UNIVERSIDADE FEDERAL DE MINAS GERAIS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Algoritmos e Estruturas de Dados III
Trabalho Prático 0 – Notação Polonesa Reversa

Caio Felipe Zanatelli

Belo Horizonte
20 de abril de 2017

Sumário

1 Introdução

A forma em que expressões matemáticas são representadas é de grande importância no que tange a resolução de problemas, visto que sua avaliação pode se tornar mais fácil e ágil de acordo com as características da notação. Neste contexto, bastante importante para o cenário computacional, se destaca a Notação Polonesa Reversa (RPN), proposta em 1954, em contraste à notação Polonesa, por Burks, Warren e Wright, e reinventada por Friedrich L. Bauer e Edsger W. Dijkstra, na década de 1960. Essa notação é dotada de uma representação não usual: os operandos sempre precedem os operadores.

Dessa forma, não há a necessidade de definição de precedência de operações, pois, como os operandos antecedem os operadores, a ordem de cálculos se torna bem definida. Por exemplo, a expressão *5 + 2*, em Notação Polonesa Reversa, seria escrita da forma *5 2 +*. Ainda, a expressão *(5 + 2) * 3*, seria escrita da forma *5 2 + 3 **. Assim, justamente pelo fato de parênteses serem dispensáveis nesta notação, sua utilização é interessante por reduzir o uso de memória, além de aumentar a velocidade operacional na solução de problemas, sendo amplamente utilizada, por exemplo, em aplicações financeiras.

Caracterizada a importância da notação supracitada, este trabalho apresenta uma implementação cujo o objetivo é, dada uma expressão RPN, com os operadores apagados e um resultado a ser obtido, listar todas as possibilidades de operadores, com soma precedendo a multiplicação, de forma que o resultado da expressão seja aquele requisitado. Para tal, a resolução da expressão foi efetuada com o emprego da estrutura de dados Pilha, sendo aplicada recursivamente para a obtenção de todas as combinações de operadores possíveis. A recursão foi adotada como estratégia pois, além de permitir a escrita com uma ordem bem definida de operadores, ela permite a reutilização de resultados obtidos nas sequências anteriores, diminuindo assim o tempo de processamento.

2 Solução do Problema

Como abordado na seção anterior, os operandos antecedem os operadores. Assim, uma abordagem *top-down* pode ser utilizada para a resolução de expressões RPN. Para tanto, uma estratégia conveniente é a utilização de uma estrutura de dados Pilha, pois esta fornece a ordem necessária de operações: identificado um operador, os últimos dois elementos da pilha devem ser os operandos. Embora este trabalho opere apenas com somas e multiplicações, a Tabela 1 ilustra bem como uma pilha pode ser utilizada para resolver uma expressão em Notação Polonesa Reversa. A expressão resolvida é *2 + ((1 + 3)*5) - 3*, a qual pode ser expressa em RPN como *2 1 3 + 5 * + 3 -*.

Tabela 1 – Resolução de uma expressão em Notação Polonesa Reversa utilizando uma Pilha

Etapa	**Entrada**	**Ação**	**Pilha**	**Descrição**
#1	2	Operando	{2}	Insere operando 2 na pilha.
#2	1	Operando	{1, 2}	Insere operando 1 na pilha.
#3	3	Operando	{3, 1, 2}	Insere operando 3 na pilha.
#4	+	Operador	{4, 2}	Retira os dois operandos do topo (1 e 3), realiza a soma (1 + 3 = 4) e insere o resultado na pilha.
#5	5	Operando	{5, 4, 2}	Insere operando 5 na pilha.
#6	*	Operador	{20, 2}	Retira os dois operandos do topo (4 e 5), realiza a multiplicação (4 * 5 = 20) e insere o resultado na pilha.
#7	+	Operador	{22}	Retira os dois operandos do topo (2 e 20), realiza a soma (2 + 20 = 22) e insere o resultado na pilha.
#8	3	Operando	{3, 22}	Insere operando 3 na pilha.
#9	-	Operador	{19}	Retira os dois operandos do topo (22 e 3), realiza a subtração (22 - 3 = 19) e insere o resultado na pilha.
#10	*None*	Resultado	{19}	Finalizadas as operações, o único valor contido na pilha é o resultado da expressão, neste caso, 19.

Uma vez definido o processo de resolução de uma expressão RPN, é necessário introduzir a estratégia utilizada para solucionar o problema dado.

2.1 Tipo Abstrato de Dados RPNExpression

A entrada fornecida contém duas linhas: a primeira contém uma sequência de operandos definidos e operadores que deverão ser descobertos, os quais são identificados com um ponto de interrogação. A segunda linha contém o resultado esperado. Como os operadores são do tipo *char* e os operandos do tipo *int*, foi necessário definir um padrão para abstrair a expressão. Para tanto, foi criado o Tipo Abstrato de Dados *RPNExpression*.

A ideia base para isso foi transformar toda a expressão em valores inteiros e armazená-los em um vetor. Ou seja, se o i-ésimo elemento for um operando, então seu valor será ele próprio. Porém, se o elemento em questão for um operador, este será representado como um identificador inteiro, no caso, foi escolhido o valor -1. Essa transformação é possível pois os valores definidos para este trabalho são estritamente inteiros positivos. Dessa forma, a avaliação da expressão se torna bem menos complexa, pois a identificação de operandos e operadores se torna intuitiva e de fácil manuseio, sem a necessidade de manipular *strings* e inteiros ao mesmo tempo. A Figura 1 ilustra a transformação da entrada nos padrões supracitados.

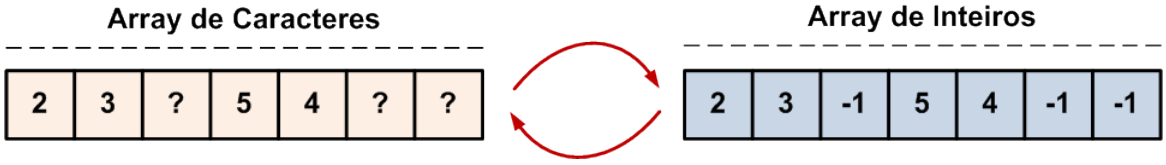

Figura 1 – Exemplo de transformação de uma expressão de entrada em uma representação simples

Por sua vez, o TAD *RPNExpression* possui os seguintes campos:

- **Elementos**: vetor de inteiros alocado dinamicante, responsável por armazenar a expressão RPN nos padrões definidos nesta seção.
- **Tamanho**: valor inteiro responsável por armazenar o tamanho da expressão RPN, isto é, o tamanho do vetor *elementos*.
- **QtdOperadores**: valor inteiro que armazena a quantidade de operadores contidos na expressão.

Além disso, esse TAD possui duas funções implementadas, as quais são descritas na sequência.

- **getRPNExpression**: transforma a *string* de entrada, a qual representa a expressão RPN, em uma variável do tipo *RPNExpression*.
- **freeRPNExpression**: desaloca o vetor de elementos da expressão RPN descrita pela variável do tipo *RPNExpression*, o qual foi alocado pela função *RPNExpression*.

2.2 Tipo Abstrato de Dados Stack (Pilha)

Como ilustrado pela Tabela 1, uma pilha é ideal para a resolução do problema. Para tanto, um TAD *Stack* foi implementado, o qual contém os seguintes campos:

- **pTopo**: um ponteiro para a célula topo da pilha, do tipo abstrato de dados criado *Node*, o

qual possui o conteúdo de cada célula da pilha. Este TAD será apresentado posteriormente.
- **Tamanho**: inteiro que armazena o tamanho atual da pilha.

As operações definidas no TAD *Stack* são as seguintes:

- **criarPilha**: cria uma pilha vazia.
- **push**: insere um elemento na pilha.
- **top**: retorna o topo da pilha, mas sem removê-lo.
- **pop**: remove o topo da pilha e o retorna.
- **getTamanho**: retorna o tamanho atual da pilha.
- **isPilhaVazia**: verifica se a pilha está vazia.
- **freePilha**: libera toda a memória utilizada/alocada pela pilha.

2.2.1 Inserção e Remoção de uma Célula na Pilha

As operações de inserção e remoção aqui descritas seguem a implementação clássica do TAD Pilha, ou seja, utiliza a ideia de LIFO (*Last in, First out*), em português, o último elemento a ser inserido, é o primeiro a ser retirado. Além disso, vale ressaltar que a pilha implementada neste trabalho é dinâmica, isto é, não possui um tamanho pré-definido. Assim, a estrutura cresce de acordo com a quantidade de operandos e operadores na expressão. A Figura 2 ilustra o processo de inserção (*push*) e remoção (*pop*) em um TAD Pilha, bem como a função *top*, que retorna a célula do topo.

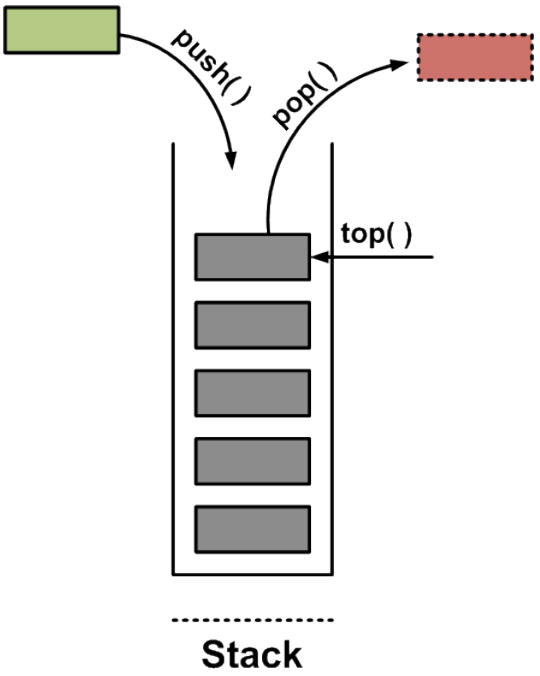

Figura 2 – Ilustração das operações de inserção e remoção em uma pilha

2.3 Tipo Abstrato de Dados Node

Como observado na seção anterior, também foi criado o tipo abstrato de dados Node. Este tipo representa uma célula do TAD *Stack*, ou seja, cada um dos elementos da pilha é um *Node*. Isso é importante de se fazer para garantir a modularidade do projeto. Assim, este mesmo TAD pode ser utilizado para outras finalidades, bastando apenas alterar o campo que contém o elemento no nodo. Dito isso, os atributos presentes nesta estrutura são:

- **Elemento**: variável do tipo *Item*, responsável por armazenar o elemento da pilha. Neste caso, *Item* foi definido como inteiro, mas pode ser redefinido para outro tipo em outras aplicações, uma vez que a modularidade do código foi preservada.
- **pProximo**: ponteiro do tipo *Node* que armazena a próxima célula da pilha, pelo fato dela ser encadeada.

2.4 Tipo Abstrato de Dados SolveRPN

Por fim, o último tipo abstrato de dados criado foi o TAD *SolveRPN*, o qual contém as funções de resolução do problema. São duas:

- **getAllSolutions**: recebe como parâmetro a expressão RPN, do tipo *RPNExpression*, e o resultado esperado, do tipo inteiro. Esta função, então, cria uma pilha e aloca um *array* de caracteres. A pilha será utilizada na função recursiva *solveRPN*, para calcular a expressão, e o *array* de caracteres será utilizado para armazenar as sequências de operações correspondentes (soma e multiplicação). Feito isso, a função *solveRPN* é invocada.
- **solveRPN**: responsável por calcular todas as combinações de operadores possíveis para a expressão dada, imprimindo aquela cujo o resultado é o mesmo do esperado. Um ponto importante a ser destacado nesta função, e que será abordado em seguida, é que se trata de uma função recursiva, o que permite a reutilização de resultados já calculados, diminuindo assim o tempo de processamento. Seus parâmetros são: a expressão, do tipo *RPNExpression*; um inteiro que representa o índice atual da análise da expressão, pois necessita desta informação devido à recursão; o resultado esperado da expressão, do tipo inteiro; um *array* de caracteres, o qual armazenará as sequências de operações calculadas; um índice para a sequência de operadores, do tipo inteiro, para escrever o i-ésimo operador na posição correta do vetor; e uma pilha, passada por referência, que armazenará os operandos da expressão, bem como os resultados parciais.

2.4.1 Função solveRPN, uma Abordagem Recursiva

O problema em questão pode ser avaliado com uma interpretação recursiva. Pede-se para calcular todas as possíveis sequências de operadores para se atingir determinado resultado, com a regra de que a soma tem precedência sobre a multiplicação. Ou seja, com uma análise simples deste enunciado, é possível perceber uma solução clara: criar uma árvore de recursão, chamando primeiro a função para soma e depois para a multiplicação. Com os devidos cuidados para manipular a pilha que é transferida de *branch* a *branch* na recursão, e impondo condições para que a execução termine, o problema é facilmente resolvido. De forma a ilustrar esta abordagem, a Figura 3 apresenta uma árvore de recursão exemplo para a expressão, em Notação Polonesa Reversa, *2 3 ? 5 4 ? ?*.

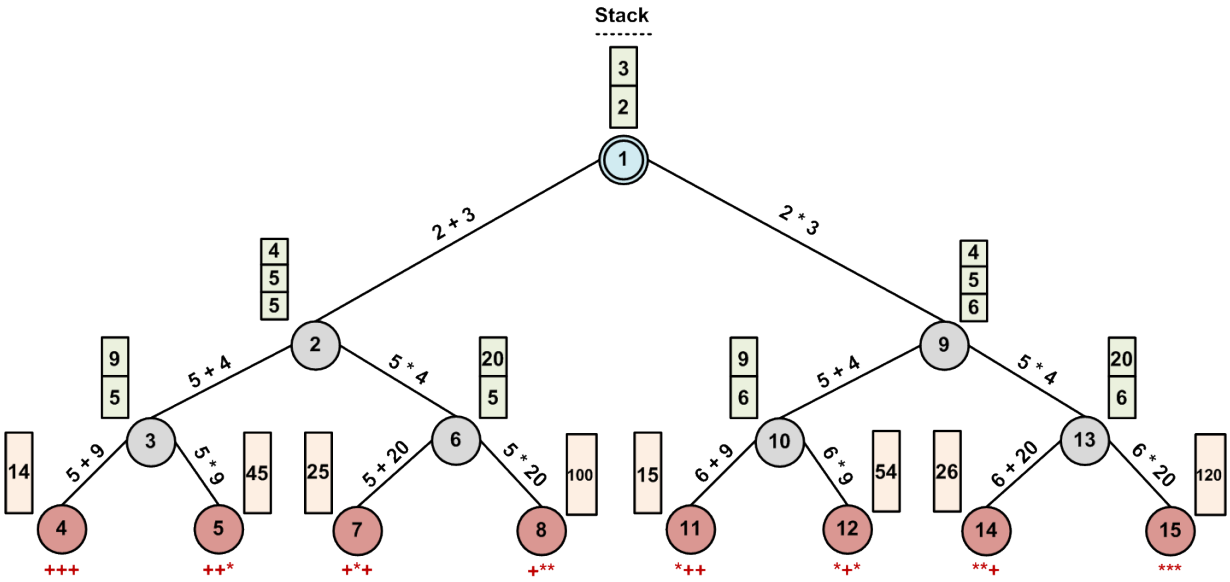

Figura 3 – Exemplo de árvore de recursão para a expressão *2 3 ? 5 4 ? ?*

O estado 1, em azul, é o início da recursão. Os operandos foram colocados em uma pilha ao lado de cada estado, emitindo a fase de empilhamento, para melhor visualização. No estado inicial, os operandos 2 e 3 já estão na pilha, e então a recursão é iniciada primeiro para a soma (arestas à esquerda) e depois para a multiplicação (arestas à direita). Os estados na cor rosa (4, 5, 7, 8, 11, 12, 14 e 15) são os estados de término, onde os resultados finais da expressão são obtidos ao variar os operadores, indicado pelo elemento único na pilha. Como esperado, a sequência de operadores, indicado também na cor rosa embaixo de cada estado de término, segue a ordem proposta inicialmente, isto é, soma tem precedência sobre a multiplicação.

Além disso, a Figura 3 também fornece uma informação de grande relevância: os resultados parciais são reaproveitados. Lendo a árvore a partir das folhas, isto é ainda mais claro. Para calcular uma sequência S_i, o resultado da sequência S_{i-1} é utilizado. Além disso, como os valores são estritamente positivos, se em algum nível da árvore o resultado exceder aquele esperado, a operação pode ser abortada, e a execução retorna ao nó pai. Assim, essas duas otimizações diminuem significativamente o tempo de processamento necessário para a resolução do problema.

3 Análise de Complexidade

Nesta seção serão apresentadas as análises de complexidade temporal e espacial do algoritmo desenvolvido.

3.1 Complexidade Temporal

A primeira função a ser executada é a *getExpression*, responsável por transformar a *string* de entrada em uma representação mais simples de se trabalhar, como já explanado. Este processo, por sua vez, percorre o vetor de caracteres da entrada apenas uma vez e, para cada *substring*, verifica se é um operando ou um operador, atribuindo seu valor ou o identificador -1 ao vetor

elementos da expressão RPN. Como cada operação individual custa $O(1)$, e são executadas n vezes, onde n é o número de *substrings* da entrada, isto é, a quantidade de operadores somada à quantidade de operandos, temos que a complexidade temporal desta função é $O(n)$. A função *freeRPNExpression*, por sua vez, apenas desaloca o vetor *elementos*, o que é $O(1)$.

Na estrutura de dados Pilha (*Stack*), temos as seguintes complexidades: *getTamanho* e *isPilhaVazia* apenas retornam o campo *tamanho* e uma comparação verificando se o *tamanho* é igual a 0, respectivamente. Como essas operações são constantes, ambas as funções são $O(1)$. A inserção e remoção são feitas utilizando apenas o topo da pilha e, como existe, no TAD, um ponteiro para o topo, este acesso é realizado com um número de operações constantes e, portanto, *push* e *pop* também são $O(1)$. Como a função *top* também apenas acessa o topo da pilha e o retorna, ela também é $O(1)$. Já a função *criarPilha*, executa apenas duas operações de atribuição e, portanto, é $O(1)$. Por fim, a função *freePilha* percorre todas as células da estrutura para desalocá-las, sendo cada *free* $O(1)$, e sendo $m < n$ o número de elementos na pilha, esta função é $O(n)$.

A função *getAllSolutions*, por sua vez, faz poucas operações. Inicialmente, cria uma pilha e aloca um vetor de caracteres, o que é $O(1)$, como mostrado anteriormente. Em seguida, chama a função *solveRPN*, que será avaliada na sequência. Efetua também a liberação da pilha e do vetor de caracteres, o que é $O(1)$, como já visto. Assim, até o momento, temos que a complexidade do algoritmo é $O(n) + O(1) + Complexidade_{solveRPN}$, isto é, $O(n) + Complexidade_{solveRPN}$.

Para avaliar a complexidade da função *solveRPN*, vamos utilizar o fato de que a árvore de recursão é completa e cheia, isto é, a árvore é estritamente binária e todos os nós folhas estão no mesmo nível. Isso ocorre devido ao fato de que, para cada estágio da recursão, sempre duas direções são exploradas: à esquerda para soma e à direita para multiplicação, como é possível observar na Figura 3. Dito isso, é importante ressaltar, também, que as operações contidas na função em questão são de inserção e remoção na pilha, além de operações aritméticas, com custo total $O(1)$. Assim, no pior caso, todos os nós da árvore são visitados, e a recursão só termina quando todos os nós folhas tiverem sido atingidos. Com isso, pode-se afirmar que o custo para resolver o problema da Notação Polonesa Reversa é equivalente ao custo de percorrer todos os nós da árvore de recursão e, portanto, é equivalente ao número de nós presentes na árvore. Dessa forma, seja k o número de operadores da expressão, o que é equivalente a altura da árvore. Considerando que a árvore começa no nível um e termina no nível k, temos que o custo para percorrer todos os nós nela presentes é dado pelo seguinte cálculo:

$$T(k) = \sum_{i=0}^{k-1} 2^i = 1 \cdot \frac{2^k - 1}{2 - 1}$$

$$\therefore T(k) = 2^k - 1$$

Neste ponto, retomemos a notação de que m é a quantidade de operandos, k é a quantidade de operadores e n é o número de elementos presentes na expressão, isto é, $n = m + k$. Temos que, para cada operando adicionado, é necessário inserir um operador. Assim, a quantidade de operadores pode ser definida como $k = m - 1$. Isolando m em $n = m + k$, temos que $m = n - k$. Desses dois resultados, obtemos que $k = \frac{n-1}{2}$. Dessa forma, provamos que a complexidade de tempo da função *solveRPN* é 2^n. Assim, agrupando com os resultados obtidos anteriormente, temos que a complexidade temporal deste algoritmo é $O(n) + O(2^n)$, ou seja, $O(2^n)$. Entretanto, embora a complexidade seja exponencial, como já citado seu tempo de execução é bem reduzido com o reuso de resultados calculados em níveis acima da recursão e do abandono dos cálculos caso o resultado obtido ultrapasse o valor esperado.

3.2 Complexidade Espacial

A utilização de espaço pelo algoritmo ocorre em duas situações: armazenamento da expressão e alocação de operandos e resultados na pilha. No primeiro caso, sendo n a quantidade de operadores somada à quantidade de operandos, ignorando as constantes para a análise assintótica, a complexidade espacial para tal é $O(n)$, pois é necessário alocar n posições na memória para guardar estes valores. Já para a pilha, são inseridos, no máximo, m elementos, onde m é o número de operandos. Isso se deve ao fato de que, para cada operador encontrado, dois elementos são removidos da pilha e o resultado de uma operação entre esses dois elementos é inserido na estrutura. Ou seja, serão inseridos, no máximo, m elementos, como é possível perceber na Figura 3. Sendo k a quantidade de operadores e observando que $m = k + 1$ e que $n = m + k$, temos que $n = 2m - 1$, ou seja, $m = \frac{n+1}{2}$. Temos, portanto, que a complexidade espacial para as operações na pilha é $O(n)$. Dessa forma, a complexidade espacial total deste algoritmo é $O(n)$.

4 Avaliação Experimental

Com o intuito de visualizar melhor o comportamento do tempo de execução do algoritmo quando submetido a variações da entrada, esta seção apresenta dois gráficos considerando alguns casos de teste exaustivos criados.

4.1 Análise Experimental do Pior Caso

Como abordado ao longo deste texto, o pior caso para o algoritmo implementado ocorre quando toda a árvore de recursão é explorada, isto é, em nenhum momento dos cálculos intermediários o valor obtido excedeu aquele esperado, fazendo com que todas as subárvores de execução para cada nó sejam percorridas. O gráfico da Figura 4 ilustra esta situação, comprovando o comportamento assintótico exponencial introduzido na seção anterior.

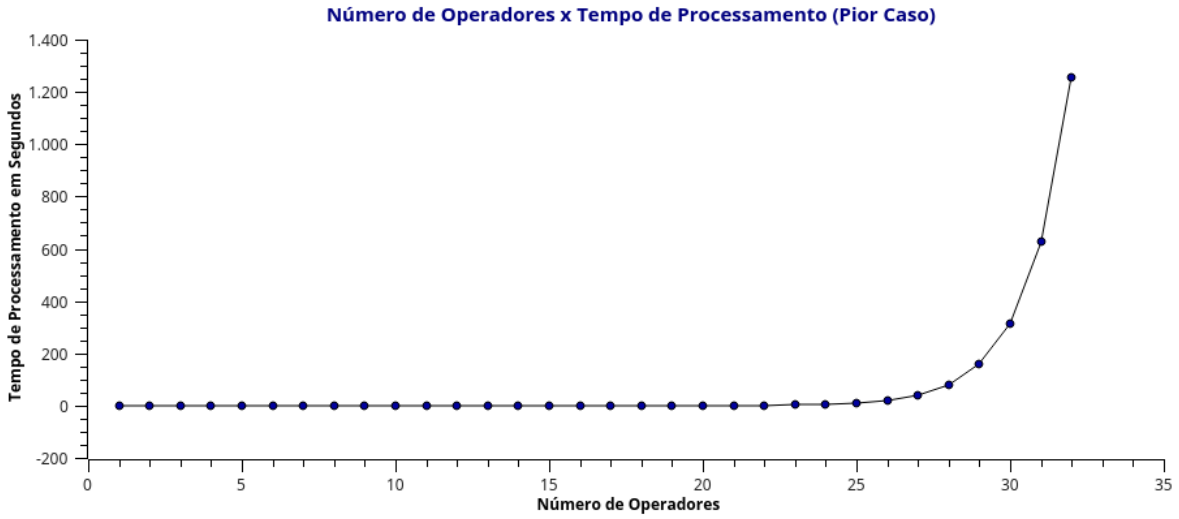

Figura 4 – Gráfico do tempo de execução em relação ao número de operadores para o pior caso

4.2 Análise Experimental do Caso Médio

O pior caso é uma situação que pode não ocorrer com tanta frequência, dada a distribuição não uniforme dos números. Dessa forma, se o resultado esperado for um valor intermediário no conjunto de operandos, isto é, ora exceder o resultado parcial na pilha e ora não, algumas subárvores serão dispensadas na análise da expressão, o que diminuirá o tempo de execução em alguns pontos. Isso é notável ao observar o gráfico da Figura 5, em que a curva exponencial é bem mais suave do que aquela da Figura 4. Vale ressaltar, porém, que a complexidade ainda é exponencial, mas a taxa de crescimento da curva é menor, podendo, inclusive, haver picos no tempo de execução para determinadas entradas, como ocorre no intervalo $[48, 50]$ e $[58, 60]$. Com isso, as otimizações decorrentes da recursão, apresentadas anteriormente, se comprovam satisfatórias.

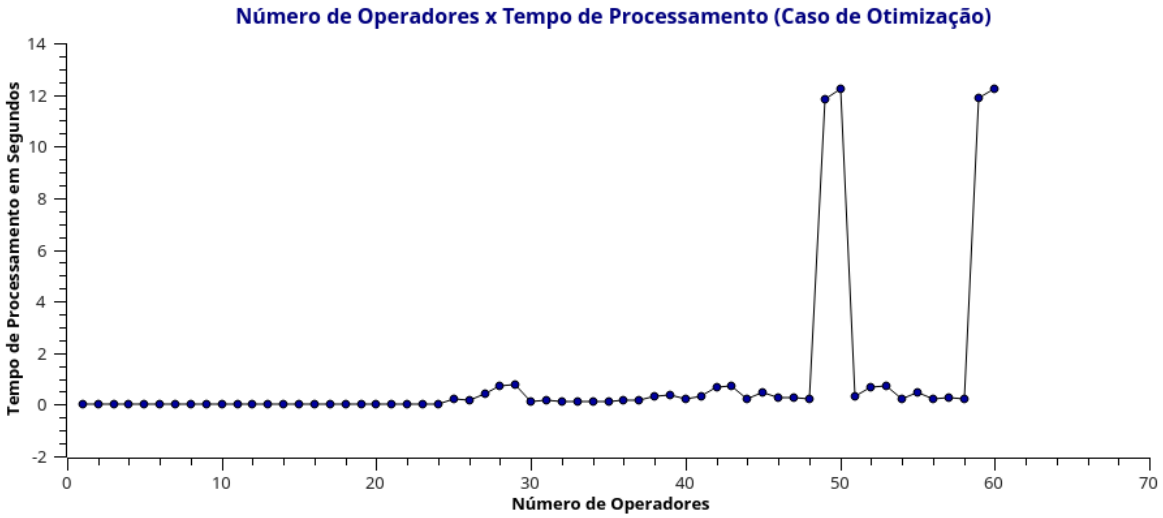

Figura 5 – Gráfico do tempo de execução em relação ao número de operadores para o caso médio

5 Conclusão

As notações para a escrita de expressões matemáticas, portanto, são de suma importância, visto que podem torná-las mais intuitivas e até reduzir custos computacionais para solucioná-las, como é o caso da Notação Polonesa Reversa. Assim, este trabalho apresentou uma solução para o problema de descoberta de todas as combinações de sequências de operadores em uma expressão RPN que resultam em um resultado estabelecido. Além disso, ficou comprovado que a estratégia de resolução por recursão foi bem proveitosa para o problema em questão, uma vez que permite a reutilização de resultados anteriores, reduzindo assim o tempo de execução para casos médios.

6 Referências

CORMEN, THOMAS H. et al. Introduction to Algorithms. Cambridge, 3ed. MIT Press, 2001.

Wikipedia. Reverse Polish Notation. Acesso em 20 abr 2016. Disponível em: https://en.wikipedia.org/wiki/Reverse_Polish_notation