UNIVERSIDADE FEDERAL DE MINAS GERAIS

Lucas Rafael Costa Santos

2021017723

TRABALHO PRÁTICO 1

Expressões Lógicas e Satisfabilidade

Belo Horizonte - MG, 15 de outubro de 2023

1. Introdução

Este trabalho tem como objetivo testar os conhecimentos do aluno quanto ao desenvolvimento e aplicação de estruturas de dados. Para isso foi proposto um problema que lida com exemplos de lógica, sendo eles a avaliação de expressões e a satisfatibilidade em expressões lógicas com quantificadores. O primeiro problema consiste na avaliação de uma fórmula lógica com variáveis binárias e operadores lógicos, seguindo uma ordem de precedência específica. Já o segundo problema aborda a questão de determinar se existe uma valoração que satisfaça uma expressão lógica, considerando quantificadores ∃ e ∀, com a restrição de que no máximo cinco variáveis podem ser quantificadas.

2. Método

2.1 Estrutura de dados

Para a avaliação das expressões a estrutura de dados utilizada foi a Pilha, pois é uma estrutura que se comporta bem e mantém um controle sobre a ordem de precedência dos operadores, garantindo que as operações sejam realizadas na sequência correta, de maneira que respeite as regras lógicas.

Uma Pilha é uma estrutura de dados que serve como uma coleção de elementos, e permite o acesso a somente um item de dados armazenado – o último item que foi inserido na estrutura (item do topo), o qual pode ser lido ou removido. Em outras palavras, o primeiro objeto a ser inserido na pilha é o último a ser removido. Essa política é conhecida pela sigla LIFO (*Last-In-First-Out*).

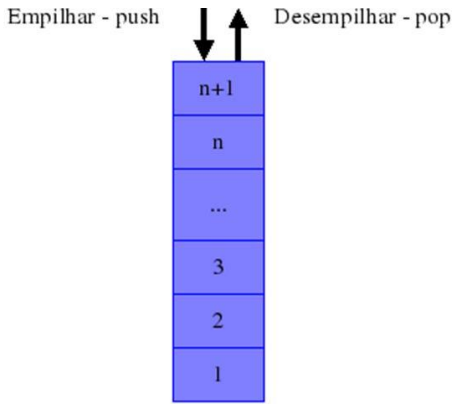

Figura 1 - Pilha

2.2 Classes

Foi desenvolvida a classe "Pilha" para a implementação de uma estrutura de pilha. Essa classe oferece funcionalidades básicas, incluindo um construtor e um destrutor. Além disso, a

classe conta com métodos para verificar se a pilha está vazia, adicionar ou remover elementos (por meio das funções "Empilha" e "Desempilha"), limpar a pilha, imprimir os elementos, obter o valor no topo da pilha e determinar o tamanho da pilha.

```cpp
#include <iostream>
#include <cstdlib>
#include <stdexcept>

//Estrutura Nó
struct Node {
    int data;
    Node* next;
};

//Classe Pilha
class Pilha {

    public:
        Pilha();
        ~Pilha();
        bool Vazia() const;
        void Empilha(int item);
        int Desempilha();
        void Limpa();
        void Imprime();
        int Topo() const;
        int Tamanho() const;

    private:
        Node* topo;
        int tamanho;
};
```

Figura 2 - Classe Pilha

2.2 Funções

Para analisar as expressões lógicas foram implementadas as seguintes funções:

```cpp
#include <string>
#include <iostream>
#include <cstdlib>
#include <cstring>

#include "pilha.hpp"

//Funções para avaliar uma expressão
bool ehOperador(char c);
int precedencia(char c);
std::string simplificarExpressao(const std::string& p);
int avaliarExpressao(const std::string& p, const std::string& s);
int avaliarSast(const std::string& p, const std::string& s);
void gerarCombinacoes(std::string& str, std::string* combinations, int& numCombinations);
void satisfatibilidade(const std::string& p, const std::string& s);
```

Figura 3 - Funções

- bool ehOperador: Função que verifica se o caractere é um operador (‘|’, ‘&’ ou ‘~’)

- int precedencia: Função que retorna a prioridade entre os operadores.

- std::string simplificarExpressao: Esta função simplifica uma expressão removendo espaços em branco e combinando múltiplas negações consecutivas.

- int avaliarExpressao: Esta função avalia uma expressão lógica dada (em notação infix) usando pilhas para manter controle das operações e operandos.

- int avaliarSast: Similar à função ‘avaliarExpressao’ esta função é específica para avaliar funções de satisfatibilidade.

- void gerarCombinacoes: Essa função gera todas as combinações possíveis de uma string 'str', onde 'a' e 'e' são substituídos por '0' e '1'. As combinações são armazenadas no array 'combinations'.

- void satisfatibilidade: Essa função lida com a satisfatibilidade de uma expressão lógica. Ela gera todas as combinações possíveis para a string 's', avalia a expressão lógica 'p' para cada combinação e determina se a expressão é satisfeita (retornando '1') ou insatisfeita ('0').

3. Análise de Complexidade

Para fazer uma análise de complexidade das funções apresentadas é importante observar o número de operações realizadas em relação ao tamanho das entradas. Sendo assim, vamos analisar cada função separadamente:

- Função ehOperador:

  Esta função realiza um número constante de comparações (3) para verificar se o caractere passado é um operador lógico. Portanto, a complexidade é constante, O(1).

- Função precedencia:

  Do mesmo modo que a função ehOperador, esta função realiza um número constante de comparações (3) para determinar a precedência de um operador lógico. Portanto, a complexidade também é O(1), constante.

- Função simplificarExpressao:

  Esta função percorre a string p uma vez, caracter por caracter, e realiza operações simples com complexidade constante. Portanto, a complexidade é linear em relação ao tamanho da string p, ou seja, O(n), onde n é o tamanho de p.

- Função avaliarExpressao:

  Esta função percorre a string expressaoSimplificada uma vez, caracter por caracter. Em cada iteração, pode realizar operações na pilha que têm complexidade constante. Portanto,

a complexidade é linear em relação ao tamanho da string expressaoSimplificada, ou seja, O(n), onde n é o tamanho de expressaoSimplificada.

- Função avaliarSast:

Similar à função avaliarExpressao, ou seja, O(n).

- Função gerarCombinacoes:

Esta função itera sobre as possíveis combinações de uma string str. O número de combinações geradas é 2^n, onde 'n' é o tamanho da string str. Portanto, a complexidade é exponencial (O(2^n)). Sendo assim, a complexidade é exponencial, O(2^n), onde n é o tamanho de str.

- Função satisfatibilidade:

A função satisfatibilidade gera todas as combinações possíveis de s usando a função gerarCombinacoes e, em seguida, avalia a expressão lógica para cada combinação usando avaliarSast. A complexidade depende da quantidade de combinações possíveis, que é 2^n, onde 'n' é o tamanho da string s. Portanto, a complexidade é exponencial (O(2^n)).

Portanto, é possível concluir que a complexidade do código varia de linear (O(n)) para funções que percorrem ou avaliam a expressão simplificada a exponencial (O(2^n)) para a geração de todas as combinações possíveis na função gerarCombinacoes e satisfatibilidade.

Agora, vamos discutir o que aconteceria com o algoritmo de satisfatibilidade caso a restrição de apenas 5 variáveis quantificadas fosse removida:

Sem Restrição de Variáveis Quantificadas: Se a restrição de apenas 5 variáveis quantificadas for removida, isso significa que o número de variáveis na string de valores s pode ser qualquer número, não apenas 5. Nesse caso, o número de combinações possíveis de valores aumentaria exponencialmente com o número de variáveis.

Complexidade de Tempo: A complexidade de tempo da função satisfatibilidade é O(2^n), onde 'n' é o tamanho da string de valores s. Se o número de variáveis for grande, a complexidade de tempo aumentaria significativamente, tornando o algoritmo muito mais lento.

Complexidade de Espaço: Além disso, a complexidade de espaço também aumentaria, uma vez que as combinações possíveis de valores para cada variável precisam ser armazenadas temporariamente na memória. Isso pode levar a um uso significativo de memória em casos com muitas variáveis.

4. Estratégias de Robustez

A fim de assegurar a robustez do programa, foi utilizada a biblioteca stdexcept, resultando na implementação de diversas estratégias de robustez no código, entre elas:

- Validação de entrada:

O código verifica o número correto de argumentos de linha de comando (argc) e exibe mensagens de erro se o número de argumentos estiver incorreto.

Ele verifica se a opção fornecida (-a ou -s) é válida e se as expressões, valores e variáveis estão vazios ou fora de alcance, exibindo mensagens de erro apropriadas.

```
//Argumentos: -a <expressao_logica> <string_valores>
if (argc != 4) {
    std::cerr << "Uso: " << argv[0] <<
" <opcao> <expressao_logica> <string_valores>" << std::endl;
    return 1;
}
```

```
if (opcao != "-a" && opcao != "-s") {
    std::cerr << "Opcao invalida: " << opcao << std::endl;
    return 1;
}

if (expressao.empty()) {
    std::cerr << "Expressao invalida: " << expressao << std::endl;
    return 1;
}

if (valores.empty()) {
    std::cerr << "Valores invalidos: " << valores << std::endl;
    return 1;
}
```

- Tratamento de exceções:

O código usa exceções (por exemplo, std::runtime_error) para lidar com erros, como pilha vazia ou variáveis fora de alcance.

Ele captura essas exceções e exibe mensagens de erro informativas.

```
else if (isdigit(token)) {
    std::string::size_type variavel = token - '0';
    if (variavel < 0 || variavel >= s.length()) {
        throw std::runtime_error(
"Variável fora de alcance na string de valores.");
    }
```

- Tratamento de recursos limitados:

O código aloca e libera recursos (memória) de forma adequada ao trabalhar com pilhas.

5. Análise Experimental

Ao utilizar o comando ‘make mem’ o Valgrind é executado para as seguintes entradas: -a "((0 & 1 | 2) & 3) & (4 | 5) & (6 | 7) & ~ 8 | ~ ~ 9" "0101100111". Como resultado disso, temos:

```
valgrind --leak-check=full ./bin/tp1.out -a "((0 & 1 | 2) & 3) & (4 | 5) & (6 | 7) & ~ 8 | ~ ~ 9" "0101100111"
==1832== Memcheck, a memory error detector
==1832== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==1832== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==1832== Command: ./bin/tp1.out -a ((0\ &\ 1\ |\ 2)\ &\ 3)\ &\ (4\ |\ 5)\ &\ (6\ |\ 7)\ &\ ~\ 8\ |\ ~\ ~\ 9 0101100111
==1832==
1
==1832==
==1832== HEAP SUMMARY:
==1832==     in use at exit: 0 bytes in 0 blocks
==1832==   total heap usage: 38 allocs, 38 frees, 74,355 bytes allocated
==1832==
==1832== All heap blocks were freed -- no leaks are possible
==1832==
==1832== For lists of detected and suppressed errors, rerun with: -s
==1832== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Figura 4 - Valgrind

Isso significa que o no final da execução do programa, não há mais memória alocada no heap e que todos os blocos de memória alocados durante a execução do programa foram liberados corretamente. Ou seja, que não há vazamentos de memória. Além disso, Error Summary indica que o Valgrind não detectou nenhum problema de alocação de memória.

Utilizando o comando ‘make callgrind’ o Callgrind é executado para as mesmas entradas. E como resultado disso temos a criação de um arquivo callgrind.out.2144 e a seguinte saída no terminal:

```
valgrind --tool=callgrind ./bin/tp1.out -a "((0 & 1 | 2) & 3) & (4 | 5) & (6 | 7) & ~ 8 | ~ ~ 9 " "0101100111"
==2144== Callgrind, a call-graph generating cache profiler
==2144== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==2144== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2144== Command: ./bin/tp1.out -a ((0\ &\ 1\ |\ 2)\ &\ 3)\ &\ (4\ |\ 5)\ &\ (6\ |\ 7)\ &\ ~\ 8\ |\ ~\ ~\ 9\  0101100111
==2144==
==2144== For interactive control, run 'callgrind_control -h'.
1
==2144==
==2144== Events    : Ir
==2144== Collected : 2343252
==2144==
==2144== I   refs:      2,343,252
```

Figura 5 – Callgrind

E o seguinte conteúdo na parte inicial do arquivo:

```
# callgrind format
version: 1
creator: callgrind-3.18.1
pid: 2144
cmd:  ./bin/tp1.out -a ((0 & 1 | 2) & 3) & (4 | 5) & (6 | 7) & ~ 8 | ~ ~ 9  0101100111
part: 1


desc: I1 cache:
desc: D1 cache:
desc: LL cache:

desc: Timerange: Basic block 0 - 372935
desc: Trigger: Program termination

positions: line
events: Ir
summary: 2343252
```

Figura 6 - Callgrind

Com isso, vemos que a saída mostra informações gerais sobre a execução do programa, incluindo o número de instruções referenciadas (I refs), que foi de 2.343.252 e que o intervalo de tempo durante o qual o perfil foi coletado abrange o "Basic block 0 - 372935."

Por fim, ao utilizar o comando ‘make cachegrind’ o Cachegrind é executado para as mesmas entradas: -a "((0 & 1 | 2) & 3) & (4 | 5) & (6 | 7) & ~ 8 | ~ ~ 9" "0101100111". E como resutado disso temos a criação de um arquivo cachegrind.out.2036 e a saída no terminal:

```
valgrind --tool=cachegrind ./bin/tp1.out -a "((0 & 1 | 2) & 3) & (4 | 5) & (6 | 7) & ~ 8 | ~ ~ 9 " "0101100111"
==2036== Cachegrind, a cache and branch-prediction profiler
==2036== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==2036== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==2036== Command: ./bin/tp1.out -a ((0\ &\ 1\ |\ 2)\ &\ 3)\ &\ (4\ |\ 5)\ &\ (6\ |\ 7)\ &\ ~\ 8\ |\ ~\ ~\ 9\  0101100111
==2036==
--2036-- warning: L3 cache found, using its data for the LL simulation.
1
==2036==
==2036== I   refs:      2,344,249
==2036== I1  misses:        2,203
==2036== LLi misses:        2,089
==2036== I1  miss rate:      0.09%
==2036== LLi miss rate:      0.09%
==2036==
==2036== D   refs:        776,944  (569,134 rd   + 207,810 wr)
==2036== D1  misses:       16,333  ( 13,852 rd   +   2,481 wr)
==2036== LLd misses:        9,310  (  7,713 rd   +   1,597 wr)
==2036== D1  miss rate:       2.1% (    2.4%     +     1.2%  )
==2036== LLd miss rate:       1.2% (    1.4%     +     0.8%  )
==2036==
==2036== LL refs:          18,536  ( 16,055 rd   +   2,481 wr)
==2036== LL misses:        11,399  (  9,802 rd   +   1,597 wr)
==2036== LL miss rate:        0.4% (    0.3%     +     0.8%  )
```

Figura 7 - Cachegrind

Analisando a saída é possível notar alguns tópicos:

- Referências à Memória:

    O programa realizou um total de 2.344.249 referências à memória.

- Cache de Instruções (I):

    O cache de instruções (I) teve 2.203 misses (quando as instruções solicitadas não estavam no cache).

    A taxa de misses no cache de instruções foi de 0.09%.

- Cache de Dados (D):

    O cache de dados (D) teve 16.333 misses.

    A taxa de misses no cache de dados foi de 2.1%.

- Cache de Dados Nível 1 (D1):

    O cache de dados de nível 1 (D1) teve 13.852 reads (leituras) e 2.481 writes (escritas).

    A taxa de misses no cache de dados de nível 1 foi de 2.4% para leituras e 1.2% para escritas.

- Cache de Dados Nível de Linha (LLd):

    O cache de dados de nível de linha (LLd) teve 7.713 reads e 1.597 writes.

    A taxa de misses no cache de dados de nível de linha foi de 1.4% para leituras e 0.8% para escritas.

- Cache Total (LL):

    O cache total (LL) teve 16.055 reads e 2.481 writes.

    A taxa de misses no cache total foi de 0.3% para leituras e 0.8% para escritas.

A partir dessas informações, é possível concluir que o programa apresentou uma taxa relativamente baixa de misses de cache (em torno de 2.1% para D1 e 1.2% para LLd) para operações de leitura e uma taxa ainda menor de misses de cache (0.4% para LL) no geral.

Isso sugere que o programa tem um bom comportamento em relação ao uso de cache, o que pode ajudar a melhorar o desempenho, reduzindo os acessos à memória principal.

6. Conclusão

Neste trabalho, abordamos dois problemas fundamentais da lógica: a avaliação de expressões lógicas e o problema da satisfabilidade com quantificadores. Desenvolvemos então, soluções para ambos os problemas, explorando técnicas e estruturas de dados específicas, como o uso de pilhas.

Este projeto proporcionou-nos uma compreensão mais profunda da lógica computacional e das estruturas de dados necessárias para resolver problemas lógicos complexos. Além disso, demonstrou a importância da análise de complexidade do código, destacando como entradas mais extensas podem impactar o desempenho de um programa. Também aprendemos como a análise experimental pode ser uma ferramenta diferencial para verificar a consistência do código e identificar possíveis problemas de alocação de memória.

7. Compilação e execução

Para compilar o programa, basta utilizar o comando ‘make’. Para executá-lo acesse o executável gerado com o comando ‘./bin/tp1.out’ com os argumentos ‘-a’ (caso queira avaliar uma expressão) ou ‘-s’ (caso queira verificar a satisfatibilidade) seguido por ‘”expressão”’ e ‘valoração’:

‘./bin/tp1.out’ -a “expressão” valoração

‘./bin/tp1.out’ -s “expressão” valoração

8. Bibliografia

Grancursosonline. Estrutura de Dados - Pilhas. Disponível em: https://blog.grancursosonline.com.br/estrutura-de-dados-pilhas/. Acesso em: 12 de outubro de 2023.

Boson Treinamentos. Estruturas de Dados - Pilhas. Disponível em: http://www.bosontreinamentos.com.br/estruturas-de-dados/estruturas-de-dados-pilhas/. Acesso em: 12 de outubro de 2023.

Instituto de Matemática e Estatística da Universidade de São Paulo. Pilha. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/pilha.html. Acesso em: 12 de outubro de 2023.

Slides da Disciplina.