Prof. Wagner Meira Jr. DCC / ICEx /UFMG

DCC205 -Estrutura de Dados 2023.2

Trabalho Prático 1 – Expressões lógicas e satisfatibilidade

Lucca Alvarenga de Magalhães Pinto – lucca.alvarenga@dcc.ufmg.br

Matrícula: 2021036736

1 INTRODUÇÃO

O trabalho consiste na implementação de um programa para analisar o resultado de fórmulas booleanas e é divido em duas partes, a primeira relacionada a avaliação de expressões lógicas, e a segunda referente a verificar se existe alguma ordem de valores binários em que a expressão lógica seja verdadeira.

2 MÉTODO

2.1 Configurações da máquina

O código foi executado nas seguintes configurações de ambiente:

- Sistema operacional: Ubuntu 20.04.6 LTS (Focal Fossa)
- Linguagem de programação: C++
- Compilador: gcc (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0
- Processador: 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GHz
- Memória: 15Gb

2.2 Estrutura de dados

Para resolução dos problemas foi adotado duas estruturas de dados, uma *Pilha* que armazena um *struct Node* para avaliar a expressão lógica de acordo com uma valoração, e um *Vector* de *Tuplas* para armazenar todas as possíveis combinações da string binária de valoração de acordo com os quantificadores dados.

Em vez de optar por estruturas de dados generalizadas que poderiam se relacionar com qualquer tipo de dado armazenado, essas estruturas foram feitas de forma especifica a solução do trabalho com o objetivo de otimizar os

problemas em questão. Foi definido um limite máximo para a quantidade de elementos que a pilha pode conter de acordo com o enunciado do projeto.

No código fornecido, o uso de uma pilha é justificado pela necessidade de rastrear e manipular a ordem de operações e operandos em expressões infixas e postfixas, bem como na avaliação de expressões lógicas. A pilha é a escolha apropriada para essas tarefas devido à sua natureza de "last-in, first-out" (LIFO), que permite controlar a precedência dos operadores e os escopos das expressões. Nesse contexto, não foi feito uma lista encadeada pois ela não ofereceria a mesma facilidade para adicionar e remover elementos na parte superior, além de que a lista possui mais complexidade ao percorrer ela.

Já o uso do vetor foi pela necessidade de armazenar e manipular uma lista de combinações de valores, além de realizar operações de inserção e remoção de elementos. Um vetor é uma escolha natural para armazenar uma lista ordenada de elementos, pois permite um acesso eficiente aos elementos por índice, bem como inserções e remoções de elementos.

2.2.1 Avaliação de expressões lógicas

Dado uma expressão lógica com variáveis numeradas e os valores delas, a avalição foi feita por 4 funções principais: a função “*avaliar”* que utiliza 3 funções em sequência para tirar os espaços em branco da fórmula, substituir os valores nela, e converte a expressão infixa para posfixa para então avaliar a fórmula.

A função “*tirar_espacos*” percorre pelos caracteres da string e verifica se não é um espaço em branco. A função “*substituir_valores*” troca cada dígito referente a cada variável da string fórmula por seu valor correspondente na string de valoração. Já a função “*pos_fixado*” converte uma expressão lógica na notação “operadores entre operandos” para “operadores após operandos” com o auxílio de uma “*Pilha*”, em resumo, a ideia é utilizar a pilha para manter o controle dos operadores e garantir que a ordem dele seja preservada de acordo com sua precedência.

2.2.1 Satisfabilidade

Neste problema, as funções principais são “satisfaz”, “*gerar_combinacoes*” e “*compara_str*”. Com a entrada de uma expressão lógica e uma string binária de valores com quantificadores matemáticos (“para todo” ou “existe”), a primeira função chama a segunda para criar e retornar todas as possíveis combinações desses quantificadores, ou seja, troca cada letra na string de valores por 0 e 1. Como o enunciado do trabalho define um máximo de quantificadores como 5, existira no máximo 32 combinações.

Como essas possibilidades foram implementadas como Tuplas(string combinação, int flag), depois de criar o vetor de cominações é avaliado cada uma pela função “*avaliar”* da primeira parte*,* e armazenado o resultado em seu elemento flag referente. Dessa forma, é criado um vetor com todas as possíveis formas de valoração e sua reposta na fórmula booleana.

Por último, as tuplas do vetor são avaliadas da seguinte forma: a string da tupla da posição 1 é compara com a da tupla da posição 2, é criado uma nova tupla com base nessa comparação, a tupla da posição 1 se torna a nova tupla, a tupla da posição 2 é deletada, depois é comparado a tupla da posição 3 com a da posição 4, e assim sucessivamente. Quando o tamanho do vetor for igual a esse ciclo é terminado.

Com isso, a string da tupla da posição ‘i’ é comparada com da posição ‘i+1’, e é verificado qual o resultado das duas, com isso é gerado uma nova tupla com base em 3 possiblidades:

- Se as duas combinações forem verdadeiras, então é chamada a função “*comparar_str*” para verificar quais índices possuem caracteres diferentes entre as duas strings:
    - A nova string será aquela que tenha um quantificador “para todo”. Caso não tenha em nehuma das duas ela pode ser qualquer uma, mas pra deixar um padrão caso não tenha ela será a primeira string.
    - Se uma string tiver um “para todo” e a outra tiver “1” no mesmo índice, significa que aquela variável é verdadeira sempre, logo, a nova string terá aquele índice como “a”.

- Se a string não tiver feito o passo acima, então ela verifica se tem algum índice em que as duas strings tenham um caractere diferente entre si, se tiver significa que aquela variável é verdadeira sempre, logo, a nova string terá aquele índice como “a” também.
- No final é criado uma Tupla que tenha como combinação essa nova string.

- Se as duas combinações forem falsas, então é criado uma tupla com combinação 0 para indicar que nenhuma possiblidade dá uma resposta correta na fórmula.
- Se uma combinação for verdadeira e a outra falsa, então cria uma Tupla com a que deu verdadeira na expressão.

Ao passar por uma dessas opções, é inserido essa nova tupla na posição ‘i’, o que irá fazer todo os elementos do vetor na posição ‘i’ e ‘> i’ irem para a próxima posição da direita do vetor. Assim, é deletado a posição ‘i+1’ e ‘i+2’ do vetor, esses passos ocorrem até o vetor ter tamanho que será a resposta de satisfabilidade.

3 ANÁLISE DE COMPLEXIDADE

Abaixo é identificando a complexidades das principais funções do projeto para cada arquivo:

1. **Utils.hpp**

1.1. Tirar_espacos: A função percorrer a string de entrada, caractere por caractere, uma única vez. Para cada caractere ela realiza uma operação simples de verificar se é um espaço em branco. A complexidade de tempo é linear e é O(n), onde ‘n’ é o comprimento da string de entrada.

1.2. Substituir_valores: A ideia é semelhante a função acima, mas a operação é verificar se o caractere é um número. A complexidade de tempo é linear e é O(n), onde ‘n’ é o comprimento da string de entrada.

1.3. Ordem: A função consiste em uma série de instruções condicionais, onde ela verifica o valor do caractere de entrada c e atribui um valor inteiro correspondente. Como essas verificações são executadas uma única vez, a complexidade de tempo é constante e é O(1).

1.4. Pos_fixado: A função percorre a string de entrada uma única vez, caractere por caractere. Em cada passo do loop, a função executa operações condicionais para decidir como manipular o caractere atual e atualiza a string result. A função também chama métodos como ‘p.insert()’, ‘p.pop()’, ‘p.get_topo()’, e ‘ordem()’, cuja complexidade de tempo depende do número de elementos na pilha. Portanto, em média a complexidade de tempo é dominada pela iteração da string de entrada e é linear, ou seja, O(n), onde 'n' é o comprimento da string formula.

1.5. Valor_operacao_logica: A função consiste em uma série de instruções condicionais que verificam o valor do operador lógico e executam a operação lógica correspondente. Independentemente do operador, apenas uma operação é executada, e isso é feito em tempo constante. Portanto, a complexidade de tempo da função é constante, O(1), porque o tempo necessário para executar a função não depende do tamanho dos valores booleanos ou do operador.

1.6. Avaliar: A função percorre a string formula, que representa a expressão lógica na forma postfix, uma vez. Dentro do loop for, a função realiza uma série de operações condicionais, incluindo inserções e remoções de elementos da pilha, além de chamadas à função ‘*Valor_operacao_logica’*. As operações na pilha, como inserir e remover elementos, bem como as operações condicionais, são todas executadas em tempo constante. Portanto, a complexidade de tempo da função é linear em relação ao tamanho da string formula, que é O(n), onde 'n' é o comprimento da string formula. No pior caso, a pilha pode conter todos os operadores e operandos da expressão, resultando em uma complexidade de espaço O(n), onde 'n' é o comprimento da string formula.

1.7. Gerar_Combinacoes: A função é implementada de forma recursiva, onde ‘index’ é o parâmetro que controla a recursão. A cada chamada recursiva, a função verifica o valor de ‘index’ e toma uma decisão com base no valor de ‘str[index]’. Cada chamada recursiva adiciona 1 ao valor de ‘index’ e prossegue para a próxima chamada. O número total de chamadas recursivas é controlado pelo comprimento da string ‘str’, pois ‘index’ aumenta de 1 em 1 até que index seja igual ao comprimento da string. Portanto, a complexidade de tempo é O(2^n), onde 'n' é o comprimento da string ‘str’. Isso ocorre porque a função realiza duas chamadas recursivas para cada caractere 'a' ou 'e' na string, resultando em um número exponencial de chamadas recursivas.

Como pode ter no máximo 5 caracteres na valoração, então o número máximo de chamadas recursivas será limitado a 2^5 = 32 chamadas.

1.8. Comparar_str: A complexidade de tempo da função é linear, uma vez que ela percorre as duas strings de entrada uma vez para determinar se 'a' ou '1' estão presentes. A complexidade de espaço é também linear, pois a string ‘*result*’ é criada e modificada com base no comprimento das strings de entrada.

1.9. Satisfaz: No geral, a complexidade de tempo da função satisfaz é dominada pela chamada à função gerar_combinacoes, que é O(2^n), onde 'n' é o comprimento da string valoração. As outras operações dentro do loop ‘while’ têm um impacto menor no tempo total, mas contribuem para tornar a complexidade global da função mais complexa. A complexidade de espaço é também O(2^n) devido ao espaço ocupado pelo vetor.

2. Pilha

2.1. Insert: A função insert é responsável por adicionar um elemento no topo da pilha. A complexidade de tempo é dominada pelas operações executadas dentro da função, que incluem: Verificação se a pilha ultrapassou o limite (O(1)). Alocação de memória para o novo nó (O(1)). Atualização do ponteiro _topo (O(1)). Incremento do tamanho _tamanho (O(1)). Portanto, a complexidade de tempo da função insert é O(1) no pior caso, pois as operações são executadas em tempo constante

2.2. Pop: A função pop é responsável por remover o elemento no topo da pilha. A complexidade de tempo é dominada pelas operações executadas dentro da função, que incluem: Verificação se a pilha está vazia (O(1)). Atualização do ponteiro _topo (O(1)). Decremento do tamanho _tamanho (O(1)). Liberação da memória ocupada pelo nó removido (O(1)). Portanto, a complexidade de tempo da função pop é O(1) no pior caso, pois as operações são executadas em tempo constante.

2.3. Print: A função imprimir é responsável por criar uma representação em forma de string dos elementos da pilha. Ela percorre a pilha para criar a representação. A complexidade de tempo é O(n), onde 'n' é o número de elementos na pilha, pois a função percorre cada elemento da pilha uma vez para construir a representação da string. Portanto, a complexidade de tempo da função imprimir é linear em relação ao número de elementos na pilha.

3. Vector

3.1. Insere_fim: Complexidade de tempo: O(1). A função insere um elemento no final da lista vinculada. As operações executadas são de tempo constante, independentemente do tamanho da lista

3.2. Insere_posicao: Complexidade de tempo: O(n). A função insere um elemento em uma posição específica da lista vinculada. No pior caso, ela precisa percorrer a lista até a posição desejada, o que leva a uma complexidade de tempo linear em relação à posição

3.3. Remove_posicao: Complexidade de tempo: O(n). A função remove um elemento de uma posição específica na lista vinculada. No pior caso, ela precisa percorrer a lista até a posição desejada, o que leva a uma complexidade de tempo linear em relação à posição.

3.4. Remove_tudo: Complexidade de tempo: O(n). A função remove todos os elementos da lista vinculada. Ela precisa percorrer a lista e excluir cada nó individualmente, resultando em uma complexidade de tempo linear em relação ao número de elementos na lista.

3.5. Todos os Gets: Complexidade de tempo: O(n). A função retorna o valor de um elemento em uma posição específica na lista vinculada. No pior caso, ela precisa percorrer a lista até a posição desejada, levando a uma complexidade de tempo linear em relação à posição.

3.6. Imprimir: Complexidade de tempo: O(n). A função percorre a lista vinculada uma vez, onde 'n' é o número de elementos na lista. Para cada elemento, ela realiza uma operação de impressão, o que a torna linear em relação ao número de elementos na lista. Portanto, a complexidade de tempo é O(n).

4 ESTRATÉGIA DE ROBUSTEZ

No meu projeto, tomei várias medidas para garantir a qualidade do código e a identificação de erros. Aqui estão algumas das principais práticas que adotei:

Tratamento de Exceções: Implementei exceções em funções que podem encontrar erros durante a execução, como estouro de pilha ou acesso a memória inválida. Isso torna o código mais robusto e permite a identificação e tratamento adequado de problemas, caso ocorram.

Testes com Doctest: Criei testes abrangentes para cada classe do projeto usando o framework Doctest. Isso incluiu testes de unidade para as funções e métodos, bem como testes de integração para garantir que as classes funcionassem bem juntas. Os testes ajudaram a identificar problemas no código e a verificar se o código estava produzindo os resultados esperados.

Verificação de Memória com Valgrind: Usei a ferramenta Valgrind para verificar a consistência da memória no código. Isso ajudou a garantir que o código não tivesse problemas de gerenciamento de memória que podiam causar falhas ou problemas de desempenho.

No geral, essas práticas ajudaram a garantir que o código fosse mais coeso, livre de erros e eficiente.

5 ANÁLISE EXPERIMENTAL

Na pasta ‘test’ do trabalho adicionei as saídas do grpof ao realizar um comando ‘s’ com uma valoração com 1, 5, 12 letras, percebe-se

No cenário com 1 quantificador: A maioria das funções apresenta tempo acumulado de 0 segundos, o que significa que essas funções não contribuem significativamente para o tempo total de execução do programa. As funções ‘Vector::get_flag_em’ e ‘Vector::get_combinacao_em’ são as que mais consomem tempo.

No cenário com 5 quantificadores: As mesmas observações se aplicam, mas agora a quantidade de chamadas de funções aumentou, resultando em tempos de execução levemente maiores. Ainda assim, a maioria das funções continua sendo rápida, com tempos acumulados de 0 segundos.

No cenário com 12 quantificadores: O tempo de execução aumentou significativamente para várias funções, especialmente ‘Vector::get_flag_em’, ‘Vector::get_elemento_em’, ‘Pilha::pop’, ‘Vector::remove_posicao’ e ‘Vector::insere_posicao’. Estas funções estão consumindo uma parcela considerável do tempo total.

6 CONCLUSÃO

Portanto, o programa criado de acordo com as especificações propostas resolve os problemas de análise de expressões lógicas por meio das estruturas de dados Pilha e Vector. Durante a modelagem do projeto, a maior dificuldade foi passar o conhecimento teórico em um código eficiente. O trabalho mostrou na prática que em um determinado contexto uma estrutura de dados especifica será mais adequada para aquela solução, nesse caso a implementação de uma Árvore Binária poderia levar a uma maior eficiência. Assim, é importante avaliar o contexto da aplicação de forma generalizada e traçar um planejamento inicial mais elaborado com o objetivo de escolher a melhor opção de algoritmo para atender aos requisitos do problema.

BIBLIOGRAFIAS

Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas dedados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Pythonds (IME-USP). Infix, Prefix, and Postfix Expressions. Disponível em: <https://panda.ime.usp.br/pythonds/static/pythonds_pt/02-EDBasicos/InfixPrefixandPostfixExpressions.html>. Acesso em: 9/10/2023.

Flavio Vinicius Del Padre. PDS2 - UFMG. Disponível em: https://flaviovdf.io/pds2-ufmg/. Acesso em: 9/10/2023.

GeeksforGeeks. Introduction to Stack Data Structure and Algorithm Tutorials. Disponível em: https://www.geeksforgeeks.org/introduction-to-stack-data-structure-and-algorithm-tutorials/. Acesso em: 9/10/2023.

APÊNDICE – INSTRUÇÕS PARA COMPILAÇÃO

Para executar o código do trabalho deve ser feito os seguintes passos. Na pasta raiz do projeto execute o comando ‘make al’ para realizar a compilação do programa. Para rodar o arquivo executável gerado acesse a pasta ‘bin’ do projeto que contém o executável ‘tp1.out’. A execução considera as seguintes flags: ‘-a’ para avaliar a expressão lógica com base em uma valoração sem quantificadores, referente a primeira parte do projeto, ou ‘-s’ para avaliar a satisfabilidade de expressão considerando uma valoração que tenha quantificadores. Para apagar o executável na pasta ‘bin’ e os arquivos ‘.o’ gerados na compilação, digite ‘make clean’.

1. $ make all
2. $ ./bin/tp1.out opção formula_lógica valores
3. $ make clean

Exemplo para o comando ‘-a’:

1. $ make all
2. $ ./bin/tp1.out -a "~ ~ 0 | 1" 10
3. $ make clean

Exemplo para o comando ‘-s’:

1. $ make all
2. $ ./bin/tp1.out -s "0 | 1" ea
3. $ make clean