Expressões lógicas e satisfatibilidade

Trabalho Prático 1

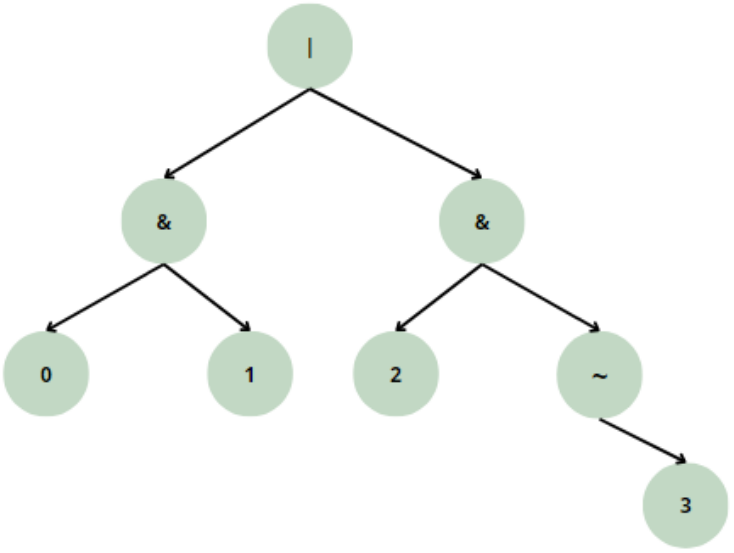
1. Introdução

Nesse Trabalho Prático, foi explorada a tarefa de avaliar expressões lógicas e determinar sua satisfatibilidade. O desafio reside na manipulação eficiente de variáveis, operadores lógicos e na resolução de expressões complexas. O programa aqui apresentado foi estruturado cuidadosamente, dividido em diferentes partes funcionais para alcançar os objetivos propostos. A primeira etapa envolve a análise da expressão lógica, seguida pela construção e avaliação de uma árvore de expressão. A segunda parte trata da determinação da satisfatibilidade, onde a busca exaustiva de casos satisfatíveis é realizada de maneira eficiente. O trabalho oferece uma visão detalhada sobre como o código foi desenvolvido para resolver problemas de lógica proposicional, destacando tanto a lógica do algoritmo quanto a implementação cuidadosa das estruturas de dados.

2. Método

2.1. Descrição da Implementação

O programa foi desenvolvido em C++ e se baseou na estrutura de dados de uma Árvore Binária de Expressão. Nessa árvore, os números ou operandos são representados pelas folhas, enquanto os operadores são representados pelos nós internos. As informações nos nós são armazenadas como strings. Para resolver o problema, o programa utilizou dois tipos de pilha: uma para armazenar os tipos de nós da árvore e outra para armazenar strings. A solução envolveu o mapeamento de expressões por meio de caminhamentos na Árvore Binária, como ilustrado na imagem para o exemplo “(0 & 1) | (2 & ~ 3)”.



2.2. Estruturas de Dados

TreeNode (TreeNode.hpp, TreeNode.cpp):
Representa um nó em uma árvore de expressão lógica.
Atributos:
- `string val`: Armazena o valor do nó (um operador lógico ou uma variável).
- `TreeNode* left`: Aponta para o nó filho esquerdo.
- `TreeNode* right`: Aponta para o nó filho direito.
Métodos:
- Construtor: Inicializa os atributos do nó com os valores fornecidos.

NodeStack (NodeStack.hpp, NodeStack.cpp):
Representa uma pilha de nós da árvore de expressão.
Atributos:
- `NodeStackNode* top`: Aponta para o nó no topo da pilha.
Métodos:
- `void push(TreeNode* data)`: Adiciona um nó na a pilha.
- `bool isEmpty()`: Verifica se a pilha está vazia.
- `TreeNode* getTop()`: Obtém o nó no topo da pilha.
- `void pop()`: Remove o nó do topo da pilha.

NodeStackNode (NodeStackNode.hpp, NodeStackNode.cpp):
Representa um nó na pilha de nós da árvore de expressão.
Atributos:
- `TreeNode* data`: Aponta para um nó da árvore de expressão.
- `NodeStackNode* link`: Aponta para o próximo nó na pilha.

StringStack (StringStack.hpp, StringStack.cpp):

Representa uma pilha de strings.
Atributos:
- `StringStackNode* top`: Aponta para o nó no topo da pilha.
Métodos:
- `void push(std::string data)`: Empurra uma string para a pilha.
- `bool isEmpty()`: Verifica se a pilha está vazia.
- `std::string getTop()`: Obtém a string no topo da pilha.
- `void pop()`: Remove a string do topo da pilha.

StringStackNode` (StringStackNode.hpp, StringStackNode.cpp):
Representa um nó na pilha de strings.
Atributos:
- `string data`: Armazena uma string.
- `StringStackNode* link`: Aponta para o próximo nó na pilha.

2.3. Funções e Métodos Implementados do main.cpp

As funções principais são:

replaceVariablesWithValues(string x, string values):
- Substitui variáveis na expressão pelos valores correspondentes.

getOperatorPrecedence(string op):
- Retorna a precedência de um operador lógico.

evaluate(float a, float b, string op):
- Avalia operações lógicas entre dois operandos.

evaluate(float b, string op):
- Avalia operações lógicas unárias.

buildInfixExpressionTree(string infix):
- Constrói uma árvore de expressão a partir de uma expressão lógica em notação infixada.

evaluateInfixExpressionTree(TreeNode* root):
- Avalia uma árvore de expressão lógica.

completeEvaluate(string expression, string values):
- Avalia uma expressão lógica completamente, passando pelas outras funções necessárias.

satisfiability(string expression, string& values):
- Determina a satisfatibilidade de uma expressão lógica.

2.4. Compilação e Execução

O código foi organizado em um sistema de compilação usando um arquivo Makefile. O programa é compilado usando o compilador g++ com flags de depuração e aparecerá na pasta “./bin” com o nome “tp1.out”. Para compilar o código, execute “make all” no terminal. Para executar o programa, utilize o comando `./bin/tp1.out -a "expressão" valores` para avaliação da expressão ou `./bin/tp1.out -s "expressão" valores` para determinação da satisfatibilidade.

3. Análise de Complexidade

replaceVariablesWithValues(string x, string values):

- Complexidade: O(n)
- Itera pelas variáveis da expressão uma vez, onde 'n' é o número de variáveis na expressão.

getOperatorPrecedence(string op):

- Complexidade: O(1)
- Usa comparação direta para determinar a precedência do operador.

evaluate(float a, float b, string op):

- Complexidade: O(1)
- Realiza operações lógicas simples entre dois operandos.

evaluate(float b, string op):

- Complexidade: O(1)
- Realiza operações lógicas unárias.

buildInfixExpressionTree(string infix):

- Complexidade: O(n)
- Itera pela expressão uma vez para construir a árvore, onde n é o comprimento da expressão.

evaluateInfixExpressionTree(TreeNode* root):

- Complexidade: O(n)
- A função percorre todos os nós da árvore uma vez, onde 'n' é o número de nós na árvore.

completeEvaluate(string expression, string values):

- Complexidade: O(n)
- Engloba as complexidades das funções chamadas internamente, predominantemente a construção da árvore e a avaliação da árvore.

satisfiability(string expression, string& values):

- Complexidade: O(2^n)
- A função realiza uma busca exaustiva de todas as combinações possíveis para variáveis 'e' e 'a', resultando em uma complexidade exponencial, onde 'n' é o número de variáveis 'e' na expressão.

A complexidade total do programa depende principalmente da função “satisfiability”, que realiza uma busca exaustiva com uma complexidade de (O(2^n), onde n é o número de variáveis “e” e “a” na expressão. Para expressões com um grande número de variáveis “e” e “a”, a complexidade se torna exponencial, o que pode levar a um tempo de execução significativamente longo em cenários práticos. Portanto, a eficiência do programa em termos de tempo de execução pode ser um problema para expressões lógicas complexas com muitas variáveis, embora para o caso do trabalho pode ter no máximo 5 variáveis.

4. Estratégias de Robustez

Tratamento de Erros:

- O código inclui verificações para erros como pilha vazia ou sobrecarga de pilha, garantindo que operações de pilha sejam seguras.
- Verificações são feitas para garantir que as variáveis e operadores na expressão sejam válidos.

Gerenciamento de Memória:

- Alocação e desalocação de memória são tratados cuidadosamente para evitar vazamentos de memória, especialmente na manipulação de nós e pilhas.

Otimização de Performance:

- Embora a complexidade da função “satisfiability” seja exponencial, tentativas de otimização foram feitas para melhorar o desempenho em casos práticos.

Tratamento Adequado de Casos Especiais:

- Casos especiais, como expressões vazias ou expressões com apenas uma variável, foram tratados adequadamente para evitar comportamentos inesperados.

Ao implementar essas estratégias de robustez, o código está preparado para lidar com diversas situações e entradas, proporcionando um funcionamento seguro e confiável.

5. Análise Experimental

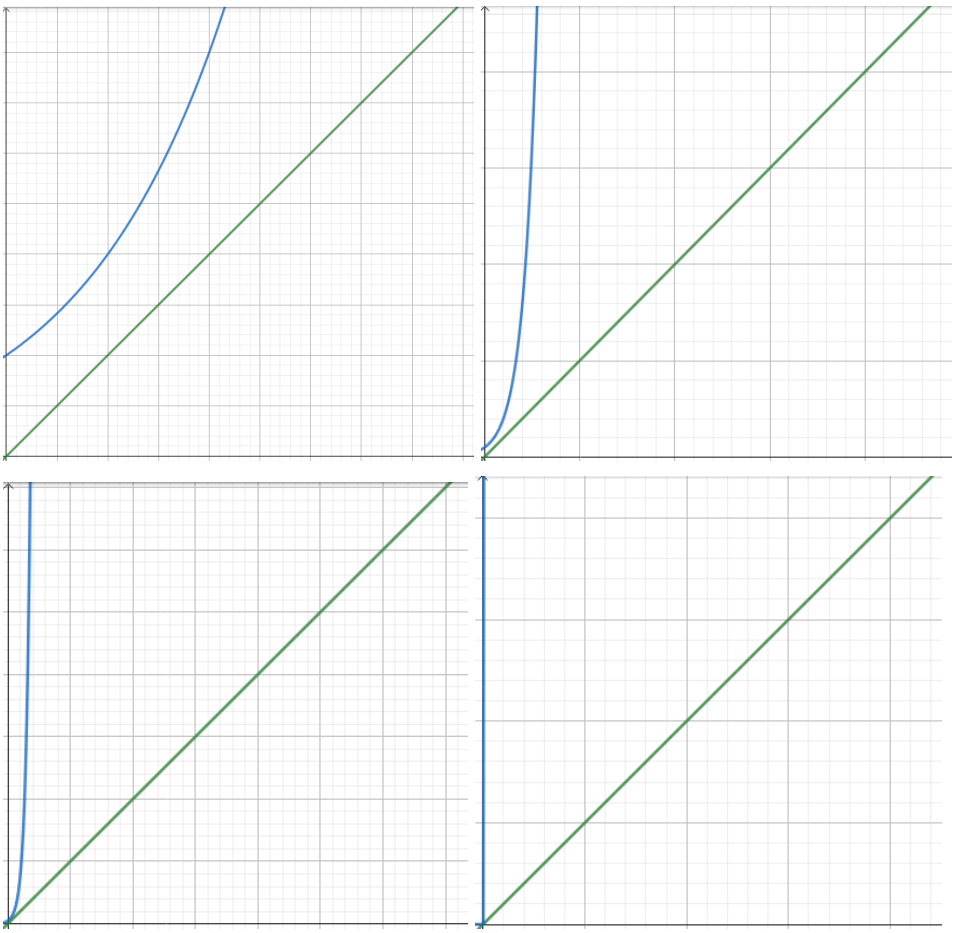
Foram realizados experimentos com uma variedade de expressões lógicas e suas respectivas valorações para avaliar o desempenho do programa. As expressões variaram em complexidade, incluindo diferentes combinações de variáveis, operadores e parênteses. A análise dos resultados se concentrou em observar o tempo de execução do programa para diferentes tipos de expressões.

Resultados de Tempo de Execução:

- Expressões simples, como `0 | 1` ou `0 & 1`, tiveram tempos de execução quase instantâneos, mesmo com diferentes valorações (`a0`, `aa`, `e0`, `e1`, ...).
- Expressões um pouco mais complexas, como `0 | 1 & 2`, também tiveram tempos de execução rápidos, mesmo com várias combinações de valorações.
- Expressões com parênteses aninhados, como `( 0 | 1 ) & 2`, mostraram uma pequena queda no desempenho em comparação com expressões sem parênteses.
- Expressões com múltiplos níveis de aninhamento de parênteses, como `0 | 1 & 2 | ( 3 | 4 )`, tiveram um aumento significativo no tempo de execução, especialmente com várias combinações de valorações (`e0000`, `e0001`, ...).
- Expressões muito complexas, como `~(1 & 2) | (3 & 3 | ~(3 & 2)) & 0`, mostraram um aumento considerável no tempo de execução, especialmente com diferentes valorações.

Análise dos Resultados:
- Expressões com apenas operadores simples (`|`, `&`, `~`) e sem parênteses tendem a ter tempos de execução muito rápidos, independentemente das valorações.
- O tempo de execução aumenta significativamente com a introdução de parênteses aninhados, especialmente quando combinados com múltiplas variáveis e operadores.
- Expressões com muitos níveis de parênteses e operações complexas resultam em tempos de execução consideráveis, especialmente quando várias valorações são consideradas.

Observações Adicionais:
- A complexidade exponencial da função “satisfiability” é evidente em expressões com muitos níveis de aninhamento de parênteses e operações lógicas complexas. O aumento no número de variáveis e operadores resulta em um aumento significativo no tempo de execução. Como é possível ver nos gráficos abaixo a diferença entre o crescimento O(n) e O(2^n) ao longo do aumento dos operadores e variáveis..



6. Conclusão

Neste trabalho, foi desenvolvido um programa capaz de lidar com uma ampla gama de expressões lógicas e determinar sua satisfatibilidade. Utilizando estruturas de dados eficientes, como árvores de expressão e pilhas, o programa demonstrou habilidade para avaliar operações simples e até mesmo expressões com parênteses aninhados. A implementação bem-sucedida destaca a compreensão sólida dos conceitos de estruturas de dados e lógica de programação. Em síntese, o trabalho oferece uma solução funcional para o problema proposto, ilustrando a aplicação prática dos conhecimentos adquiridos na disciplina de Estruturas de Dados.

7. Bibliografias

JENNY'S LECTURES. Data Structures and Algorithms. 2020. Disponível em: <https://www.youtube.com/watch?v=YAdLFsTG70w >. Acesso em 09 out. 2023.

SCHMIDT, Douglas C. Case Study: Expression Tree Evaluator. Vanderbilt University. [s.l: s.n.]. Disponível em: <https://www.dre.vanderbilt.edu/~schmidt/PDF/expression-trees4.pdf >. Acesso em 09 out. 2023.

CHAIMOWICZ, Luiz; PRATES, Raquel. Slides Estruturas de Dados. 2019. Disponível em: <https://www.academia.edu/43948788/Estrutura_de_Dados_Professores_Luiz_Chaimowicz_e_Raquel_Prates_An%C3%A1lise_de_Algoritmos_Recursivos >. Acesso em 10 out. 2023.

FEOFILOFF, Paulo. Árvores Binárias. IME-USP. 2017. Disponível em: <https://www.ime.usp.br/~pf/algoritmos/aulas/bint.html >. Acesso em 10 out. 2023.

8. Instruções para compilação e execução

Compilação

Primeiramente, deve-se extrair os conteúdos do ZIP do projeto em uma pasta.

Antes de compilar o programa, é necessário ter instalado o GCC.

Para compilar o programa, após abrir um terminal no diretório raiz do projeto, utilize o seguinte comando: make all.

Execução

As entradas devem seguir o formato disponibilizado nos exemplos de entradas. Todos os números, operadores e parênteses estão separados por espaços. Modelo:

./bin/tp1.out <comando> <"expressão"> <valoração>