TRABALHO PRÁTICO N.01
Pontifícia Universidade Católica de Minas Gerais
11/2023

Fundamentos Teóricos da Computação

Marcus Navarro Gabrich / Matrícula: 700185
Rafael Murta Lopes / Matrícula: 690213

1 Introdução

Este trabalho centra-se na importância das transformações de gramáticas e na implementação de algoritmos para converter gramáticas em Forma Normal de Chomsky (FNC) e Segunda Forma Normal (2NF). As transformações gramaticais são fundamentais no contexto de linguagens formais e gramáticas, desempenhando um papel crucial na análise e manipulação de estruturas gramaticais. A capacidade de representar as regras de uma gramática em formas normais específicas simplifica significativamente o processo de compreensão e processamento, proporcionando vantagens na resolução de diversos problemas associados às linguagens formais.
A Forma Normal de Chomsky é uma representação especial de uma gramática livre de contexto em que todas as produções têm um formato específico, facilitando a aplicação de algoritmos de análise sintática e simplificando a interpretação da linguagem definida pela gramática. A Segunda Forma Normal, por sua vez, é uma adaptação para gramáticas em que as produções são limitadas a ter no máximo dois não-terminais do lado direito, contribuindo para uma estrutura mais organizada e eficiente em determinadas aplicações.
Ao abordar a implementação de algoritmos específicos para realizar essas transformações, este trabalho busca fornecer uma contribuição prática para a compreensão e aplicação dessas técnicas. Os algoritmos CYK (Cocke-Younger-Kasami) e uma versão modificada denominada `ModifiedCyk` são apresentados para verificar se uma gramática gera uma determinada entrada, sendo essenciais na validação e análise de expressões em linguagens formais. As classes `ChomskyNormalForm` e `SecondNormalForm` detalham os processos de conversão de gramáticas para FNC e 2NF, respectivamente, destacando a importância dessas formas normais na simplificação e otimização de estruturas gramaticais.
Assim, ao compreender a relevância das transformações de gramáticas e explorar a implementação de algoritmos específicos, este trabalho visa contribuir para a compreensão prática e teórica das linguagens formais, oferecendo uma base sólida para a aplicação dessas técnicas em diversas áreas, como compiladores, processamento de linguagem natural e análise sintática.

2 Implementações de Código

Descrevemos abaixo as implementações de código desenvolvidas para realizar transformações em gramáticas.

2.1 Leitura de Regras Gramaticais

A leitura das regras gramaticais é a primeira etapa do processo a ser feito após criar uma gramática. Em nossa implementação os terminais, variáveis e regras de produção são definidos pela função `readGramatica` presente na classe `Gramatica`.

2.1.1 Função `loadModel`

Essa é a única função chamada pela `readGramatica` recebendo o caminho para o arquivo de texto contendo os terminais, estados e regras de produção da gramática de uma forma amigável ao usuário e retorna os terminais, variáveis e regras de produção já tratados de forma a facilitar o seu uso no programa. Para isso o arquivo de texto é aberto e é realizado uma pré limpeza da entrada, com uma variável K contendo os terminais, uma variável V contendo as variáveis e uma variável P contendo as regras de produção. A função `cleanAlphabet` é chamada para as variáveis K e V e `cleanProductions` chamada para a variável P.

2.1.2 Função `cleanProductions`

Essa função recebe uma string contendo as regras de produção e retorna uma lista de tuplas representando as produções.

2.1.3 Função `cleanAlphabet`

Essa função recebe uma string contendo a uma sequência de terminais ou estados e retorna uma lista de símbolos.

2.1.4 Funcionalidade e Importância

A funcionalidade das funções de leitura de regras gramaticais é fundamental para o fluxo de trabalho subsequente. Ela simplifica e automatiza a leitura de regras gramaticais a partir de um arquivo externo, garantindo que essas regras estejam prontas para serem processadas pelos algoritmos de transformação gramatical. A correta execução desta etapa inicial é crucial para assegurar a qualidade e consistência das análises e transformações gramaticais realizadas ao longo do projeto.

2.2 Implementação de Formas Normais de Chomsky e Segunda Forma Normal

Essas formas normais são essenciais para simplificar e otimizar as regras gramaticais, facilitando análises sintáticas e processamentos subsequentes. Nesta seção, examinaremos detalhadamente a implementação dessa funcionalidade crucial.

2.2.1 `ChomskyNormalForm`

A função `cfgToCnf` é responsável pela conversão de gramáticas para a Forma Normal de Chomsky. Ela funciona do seguinte modo: Recebe uma gramática, logo em seguida chama a função `methods.defineVariables` que organiza as variáveis ainda não utilizadas para facilitar o uso. Depois chama as funções de conversão `methods.START`,

`methods.TERM` `methods.BIN`, `methods.DEL` e `methods.UNIT`. Por fim chama a função `methods.startingRuleFirst` que move a regra inicial para o início da lista de regras.

2.2.2 `SecondNormalForm`

A função `cfgTo2nf` é responsável pela conversão de gramáticas para a Segunda Forma Normal. Ela funciona de modo semelhante ao da função `cfgToCnf`, mas como diferença ao invés de utilizar as 5 funções de conversão, utiliza apenas a `methods.BIN` e não tem a necessidade de chamar a `methods.startingRuleFirst`.

2.2.3 Funcionalidade e Importância

A implementação dessas funções é crucial para o processo de normalização de gramáticas, proporcionando uma estrutura que simplifica e otimiza as regras, preparando-as para análises sintáticas e processamentos subsequentes. A Forma Normal de Chomsky e a Segunda Forma Normal são padrões fundamentais em linguagens formais, e a correta execução desses algoritmos é essencial para garantir a consistência e a eficácia das transformações realizadas nas gramáticas.

2.3 Algoritmos para Verificação de Gramáticas

O arquivo `algorithms.py` contém implementações dos algoritmos fundamentais para a verificação de gramáticas. Estes algoritmos desempenham um papel crucial no contexto de linguagens formais e gramáticas, oferecendo meios eficazes para determinar se uma dada gramática pode gerar uma determinada sequência de símbolos. Nesta seção, abordaremos detalhadamente as duas principais classes contidas neste arquivo: `Cyk` e `ModifiedCyk`.

2.3.1 `Cyk`

A classe `Cyk` implementa o algoritmo Cocke-Younger-Kasami (CYK), utilizado para verificar se uma gramática gera uma determinada sequência de símbolos. O método principal é o `run`.
O método utiliza uma tabela dinâmica para verificar se a gramática definida pelas regras de entrada pode gerar a sequência de símbolos fornecida. O algoritmo é eficiente e amplamente utilizado em análise sintática.

2.3.2 `ModifiedCyk`

A classe `ModifiedCyk` implementa uma versão modificada do algoritmo CYK. Seu método principal também é o `run`, mas antes de ser chamado é necessário chamar o `methods.anulavel` que retorna as variáveis anuláveis.
Esta versão modificada se baseia na versão original, mas utiliza de três funções extras, a `inverseUnitGraph` que recebe a gramática e as variáveis anuláveis e retorna o grafo de unidade inverso, a `discoverReach` que recebe o grafo e a lista de palavras e retorna a lista de símbolos alcançáveis e a `dfs` que recebe um grafo e o nó inicial e retorna a lista de nós alcançáveis.

2.3.3 Funcionalidade e Importância

Os algoritmos implementados em `algorithms.py` são essenciais para a verificação prática de gramáticas. Eles oferecem meios para determinar se uma gramática pode gerar uma determinada sequência de símbolos, sendo fundamentais em processos de análise sintática e transformações gramaticais. O algoritmo CYK, em particular, é amplamente utilizado e reconhecido na área de linguagens formais.

3 Experimentos e Resultados

Realizamos experimentos utilizando diversas gramáticas de tamanhos variados como entrada para avaliar a eficácia das transformações implementadas. Os resultados como se pode ver na Figura 1 demonstram o ganho de performance do cyk modificado frente o original, que gasta apenas 70% do tempo originalmente gasto.

```
-------------------------------------------------------------------------------------------
| media cyk:0.0162s, media modified_cyk:0.01142s |
-------------------------------------------------------------------------------------------
```

Figure 1: Média dos tempos dos algoritmos.

Os testes foram realizados para as seguintes gramaticas:

Gramática 1

Variáveis:

$$[' S',' A',' B',' C',' D',' E',' F',' G',' H',' I',' J']$$

Terminais:

$$[' a',' b']$$

Regras: S $\rightarrow A \mid B$
$A \rightarrow aE \mid \varepsilon$
$B \rightarrow aF \mid bI$
$C \rightarrow \varepsilon$
$D \rightarrow bJ$
$E \rightarrow aEE \mid aFG \mid bC$
$F \rightarrow aEF \mid aFH \mid bD$
$G \rightarrow b$
$H \rightarrow bD$
$I \rightarrow aEFG \mid b$
$J \rightarrow aFH \mid bD$

Resultados:

- Utilizando CYK: "aaabbb" pertence à gramática
  Utilizando CYK Modificado: "aaabbb" pertence à gramática
  Tempo de execução CYK: 0.00028
  Tempo de execução CYK Modificado: 0.0003

- Utilizando CYK: "aaaaaaaaaaaaaaabbbbbbbbbbbbbbbb" não pertence à gramática
  Utilizando CYK Modificado: "aaaaaaaaaaaaaaabbbbbbbbbbbbbbbb" não pertence à gramática
  Tempo de execução CYK: 0.01309
  Tempo de execução CYK Modificado: 0.01151

- Utilizando CYK: "ab" pertence à gramática
  Utilizando CYK Modificado: "ab" pertence à gramática
  Tempo de execução CYK: 0.00016
  Tempo de execução CYK Modificado: 0.00011

Gramática 2

Variáveis:

$$[' E']$$

Terminais:

$$[' t', ' +', ' *', ' (', ')']$$

Regras:

$$E \rightarrow E + E \mid E \mid E * E \mid E \mid (E) \mid t$$

Resultados:

- Utilizando CYK: "t+t+t+t+t+t+(t)" pertence à gramática
  Utilizando CYK Modificado: "t+t+t+t+t+t+(t)" pertence à gramática
  Tempo de execução CYK: 0.00168
  Tempo de execução CYK Modificado: 0.00126

- Utilizando CYK: "()" não pertence à gramática
  Utilizando CYK Modificado: "()" não pertence à gramática
  Tempo de execução CYK: 0.00029
  Tempo de execução CYK Modificado: 4e-05

- Utilizando CYK: "(t)+t" pertence à gramática
  Utilizando CYK Modificado: "(t)+t" pertence à gramática
  Tempo de execução CYK: 0.00087
  Tempo de execução CYK Modificado: 0.00012

Gramática 3

Variáveis:

$$[' P', ' Z']$$

Terminais:

$$[' 0', ' 1']$$

Regras: P $\rightarrow 0P0 \mid 1P1 \mid 0 \mid 1 \mid Z$
$Z \rightarrow \varepsilon$
Resultados:

- Utilizando CYK: "0000001111111" pertence à gramática
  Utilizando CYK Modificado: "0000001111111" não pertence à gramática
  Tempo de execução CYK: 0.00044
  Tempo de execução CYK Modificado: 0.00042

- Utilizando CYK: "1" pertence à gramática
  Utilizando CYK Modificado: "1" pertence à gramática
  Tempo de execução CYK: 5e-05
  Tempo de execução CYK Modificado: 3e-05

Gramática 4

Variáveis:

$$[' E', ' T', ' F']$$

Terminais:

$$[' t', ' +', ' *', ' (', ')']$$

Regras: E $\rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid t$
Resultados:

- Utilizando CYK: "t+t+t+t+t+t+(t)" pertence à gramática
  Utilizando CYK Modificado: "t+t+t+t+t+t+(t)" pertence à gramática
  Tempo de execução CYK: 0.0007
  Tempo de execução CYK Modificado: 0.00061

- Utilizando CYK: "()" não pertence à gramática
  Utilizando CYK Modificado: "()" não pertence à gramática
  Tempo de execução CYK: 9e-05
  Tempo de execução CYK Modificado: 3e-05

Resultados finais:

- Media de tempo de execução CYK: 0.01798
- Media de tempo de execução CYK Modificado 0.01207

4 Conclusão

Este trabalho apresentou implementações dos algoritmos CYK e CYK Modificado. Os experimentos realizados demonstram o ganho de performance proposto pelo CYK Modificado. No entanto, vale ressaltar que existem limitações e possíveis melhorias que podem ser investigadas em trabalhos futuros.