTree (std::string item):***
   * Método público para construir a árvore binária.
   * Chama a função recursiva buildRecursive() para construir a árvore.
4. **Void BuildRecursive(NodeType &p, std::string item, int pos, NodeType parent = nullptr):**
   * Função recursiva que constrói a árvore binária a partir de uma string item.
   * Usa os caracteres 'a' e 'e' presentes na string para criar ramos na árvore, até que as folhas representem todas as possibilidades possíveis de valores que aquela string pode tomar.
   * Mantém uma referência para o nó pai.
5. **Void Clear(NodeType &p):**
   * Função que limpa a árvore binária de forma recursiva.
   * Libera a memória alocada para os nós.
6. ***Void TraverseAndEvaluate(std::string expression, NodeType \*p):***
   * Caminha na árvore binária a fim de calcular os resultados das folhas com base na expressão expressada.
   * Para isso, usa o objeto ExpEvaluator (verificando se, para cada possibilidade, ela satisfaz, ou não, a expressão).
7. ***NodeType\* GetRoot():***
   * Retorna a raiz da árvore.
8. ***Int GetHeight(NodeType node):***
   * Calcula a altura da árvore a partir de um nó específico.
9. ***Std::string ExtractEAndA(const std::string& input):***
   * Extrai os caracteres 'e' e 'a' da valoração e cria uma nova string só com eles.
10. ***Void ApplyOperationRecursive(NodeType p, char operation, int level):***
    * Aplica uma operação lógica ('a' para AND ou 'e' para OR) a todos os pares de nós em um nível da árvore.
11. ***Void ApplyOperationsFromBottom(const std::string& operations):***
    * Aplica as operações lógicas (AND ou OR), com base nos quantificadores passados na valoração, um nível por vez, da parte inferior da árvore até a raiz.
    * Usa a função ApplyOperationRecursive para isso.
12. ***Std::string EvaluateRootChildren():***
    * Avalia os filhos da raiz da árvore e retorna resultados com base em algumas condições.
13. ***Std::string GetLeafItems(NodeType p, int targettResult):***
    * Recursivamente obtém os itens dos nós folha com um resultado desejado.
14. ***Std::string ANotation(std::string input):***
    * Converte as respostas para a notação que utiliza um "a" e retorna uma string resultante.

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

1. **Análise de complexidade**
   1. **Complexidade de tempo**

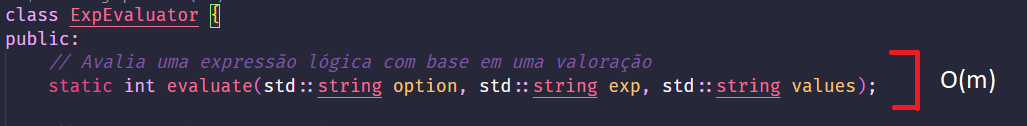


***Pilhas (CharStack e NumStack):***

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.

***Árvore binária (BinaryTree):***

1. Construtor:
   * Complexidade de Tempo: O(1)
   * A função apenas faz uma atribuição de valor fixo.
2. Destrutor:
   * Complexidade de Tempo: O(n)
   * A função clear() é chamada para limpar a árvore, e essa função percorre todos os nós da árvore uma vez.
3. **buildTree(std::string item)**:
   * Complexidade de Tempo: O(n)
   * O tempo de execução depende do tamanho da string item, pois a função buildRecursive() é chamada uma vez para cada caractere na string.
4. buildRecursive(NodeType &p, std::string item, int pos, NodeType parent = nullptr):
   * Complexidade de Tempo: O(2^n)
   * Essa função é recursiva e cria uma árvore binária de todas as combinações possíveis de "a" e "e" na string item. O número total de chamadas cresce exponencialmente com o tamanho da string.
5. clear(NodeType &p):
   * 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. **traverseAndEvaluate(std::string expression, NodeType \*p)**:
   * Complexidade de Tempo: O(n)
   * A função avalia cada nó da árvore exatamente uma vez.
7. **getRoot**():
   * Complexidade de Tempo: O(1)
   * A função simplesmente retorna a raiz da árvore, o que é uma operação de tempo constante.
8. getHeight(NodeType node):
   * Complexidade de Tempo: O(n)
   * A função percorre toda a árvore para calcular a altura.
9. **extractEAndA(const std::string& input)**:
   * Complexidade de Tempo: O(n)
   * A função percorre a string de entrada uma vez.
10. applyOperationRecursive(NodeType p, char operation, int level):
    * Complexidade de Tempo: O(n)
    * A função percorre a árvore em um nível específico uma vez.
11. **applyOperationsFromBottom(const std::string& operations):**
    * Complexidade de Tempo: O(n)
    * A função percorre a árvore uma vez para aplicar as operações.
12. **evaluateRootChildren()**:
    * Complexidade de Tempo: O(1)
    * A função faz verificações condicionais simples com base nos resultados dos filhos da raiz.
13. getLeafItems(NodeType p, int targetResult):
    * Complexidade de Tempo: O(n)
    * A função percorre a subárvore a partir do nó p para encontrar os nós folha com um resultado desejado.
14. **aNotation(std::string input)**
    * Complexidade de Tempo: O(n\*log(m)).
    * A função extrai e processa os números da string de entrada. Onde n é o número de números na entrada e m é o número médio de dígitos por número.



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.
   1. **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.

*BinaryTree (Árvore binária):*

* Em uma árvore binária completa, o número de nós é dado por uma potência de 2. Se a árvore tem h níveis (começando em 0), então o número total de nós é 2^(h+1) - 1. Por exemplo, em uma árvore com 3 níveis, teremos 2^(3+1) - 1 = 15 nós.
* A complexidade de espaço de uma árvore binária completa é, portanto, O(2^h), onde 'h' é a altura da árvore. Isso significa que a complexidade de espaço cresce exponencialmente à medida que a altura da árvore aumenta (como só podem existir, no máximo, 5 quantificadores em uma valoração, essa complexidade acaba não sendo um problema).

1. **Estratégias de robustez**

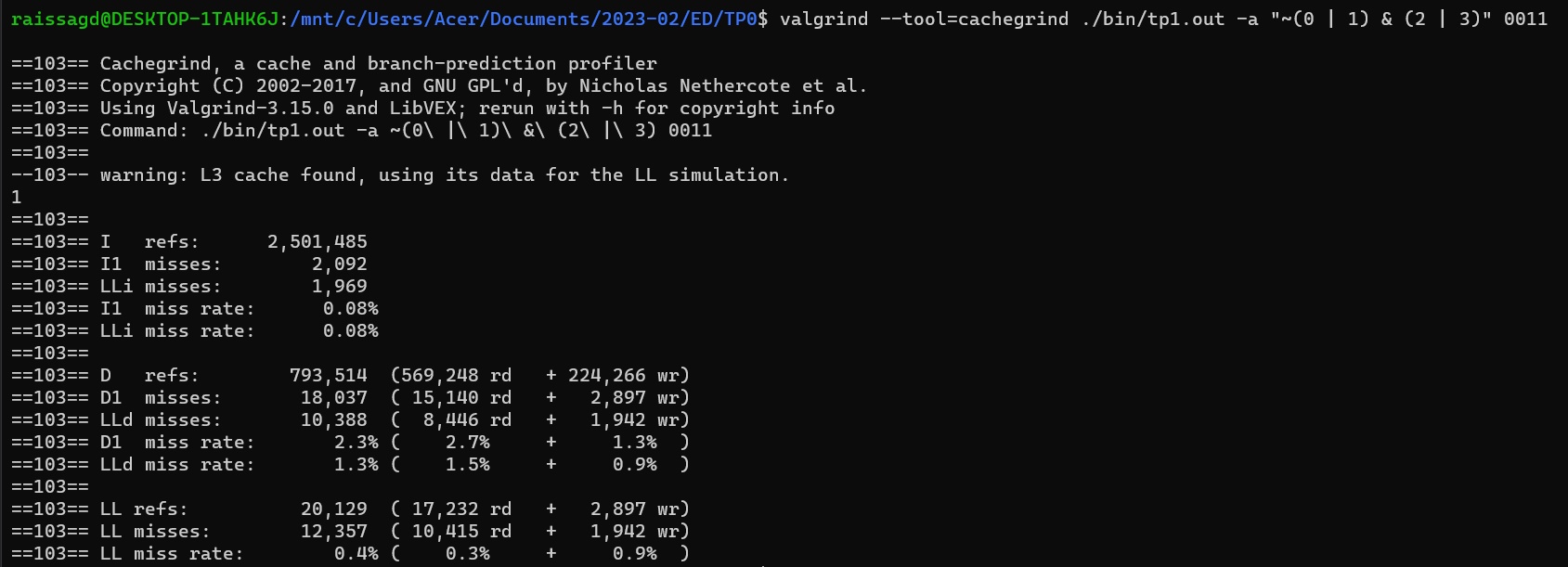
O código utiliza exceções (std::runtime\_error) para notificar erros durante a execução. As exceções são lançadas nas seguintes ocasiões:

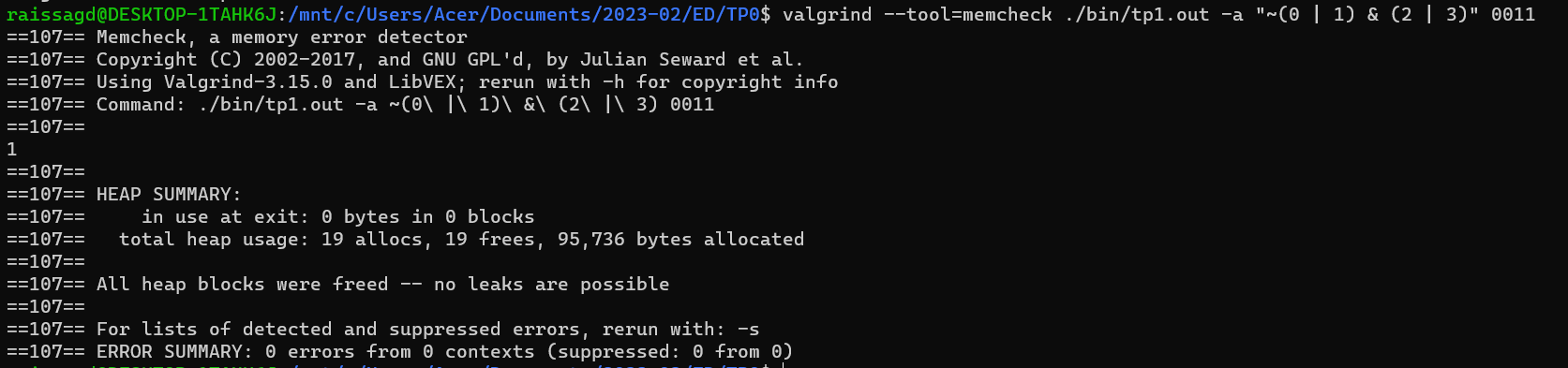
1. **Limite de Quantificadores**: Para evitar cenários de grande complexidade (e tal qual esclarecido no enunciado do trabalho), o código impõe um limite de até 5 quantificadores em uma expressão.
2. **Verificação de Correspondência de Variáveis**: O código verifica se o número de variáveis na expressão 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).

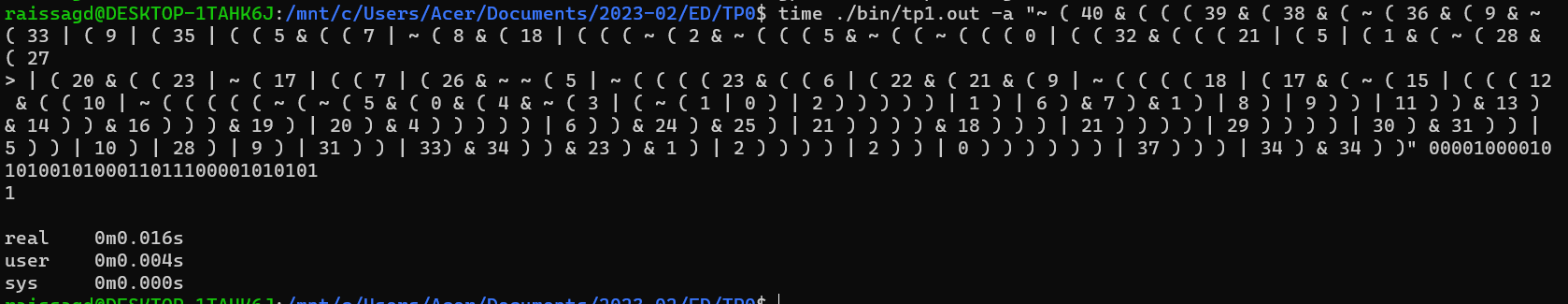
Essas estratégias tornam o código mais robusto, ajudando a detectar e lidar com potenciais problemas e garantindo que a lógica da avaliação de expressões, com ou sem quantificadores, seja executada corretamente.

1. **Análise Experimental**

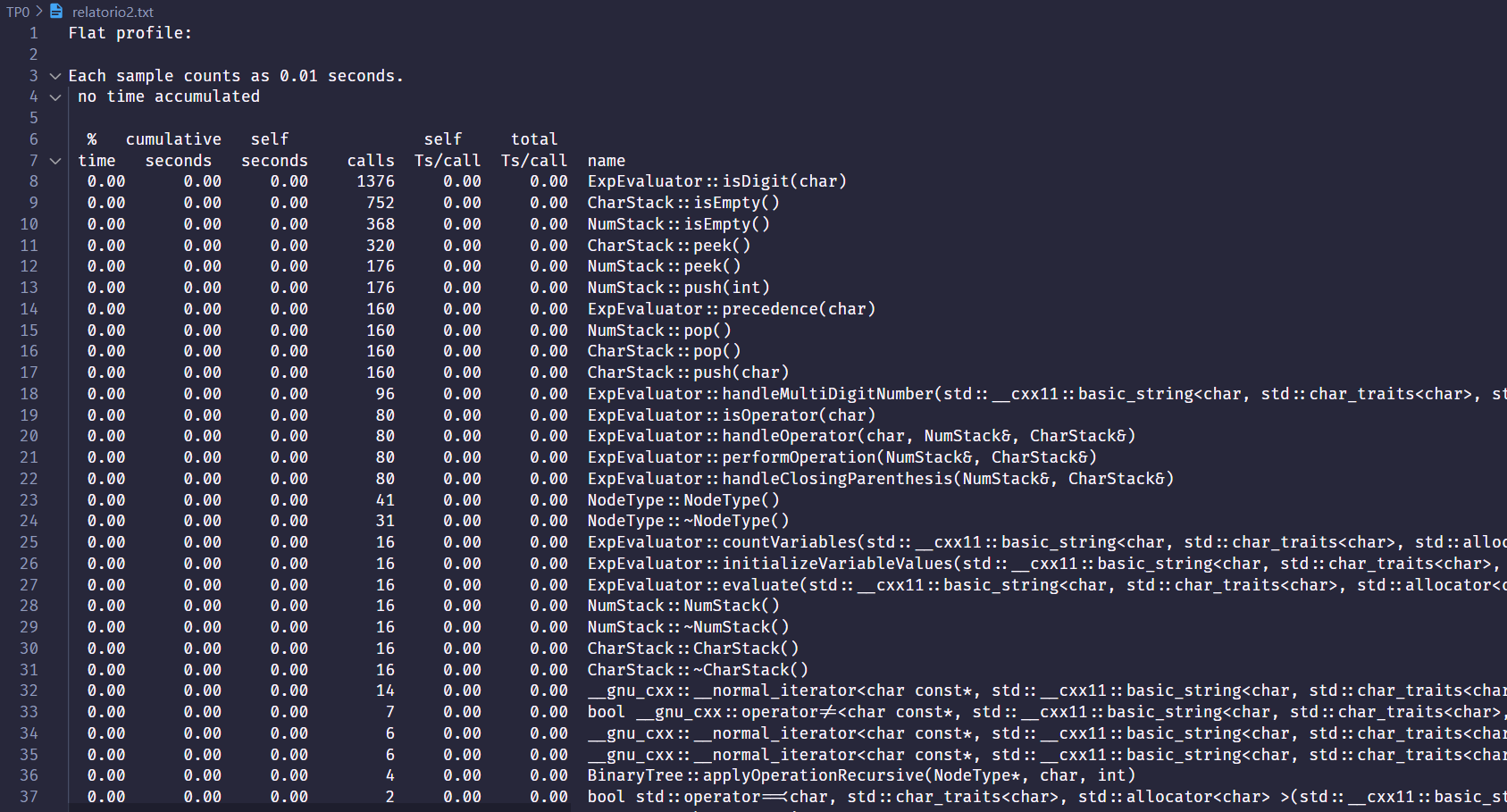
A análise experimental foi utilizada para avaliar o uso de memória durante a execução de um programa. Permitindo identificar possíveis vazamentos de memória e otimizar a alocação e desalocação de memória pelo programa. No caso do código apresentado, foram utilizadas ferramentas como o Valgrind e o Gprof para realizar essa análise. O Valgrind é uma ferramenta que permite identificar vazamentos de memória, enquanto o Gprof é uma ferramenta que permite medir o tempo de execução de cada função do programa. Para utilizar o Valgrind, basta executar o programa com o comando "valgrind nome\_do\_programa". O Valgrind irá monitorar o uso de memória pelo programa e exibir possíveis erros de vazamento de memória. Já o Gprof é utilizado para medir o tempo de execução de cada função do programa. Para utilizar o Gprof, é preciso compilar o programa com a opção "-pg" e executá-lo com o comando "gprof nome\_do\_programa gmon.out > relatorio.txt". O Gprof irá gerar um relatório de desempenho do programa, mostrando o tempo de execução de cada função. Além disso, podemos utilizar o comando "time" no terminal para medir o tempo total de execução do programa. Basta executar o comando "time nome\_do\_programa arquivo\_de\_entrada" e o terminal irá exibir o tempo de execução do programa.







Em relação ao avaliador de expressões lógicas usando pilhas, os dados do Valgrind indicam que, em termos de cache, o programa parece ter um bom desempenho, com uma taxa de falha relativamente baixa em ambas as caches L1 e LL. Isso significa que a maioria das instruções e dados é encontrada na cache, o que ajuda a minimizar a latência de acesso à memória principal. Além disso, não houveram vazamentos de memória, e o programa roda em pouco tempo, mesmo para grandes entradas. Um comportamento semelhante foi demonstrado na árvore binária.



Com base nos resultados do Gprof, as funções do programa não consomem um tempo significativo. As funções mais chamadas envolvem operações da pilha. As funções da árvore binária, em geral, parecem gastar ainda menos tempo, o que mostra que foram efetivamente implementadas.

1. **Conclusão**

O presente trabalho abordou tanto o desafio de criar um avaliador de expressões lógicas, quanto a resolução do problema da satisfabilidade, utilizando, para isso, estruturas de dados como pilhas e árvores binárias, 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 nessa área.

1. **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.

1. **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*