Aplicação para determinização e minimização de autômatos finitos

Edson Lemes da Silva[1]

[1]Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)
Caixa Postal 181 – 89.802-112 – Chapecó – SC – Brasil

edson_lemes@live.com

Abstract. *This paper describes some theoretical aspects for geration, determinization and minimization of finite Automata. Conversely, it presents an application for the referred theorems, constructed from regular grammar and tokens. In this way, the development details and their particularities are presented.*

Resumo. *Este trabalho descreve alguns aspectos teóricos para geração, determinização e minimização de autômatos finitos. Por outro lado, apresenta uma aplicação para os teoremas referidos, construídos a partir de gramáticas regulares e tokens. Desta forma, são apresentados os detalhes de desenvolvimento e suas particularidades.*

1. Introdução

O processo de determinização permite transformar um autômato finito não-determinístico em um determinístico, esta tarefa é importante para eliminar ambiguidades entre as transições, e também facilita a verificação dos estados durante o processo de derivação. Em paralelo a isto, a minimização permite eliminar estados não produtivos, ou seja, aqueles que não interferem nas produções de acordo com a gramática da linguagem.

A aplicação descrita aqui atende a leitura de gramáticas regulares e tokens, o processo de determinização e a minimização do autômato finito. A linguagem de programação usada é o Python, juntamente com algumas classes que já estão implementadas.

2. Autômatos finitos

Os autômatos finitos (AF) podem ser construídos sobre uma gramática regular (GR) ou uma sequência de símbolos que representam tokens. De acordo com [Menezes 2000, p. 69], um autômato finito é descrito por um sistema de estados finitos, com aplicações em modelos de estudos, tais como compiladores, linguagens formais e semântica formal.

Analisando de forma detalhada, os AFs podem apresentar duas características estruturais. Deste modo, um autômato finito pode ser determinístico (AFN) ou não-determinístico (AFND). A diferença recorrente entre eles, descreve a forma como ocorrem as transições de um estado para outro conforme o símbolo de entrada. Desta maneira, em um AFN há exatamente um estado possível em cada transição, enquanto no AFND podem haver múltiplos estados conforme os símbolos da linguagem.

Um autômato finito não-determinístico pode ser transformado em um determinístico, já que existe uma relação de equivalência entre ambos. Este processo é chamado de determinização de autômatos finitos. Esta rotina permite tornar os estados

bem definidos e sequenciais, facilitando principalmente a implementação de analisadores léxicos. A determinização ocorre analisando e criando novos estados para eliminar os indeterminismos, uma vez que estas situações dificultam o tratamento e verificação das possíveis palavras que pertencem à linguagem descrita pela gramática.

O outro processo a descrito é a minimização de autômatos finitos determinísticos. De acordo com o [Menezes 2000, p. 122], este processo sobre um AFD não influencia no tempo de processamento do autômato em si, mas sim no números de estados, ou seja, na complexidade de espaço. Em outras palavras, este processo busca gerar um AFD mínimo e equivalente. O algoritmo de minimização é dividido em três partes: eliminação de estados inalcançáveis, mortos e aplicação de classes de equivalência.

3. A aplicação

Este projeto visa aplicar os teoremas de determinização e minimização. A primeira parte refere-se a leitura de tokens ou gramática regular. Estes dados são carregados a partir de um arquivo de texto. Para a leitura de gramática regular, ela deve estar definida com a notação de BNF (Backus-Naur Form), os itens que não estão neste padrão são detectados como tokens. Uma vez que há tokens e uma gramática compostos no mesmo AF, então apenas o estado inicial é compartilhado, os demais são exclusivos para reconhecimento de tokens ou palavras da gramática. Portanto, este autômato gerado terá grandes possibilidades de ser não-determinístico.

Após a construção do AFND, é feito a aplicação do teorema de determinização, e assim é obtido um AFD equivalente. Posteriormente, sob o AFD gerado, é feito a minimização para eliminar possíveis estados inúteis, ou seja, inalcançáveis ou mortos. Contudo, não é feito as classes de equivalência, pois isto pode tornar o autômato capaz de reconhecer situações que não são representados tantos pelos tokens, quanto pela GR.

A última parte da aplicação, preenche os estados não mapeados com um de erro, este é incluído após a minimização para não aumentar o tamanho do AFD de forma desnecessária.

4. A implementação

Inicialmente, é preciso descrever o ambiente de programação. A linguagem utilizada para a aplicação, foi o Python; utilizando-se de alguns módulos (classes) já implementados para esta linguagem.

Como forma de compatibilidade, esta aplicação foi desenvolvida sob a versão 2.7.12 do python. Os módulos utilizados e importados para o código são: `collections`, `copy`, `itertools e terminalTables` (pode ser instalado via pip em um terminal Linux, utilizando o comando "`pip install terminaltables`").

Internamente, existem 6 funções principais. As demais são consideradas secundárias, pois são processos chamados dentro destas funções. Assim, estarão descrita aqui as funções em ordem de execução.

Na implementação, estas funções estão dentro de uma classe, chamada de `determinizeAfnd`. Uma instância desta classe, recebe um parâmetro do tipo

`String`, que representa o nome e o caminho do arquivo de leitura (onde estão descritos os tokens e a gramática da linguagem).

A estrutura para armazenar o AFND lido é do tipo `defaultdict`, um dicionário baseado em chave e valor; chamada de `auto` na aplicação. Neste caso, cada estado é representada com um número inteiro (usado como chave), enquanto os valores são listas de inteiros (estados), cada uma dela representa as transições para um dado estado conforme os símbolos do alfabeto da linguagem. A identificação de indeterminismos acontece através de aninhamento de listas, ou seja, cada posição de uma lista, pode conter um vetor que representa os estados de transição para um símbolo específico.

Como forma de implementação, os estados são armazenados como números inteiros, a partir do número zero (sempre representa a regra inicial). Para a visualização dos estados, foi criado um mapeamento de cada um deles para uma forma representativa, estes dados são armazenados na variável `mapGrammar`, do tipo `dict`.

Os símbolos que fazem parte da GR (que dão nomes às regras), são mapeados conforme a sua forma escrita do arquivo de leitura, com exceção das produções que possuem estados terminais (apenas com símbolos do alfabeto da linguagem), neste caso nomeia-se para "TE". Os estados que fazem parte de tokens, são chamados de T_i, onde i representa um número inteiro que identifica o estado. Assim, para diferenciar as transições que são finais, concatena-se um valor "f" a eles.

A primeira função (método), é chamada de `fileReader`, ela é a responsável por fazer a leitura do arquivo de entrada. Assim, é feito o processamento para identificação dos itens que fazem parte do texto. A função lê linha por linha (até encontrar um separador), então identifica cada uma de acordo com as suas características, isto é, se existem caracteres que definem uma regra de uma GR (dentro da forma BNF), caso contrário são considerados como tokens. Desta maneira, são chamadas as funções `grammarReader` ou `tokenReader`, respectivamente.

A `grammarReader` é responsável por analisar a regra recebido, uma vez que os parâmetros lidos anteriormente fazem parte de uma GR. A verificação de uma regra de uma gramática dividi-se em duas partes, a primeira identifica o símbolo que da nome à regra. Caso este é um símbolo que ainda foi adicionado na estrutura de estados, então acrescenta-se no vetor.

Posteriormente é verificado as produções desta regra, novamente verificando se os símbolos fazem parte do conjunto de estados, caso algum deles não pertencem a estrutura, então adiciona-se respeitando as transições conforme os símbolos do alfabeto da linguagem.

Em contra-partida, a função `tokenReader` cria um novo estado para cada símbolo deste token, com exceção do inicial (caso ele exista). Assim, acrescenta-se na estrutura `auto` os estados e suas transições conforme os símbolos que fazem parte do alfabeto da linguagem.

Após a leitura do AFND, o próximo passo é executar a função de determinização do autômato finito, chamada de `determinize`, ela verifica para cada chave da estrutura `auto`, todos os valores, ou seja, todas as listas correspondentes. Essa análise procura por posições onde o tamanho é maior que um, isto é, há mais um estado de transição para

um mesmo símbolo. Desta maneira, cria-se um estado novo que substitui aqueles dos quais fazem parte deste indeterminismo, criando um apontamento para cada estado da composição deste novo estado. Estes "ponteiros" são armazenados na forma de chaves na estrutura `mapGrammar`.

A próxima etapa da determinização verifica os novos estados criados, mapeando as suas transições. Caso haja novos indeterminismos, a função cria os novos estados até que todo o autômato esteja determinizado.

A partir do ponto que a aplicação determinizou o autômato, é a chamado a função `minimize` que permite minimizar o AFD, ou seja, eliminar estados inalcançáveis e mortos. Essa eliminação ocorre através de uma busca em profundidade (DFS) na estrutura `auto`. Primeiramente, é lançado uma busca dfs a partir do estado inicial; obtendo todos aqueles que são visitados a partir do início. Aqueles que não são alcançáveis, a aplicação elimina. A outra parte da minimização procura estados mortos, novamente é usado uma busca dfs, mas desta vez para cada um é lançado uma busca, neste caso verifica-se entre os visitados se algum deles é estado final, caso contrário o mesmo é removido.

A última parte da aplicação preenche os estados não mapeados nas transições através da função `fill`, este processo é feito após a minimização e a determinização para evitar a criação excessiva de estados. Como forma de erro, um estado "X" é acrescido no fim do AFD.

4.1. Execução do algoritmo

A execução do algoritmo é feita a partir do interpretador Python. Primeiramente, deve-se importar o arquivo que contém a classe implementada, que esta no arquivo "detAfnd.py". O próximo passo é criar uma instância para esta classe, passando como parâmetro o nome do arquivo de leitura.

Assim, após a instanciação, deve-se chamar o método (função) `determinize`, ele se encarrega de chamar as demais funções do algoritmo, isto é, de minimização e preenchimento dos estados não mapeados.

Alternativamente, está implementado uma função chamada de `printTable`, ela imprime o autômato finito em forma de tabela; com o intuito de permitir uma melhor visualização dos dados. Este método pode ser chamado a qualquer momento, ou seja, pode chamá-lo para imprimir o autômato finito antes do processo de determinização, da mesma maneira após este processo ser finalizado.

Outro aspecto de visualização do processo, é referente à determinização do autômato finito. Enquanto os estados são analisados pela função `determinize`, no console da aplicação são imprimidos informações sobre o andamento do processo. Neste caso, são mostrados os estados onde há indeterminismos, e quais estados e símbolos estão envolvidos. Quando verifica-se os novos estados criados, a aplicação mostra na tela onde há um novo indeterminismo, ou se não for o caso, quais símbolos tiveram as suas transições já determinizadas.

Para a saída do programa (autômato finito determinístico) ser visualizada de forma clara, é possível exportar a tabela gerada para um arquivo externo. Esta saída é armazenada em arquivo de texto, nesta aplicação nomeado de "automato.txt". Contudo, apenas o AFD minimizado é exportado, os demais são apenas imprimidos na tela durante

a execução do algoritmo. Como exemplo, esta disponível um arquivo chamado de "test.py", ele cria uma instância e executa-o.

```
from detAfnd import *

test = determinizeAfnd('test.in')
test.printTable()
test.determinize()
test.printTable() #Gera o arquivo 'automato.txt' nessa chamada
```

Exemplo 1. Execução do algoritmo.

5. Conclusão

No trabalho foi demonstrado as operações que podem ser feitas sobre um autômato finito, tais como o teorema de determinização e seus objetivos; e da mesma maneira também o teorema de minimização, com a exceção do uso de classes de equivalência.

A aplicação desenvolvida permite gerar, determinizar e minimizar um autômato finito construído a partir de uma gramática regular, juntamente com a composição de tokens lidos a partir de um arquivo de texto. Toda a implementação foi desenvolvida na linguagem Python.

A documentação sobre a implementação descreveu as funções principais de forma detalhada, assim como no código-fonte, onde estão descritas de forma resumida as funcionalidades de cada uma delas.

Referências

Menezes, P. F. B. (2000). *Linguagens formais e autômatos*. Editora Sagra Luzzatto, 3rd edition.