Identificação de sentenças em LLCs

Henrique Castro e Silva[1], **Leonardo Caetano Gomide**[1]

[1]Departamento de Ciência da Computação - PUC Minas

leocgomide@gmail.com, henrique.castros@outlook.com

1. Introdução

No vasto domínio da teoria da computação, as Gramáticas Livres de Contexto (GLCs) emergem como uma ferramenta fundamental para descrever a estrutura sintática de linguagens, abrindo caminho para uma compreensão mais profunda das complexidades inerentes à manipulação de linguagens formais. Enquanto as GLCs desempenham um papel central na modelagem de gramáticas de linguagens de programação, compiladores e processadores de linguagem natural, a exploração das derivações dentro desse contexto torna-se uma área de estudo igualmente crucial.

2. Algoritmo

O algoritmo CYK (Cocke–Younger–Kasami) nomeado em homenagem a seus 3 inventores independentes na década de 60 é um algoritmo que consegue descobrir se uma sentença *w* é derivada por uma GLC *G*. Ele utiliza da técnica de computação dinâmica para armazenar derivações parciais da sentença até que se obtenha todos os símbolos que podem derivar a sentença inteira, e se o símbolo inicial *s* estiver nesse conjunto, a sentença é derivável pela gramática. O algoritmo tem uma complexidade de $\mathbf{O(n^3 * |G|)}$ sobre o tamanho n da sentença.

O CYK original nescessita que a GLC esteja na Forma Normal de Chomsky (CNF), cuja operação de conversão a partir de uma gramática G qualquer pode ser exponencial e com um entendimento mais complexo que com uma forma menos restritiva. [Lange and Leiß 2009] apresentaram uma modificação do algoritmo que trabalha com uma forma normal binária, menos restritiva que a CNF, possuindo apenas uma restrição: as produções não podem produzir mais que dois símbolos (terminais ou não).

3. Implementação

A implementação de nosso trabalho foi feita em Python 3 por possuir a estrutura de dados conjunto (set) nativa, facilitando a implementação e melhorando a legibilidade do código. A classe principal do código é a *ContextFreeGrammar* representando uma CFG que é implementada tanto pela *ChomskyNormalForm* quanto pela *BinaryNormalForm*. Um script utilitário *LanguageGenerator.py* foi utilizado para gerar arquivos com várias sentenças programaticamente para facilitar que as Gramáticas sejam testadas.

Todas as gramáticas foram definidas manualmente, tomando como base gramáticas apresentadas em classes ou exemplos encontrados online.

3.1. Gramáticas

O arquivos das gramáticas define uma produção por linha, com a variável e sua produção sendo separada por $\therefore$. A variável contida na primeira linha do arquivo é considerada a

variável de partida, os símbolos que aparecem pelo menos uma vez do lado esquerdo de uma produção são considerados não terminais e os demais são considerados terminais, um exemplo de gramática é a seguinte:

$$P : 0P0$$
$$P : 1P1$$
$$P : 0$$
$$P : 1$$
$$P :$$

Essa gramática representa a gramática escrita convencionalmente como:

$$G : P \rightarrow 0P0 \mid 1P1 \mid 0 \mid 1 \mid \lambda$$

3.2. Linguagens

Os arquivos de linguagens são arquivos que possuem sentenças produzidas por uma Linguagem qualquer. Cada linha no arquivo representa uma sentença, e esses arquivos serão utilizados posteriormente para checar as linguagens das gramáticas.

Os arquivos das linguagens foram gerados pelo script *LanguageGenerator.py*, que possui o método *generateLanguage*, que gera o fecho de até tamanho n de um alfabeto passados por parâmetro e checa se cada sentença será adicionada à linguagem com uma função que recebe a sentença e retorna se ela pertence ou não a linguagem.

3.3. ContextFreeGrammar

A classe *ContextFreeGrammar* é uma representação genérica de GLCs, possuindo 4 atributos iguais à quartupla (V, Σ, P, S) similar à utilizada em [Vieira 2006]. A classe também possui um método *from_file* que instancia um novo objeto a partir de um arquivo.

A classe também possui alguns métodos auxiliares que serão utilizados para facilitar a geração das formas normais, como remoção de produções lambda, remoção de produções unitárias, adicionar novo início e remoção de produções longas.

3.4. ChomskyNormalForm

A classe *ChomskyNormalForm* é uma classe que implementa *ContextFreeGrammar*, com os mesmos 4 atributos (V, Σ, P, S) e é uma representação de uma gramática que tem suas produções de acordo com as regras de formas normais de Chomsky:

$$A \rightarrow BC \mid (A, B, C) \in V$$
$$A \rightarrow a \mid A \in V, a \in \Sigma$$
$$S \rightarrow \lambda$$

Essa classe também possui um método que dada uma GLC qualquer realiza os passos necessários para transformar essa GLC na Forma Normal de Chomsky:

0. Opcionalmente, remove produções inúteis (duplicadas e/ou que não terminam)
1. Adicionar uma nova variável de partida, caso a variável de partida original apareça do lado direito de alguma produção
2. Remove produções lambda
3. Remove produções unitárias
4. Converte produções restantes

Tendo a gramática representada na FNC, podemos implementar o algorítmo CYK original para validar se uma sentença pode ser produzida por uma gramática, implementado no método com o nome *is_in_language*, e utilizando esse método, foi implementado um método que dada uma gramática na FNC, e um arquivo de linguagem, avaliasse todas as sentenças do arquivo para validar se a gramática reconhece ou não todas as sentenças dessa linguagem.

3.5. BinaryNormalForm

A classe *BinaryNormalForm* é uma classe que implementa *ContextFreeGrammar*, com os mesmos 4 atributos (V, Σ, P, S) e é uma representação de uma gramática que tem suas produções de acordo com as regras de formas normais binárias (2NF):

$$A \to \alpha \mid A \in V, |\alpha| \leq 2$$

A forma normal binária é mais relaxada que a forma normal de Chomsky, tendo somente um passo obrigatório - Reduzir o tamanho das produções. Em nossa implementação, foram também implementados três passos opcionais que podem ser ativados por parâmetros quando chamando a função de conversão: remover produções inúteis, introduzir novo início e remover produções lambda.

Por ser mais relaxada, a 2NF precisa de construir de dados adicional para conseguir verificar se uma sentença pode ser produzida pela gramática. O passo a passo para construir o grafo unitário inverso é descrito em [Lange and Leiß 2009]. O único ponto a se notar em nossa implementação é que ao calcular o grafo unitário inverso, é retornado além de $(V, \breve{\mathcal{U}}_G)$, também retornamos uma função *Ugstar*, que dado um conjunto de símbolos w, retorna todos os símbolos alcançáveis a partir desse conjunto, para facilitar o uso quando formos realizar o CYK modificado para 2NF.

Tendo a 2NF e o grafo $(V, \breve{\mathcal{U}}_G)$, é possível implementar o CYK modificado introduzido em [Lange and Leiß 2009], e da mesam forma que foi feito para *ChomskyNormalForm*, foram implementados os métodos de checar uma sentença (*is_in_language*) e a linguagem inteira (*check_language*).

4. Experimentos

Como experimentos, foram definidas 12 gramáticas com alfabetos e linguagens reconhecidas distintas e foram gerados 8 arquivos de linguagem com o script de geração de linguagens. Para cada gramática tentamos reconhecer todas as sentenças de cada uma das linguagens e mostramos em tela se cada linguagem é reconhecida ou não.

Na pasta *test* são mostrados mais detalhes sobre o reconhecimento de cada linguagem por cada gramática, mostrando o resultado para cada sentença da linguagem e quantas sentenças foram aceitas e rejeitadas.

SO	Processador	Memória RAM	Tempo CNF	Tempo BNF
Windows 10	Intel i5-7200 2	20GB	14.694s	5.646s
Windows 10	Intel i5-8265 2	8GB	14.406s	5.958s
MacOS	**M2**	**16GB**	**6.040s**	**2.128s**

Tabela 1. Tempo de execução em diferentes computadores.

4.1. Resultados

Para testar nossa solução testamos o programa em três computadores para termos comparação do desempenho. Em cada computador o programa foi executado 5 vezes para podermos ter um limiar do tempo médio.

Conforme constado na 4.1, o processador do *M2* o desempenho foi bem superior aos processadores *Intel*, enquanto ter mais ou menos memória RAM não aparentou ter grandes ganhos no desempenho.

Outro ponto que é interessante notar que o CYK para 2NF executou mais rápido, apesar de ter que trabalhar com a estrutura adicional.

5. Conclusão

Com os dados da sessão anterior podemos ver o ganho ao utilizar o método executado nas gramáticas no formato BNF, demonstrando que, apesar do maior gasto espacial, o método executa mais rápido que o que utiliza gramáticas CNF.

Outro ponto interessante que podemos notar na sessão anterior é a importância do processador para o tempo de execução do programa. Pode-se notar que com a utilização do processador *M2*, foi obtido o melhor resultado em ambas tarefas. Seria preciso mais testes e mais computadores, porém pode-se formular a hipótese que o principal gargalo do programa é o processamento e não a complexidade espacial, abrindo portas para trabalhos futuros que abordem paralelismo e outras otimizações.

Referências

Lange, M. and Leiß, H. (2009). To cnf or not to cnf? an efficient yet presentable version of the cyk algorithm. *Informatica Didactica*, 8.

Vieira, N. (2006). *Introdução aos fundamentos da computação: Linguagens e máquinas*. Pioneira Thomson Learning.