PUC Minas
A B A K Ó S
Instituto de Ciências Exatas e Informática

Revista Abakós do Instituto de Ciências Exatas e de Informática Pontifícia Universidade Católica de Minas Gerais*

Model - Magazine Abakós - ICEI - PUC Minas

João Pedro Lobato de Pinho[1]
Vinícius Henrique Giovanini[2]

Resumo

Este artigo apresenta a implementação original do algoritmo CYK, o qual determina se uma palavra está contida ou não em uma determinada linguagem. No entanto, para garantir o funcionamento correto desse algoritmo, é necessário que a gramática esteja na Forma Normal de Chomsky (CNF). Para abordar esse requisito, o artigo inclui a implementação de um algoritmo de conversão, um passo preliminar necessário antes da execução do algoritmo principal. Outro ponto abordado no artigo diz respeito à codificação do CYK modificado, que compartilha o mesmo objetivo do algoritmo original, mas adota abordagens iniciais distintas. Neste caso, a gramática é assumida na forma 2NF, não na CNF. Para superar essa diferença, o algoritmo modificado emprega a binarização da gramática como etapa inicial. Vale ressaltar que, apesar das diferenças nos passos iniciais, ambos os algoritmos têm como meta determinar se uma palavra pertence à linguagem especificada. O CYK modificado, no entanto, destaca-se por sua menor complexidade na implementação, uma vez que atinge o mesmo resultado com menos dificuldade no processo de solução.

*Artigo apresentado à Revista Abakos
[1]Programa de Bacharelado em Ciências da Computação da PUC Minas, Brasil– joao.pinho.1275438@sga.pucminas.br
[2]Programa de Bacharelado em Ciências da Computação da PUC Minas, Brasil– vgiovanini@sga.pucminas.br

Abstract

This article presents the original implementation of the CYK algorithm, which determines whether a word is contained in a particular language or not. However, to ensure the correct operation of this algorithm, it is necessary for the grammar to be in Chomsky Normal Form (CNF). To address this requirement, the article includes the implementation of a conversion algorithm, a necessary preliminary step before executing the main algorithm. Another point addressed in the article concerns the coding of the modified CYK, which shares the same goal as the original algorithm but adopts different initial approaches. In this case, the grammar is assumed to be in 2NF, not CNF. To overcome this difference, the modified algorithm employs grammar binarization as an initial step. It is worth noting that despite the differences in the initial steps, both algorithms aim to determine whether a word belongs to the specified language. The modified CYK stands out, however, for its lower implementation complexity, achieving the same result with less difficulty in the solution process.

1 INTRODUÇÃO

A análise de sentenças em linguagens livres de contexto (LLC) desempenha um papel crucial em diversos domínios, como o projeto de compiladores, bioinformática e linguística. O problema de verificar a pertinência de uma sentença em relação a uma LLC, dada sua gramática, é fundamental em muitas aplicações práticas. Neste contexto, o algoritmo CYK, proposto por Cocke, Younger e Kasami, destaca-se como uma abordagem eficaz para resolver esse desafio, empregrando programação dinâmica e alcançando uma complexidade de O(n^3) para uma gramática de tamanho fixo.

Entretanto, a aplicação do algoritmo CYK exige que a gramática esteja na forma normal de Chomsky (CNF), tornando sua compreensão, ensino e implementação mais complexos. A transformação de uma gramática livre de contexto (GLC) para a CNF pode ser custosa, com um potencial custo exponencial se não for cuidadosamente implementada. Para superar essas limitações, Lange e Leiβ propuseram em 2009 uma modificação do algoritmo CYK, adaptando-o para lidar com gramáticas em forma normal binária (2NF). Essa abordagem simplifica significativamente a implementação, eliminando a necessidade de converter a gramática para CNF.

Este trabalho propõe a implementação e comparação de dois métodos para verificar a pertinência de uma sentença em uma LLC: (i) o algoritmo CYK original e (ii) o algoritmo CYK modificado proposto por Lange e Leiβ. Ambas implementações foram feitas em Java e receberam como entrada uma descrição de uma GLC, juntamente com a(s) sentença(s) a serem verificadas. Os resultados foram registrados em arquivos de saída, e experimentos foram conduzidos para avaliar o desempenho em questão de tempo das duas estratégias aplicadas a gramáticas de diferentes tamanhos.

2 AMBIENTE DE DESENVOLVIMENTO

O ambiente de desenvolvimento onde foram realizado todos os experimentos com os tipos de gramática tem as características listadas abaixo:

1: **Sistema Operacional:** Windows 10.

2: **Processador:** Intel(R) Core(TM) i5-6200U CPU @ 2.30 GHz 2.40GHz.

3: **Memória Ram:** 8 GB.

4: **JDK:** Eclipse Temurin JDK with Hotspot 17.0.5+8 (x64)

Os códigos foram desenvolvidos no visual studio code com a criação de projetos padrão para java. Para realizar os experimentos foram utilizados tipos de gramática com diferentes tamanho, com o objetivo de avaliar o desempenho do algoritmo em diferentes casos.

A avaliação do desempenho foi realizada medindo o tempo de execução do algoritmo em cada caso, considerando o tempo total para encontrar se a frase pertence ou não a gramática.

Para a execução do algoritmo foram utilizados argumentos padrões de uma execução em Java nas depurações do visual studio code, sem nenhuma alteração. O algoritmo espera de entrada o nome de um arquivo (para gramática) e um nome de outro arquivo contendo as frases de teste. A saída é direcionada para um arquivo com nome "resultados-original.txt"ou "resultados-modificado.txt"a depender da versão do CYK escolhida para execução.

3 DESENVOLVIMENTO

Nas seções a seguir, serão abordados detalhes sobre a implementação dos métodos e algoritmos feitos em Java.

3.1 Classe Gramatica

A classe `Gramatica` desempenha o processamento e manipulação de gramáticas. Seu código é projetado para a leitura de gramáticas a partir de arquivos de texto, facilitando a preparação dessas estruturas para posterior análise pelos algoritmos CYK. Abaixo, são destacados alguns pontos importantes relacionados à classe:

- **Leitura de Gramáticas**: O método `readGrammar` permite a leitura de uma gramática a partir de um arquivo dado como parâmetro na execução. Durante esse processo, a classe atenta para a estrutura de não-terminais e regras associadas, mapeando-as em um formato que facilite operações subsequentes.
- **Estrutura de Dados**: A gramática lida é representada internamente por um mapa, onde cada não-terminal é associado a uma lista de regras. Essa estrutura é central para as manipulações e verificações realizadas pela classe.
- **Restrições da Gramática Lida**:
    1. **Ausência de Números**: A gramática não pode conter números, seja em não-terminais ou terminais. A representação é restrita a letras e caracteres especiais, excluindo a exclamação "!". Números podem ser gerados aleatoriamente durante o processo de conversão para CNF.
    2. **Uso do Símbolo "!"para Transição Vazia**: O símbolo de exclamação "!"é adotado para denotar uma transição vazia (lambda - λ). Essa escolha contribui para a expressividade da gramática e é especialmente relevante na manipulação de linguagens formais.
    3. **Possibilidade de Formatos 2NF ou CNF**: A gramática lida pode estar ou não no formato de Forma Normal Binária (2NF) ou Forma Normal de Chomsky (CNF). A classe é capaz de lidar com ambas as situações, permitindo flexibilidade quanto à entrada do usuário.

3.2 Conversão para a Forma Normal de Chomsky

A Forma Normal de Chomsky (FNC ou CNF) é uma representação específica de gramáticas livres de contexto que simplifica a estrutura das regras de produção. Uma gramática está na FNC se todas as suas regras de produção são de dois tipos:

1. **Regras de Produção Binárias:** Cada regra de produção na FNC tem a forma `A -> BC`, onde `A`, `B` e `C` são símbolos não-terminais. Essas regras garantem que cada não-terminal gere exatamente dois símbolos, simplificando a estrutura da gramática.

2. **Regras de Produção Terminais:** Cada regra de produção terminal tem a forma `A -> a`, onde `A` é um não-terminal e `a` é um terminal. Essas regras permitem que os não-terminais gerem terminais individualmente.

O processo de conversão para a FNC envolve vários passos, como a remoção de transições vazias, transições unitárias e símbolos inúteis. Além disso, é necessário converter terminais em não-terminais e ajustar as regras correspondentes. Duplas de maiúsculas em regras também são convertidas em novos não-terminais. Cada um desses passos contribui para a simplificação da gramática e a conduz para a Forma Normal de Chomsky. A seguir, estão descritos cada um dos passos principais.

3.2.1 Remoção de Transições Vazias

O primeiro passo consiste na remoção das transições vazias da gramática. Este processo envolve a identificação dos não-terminais que possuem transições vazias, a remoção do símbolo de transição vazia (!), e o ajuste das regras afetadas.

Para remover as transições vazias, primeiramente é necessário obter a lista com todas as transições que geram vazio na gramática, sejam essas transições diretas ou indiretas. Diretas seriam aquelas que tem "!"(λ) diretamente em sua regra, e indiretas seriam aquelas que contém um não-terminal isolado pertencente a lista de transições vazias.

O método pega as transições vazias de forma recursiva, começa a lista inicialmente com o símbolo "!"e ao encontrar um não-terminal que gera esse símbolo, o adiciona na lista e chama recursivamente o método, passando pela gramática novamente. Na próxima passada, ele irá procurar novamente pelo símbolo "!"juntamente com a ocorrência isolada do novo não-terminal adicionado, até adicionar todas as ocorrências diretas e indiretas possíveis.

O próximo passo, com a lista de transições vazias em mãos, envolve a remoção das regras necessárias e a substituição por possíveis derivações. O pseudo-código abaixo mostra como foi feita a implementação.

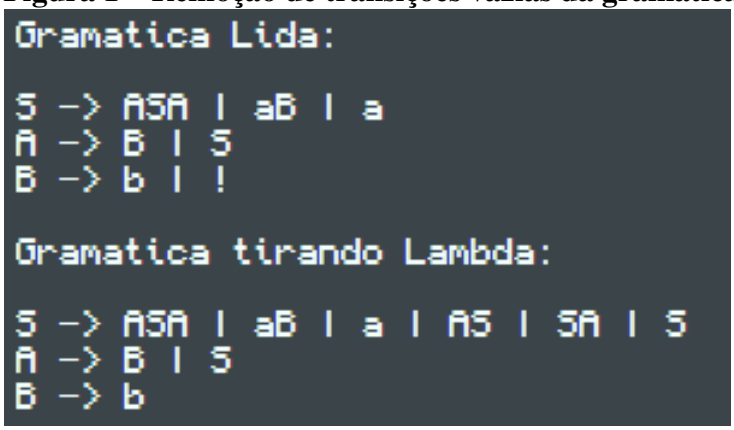
Algorithm 1: Removendo Lambda da Gramática

Require: $transicoesVazias$: lista, glc: gramática original
Ensure: Gramática atualizada sem produções vazias

```
1:  glcCopy ← nova instância de LinkedHashMap contendo cópia de glc
2:  for all (naoTerminal, regras) em glc do
3:      regrasCopy ← cópia de regras excluindo o símbolo lambda ("!")
4:      for all elementoLista em regras do
5:          for all eachTV em transicoesVazias do
6:              if (elementoLista.contém(eachTV)) then
7:                  if (elementoLista.comprimento == 2) then
8:                      resultado ← substituir eachTV por uma string vazia em elementoLista
9:                      if (!regrasCopy.contém(resultado)) then
10:                         regrasCopy.adicionar(resultado)
11:                     end if
12:                 else
13:                     possibilidades ← Derivacoes.derivacaoPalavra(elementoLista, eachTV[0])
14:                     for all string em possibilidades do
15:                         if (string ≠) then
16:                             regrasCopy.adicionar(string)
17:                         end if
18:                     end for
19:                 end if
20:             end if
21:         end for
22:     end for
23:     glcCopy.adicionar(naoTerminal, regrasCopy)
24: end for
25: return glcCopy
```

O algoritmo acima tem como objetivo remover as produções vazias da gramática dada por parâmetro. O algoritmo percorre cada não terminal e suas regras na gramática original, criando uma cópia. Durante esse processo, elimina o símbolo lambda ("!") se presente nas regras. Em seguida, para cada regra, verifica se há transições vazias e realiza as substituições necessárias, adicionando os resultados em uma cópia das regras. A gramática copiada é então atualizada com as regras modificadas, resultando em uma nova gramática sem produções vazias. A imagem abaixo mostra a execução da remoção de lambda.

Figura 1 – Remoção de transições vazias da gramática



Fonte: Elaborado pelo autor.

3.2.2 Remoção de Transições Unitárias

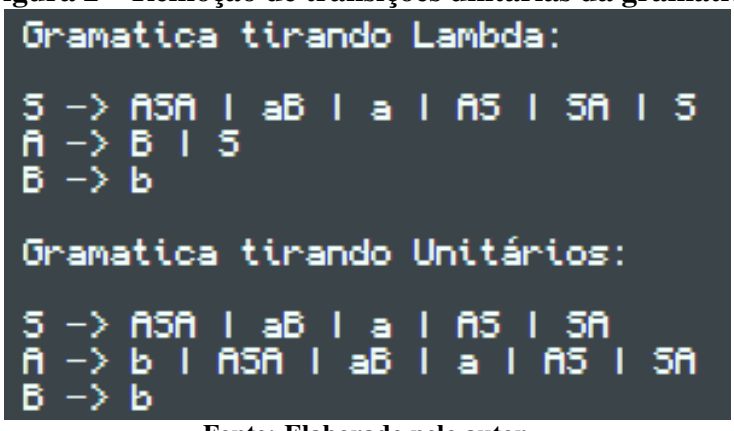
O segundo passo lida com as transições unitárias na gramática, eliminando regras em que um não-terminal se converte diretamente em outro não-terminal. O pseudo-código abaixo descreve os passos.

Algorithm 2: Remoção de Transições Unitárias

1: **Entrada:** Gramática Livre de Contexto (GLC)
2: **Saída:** GLC sem transições unitárias
3: Remover regras que geram elas mesmas
4: Remover transições unitárias copiando conteúdo
5: **return** GLC sem transições unitárias

A mesma gramática agora com a execução desse passo fica da seguinte forma:

Figura 2 – Remoção de transições unitárias da gramática



Fonte: Elaborado pelo autor.

O terceiro passo visa remover símbolos inúteis da gramática, ou seja, não-terminais que não são alcançados a partir da regra inicial, regras que entram em loop, regras iguais, dentre outras. Em geral, esse passo não realiza grandes operações na gramática, depende muito de como está a sua configuração inicial para serem necessárias remoções ou ajustes para sua

aplicação.

3.2.3 Conversão de Terminais

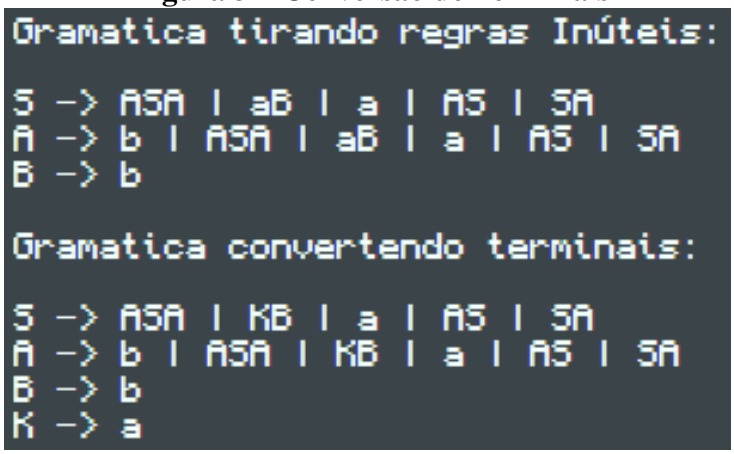
O quarto passo envolve a conversão de terminais em não-terminais, gerando novos não-terminais para cada terminal e ajustando as regras correspondentes. O algoritmo pode ser representado da seguinte forma:

Algorithm 3: Conversão de Terminais

1: **Entrada:** Gramática Livre de Contexto (GLC)
2: **Saída:** GLC com terminais convertidos
3: Para cada terminal, gerar um novo não-terminal
4: Substituir terminais nas regras pelos novos não-terminais
5: **return** GLC com terminais convertidos

Aqui são geradas letras aleatórias para substituírem os terminais, caso essa letra já exista no conjunto de não-terminais, é adicionado um número a letra, ou seja, se a gramática tem um não-terminal `A` e a letra aleatória para substituir determinado terminal seja novamente `A`, o resultado será `A` seguido de zero, e soma-se um para cada nova ocorrência de outra letra aleatória que gere `A` novamente. Por isso a entrada de arquivos não pode conter números na leitura da gramática, estando restrita a letras de a à z e símbolos especiais, excluindo o "!". A imagem abaixo mostra como fica a execução do quarto passo.

Figura 3 – Conversão de Terminais



Fonte: Elaborado pelo autor.

Pode-se observar que apenas os terminais que estão seguidos de um não-terminal são convertidos para novos não-terminais, isso se deve ao fato de que terminais isolados já estão conforme a Forma de Chomsky.

3.2.4 Conversão de Não-Terminais

O quinto passo consiste na conversão de não-terminais remanescentes, gerando novos não-terminais para substituir duplas de maiúsculas nas regras existentes. Segue a mesma ideia da geração de letras aleatórias para substituição. O algoritmo correspondente pode ser descrito como segue:

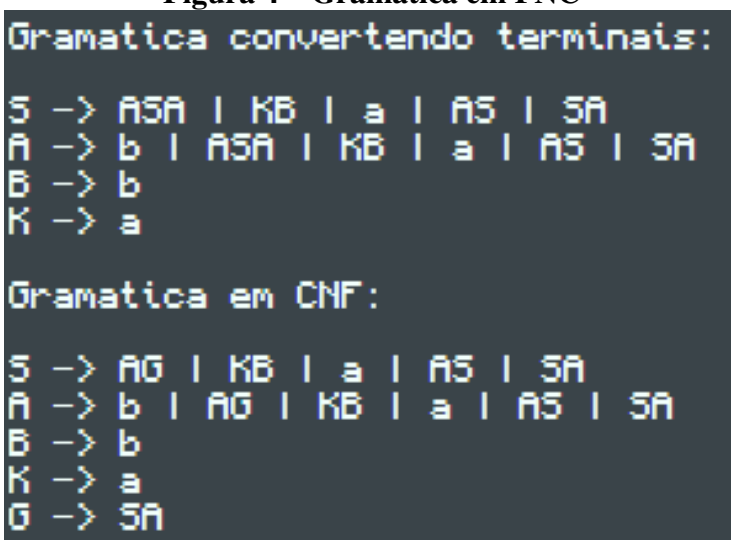
Algorithm 4: Conversão de Não-Terminais

1: **Entrada:** Gramática Livre de Contexto (GLC)
2: **Saída:** GLC com não-terminais convertidos
3: Para cada dupla de maiúsculas nas regras, gerar um novo não-terminal
4: Substituir duplas de maiúsculas pelos novos não-terminais
5: **return** GLC com não-terminais convertidos

Por fim, a classe `FormaNormalChomsky` coordena todos os passos mencionados anteriormente para produzir uma gramática na Forma Normal de Chomsky. A implementação completa dessa conversão é obtida pela aplicação sequencial dos algoritmos descritos.

Este conjunto de passos assegura que a gramática original seja convertida para a Forma Normal de Chomsky, facilitando análises e manipulações subsequentes e sendo possível sua utilização no algoritmo original CYK.

Figura 4 – Gramática em FNC



Fonte: Elaborado pelo autor.

4 CONVERSÃO PARA 2NF

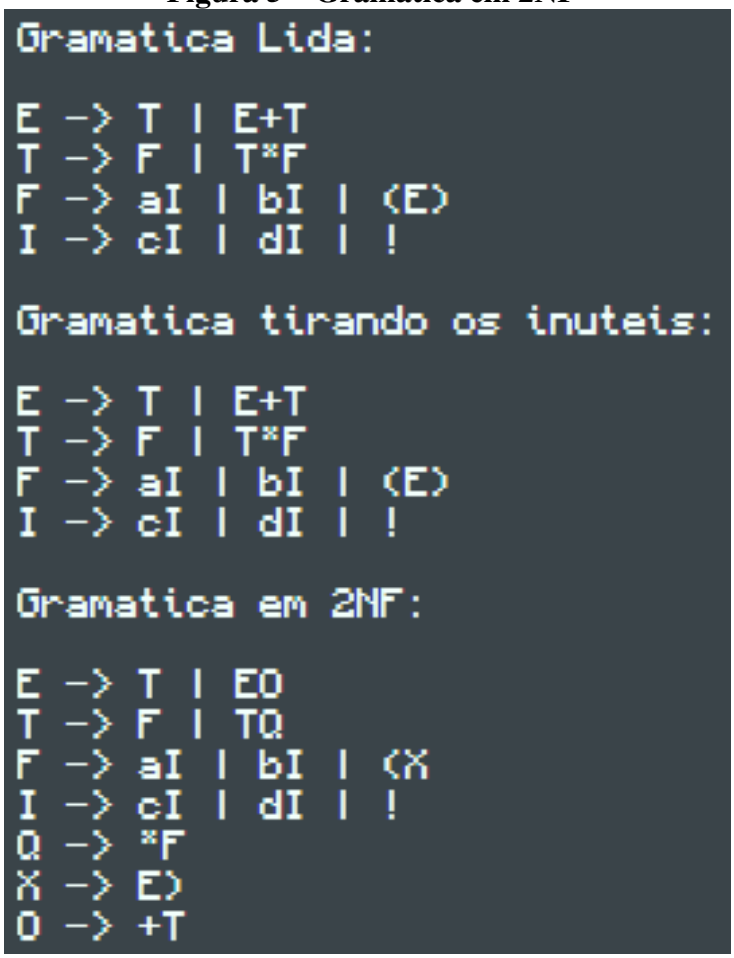
A 2NF é uma representação especial para gramáticas livres de contexto descrita no artigo (LANGE; LEISS, 2009). Uma gramática está na 2NF (forma binária) quando todas as regras de produção são da forma dois elementos ou apenas um. A seguir está um resumo do método

utilizado:

1: **Entrada:** Gramática Livre de Contexto (GLC)
2: **Saída:** GLC na Forma Normal de Chomsky (2NF)
3: $glc \leftarrow$ gramática de entrada
4: $glc \leftarrow$ `RemoverTransicoesInuteis.removerInuteis`(glc)
5: $glc \leftarrow$ `ConverterNaoTerminais.converterNaoTerminais`(glc)
6: **return** glc

A classe `Forma2NF` implementa parte da conversão para a Forma Normal de Chomsky, quando é feito a binarização na conversão de não-terminais, o código é apenas reutilizado. A figura abaixo mostra a mesma gramática utilizada de exemplo no artigo convertida para 2NF.

Figura 5 – Gramática em 2NF



Fonte: Elaborado pelo autor.

5 ALGORITMO CYK ORIGINAL

O algoritmo CYK (Cocke-Younger-Kasami) é um algoritmo de parsing bottom-up para determinar se uma dada string pode ser gerada por uma gramática livre de contexto (GLC). Ele

utiliza uma tabela dinâmica para armazenar informações sobre a derivação da string a partir das regras da gramática.

Entrada: Gramática Livre de Contexto (GLC), Palavra de Entrada
Saída: Verdadeiro se a palavra pertence à gramática, Falso caso contrário
Inicializar uma tabela $tabela$ com dimensões $n \times n$, onde n é o comprimento da palavra.
for i de 0 até $n - 1$ **do**
    Preencher $tabela[i][i]$ com não-terminais que geram o terminal correspondente na palavra.
end for
for j de 1 até $n - 1$ **do**
    for i de $j - 1$ até 0 **do**
        for h de i até $j - 1$ **do**
            for Cada não-terminal A na gramática **do**
                for Cada regra $A \rightarrow BC$ na gramática **do**
                    if $tabela[i][h]$ contém B e $tabela[h + 1][j]$ contém C **then**
                        Adicionar A a $tabela[i][j]$
                    end if
                end for
            end for
        end for
    end for
end for
return **Verdadeiro** se $tabela[0][n - 1]$ contém o símbolo inicial da gramática, **Falso** caso contrário.

O algoritmo CYK preenche a tabela com informações sobre as derivações possíveis e verifica se a palavra pode ser derivada a partir do símbolo inicial da gramática. A implementação foi feita em uma classe separada e foram realizados alguns testes para verificar o tempo gasto para executar o algoritmo, juntamente com a conversão para a gramática para a Forma Normal de Chomsky, já que o CYK original recebe a gramática nessa forma. A seguir foram documentados os testes realizados com diferentes tamanhos de gramática e diferentes tamanhos de sentença.

O primeiro teste foi com a gramática presente no artigo, descrita abaixo:

$$E \rightarrow T \mid E + T$$
$$T \rightarrow F \mid T * F$$
$$F \rightarrow aI \mid bI \mid (E)$$
$$I \rightarrow cI \mid dI \mid !$$

A gramática possui 4 não-terminais e 10 regras (antes da conversão para FNC). As letras minúsculas e caracteres especiais são terminais. Os testes foram feitos com diversas sentenças

que pertencem ou não à gramática, como: (ac+b)*a, baabab, adccc, dentre outros padrões, inclusive com sentenças que possuíam letras e símbolos não pertencentes ao alfabeto. O tempo médio de conversão para CNF foi de 28 milissegundos, e o tempo de testes para cada sentença de diferentes tamanhos no CYK original pode ser observado no gráfico abaixo:

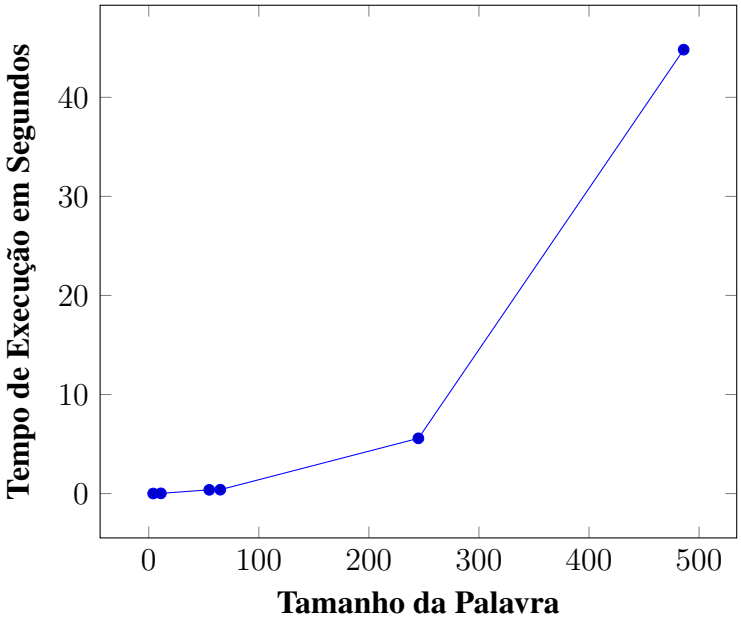


Figura 6 – Teste para a Gramática.

O segundo teste foi com uma gramática já em FNC, descrita abaixo:

$$S \rightarrow SE \mid a \mid AX \mid AB$$
$$L \rightarrow AX \mid AB$$
$$E \rightarrow a \mid AX \mid AB$$
$$A \rightarrow ($$
$$B \rightarrow)$$
$$X \rightarrow SB$$

A gramática possuí dessa vez 6 não-terminais, três terminais e 12 regras. Novamente os testes foram feitos com sentenças aleatórias de diversos tamanhos que pertencem ou não a gramática, como (a), aaa, ((())), dentre outras. O tempo médio para a conversão para CNF, foi novamente 28 milissegundos, mesmo a gramática já estando na Forma Normal de Chomsky, o método passa para verificar sua adequação, o gráfico abaixo mostra o tempo obtido para diferentes tamanhos de palavras:

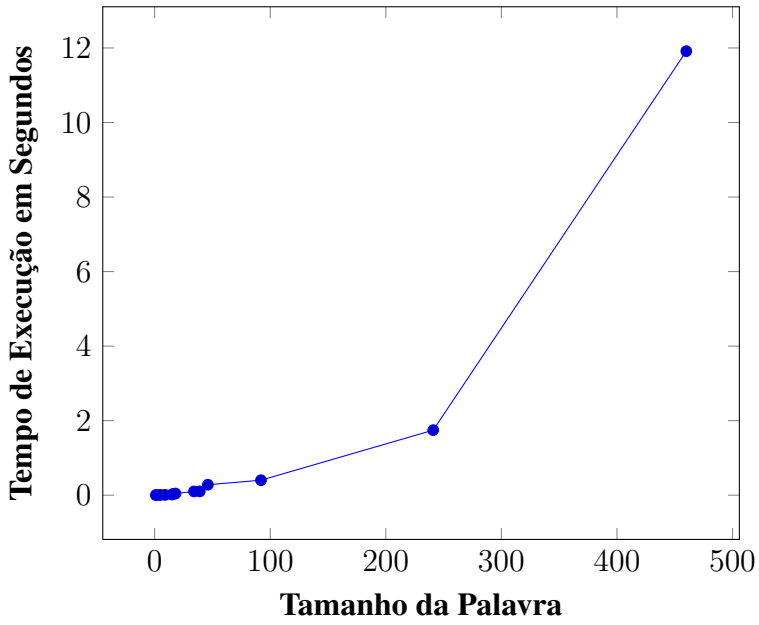


Figura 7 – Teste para segunda Gramática.

6 ALGORITMO CYK MODIFICADO

O algoritmo Cocke-Younger-Kasami Modificado (CYK Modificado) é uma variação do algoritmo CYK, proposta no artigo e também é utilizada para verificar se uma determinada palavra pertence a uma Gramática Livre de Contexto (GLC). O método recebe a gramática original, uma outra gramática U_G que contém a relação unitária como é descrita no artigo e a sentença a ser verificada.

O primeiro passo é gerar a gramática U_G a partir da 2NF, para isso são seguidas algumas rotinas, começando com obter uma gramática com somente unitários, como no exemplo abaixo:

Figura 8 – Gramática de Unitários

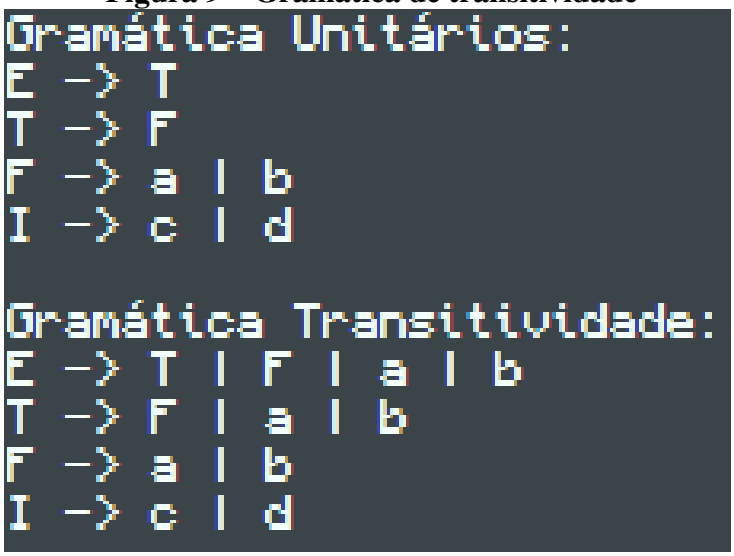
```
Gramática Lida:
E -> T | EX
T -> F | TZ
F -> aI | bI | (N
I -> cI | dI | !
X -> +T
Z -> *F
N -> E)

Gramática Unitários:
E -> T
T -> F
F -> a | b
I -> c | d
```

Fonte: Elaborado pelo autor.

Nesse passo, é importante observar que regras que geram vazio também devem ser adi-

cionadas mesmo não sendo propriamente unitárias, como em "aI, bI, cI e dI". Em seguida o próximo passo é obter a gramática com a transitividade de cada elemento, isto é, a partir de um elemento X, quais outros ele pode alcançar:

Figura 9 – Gramática de transitividade

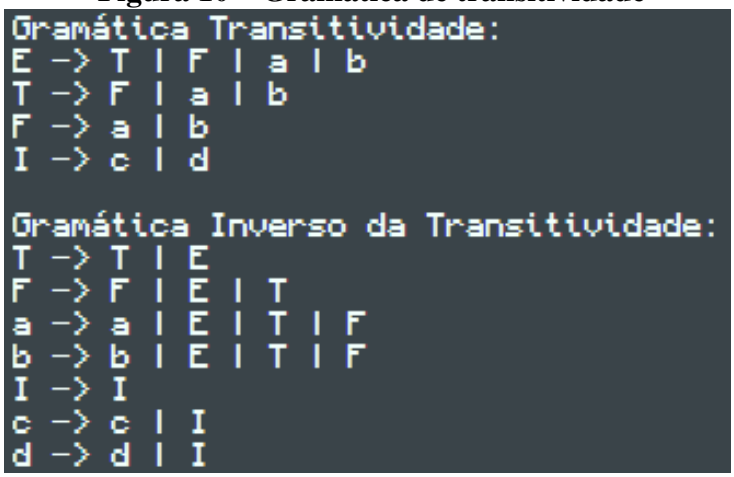


Fonte: Elaborado pelo autor.

Na imagem acima, o elemento `E` alcança `T`, que por sua vez alcança `F` e `F` alcança `a` e `b`. A mesma lógica é seguida para os outros elementos.

Por fim, para obter a gramática U_G cria-se uma nova gramática que a lógica agora é que, dado um elemento x, quais elementos chegam em x, como na imagem abaixo:

Figura 10 – Gramática de transitividade



Fonte: Elaborado pelo autor.

Com a gramática original e a gramática U_G em mãos agora é possível realizar a implementação do algoritmo CYK modificado. A implementação em Java utiliza duas tabelas, `tabela` e `tabela1`, para armazenar conjuntos de não-terminais associados a subpalavras da entrada. O algoritmo segue os seguintes passos:

1. **Inicialização:** Duas tabelas são criadas e inicializadas com listas vazias.

2. **Preenchimento da Tabela Inicial:** As regras da gramática inversa são utilizadas para preencher a tabela inicial com não-terminais associados aos caracteres da sentença de entrada teste.

3. **Preenchimento das Tabelas:** O algoritmo utiliza uma abordagem bottom-up para preencher as tabelas, verificando combinações de não-terminais que geram substrings da palavra.

4. **Verificação Final:** A tabela final é verificada para determinar se a palavra pertence à gramática, com base na presença do primeiro não-terminal da gramática.

5. **Impressão de Tabelas:** Funções auxiliares são utilizadas para imprimir as tabelas inicial e final em arquivos de texto para fins de depuração.

Para fins de comparação, os testes foram feitos com as mesmas gramáticas e frases executadas no CYK original. Para a primeira gramática, o tempo de conversão para 2NF + o tempo para gerar a gramática U_G (pré-requisitos para o CYK Modificado) foi de 55 milissegundos, e o tempo de testes para cada sentença de diferentes tamanhos pode ser observado no gráfico abaixo:

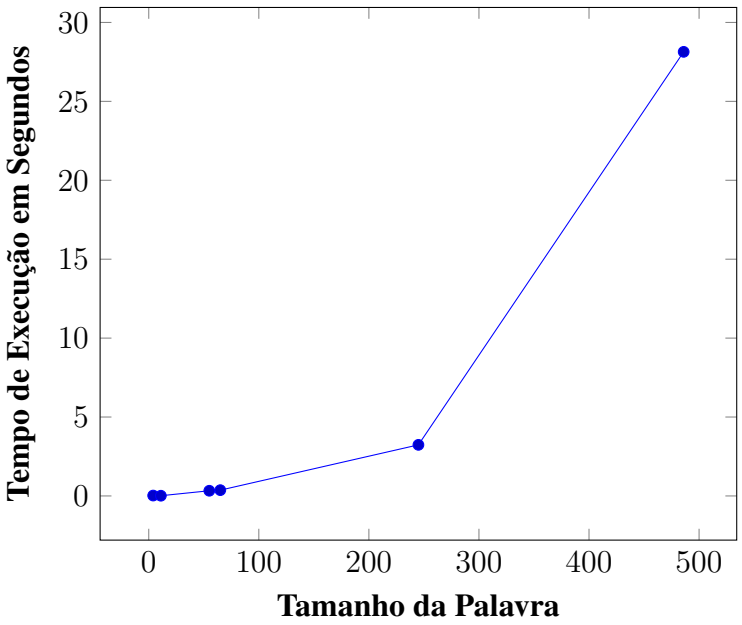


Figura 11 – Teste para a primeira Gramática.

É perceptível que o tempo médio abaixou em relação ao uso do algoritmo original. Em relação ao segundo teste, agora com o modificado e o tempo médio para a conversão 2NF + o tempo para gerar a gramática U_G foi de 39 milissegundos. O próximo gráfico mostra como foram os novos tempos para as mesmas sentenças de teste:

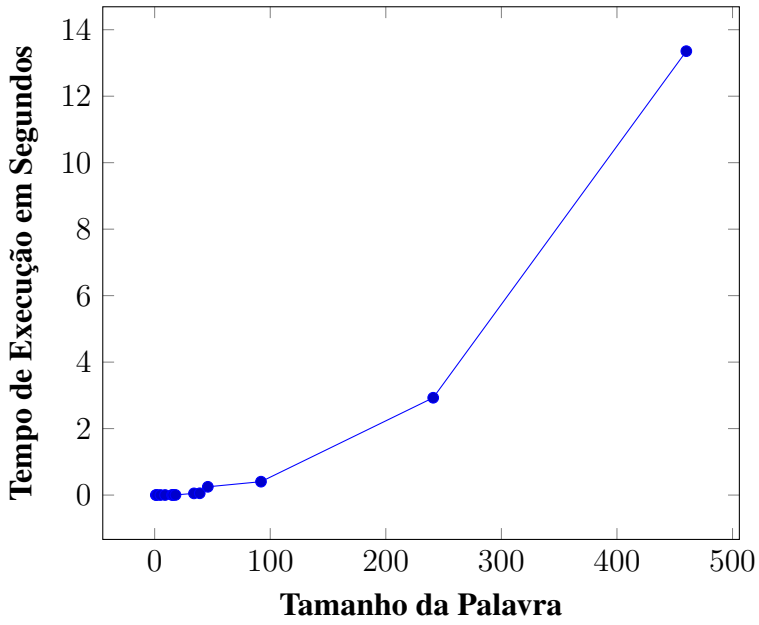
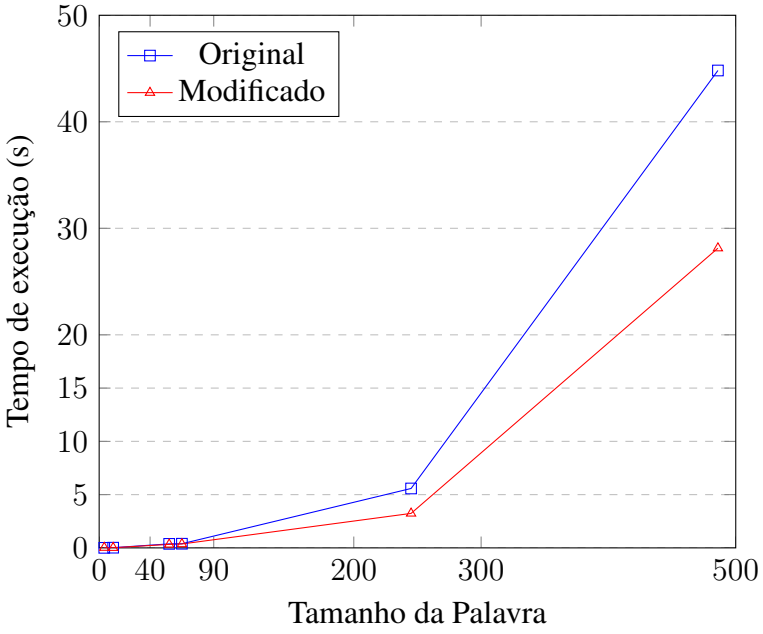
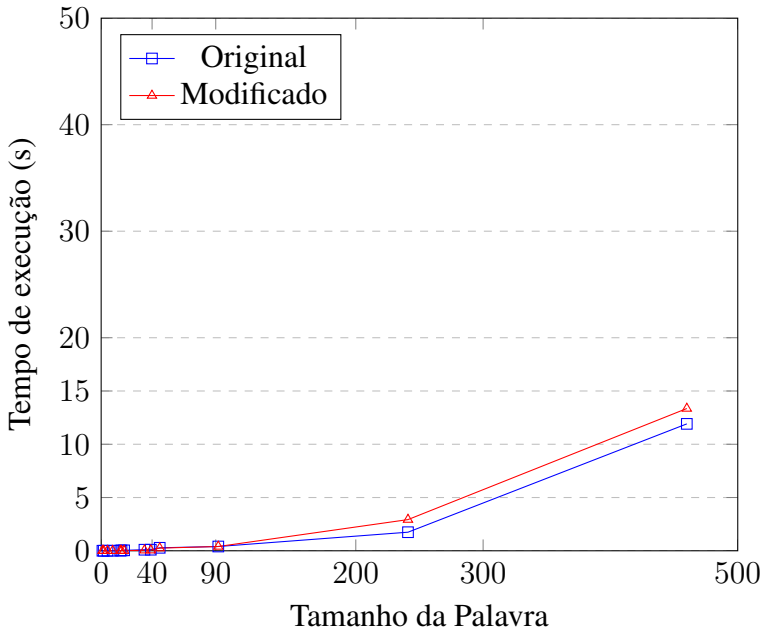


Figura 12 – Teste para segunda Gramática.

Para a segunda gramática os tempos foram menores até determinado ponto, chegando em um certo tamanho de palavra, o cyk original foi melhor que o modificado por poucos segundos. Mas, o tempo geral de teste de todas as sentenças juntas ainda é menor na versão modificada.

Por fim, os gráficos abaixo mostram um comparativo de tempo de ambas as gramáticas executadas nas duas versões do algoritmo:





É possível observar a partir dos gráficos que a quantidade de regras e não-terminais influência diretamente no tempo necessário para obter a resposta. No caso do CYK modificado, como ele usa duas tabelas com o tamanho da palavra ao invés de uma dependendo do número de regras a ser alocado e pesquisado nas tabelas sua execução tende a demorar mais.

7 CONCLUSÃO

Concluímos que, ao considerar a implementação e complexidade dos algoritmos, o CYK original demonstra ser mais exigente do que o CYK modificado. Isso se deve à sua necessidade de uma gramática na Forma Normal de Chomsky (CNF), envolvendo mais etapas quando comparada à abordagem binária. Essa característica torna a implementação do CYK original mais complexa. Quanto ao desempenho, observamos que o CYK modificado superou o original. Esse resultado é atribuído às características da gramática de entrada, à quantidade de regras e não-terminais, fatores que exercem influência direta no desempenho do algoritmo. Além disso, a menor quantidade de passos necessários para chegar à solução contribui para a eficiência do CYK modificado.

Referências

LANGE, Martin; LEISS, Hans. To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. **Informatica Didact.**, v. 8, 2009. Disponível em: <https://www.informaticadidactica.de/index.php?page=LangeLeiss2009_en>.