Trabalho Prático 1 - Expressões Lógicas e Satisfabilidade

João Correia Costa (2019029027)

Outubro de 2023, Belo Horizonte

1 Introdução

A resolução de problemas lógicos é essencial nos domínios da Ciência da Computação e da Matemática. Nesse contexto, este trabalho se concentra em abordar dois problemas clássicos em lógica, enfatizando aspectos de complexidade algorítmica e estruturas de dados implementadas em C++.

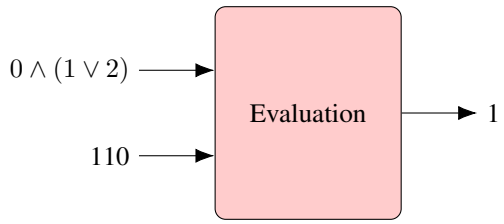
O primeiro desafio consiste em determinar o valor de verdade de expressões booleanas simples, compostas apenas por variáveis binárias e operadores lógicos ($\vee$, $\wedge$ e $\neg$). A abordagem proposta divide-se em duas etapas: a conversão da expressão no formato infix para postfix e a subsequente avaliação. Para isso, utilizamos uma pilha (`class Stack`) em cada etapa do processo.

O segundo desafio concentra-se em avaliar a satisfabilidade de expressões lógicas incluindo quantificadores existenciais ($\exists$) e universais ($\forall$). O algoritmo determina se a expressão fornecida é satisfazível ou não, identificando as soluções possíveis no primeiro caso. Essa abordadem incorpora o problema anterior e faz uso de árvores binárias (`class Tree`) para explorar as várias combinações possíveis de valores para cada variável. A árvore é construída com o uso de uma fila (`class Queue`), em seguida é realizado um caminhamento da árvore em ordem posfix para construir uma pilha que servirá a etapa final de avaliação.

2 Método

2.1 Avaliação de Expressões Booleanas

O problema proposto consiste em desenvover um algoritmo que avalia uma expressão booleana infix simples para valores especificados de cada variável binária. O diagrama abaixo caracteriza o problema em alto nível.

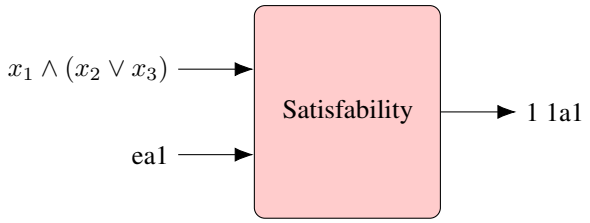


A avaliação é realizada pela função `evaluate_expression` em duas etapas. Primeiro, a expressão infix é convertida em posfix usando a função `to_posfix`. Foi utilizada como estrutura de dados uma pilha. A conversão é descrita em linguagem natural em `algorthm 1`. Em seguida, utilizou-se outra pilha para avaliar a expressão posfix, como descrito em `algorithm 2`.

Estruturas de Dados I: 2 `Stack<char>`

2.2 Satisfabilidade

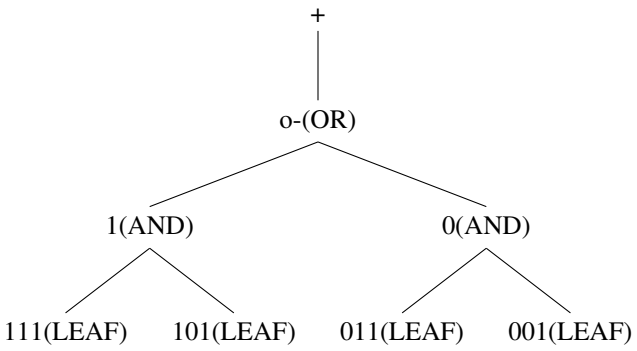
O problema de satisfabilidade booleana (SAT) envolve determinar se uma dada expressão booleana é satisfazível, ou seja, se existe uma valoração das variáveis que torna a expressão verdadeira. Neste contexto, a primeira entrada contém uma equação booleana simples, com operadores $\vee$, $\wedge$ e $\neg$. Os quantificadores existenciais e universais são codificados na segunda entrada, em que a letra `e` representa o quantificador existencial $\exists$ e a letra `a` representa o quantificador universal $\forall$. A saída do algoritmo indica `0` se a expressão não for satisfazível. Caso seja satisfazível, a saída é `1`, juntamente com uma string que codifica todas as soluções possíveis usando `1`'s, `0`'s e a letra `a` para indicar que qualquer valor satisfaz. Observe o diagrama abaixo que caracteriza o desafio e exemplifica a expressão em (1).



$$\exists x_1, \forall x_2 : x_1 \wedge (x_2 \vee 1) \tag{1}$$

Tomando como exemplo o diagrama mencionado, a expressão é considerada satisfazível, e as soluções possíveis são: $x_1 = 1, \forall x_2, x_3 = 1$. Isso implica que, para essa expressão, o valor de x_1 e x_3 devem ser `1`, enquanto que a variável x_2 pode assumir qualquer valor para que a expressão seja verdadeira.

O algoritmo desenvolvido para resolver o problema de satisfabilidade incorpora a avaliação de expressões booleanas do primeiro problema. A abordagem baseia-se na construção de uma árvore binária, representada pela classe `Tree` que explora todas as possíveis valorações das variáveis, codificadas pelos operadores $\exists$ e $\forall$. Os nós internos da árvore são representados pela estrutura `NodeT` e possuem atributos, incluindo uma `std::string substring` que indica uma valoração possível, uma função `Function function` que pode ser ($\vee$, $\wedge$ e `LEAF`) e um valor booleano `bool flag`. Além disso, os nós possuem ponteiros que conectam a árvore. Observe a árvore abaixo:



A construção da árvore é realizada pela função `_build` que utiliza uma fila `Queue` conforme especificado em `algorithm 3`. Cada nó interno recebe uma valoração incompleta e uma função $\wedge$, associada ao operador universal $\forall$, ou $\vee$, associada ao operador existencial $\exists$. Cada folha possui uma `substring` completa e uma valoração `flag` obtida por meio da função `evaluate_expression`.

Após a construção da árvore, é realizado um percurso posfix para criar uma pilha de `NodeT` na função `_traversal_stack` permitindo assim a avaliação da expressão codificada na árvore. O percurso posfix e a pilha posfix estão detalhados em `algorithm 4`. A última etapa consiste em desempilhar os nós da pilha posfix, conforme indicado em `algorithm 5`, para obter o nó resultado por meio da função `solve`. O nó resultado contém tanto a flag booleana que indica a satisfabilidade como a string que representa o conjunto de soluções possíveis no caso de satisfabilidade.

O problema da satisfabilidade é mais complexo que o anterior e a abordagem escolhida para solucioná-lo incorpora o algoritmo de avaliar expressões booleanas. A ideia em alto nível é utilizar uma árvore binária (`class Tree`) que varre todas as possiblidades de valoração das variáveis, que estão codificadas pelos operadores $\exists$ e $\forall$. Os Nós `struct NodeT` da árvore de satisfabilidade têm como atributos uma `std::string substring`, indicando uma valoração possível, uma função `Function function`, que pode ser $\wedge$, $\vee$, `LEAF` e uma valor booleano `bool flag`. Além dos ponteiros entre nós.

Estruturas de Dados II: 4 `Stack<NodeT*>`, 1 `Queue<NodeT*>`, 1 `Tree<NodeT*>`

3 Análise de Complexidade

Essa seção se dedica a analisar a complexidade assintótica, em termos de espaço e de tempo, das principais funções envolvidas nos problemas de Expressões booleanas e Satisfabilidade.

3.1 set_values

O loop `for` itera sobre cada caractere na fórmula fazendo operações simples. A verificação de espaço em branco com `std::isspace(c)` é uma operação constante `O(1)`. Quando um dígito é encontrado, há um loop `do-while` que percorre a sequência de dígitos, mas isso é feito apenas uma vez para cada dígito na entrada, portanto, ainda é uma operação `O(1)` para cada dígito. No geral, o loop `for` é executado uma vez para cada caractere na entrada, o que resulta em uma complexidade de tempo de `O(n)`, onde `n` é o comprimento da fórmula.

3.2 to_posfix

A função `set_values` é chamada primeiro, que tem uma complexidade de tempo de `O(n)`. A função principal percorre a fórmula infixa e, para cada caractere, executa operações no topo da pilha. As operações principais dentro do loop incluem a adição de caracteres à string `posfix_formula` e operações na pilha. Para caracteres como `0` e `1`, a operação é `O(1)`). Para os parênteses e operandos, as operações de empilhamento e desempilhamento na pilha podem ser `O(1)` em média, pois o número de elementos na pilha é limitado. No geral, o loop `for` é executado uma vez para cada caractere na entrada, resultando em uma complexidade de tempo de `O(n)`, onde `n` é o comprimento da fórmula.

3.3 boolean_evaluation

A função `to_posfix` é chamada primeiro e tem uma complexidade de tempo de `O(n)`, como discutido anteriormente. O loop `for` itera sobre cada caractere na fórmula de entrada. Para cada caractere, as seguintes operações são executadas: A verificação de `is_digit(c)` é uma operação `O(1)`. Para o caractere $\sim$, uma operação `_neg` é realizada em um único caractere, o que é `O(1)`. Para os operadores & e |, duas operações `_and` ou `_or` são realizadas em dois caracteres, resultando em `O(1)`. Portanto, para cada caractere na fórmula de entrada, as operações dentro do loop são realizadas em tempo constante `O(1)`.

No geral, o loop `for` é executado uma vez para cada caractere na fórmula, resultando em uma complexidade de tempo total de `O(n)`, onde `n` é o comprimento da fórmula de entrada.

3.4 Tree::_build

- **Complexidade de Tempo da Função _build**:

  1. Se a profundidade `_depth` da árvore for igual a `1`, a função cria um único nó `_root`. Isso envolve uma chamada para `evaluate_expression`, que tem uma complexidade de tempo de `O(n)` (onde `n` é o comprimento da fórmula). Portanto, a complexidade de tempo neste caso é `O(n)`.
  2. Caso contrário, a função começa a construir a árvore de maneira iterativa usando um loop. A cada iteração do loop, a função chama `_find_operator`, que tem uma complexidade de tempo de `O(1)` para encontrar o próximo operador na fórmula. Em seguida, ela cria dois nós filhos para o nó atual em cada iteração.
  3. Para cada folha criada, é necessário avaliar a expressão usando `evaluate_expression`. O número de folhas será dado por `2^_depth`, que corresponde ao número total de chamadas para `evaluate_expression` ao longo do processo.
  4. A complexidade de tempo geral da função `_build` é dominada pelo número de chamadas da função de avaliar expressões, logo temos custo exponencial `O(2^n)`.

- **Complexidade de Espaço da Função _build**:

  1. Para cada nó criado na árvore, a função aloca memória para armazenar informações sobre o nó, como sua `valuation`, `sub_string`, `function` e outros atributos. Portanto, a complexidade de espaço está relacionada ao número total de nós na árvore.
  2. Além disso, a função utiliza uma fila (representada como `Queue<char>`queue) para gerenciar os nós à medida que a árvore é construída. A quantidade de espaço necessário para manter a fila é proporcional ao número de nós presentes na árvore.
  3. A complexidade de espaço da função `_build` está relacionada ao número de nós na árvore, que corresponde a `2^_depth`.

3.5 Tree::_traversal_stack

- **Complexidade de Tempo da Função _traversal_stack**:

  1. A função realiza uma travessia pós-ordem na árvore. Ela utiliza duas pilhas, `node_stack` e `traversal_stack`, para realizar a travessia.
  2. O loop while executa enquanto o nó atual não for nulo ou a pilha node_stack não estiver vazia. Dentro do loop, as operações principais incluem adicionar e remover nós das pilhas.
  3. Quando o nó atual não é nulo, ele é adicionado às duas pilhas, e o algoritmo avança para o filho direito do nó atual. Essa operação envolve operações de empilhamento e desempilhamento, mas são executadas em tempo constante, `O(1)`.
  4. Quando o nó atual é nulo, um nó é desempilhado da pilha node_stack, e o algoritmo avança para o filho esquerdo desse nó. Novamente, envolve operações de empilhamento e desempilhamento em pilhas, mas em tempo constante `O(1)`.

5. Portanto, para cada nó na árvore, há operações de empilhamento e desempilhamento, que são executadas em tempo constante `O(1)`. Como o loop while executa para cada nó na árvore uma vez, a complexidade de tempo da função _traversal_stack é `O(n)`, onde `n` é o número de nós na árvore.

- **Complexidade de Espaço da Função _traversal_stack**:

1. A função utiliza duas pilhas, node_stack e traversal_stack, para gerenciar os nós durante a travessia. A quantidade de espaço necessário para manter essas pilhas depende do número de nós na árvore.
2. Como mencionado anteriormente, a complexidade de tempo da função é `O(n)`, onde `n` é o número de nós na árvore. Portanto, o espaço necessário para as pilhas é proporcional ao número de nós na árvore.
3. No pior caso, a árvore pode ter `n` nós, o que resultaria em um espaço adicional de `O(n)` para as pilhas.

3.6 Tree::solve

- **Complexidade de Tempo da Função solve**:

1. A função solve é responsável por resolver a árvore de expressões lógicas que já foi construída. Ela utiliza duas pilhas, stack_evaluation e `post_stack`, durante o processo de avaliação.
2. A função inicia recuperando a pilha `post_stack` com a travessia pós-ordem da árvore, o que envolve um custo de tempo de `O(n)`, onde `n` é o número de nós na árvore.
3. Em seguida, ela inicia um loop while que percorre os nós da pilha `post_stack`.
4. No loop while, a função realiza operações que envolvem adicionar e remover nós das pilhas `stack_evaluation` e `post_stack`, bem como avaliar expressões, executar operações lógicas, atualizar flags e valuations dos nós.
5. A operação de adicionar e remover nós das pilhas é executada em tempo constante `O(1)`.
6. A avaliação de expressões, execução de operações lógicas e atualizações nos nós também são realizadas em tempo constante `O(1)`.
7. O loop while é executado para cada nó na pilha `post_stack`, que é diretamente proporcional ao número de nós na árvore.
8. Portanto, a complexidade de tempo da função solve é dominada pelo loop while e é `O(n)`, onde `n` é o número de nós na árvore.

- **Complexidade de Espaço da Função solve**:

1. A função solve utiliza duas pilhas, `stack_evaluation` e post_stack, para gerenciar os nós durante o processo de avaliação.
2. A quantidade de espaço necessário para manter essas pilhas é proporcional ao número de nós na árvore.
3. Como mencionado anteriormente, a complexidade de tempo da função é `O(n)`, onde `n` é o número de nós na árvore. Portanto, o espaço necessário para as pilhas é proporcional ao número de nós na árvore.
4. No pior caso, a árvore pode ter `n` nós, o que resultaria em um espaço adicional de `O(n)` para as pilhas.

4 Estratégias de Robustez

Com o objetivo de tornar o programa mais robusto e evitar problemas com entradas inválidas, foram implementadas verificações nos inputs do terminal no arquivo `main.cpp`. Isso garante que o usuário não insira opções inválidas e, caso o faça, o programa apresenta uma mensagem de erro no terminal e encerra a execução.

Além disso, foram criadas classes de exceção, como `ExceptionEmptyStack` e `ExceptionEmptyQueue`. Nas funções de remoção das respectivas estruturas de dados, lançamos essas exceções caso a estrutura esteja vazia. Isso permite um tratamento adequado de situações excepcionais, garantindo a integridade do programa.

Entretanto, é importante destacar que o programa ainda possui limitações, uma vez que não cobre um amplo espectro de possíveis entradas de dados, presumindo que o usuário fornecerá entradas corretas. As verificações se concentram principalmente na detecção de opções inválidas.

Para manter a integridade do programa e evitar vazamentos de memória, todos os Tipos Abstratos de Dados (TADs) implementam destrutores apropriados. Além disso, foram realizados testes com o Valgrind, e nenhum erro relacionado à alocação de memória foi observado.

5 Análise Experimental

O seguinte experimento foi conduzido para avaliar o custo computacional das funções implementadas em um problema de satisfatibilidade com uma fórmula contendo 100 variáveis e 20 quantificadores. Essa configuração resulta em uma árvore binária com um total de 2^{20} folhas.

A análise de desempenho através do `gprof` (conforme apresentada na tabela) é essencial para identificar quais funções consomem a maior parte do tempo de execução no programa. As funções foram classificadas com base no tempo total de execução (cumulativo) e na quantidade de chamadas.

A função `_init` é a mais custosa, representando $26,5\%$ do tempo total de execução. É importante observar que esta função não pertence à implementação do programa, mas está relacionada à biblioteca `libgc` e pode ser considerada uma função de inicialização.

Em seguida, as funções associadas à avaliação de expressões booleanas simples apresentam o maior custo. As funções `set_values` e `to_posfix` representam aproximadamente $11,27\%$ e $10,40\%$ do tempo total de execução, respectivamente. Ambas são chamadas um grande número de vezes (1048576), o que corresponde ao número total de folhas na árvore de teste. É importante notar que a função `evaluate_expression` é chamada a mesma quantidade de vezes, o que é esperado, uma vez que invoca tanto `set_values` quanto `to_posfix`.

Funções pertencentes à classe `Stack`, como `empty`, `add`, `pop` e `peek`, consomem uma quantidade significativa de tempo e são chamadas em grande quantidade. Pode ser benéfico revisar e otimizar o desempenho dessas funções, já que são usadas de forma intensiva. Além disso, é observado que o número de chamadas da função `pop` corresponde ao número de chamadas da função `add`, o que é razoável.

Funções como `precedence`, `is_digit`, `is_operand` e `_neg` também apresentam um alto número de chamadas e contribuem para um tempo de execução considerável.

Por outro lado, as funções `Tree::_traversal_stack` e `Tree::solve` são chamadas muito poucas vezes (2 e 1, respectivamente), mas ainda contribuem para o tempo de execução.

Há várias funções que são chamadas apenas uma vez, como `Tree:: Tree`, `Queue::clear`, `Tree::_build`, etc. O tempo gasto nessas funções pode não ser significativo para o desempenho geral.

6 Conclusões

Neste trabalho, abordamos dois desafios clássicos em lógica, com foco na complexidade algorítmica e nas estruturas de dados implementadas em C++. No primeiro desafio, desenvolvemos um algoritmo para determinar o valor de verdade de expressões booleanas simples. No segundo desafio, exploramos a avaliação da satisfabilidade de expressões lógicas mais complexas, que incluíam quantificadores existenciais ($\exists$) e universais ($\forall$).

O que foi aprendido:

1. Compreensão da teoria por trás das expressões booleanas e das estruturas de dados para manipulá-las eficientemente.
2. A conversão de expressões infix para postfix é uma técnica importante para a avaliação de expressões lógicas e facilita a criação de algoritmos eficazes.
3. A incorporação de quantificadores existenciais e universais em expressões lógicas adiciona uma camada adicional de complexidade, mas pode ser abordada de maneira sistemática com o uso de estruturas de dados como árvores binárias.
4. O uso de estruturas de dados bem projetadas, como pilhas, filas e árvores, permitiu a resolução eficaz desses desafios.

Este trabalho expandiu nosso conhecimento sobre algoritmos lógicos e demonstrou a importância das estruturas de dados no contexto da lógica computacional. Além disso, foi possível verificar na prática o custo computacional das soluções implementadas. Esse trabalho remete ao lema: ”Algoritmo + Estrutura de Dados = Programa”.

7 Bibliografia

1. Chaimowicz, L. and Prates, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

2. Introduction to Algorithms, Thomas H. Cormem, Charles E. Leiserson, Ronald L. Rivest.

A Instruções para Compilação e Execução

Observação: Certifique-se de que você tenha o compilador GCC (g++) instalado em seu sistema para a compilação.

A.1 Compilação do Projeto

Para compilar o projeto, siga as instruções abaixo:

1. Abra um terminal e navegue até o diretório raiz do projeto.
2. Certifique-se de que o projeto contenha a seguinte estrutura de diretórios:

   ```
   - src/
   - obj/
   - bin/
   - test/
   - include/
   - third_party/
   ```

3. Utilize o seguinte comando para compilar o projeto:

   ```
   make
   ```

   Isso irá compilar o projeto e gerar o executável `bin/tp1.out`.

A.2 Execução do Projeto

Para executar o projeto compilado, utilize o seguinte comando:

```
./bin/tp1.out
```

Este comando executará o programa principal.

A.3 Execução de Testes e Experimentos

Se desejar executar os testes do projeto, execute:

```
make test
```

Se desejar executar o experimento com `gprof` especificado em `test/input.txt`, execute::

```
make experiment
```

A.4 Limpeza dos Arquivos Compilados

Para limpar os arquivos compilados e executáveis, utilize o seguinte comando:

```
make clean
```

Isso removerá os arquivos objetos, executáveis e quaisquer arquivos gerados durante os testes.

B Tabelas e Algoritmos

Table 1: *Function Profiling*

%	Calls	Name
26.50	-	_init
11.27	1048576	set_values
10.40	1048576	to_posfix
8.49	1399848960	Stack::empty
7.83	580911104	precedence
7.61	809500672	is_digit
6.52	503316480	Stack::add
5.93	503316480	Stack::pop
3.07	494927872	Stack::peek
3.00	94371840	_neg
3.00	1048576	evaluate_expression
2.64	198180864	is_operand
1.90	503316480	Node::Node
0.59	2	Tree::_traversal_stack
0.44	94371840	_and
0.22	10485755	Stack::add
0.15	1	Tree::T̃ree
0.07	29360118	Stack::empty
0.07	5242877	Queue::empty
0.07	2097151	Queue::pop
0.07	2097150	std::operator+
0.07	1048575	Queue::peek
0.07	1	Tree::solve
0.00	12582906	Node::Node
0.00	10485755	Stack::pop
0.00	10485755	Stack::peek
0.00	9437184	_or
0.00	2097152	Stack::clear
0.00	2097152	Stack::Stack
0.00	2097152	Stack::S̃tack
0.00	2097151	NodeT::NodeT
0.00	2097151	NodeT::ÑodeT
0.00	2097151	Queue::add
0.00	2097150	std::move
0.00	2097150	std::operator+
0.00	1048575	Tree::_evaluate
0.00	404	__gnu_cxx::__normal_iterator::base
0.00	202	__gnu_cxx::operator!=
0.00	200	__gnu_cxx::__normal_iterator::operator++
0.00	200	__gnu_cxx::__normal_iterator::operator*
0.00	21	Tree::_find_operator
0.00	21	Tuple::T̃uple
0.00	20	Queue::get_size
0.00	20	Tuple::operator=
0.00	5	Stack::clear
0.00	5	Stack::Stack
0.00	5	Stack::S̃tack
0.00	2	count_char
0.00	1	__static_initialization_and_destruction_0
0.00	1	Tree::_build
0.00	1	Tree::Tree
0.00	1	Queue::clear
0.00	1	Queue::Queue
0.00	1	Queue::Q̃ueue

Algorithm 1 Converter expressão lógica infixa para pós-fixa

1. Inicialize uma string vazia chamada posfix_formula para armazenar a fórmula pós-fixa.
2. Crie uma pilha chamada stack para auxiliar na conversão.
3. Substitua quaisquer variáveis na fórmula infixa pelos valores correspondentes fornecidos na sequência valuation.
4. Percorra cada caractere da fórmula infixa da esquerda para a direita.
5. Para cada caractere:
   (a) Se for um valor booleano '0' ou '1', adicione-o à posfix_formula.
   (b) Se for um parêntese aberto '(', empilhe-o na pilha stack.
   (c) Se for um parêntese fechado ')', desempilhe operadores da pilha stack e adicione-os à posfix_formula até encontrar um parêntese aberto correspondente.
   (d) Se for um operador (como AND, OR), enquanto houver operadores na pilha stack com maior precedência ou a mesma precedência (e associatividade à esquerda), desempilhe-os e adicione à posfix_formula. Em seguida, empilhe o operador atual na pilha stack.
6. Após percorrer toda a fórmula infixa, desempilhe quaisquer operadores restantes da pilha stack e adicione-os à posfix_formula.
7. A posfix_formula agora contém a expressão no formato pós-fixo.
8. Retorne a posfix_formula.

Algorithm 2 Avaliar expressão lógica pós-fixa com valores 0's e 1's

1. Chame a função to_posfix para converter a fórmula da sua forma infixa para a forma pós-fixa, aplicando a sequência de valores valuation quando necessário.
2. Inicialize uma pilha chamada stack para armazenar temporariamente os valores da expressão.
3. Percorra a formula no formato pós-fixo da esquerda para a direita.
4. Para cada caractere:
   (a) Se for um dígito '0' ou '1', empilhe o valor na pilha stack.
   (b) Se for o operador ' ' (NOT), desempilhe um valor da pilha, negue-o e empilhe o resultado na pilha stack.
   (c) Se for um operador binário (AND ou OR), desempilhe dois valores da pilha, aplique a operação apropriada e empilhe o resultado na pilha stack.
5. Após percorrer toda a expressão, o valor resultante estará no topo da pilha stack.
6. Verifique se o valor no topo da pilha é '1'. Se for, retorne true, indicando que a expressão é verdadeira. Caso contrário, retorne false, indicando que a expressão é falsa.

Algorithm 3 Construção de uma Árvore de Expressões Lógicas usando uma Fila

1. Inicialize o nó raiz da árvore como nulo.
2. Verifique se a profundidade (*depth*) é igual a 1.
   (a) Se for igual a 1, crie um nó folha com o valor da avaliação da fórmula (*formula*) e atribua-o como o nó raiz.
3. Caso contrário, continue com a construção da árvore:
   (a) Inicialize um índice como 0.
   (b) Use uma função (*_find_operator*) para encontrar o próximo operador e seus operandos na avaliação (*valuation*) com base no índice atual.
   (c) Crie o nó raiz da árvore com o operador e a subexpressão obtidos da função anterior.
   (d) Inicialize uma fila (queue) vazia para gerenciar os nós da árvore.
   (e) Adicione o nó raiz à fila (queue).
4. Execute um loop até que o operador encontrado seja um operador folha:
   (a) Obtenha o próximo operador e seus operandos da avaliação.
   (b) Calcule o tamanho atual da fila (queue).
   (c) Para cada nó na fila (queue) neste momento:
      i. Remova um nó da fila (queue) e defina-o como o nó atual (*current*).
      ii. Crie dois nós filhos: left e right, anexando substring do parent concatenada com ”1” ou ”0”, concatenada com a substring corrente.
      iii. Se o operador encontrado é um operador folha, avalie os valores dos nós filhos usando a função (*evaluate_expression*) e atribua os resultados aos campos *flag* dos nós.
      iv. Adicione os nós filhos à fila (queue).
5. Após a conclusão do loop, a árvore estará construída.

Algorithm 4 Avaliação a partir da Pilha Pós-Fixa

1. Inicie com duas pilhas vazias: uma para a pilha de nós e outra para a pilha de travessia.
2. Comece a travessia a partir da raiz da árvore.
3. Enquanto houver nós para serem explorados ou a pilha de nós não estiver vazia:
   (a) Se o nó atual não for nulo:
      i. Empilhe o nó na pilha de nós e na pilha de travessia.
      ii. Avance para o filho direito do nó atual, se existir, e continue a partir desse ponto.
   (b) Caso contrário:
      i. Desempilhe um nó da pilha de nós e vá para o seu filho esquerdo, se existir.

Algorithm 5 Avaliação a partir da pilha posfix

1. Inicie a pilha de avaliação como vazia.
2. Enquanto a pilha de travessia (`traversal_stack`) não estiver vazia, faça o seguinte:
   (a) Retire o nó do topo da pilha de travessia.
   (b) Se o nó for uma folha (um operando), adicione-o à pilha de avaliação.
   (c) Se o nó for um operador, desempilhe os operandos necessários da pilha de avaliação, aplique a operação representada pelo operador e coloque o resultado na pilha de avaliação.
3. Após avaliar todos os nós da pilha de travessia, a pilha de avaliação conterá o resultado final da expressão da árvore.
4. Realize as ações desejadas com base no resultado da avaliação, como imprimir, armazenar ou executar operações adicionais.