Trabalho Prático 1

Expressões Lógicas e Satisfabilidade

Arthur Araújo Rabelo
2022091552

1. Introdução

Neste trabalho, existem dois problemas que a ser resolvidos: a avaliação e a satisfabilidade de expressões booleanas, as quais possuem apenas um resultado (0 ou 1) e três operadores (&, | , ~). Na avaliação, o necessário é dizer se o resultado da expressão é falso (0) ou verdadeiro (1). No problema da satisfabilidade existe o que se chama de quantificadores. Estes criam o desafio de dizer se, para a significação de cada um (existe ou para todo), há uma solução em que a expressão seja verdadeira.

A solução que será utilizada para cada um desses problemas é diferente, ou seja, utiliza estruturas de dados diversas. A mera avaliação lógica das expressões é solucionada com pilhas, já a satisfabilidade é estruturada por meio de árvores binárias.

2. Método

2.1 Estruturas de Dados

As estruturas de dados utilizadas foram pilhas estáticas e uma árvore binária. As pilhas são alocadas estaticamente por um vetor do tipo dos elementos que cada uma vai receber. O tamanho da alocação foi feito de acordo com o máximo descrito na especificação do trabalho: 1000000 para a expressão e 100 para os valores. Já a árvore binária foi criada a partir da classe TipoNo, que é um TAD alocado dinamicamente. Uma árvore possui um ou mais nós.

2.2 Funções

A avaliação de expressões é feita pela função avaliaExpressao. O problema da satisfabilidade é resolvido pela função satisfazExpressao.

A função *avaliaExpressao* retorna se o valor booleano da expressão é 0 ou 1. Ela percorre a *string* dos valores (s) e utiliza as pilhas para chegar no resultado. Se não há problema de precedência a ser resolvido, os operadores e operandos são empilhados nas respectivas pilhas até que a expressão chegue ao final. Em seguida, os operandos são desempilhados juntamente com o operador que tiver ligado a eles para que o resultado seja calculado e inserido na pilha. Por fim, retorna-se o topo da pilha de operandos, que será o resultado final da expressão. Quando há precedência, a lógica é a mesma explicada acima, mas com a seguinte nuance: o que tem prioridade maior é tratado (desempilhado) primeiro e o resultado é novamente empilhado para ser resolvido com o resto da expressão. No caso dos parênteses, a função também fornece a prioridade necessária para tratar completamente o que está dentro deles antes de continuar. Sempre que há um parêntese de fechamento, a mesma lógica anterior de desempilhamento dos operandos e do operador e empilhamento do resultado é aplicada, de forma que a função não prossegue pela *string* s sem que tudo tenha sido solucionado.

A função satisfazExpressão é uma função recursiva que retorna um valor booleano indicando se a expressão é satisfazível ou não. A função começa a partir da raiz da árvore. A cada chamada, verifica-se pela função externa numQuantificadores se ainda existem quantificadores a serem tratados. Se sim, é feita a inserção de dois nós na árvore. Em seguida, a *string* s é percorrida e, se há um caractere ‘e’ ou ‘a’, os nós recebem como item as duas alternativas de valores possíveis para a expressão do nó anterior. O valore booleano de cada nó é dado pela própria função, a qual criará novos nós atribuídos a cada um dos anteriores até que todas as alternativas já tenham sido inseridas na árvore. Sendo assim, quando a pilha recursiva terminar, haverá o desempilhamento até a raiz, que terá valor 0 (não satisfazível) ou 1 (satisfazível). Enquanto esse processo é feito, a expressão original é alterada para satisfazer a saída requisitada.

Outras funções são utilizadas para resolver alguns detalhes da implementação, como *precedencia*, *numQuantificadores* e *operador*.

3. Análise de Complexidade

3.1 Complexidade de Tempo

avaliaExpressao:

Considerando n o tamanho da expressão p e m o tamanho da expressão s, temos a seguinte complexidade de tempo para esta função:

A função primeiro percorre a string s para verificar se existem quantificadores, o que é **O(m)**. O código percorre, em seguida, a string p, o que é **O(n)**. Para cada caractere de p éfeita uma comparação O(1), o que no total resulta em **O(n)**. Dentro do loop, as funções de empilhamento e desempilhamento são cada uma O(1). Como ocorre o empilhamento e o desempilhamento de todos os operadores e operandos, a complexidade total para essas operações é **O(n)**. Em cada while, o pior caso será também **O(n)**. Sabendo que n domina m assintoticamente, temos que:

$$O(m) + O(n) + O(n) + O(n) + O(n) + O(n) = \mathbf{O(n)}$$

satisfazExpressao:

Considerando m o número de quantificadores, temos a seguinte equação recursiva para esta função: T(m) = 2(T(m-1) + O(log 2^m)), onde T(0) = O(n). Isso quer dizer que o custo de resolver um problema de tamanho m, é o 2 vezes (2 nós a serem inseridos) o problema de resolver um problema de tamanho m-1 somado à complexidade de inserir dois nós na árvore. Se não há nenhum quantificador, a única operação é a chamada da função avaliaExpressao, a qual é O(n). Fazendo a expansão:

T(m) = 2(T(m-1) + O(log2^m))

= 2(2(T(m-2) + O(log2^(m-1))) + O(log2^m))

= 2^2(T(m-2) + O(log2^(m-1))) + 2O(log2^m)

= 2^3(T(m-3) + O(log2^(m-2))) + 2^2O(log2^(m-1)) + 2O(log2^m)

= ...

T(m) = 2^mT(0) + 2^(m-1)O(log2) + 2^(m-2)O(log2^2) + ... + 2^2O(log2^(m-2)) + 2O(log2^(m-1)) + O(log2^m)

T(m) = 2^mO(n) + O(2^(m-1)) + O(2^(m-2)) + ... + O(2^2) + O(2) + O(1)

No infinito, temos que:

T(m) = 2^mO(n) + O(2^m - 1).

Sendo assim, temos que a complexidade de tempo da função satisfazExpressao é **O(n)**, visto que é dominada assintoticamente pela chamada da função avaliaExpressao.

3.2 Complexidade de Espaço

avaliaExpressao:

Considerando que as pilhas são alocadas estaticamente com um valor constante, a complexidade de espaço para essa função é O(1).

satisfazExpressao:

Nesta função, existe a alocação dinâmica para cada nó da árvore. Sendo m o número de quantificadores, teremos $2^0 + 2^1 + \ldots + 2^m$ nós. Sendo assim, a complexidade de espaço é **O(2^m)**.

3.3 Limite de Quantificadores

Se o limite de quantificadores fosse retirado, o código não teria que ser alterado, uma vez que a sua implementação não foi feita de maneira pré-estabelecida para se adaptar a somente cinco quantificadores, mas para qualquer número. A complexidade também seria alterado, visto que para certo

número de quantificadores a influência da inserção de nós na árvore poderia se tornar mais significativa no tempo e no espaço utilizados pelo código.

4. Estratégias de Robustez

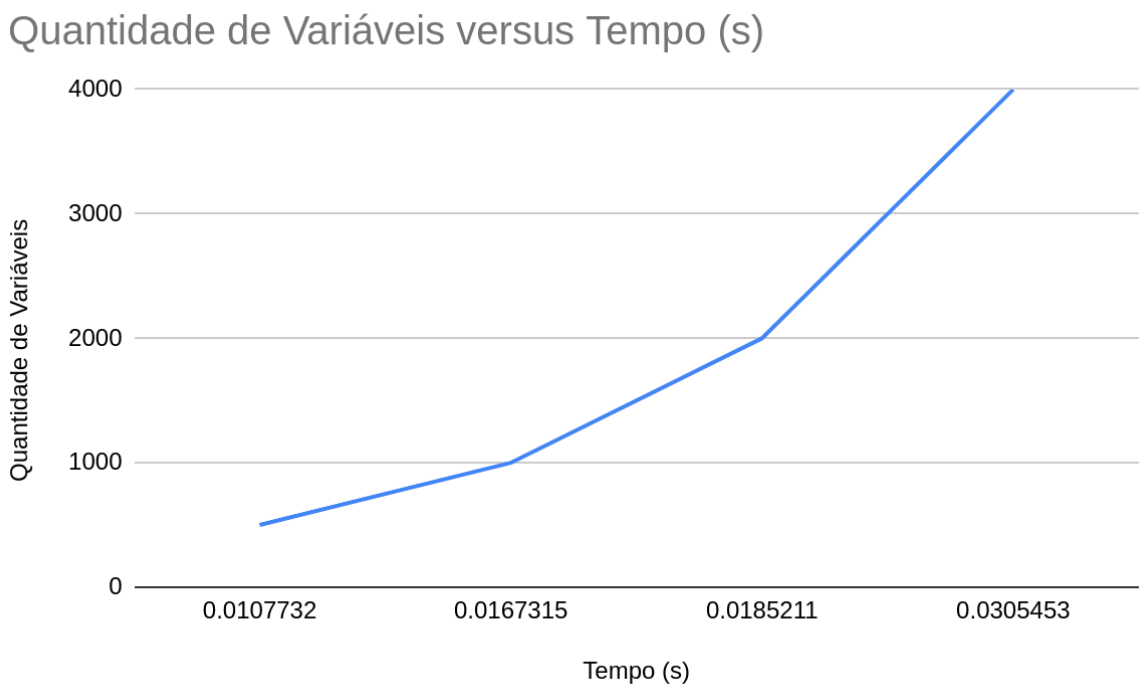
As medidas de programação defensiva foram as seguintes:

Na leitura da entrada, verifica-se se o número de argumentos está correto. Se o usuário digitar algo além do comando da operação, da string p e da string s, ou algo a menos, o código retorna que o número de argumentos é inválido. Nas funções de avaliaExpressao e satisfazExpressao, foi implementado um bloco try e catch para quando há algum caractere diferente dos operadores, operandos, quantificadores e números. Se isso ocorre o código lança a exceção de caractere inválido.

No TAD da pilha, também foram criadas duas exceções: a de pilha vazia e pilha cheia. Essas exceções são lançadas quando casos incoerentes acontecem. Por exemplo, se houver tentativa de desempilhar uma pilha que já está vazia ou empilhar em uma pilha que já atingiu o seu tamanho máximo.

5. Análise Experimental

Foi realizada uma análise para o comportamento do código em termos de tempo de relógio em relação ao número de variáveis com o número máximo de quantificadores (5). A função testada foi a satisfazExpressao, uma vez que ela contém as duas partes do problema. Pode-se verificar nos intervalos de variação do tamanho da entrada, o tempo cresce linearmente, comprovando a análise de complexidade feita acima.



Quanto aos resultados, foram feitos vários testes, incluindo os disponibilizados no VPL. Houveram alguns erros, mas que foram consertados logo em seguida.

6. Conclusão

O programa foi criado de acordo com a especificação e resolve o problema proposto de avaliação e satisfabilidade de expressões com seus respectivos requisitos. Aplicando os conceitos vistos em sala de aula e nas práticas, como estruturas de dados, análise de complexidade, análise de desempenho etc, foi possível desenvolver uma implementação que resolve um problema real.

Com efeito, houve um aprendizado de como estruturas de dados funcionam, onde e como podem ser implementadas - especificamente árvores e

pilhas. Além disso, a análise de tempo e espaço também possibilitou um maior aprofundamento em complexidade assintótica e uso de memória de um algoritmo, além das ferramentas de *clock* para medir o tempo de relógio de um programa.

7. Bibliografia

Expression Evaluation, Geeks for Geeks. Disponível em: https://www.geeksforgeeks.org/expression-evaluation/

CHAIMOWICZ, Luiz. PRATES, Raquel. Pilhas e Filas. Apresentação do PowerPoint. Disponível em: https://virtual.ufmg.br/20232/pluginfile.php/283491/mod_resource/content/2/07%20-%20PilhasFilas.pdf

CHAIMOWICZ, Luiz. PRATES, Raquel. Árvores. Apresentação do PowerPoint. Disponível em: https://virtual.ufmg.br/20232/pluginfile.php/283500/mod_resource/content/1/08%20-%20Arvores.pdf

CHAIMOWICZ, Luiz. PRATES, Raquel. Análise de Algoritmos Recursivos. Apresentação do PowerPoint. Disponível em: https://virtual.ufmg.br/20232/pluginfile.php/283461/mod_resource/content/1/04-AlgRecursivos.pdf