UNIVERSIDADE FEDERAL DE MINAS GERAIS
Nome: João Marcos Ribeiro Tolentino - 2021049536

Avaliação de expressão e satisfabilidade

1. Introdução

O problema proposto foi de trabalhar com expressões binárias e operadores lógicos "E"(∧), "OU"(∨) e "NÃO"(¬). Nesse sentido é passado na entrada a expressão e o valor: Exemplo: ./tp1.out -a "0 | 1" 10. Avaliado a expressão de acordo com a função desejada (avaliação de expressão ou satisfabilidade)

Sendo assim, a avaliação de expressão vai literalmente avaliar a expressão passada, substituindo os valores passados pelas variáveis respectivamente. Lembrando que é determinante e importante a ordem de precedência dos operadores. Sendo assim uma expressão lógica ϕ(x1, x2, . . . , xn) onde x1, x2, . . . , xn são as variáveis recebidas e analisadas para retornar de acordo com a análise do programa.

Em síntese, a função de satisfabilidade é diferente agora o importante é verificar se existe uma valoração que faz o resultado final ser 1. Caso não exista valoração válida retorna 0 e no caso de o valor da variável ser irrelevante ele retorna “a” no lugar da variável.

2. Implementação

O programa foi desenvolvido na linguagem C++, compilada pelo compilador G++ da GNU Compiler Collection.

3. Estrutura de Dados

Neste código, são usadas estruturas de dados mais básicas, como matrizes de caracteres (strings) e variáveis inteiras. Vou explicar como essas estruturas de dados são usadas no código:

const char***: (variáveis: exp e valor)**

int: resultadoFinal, tam, valor0, valor1, prox_valor, contagemParenteses, i, k, r, r2 e outras variáveis no código são do tipo int.
string: No código, copiaExp e copiaValor são objetos string que são usados para fazer cópias e manipulações das strings originais exp e valor. Isso facilita a substituição de caracteres e a construção da string de resultado.

Basicamente, as estruturas de dados usadas neste código são bastante simples. Os arrays de caracteres (strings) são usados para representar as expressões lógicas e atribuições de variáveis, e variáveis inteiras são usadas para armazenar resultados intermediários e finais da avaliação da expressão. O uso de strings é especialmente útil nesse caso para manipular e modificar as expressões lógicas e os valores durante o processo de avaliação.

4 - Funções:

Avaliar Expressão:

Esta função é responsável por avaliar uma expressão lógica dada em formato de string. Ela recebe dois argumentos: a expressão lógica (`exp`) e os valores das variáveis (`valor`) usados na expressão.

A função percorre a expressão, realiza operações lógicas (AND, OR e NOT) e retorna o resultado final como um inteiro (0 para falso, 1 para verdadeiro).

Satisfabilidade: Esta função é responsável por verificar a satisfação de uma expressão lógica, ou seja, se existe uma atribuição de valores às variáveis que torna a expressão verdadeira.

Ela recebe a mesma expressão lógica (`exp`) e os valores iniciais das variáveis (`valor`) como argumentos.
A função faz uma primeira avaliação da expressão e, se necessário, modifica temporariamente os valores para verificar a satisfabilidade.
Ela retorna uma string indicando se a expressão é satisfatível (começando com '1') ou insatisfatível (começando com '0') e, se possível, com a atribuição de valores que a torna verdadeira.

5 - Análise de Complexidade

```
// Função para verificar satisfabilidade da expressão
string satisfabilidade(const char *exp, const char *valor)
{
    int tam = strlen(exp);
    int tamanhoValor = strlen(valor);

    string copiaExp(exp);
    string copiaValor(valor);
    int k = 0;

    for (int i = 0; i < tamanhoValor; i++)
    {
        if (copiaValor[i] == 'e')
        {
            k = i;
            copiaValor[i] = '1';
        }
    }

    // Substituir dígitos na expressão pelos valores correspondentes
    for (int i = 0; i < tam; i++)
    {
        if (copiaExp[i] == '0')
        {
            copiaExp[i] = valor[0];
        }
        else if (copiaExp[i] == '1')
        {
            copiaExp[i] = valor[1];
        }
        else if (copiaExp[i] == '2')
        {
            copiaExp[i] = valor[2];
        }
    }

    // Avaliar a expressão
    int r = avaliarExpressao(copiaExp.c_str(), copiaValor.c_str());

    // Reverter a substituição de 'e' por '0' para a segunda avaliação
    for (int i = 0; i < tamanhoValor; i++)
    {
        if (i == k)
        {
            copiaValor[i] = '0';
        }
    }

    // Avaliar a expressão novamente
    int r2 = avaliarExpressao(copiaExp.c_str(), copiaValor.c_str());
    string resultado;

    // Verificar satisfabilidade
    if (r == 1 || r2 == 1)
    {
        for (int i = 0; i < tamanhoValor; i++)
        {
            if (i == k)
            {
                if (r2 == 0)
                {
                    copiaValor[i] = '1';
                }
                else
                {
                    copiaValor[i] = 'a';
                }
            }
        }
        resultado = "1 " + string(copiaValor);
```

```cpp
// Função para avaliar a expressão lógica
int avaliarExpressao(const char *exp, const char *valor) {
    int resultadoFinal = 0;
    int tam = strlen(exp);
        int valor0 = valor[0] - '0';
        int valor1 = valor[1] - '0';

    for (int i = 0; i < tam; i++) {
        if (exp[i] == ' ') {
            continue;
        }
        if (exp[i] == '0') {
            resultadoFinal = valor0;
        } else if (exp[i] == '1') {
            resultadoFinal = valor1;
        } else if (exp[i] == '~') {
            i++;
            int prox_valor = valor[exp[i] - '0'] - '0';
            prox_valor = !prox_valor;
            resultadoFinal = prox_valor;
        } else if (exp[i] == '&') {
            i++;
            if (exp[i] == ' ') {
                i++;
            }
            int prox_valor = valor[exp[i] - '0'] - '0';
            resultadoFinal = resultadoFinal && prox_valor;
        } else if (exp[i] == '|') {
            i++;
            if (exp[i] == ' ') {
                i++;
            }
            int prox_valor = valor[exp[i] - '0'] - '0';
            resultadoFinal = resultadoFinal || prox_valor;
        } else if (exp[i] == '(') {
            int contagemParenteses = 1;
            i++;
            while (contagemParenteses > 0) {
                if (exp[i] == '(') {
                    contagemParenteses++;
                } else if (exp[i] == ')') {
                    contagemParenteses--;
                }
                if (contagemParenteses < 0) {
                // Parênteses desbalanceados
                throw std::runtime_error("Parênteses desbalanceados");

                // cout << "Parenteses desbalanceados" << endl;
                return false;
            }
                i++;
            }
        }
    }


    return resultadoFinal;
```

Funções (avaliarExpressão e satisfabilidade)

AvaliarExpressao: As operações dentro dos condicionais (como AND, OR, NOT) e as operações de atribuição são operações de tempo constante O(1).

Quando a função encontra parênteses, ela pode precisar realizar um loop interno para lidar com as expressões dentro dos parênteses. Se o número de parênteses aninhados for limitado, isso ainda será uma operação O(n).

*** Portanto, a complexidade geral da função é O(n).**

Satisfabilidade: A primeira chamada para `avaliarExpressao` é uma operação O(n), onde 'n' é o comprimento da expressão lógica. A modificação temporária nos valores para verificar a satisfabilidade também é uma operação O(n).

Portanto, a complexidade geral dessa função é: O(n)

Validação da Expressão Lógica: Na função `avaliarExpressao`**,** o código verifica vários elementos da expressão lógica, como parênteses desbalanceados. Se encontrar parênteses desbalanceados, lança uma exceção (`throw std::runtime_error`) para sinalizar o erro.

6 - Estratégia e Robustez:

Evitar Comportamento Indefinido: O código lida com casos em que os valores da expressão podem exceder os limites esperados, garantindo que a avaliação da expressão seja definida e não produza resultados indefinidos ou imprevisíveis.

Exceções para Erros Críticos: Em caso de erros críticos, como parênteses desbalanceados, o código lança exceções, fornecendo informações detalhadas sobre o erro. Isso ajuda a interromper a execução e fornecer feedback sobre o erro em vez de permitir que o programa prossiga com dados incorretos.

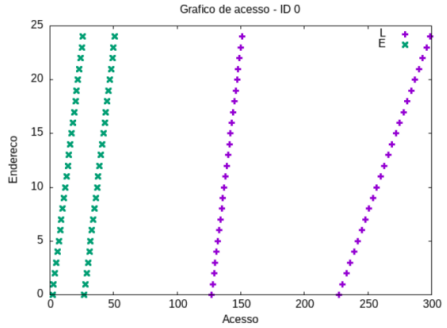
Tratamento de Entrada do Usuário: O programa lida com entrada do usuário por meio dos argumentos da linha de comando. Ele verifica se a entrada está de acordo com as opções disponíveis (-a e -s) e fornece uma mensagem de uso clara em caso de entrada incorreta.

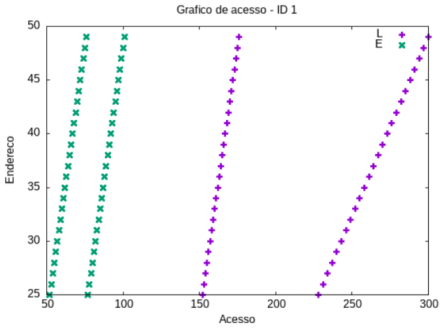
```
joao@joao-IdeaPad-3-15ALC6:~/Área de Trabalho/UFMG/Estrutura_de_dados/TP/src$ valgrind --tool=callgrind ./ma
==24834== Callgrind, a call-graph generating cache profiler
==24834== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==24834== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==24834== Command: ./main
==24834==
==24834== For interactive control, run 'callgrind_control -h'.
Uso: ./main -a/-s <expressao> <valor>
==24834==
==24834== Events    : Ir
==24834== Collected : 2331700
==24834==
==24834== I   refs:      2,331,700
joao@joao-IdeaPad-3-15ALC6:~/Área de Trabalho/UFMG/Estrutura_de_dados/TP/src$ kcachegrind callgrind.out.*
Selected  "0x00000000000202b0"
CallGraphView::refresh
CallGraphView::refresh: Starting process 0x5604711bf820, 'dot -Tplain'
CallGraphView::readDotOutput: QProcess 0x5604711bf820
CallGraphView::dotExited: QProcess 0x5604711bf820
Selected  "_dl_start"
CallGraphView::refresh
CallGraphView::refresh: Starting process 0x5604718bc900, 'dot -Tplain'
CallGraphView::readDotOutput: QProcess 0x5604718bc900
CallGraphView::dotExited: QProcess 0x5604718bc900
Selected  "_dl_relocate_object"
CallGraphView::refresh
CallGraphView::refresh: Starting process 0x5604718aca70, 'dot -Tplain'
CallGraphView::readDotOutput: QProcess 0x5604718aca70
CallGraphView::dotExited: QProcess 0x5604718aca70
Selected  "0x00000000000202b0"
CallGraphView::refresh
CallGraphView::refresh: Starting process 0x5604721b5fc0, 'dot -Tplain'
CallGraphView::readDotOutput: QProcess 0x5604721b5fc0
CallGraphView::dotExited: QProcess 0x5604721b5fc0
Selected  "(below main)"
CallGraphView::refresh
CallGraphView::refresh: Starting process 0x56047218de40, 'dot -Tplain'
CallGraphView::readDotOutput: QProcess 0x56047218de40
```

Informações sobre uso de dadose CPU (Valgrind)

`I refs`: 2,331,700 indica que o programa executou um total de 2.331.700 instruções.

Gráficos (mapas de acesso):





7. Conclusão

Dessa maneira, conclui-se que o trabalho apresenta de forma completa em relação a análise de complexidade, experimental, estratégias utilizadas, entendimento do código e etc. O esperado foi concluído com sucesso apesar de algumas dificuldades no decorrer da sua implementação. Portanto, o programa está bastante eficiente em relação ao desempenho, bom uso de boas práticas (como comentários) para modificações e leitura do código. A estrutura do código, bem como suas funções bem definidas e organizadas tornam o código mais légivel e de fácil manutenção.

A análise de desempenho usando o Valgrind revelou que o programa executou um total de 2.331.700 instruções, demonstrando sua eficiência em termos de recursos computacionais. Portanto , este projeto demonstra a importância do uso de boas práticas de programação, análise de complexidade e ferramentas de análise de desempenho para criar programas eficientes e robustos.

8. Instruções para compilar e executar o programa:

- Rodar o: “make all”
- ir dentro da pasta bin e executar:

```
./tp1.out -opcao “expressão” valor
```