Trabalho Prático 2
Fundamentos Teóricos da Computação

Rossana Oliveira Souza [1], **Rafael L. Murta**[2]

[1]Instituto de Ciências Exatas e Informática – Pontíficia Universidade de Minas Gerais (PUC MG)

Resumo. *Este artigo descreve a implementação do Segundo Trabalho Prático da disciplina de Fundamentos Teóricos da Computação do Curso de Ciência da Computação da PUC Minas, que consiste na implementação do algoritmo cyk, do cyk-modificado, bem como da forma normal de Chomsky e da segunda forma normal proposta por Lange e Leiß e a análise de complexidade desses algoritmos.*

1. Introdução

O algoritmo de Ocke-Younger-Kasami (CYK) determina se uma cadeia de caracteres pode ser gerada por uma determinada gramática livre de contexto ou não. No entanto, esse algoritmo possui algumas limitações, por ser caro computacionalmente e por não conseguir fazer a análise sintática da cadeia em qualquer formato de gramática. A versão padrão desse algoritmo tem complexidade O(N^3), sendo N o tamanho da gramática, além disso, ele possui a limitação de conseguir analisar apenas gramáticas que estejam na forma normal de Chomsky. Na literatura fala-se muito do algoritmo CYK, pois ele foi o primeiro a resolver esse problema computacionalmente. Não obstante, apesar de ter solução em tempo polinomial, é interessante que a tranformação da gramática livre de contexto para uma equivalente na forma normal de Chomsky também seja considerada ao fazer a análise de complexidade desse algoritmo, uma vez que ele não é capaz de processar uma gramática que não esteja nessa forma. É nesse ponto que o custo computacional do CYK pode se tornar caro e suas limitações se tornam um problema. Algoritmos mais rápidos e especializados têm surgido na tentativa de otimizar esse problema. Neste documento vamos entender mais sobre a CNF o algoritmo CYK e uma variação de ambos. A segunda forma normal (2NF) e o algoritmo CYK-modificado são ambos inspirados nos originais e menos caros computacionalmente.

O repositório do github com as implementações completas pode ser encontrado em: `https://github.com/RLMurta/tp2_ftc`

2. Forma Normal de Chomsky

A forma normal de Chomsky (CNF) é uma maneira de simplificar uma gramática livre de contexto. Uma gramática é dada por variados símbolos à esquerda, e suas regras, símbolos à direita. Na CNF as regras na direita não podem ter mais do que dois símbolos, sendo duas variáveis ou um teminal. Veja um exemplo:
$A \rightarrow BC|a$
$B \rightarrow b$
$C \rightarrow c$

É importante ressaltar que toda gramática na forma normal de Chomsky é livre de contexto, e inversamente, toda gramática livre de contexto pode ser transformada em uma equivalente que está na forma normal de Chomsky. Para passar uma gramática livre de contexto para a CNF deve-se seguir os seguintes passos:
1. Introduzir uma nova regra inicial S_0, que gera a regra da variável inicial anterior $S_0 \rightarrow S$;
2. Elimiar todas as regras do tipo epsilon ϵ (regras que geram vazio ou nulo);
3. Elimine os terminais que não estiverem sozinhos como produção de regra. Por exemplo $X \rightarrow xY$ pode ser decomposta como $X \rightarrow XY, Y \rightarrow x$;
4. Elimine as regrs com mais de duas variáveis. Por exemplo, a regra $X \rightarrow XYZ$, pode ser decomposta como $X \rightarrow PZ, P \rightarrow XY$

A implementação de algoritmo que transforma uma linguagem livre de contexto numa gramática equivalente CNF pode ser encontrada em: `https://github.com/RLMurta/tp2_ftc/blob/main/Cfg2Cnf.py`. Este algoritmo segue o passo a passo descrito anteriormente e mais algumas etapas para complementar a lógica de programação.
O código recebe uma GLC qualquer na forma:

```
S => XB | ABC
X => AS | aX
A => a
B => b
C => c
```

Separa as regras das variáveis geradoras, cria uma lista de variáveis e suas respectivas regras:

```
    for line in lines:
      parts=line.split("=>")
      parts[0]=parts[0].strip()
      parts[1]=parts[1].strip()
      parts[1]=parts[1].split("|")
      if not parts[0] in self.rules:
        self.rules[parts[0]]=[]
      for part in parts[1]:
        part=part.strip()
        self.rules[parts[0]].append(part)
```

Define uma nova variável inicial:

```
  def new_name_for_grammar(self, array):
    names=list(array)
    for alpha in self.alphas:
      if alpha not in names:
        return alpha
```

Onde houver regras maiores que 2 e terminais não sozinhas, novas regras são geradas, terminais são colocadas sozinhas e uma nova regra (variável), é criada para elas. Onde houver a concatenação de mais de duas variáveis, também há a separação delas e novas regras são criadas:

```
def binarization(self, array):
    for key in list(array):
        index = 0
        count = len(array[key])
        while index < count:
            length = len(array[key][index])
            if length > 2:
                values = array[key][index][1:]
                name = self.new_name_for_grammar(array)
                array[name] = []
                array[name].append(values)
                array[key][index] = array[key][index].replace(value
            index += 1
    return array
```

3. CYK

O algoritmo monta uma matriz com número de linhas linhas e colunas igual ao tamanho da palavra de entrada, a qual será analisado se pertence ou não à gramática. A matriz é preenchida de baixo pra cima e na base são inseridos os dados da própria sentença a ser computada. Na linha superior a base são anotadas as regras que geram cada um dos valores da palavra de entrada. Vamos ver um exemplo de funcionamento do algoritmo:

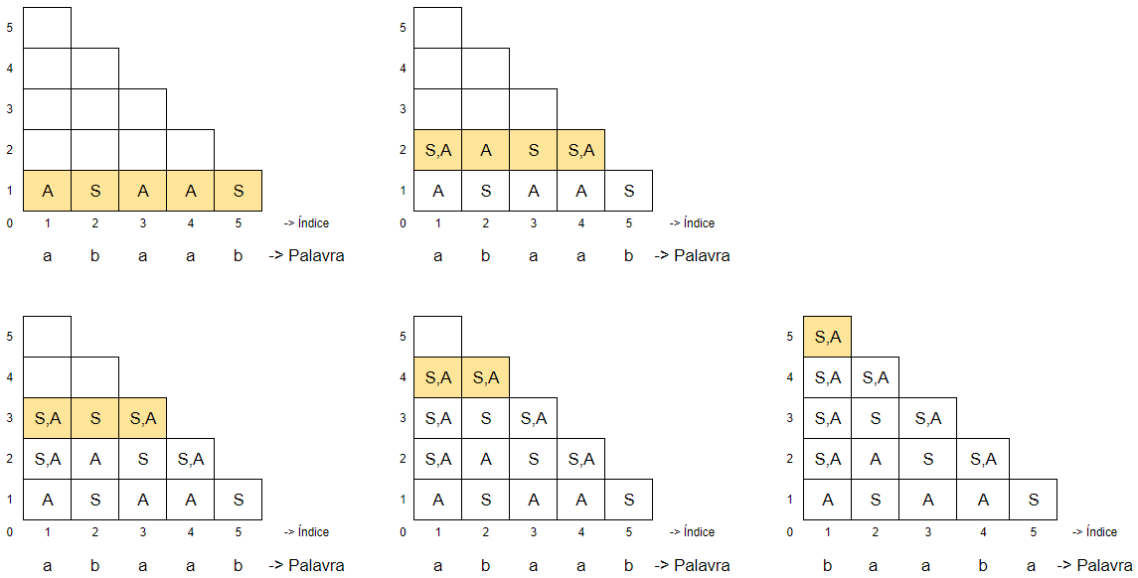

Figure 1. Exemplo de funcionamento do algoritmo CYK

Dada a gramática:
$S \rightarrow AA|AS|b$
$A \rightarrow SA|AS|a$

A partir da figura 1, vamos analisar a palavra: ”abaab”

Primeiro a palavra inicial é posicionada abaixo da base da matriz e suas respectivas regras geradoras preenchem a linha de baixo.

Em seguida, as regras são combinadas de dois em dois. Por exemplo, vamos ver a geração para baaba da seguinte maneira: ab, ba, aa, ab, o que implica em quatro regras geradoras. Como a "ab" é gerado por A e por S, temos S,A. Depois, "ba" é gerado apenas pela regra A. Observe que A tem o poder de gerar SA, S pode gerar "b" e A pode gerar "a", por isso "ba" é gerado por A. Dessa forma, até chegar ao topo da matriz as combinações vão sendo feitas. Na linha superior, as combinações são de três em três unidades da palavra, na superior de quatro em quatro, até que a palavra seja analisada por inteiro.

Note que, se a variável de partida da gramática, ao final do processo, estiver posicinada no topo da matriz, então a palavra será reconhecida por aquela gramática, caso contrário, a palavra de entrada não pode ser gerada por aquela gramática.

4. Segunda Forma Normal

A segunda forma normal, também conhecida como 2NF, representa gramáticas livres de contexto, bem como as gramáticas livres de contexto podem ser transformadas para uma 2NF equivalente. O que diferencia a CNF da 2NF é que a segunda tem menos restrições que a primeira. A segunda forma normal aceita varáveis concatenadas com terminais, por exemplo, o que não seria aceito pela normalização de Chomsky.

Nesssa forma as regras podem gerar, além da forma normal de Chomsky, varáveis sozinhas e terminais concatenados com variáveis. Veja um exemplo:
A gramática livre de contexto:
$S => XB|A|abXAB$
$X => AS|B$
$A => a$
$B => b$
Versão 2NF equivalente:
$S \rightarrow XB|A|aC$
$X \rightarrow AS|B$
$A \rightarrow a$
$B \rightarrow b$
$C \rightarrow bD$
$D \rightarrow XE$
$E \rightarrow AB$

Para transformar uma gramática LC na forma 2NF apenas a etapa de limitar o tamanho das regras à dois e criar novas variáveis para simplificar regras maiores são utilizadas. Sua transformação não parece significativamente não distante da forma CNF, mas o que a torna especial é que seu custo computacional. Pois, ao elimitar as outras etapas, que são etapas mais caras da transformação para CNF, o algoritmo fica cosideravelmente menos caro. Infortunamente, o algoritmo que transforma uma GLC para 2NF implementados no código fonte deste trabalho, tem sim custo computacional maior e performance um pouco inferior à transformação original de GLC para CNF, o que contraria a proposta

de Lang e Leiss. Porém, apesar de não reproduzir fidedignamente o que foi proposto pelos autores, as etapas de transformação e o que torna o algoritmo de Lang e Leiss especial foi entendido e o objetivo deste documento é também fazê-lo ser entendido.

5. CYK Modificado

Inspirado no CYK original, esse algoritmo modificado tem a intenção, não de fazer um processamento diferente do CYK original para validar se ela pode ou não ser gerada por uma dada gramática livre de contexto, mas sim de torná-lo apto a analisar uma palavra a partir de uma gramática no formato 2NF no lugar da forma normal de Chomsky. A segunda forma normal, por ser menos restritiva, é menos cara computacionalente de ser gerada a partir de uma gramática LC qualquer e menos cara de ser processada. A diferença principal entre o CYK original e o modificado está nas funções de criação de um grafo de unidade inversa e nullable que são responsáveis respectivamente por conseguir o grafo inverso ao grafo da gramática, que depois é usado para saber qual variáveis resultam em quais símbolos finais e por modificar para vazio todas as variáveis que resultam em vazio.

6. Análise de complexidade e tempo

O algoritmo CYK original tem complexidade $O(n^3)$, sendo n o tamanho da gramática. No entanto, o algoritmo que transforma CFG para CNF podem ter complexidade exponencial. A melhoria proposta por Lang e Leiss leva em consideração que a transformação de CFG para 2NF tem aumento linear no tamanho da gramática. Já o algoritmo CYK-modificado, a princípio, deveria ter a mesma performance que o original. A transformação da gramática LC para 2NF implementada nesse trabalho teve performance equivalente à transformação CFG para CNF. No entanto o algoritmo CYK-modificado tem complexidade $O(n^5)$

Para a seguinte gramática:
$S => XB|A|abXAB|AAABBB|AB|BA|P$
$X => AS|B|Q$
$A => a$
$B => b$
$P => p|XA|S$
$Q => BP|PB|PA|AP$

Os tempos de conversão são:
GLC para CNF: 0.0001553 s
GLC para 2NF: 0.0003283 s

Na figura 2 os tempos de processamento dos algoritmos cyk e cyk-modificado são apresentados.

Vamos analisar um exemplo com uma gramática maior: $S => XB|A|abXAB|AAABBB|AB|BA|f|afb|uio|j|JKL$
$X => AS|B|Q|SXABPQJKL|p|pP|J$
$A => a|P|K|abjklp$
$B => b|A|AB$
$P => p|XA|S$

Tamanho da palavra em caracteres	CYK	CYK modificado
64	0,02637	0,213995
320	1,6535	21,1965999
640	13,314855	368,9149173

Figure 2. Tempos de processamento CYK e CYK-modificado em segundos

$Q => BP|PB|PA|AP$
$J => j|jJ|JJ|JKL$
$K => kj|kk|kK|L$
$L => l|ll|lL|lM$
$P => S|A|B|SABK|p$

Os tempos de conversão são:
GLC para CNF: 0.000126 s
GLC para 2NF: 0.000363 s

Tempos de processamento:

Tamanho da palavra em caracteres	CYK	CYK modificado
64	0,01821	1,68903
320	1,47824	199,668133
640	12,30896	1.745,28475

Figure 3. Tempos de processamento CYK e CYK-modificado em segundos

Agora uma gramática menor:
$S => S + T|T$
$T => T * F|F$
$F => (S)|t$
Os tempos de conversão são:
GLC para CNF: 0.0003199 s
GLC para 2NF: 0.0002427 s

Tempos de processamento:

Tamanho da palavra em caracteres	CYK	CYK modificado
64	0,01395	0,038659
320	1,120697	4,076915
640	9,24351	33,4379658

Figure 4. Tempos de processamento CYK e CYK-modificado em segundos

7. Conclusão

O algoritmo CYK-modificado e a conversão de gramática LC para a forma 2NF não é mais difícil de ser aprendida que o original, aliás, a conversão de LC para 2NF é apenas uma das etapas da conversão de GLC para CNF. O que a torna mais fácil de ser aprendida e implementada.

Apesar da proposta de Lang e Leiss, inicialmente, era ter um algoritmo CYK-modificado com a mesma performance da original, o algoritmo implementado neste trabalho teve tempo maior na maioria dos casos. Isso ocorreu devido as modificações implementadas que utilizam métodos menos eficientes.

8. Referências

TO CNF or not to CNF? An Efficient Yet Presentable Version of the CYK Algorithm. Informatica didactica, File:///C:/Users/Rosana/Downloads/LangeLeiss2009.pdf, p. 1-21, 29 jun. 2009.

CFGSOLVER. [S. l.], 4 nov. 2013. Disponível em: https://github.com/nikoladimitroff/CfgSolver. Acesso em: 15 nov. 2022.

CYK Algorithm. [S. l.], 22 fev. 2020. Disponível em: https://github.com/mynttt/CYK-algorithm. Acesso em: 26 nov. 2022.

CYK-MODIFICADO. [S. l.], 29 set. 2021. Disponível em: https://github.com/Kymberlly/CYK-Modificado. Acesso em: 30 nov. 2022.