Relatório de Implementação do projeto final da disciplina de Linguagens Formais e Autômatos

Matheus Dias Negrão[1], Vinicius Dos Reis de Jesus[1]

[1]Curso de Ciência da Computação – Universidade Federal da Fronteira Sul (UFFS)

Chapecó – SC – Brazil

matheus.negrao@uffs.estudante.edu.br, vinicius.reis@uffs.estudante.edu.br

Abstract. *This paper describes and explains the planning and execution of the final project implementation of the discipline of Formal Languages and Automata. In the same are described and developed the concepts, and the theoretical reference, to understand the construction of an application, with the objective of: to generate a finite non-deterministic automaton, to eliminate epsilon existing transitions, to determine it and to minimize the resulting automaton without the use of equivalence classes. The application was developed using the Python programming language in version 3. The objective of the project developed is the validation of the knowledge acquired during the semester.*

Keywords. deterministic automata, minimizate

Resumo. *Este artigo descreve e explica o planejamento e a execução da implementação do projeto final da disciplina de Linguagens Formais e Autômatos. No mesmo estão descritos e desenvolvidos os conceitos, e o referencial teórico, para compreensão da construção de uma aplicação, com o objetivo de: gerar um autômato finito não determinístico, eliminar épsilon transições existentes, determinizá-lo e minimizar o autômato resultante, sem o uso de classes de equivalência. A aplicação foi desenvolvida utilizando a linguagem de programação Python na versão 3. O objetivo do projeto desenvolvido é a validação dos conhecimentos adquiridos durante o semestre letivo.*

Palavras-Chave. autômato determinístico, minimização

1. Introdução

Entende-se como linguagem formal o estudo de modelos matemáticos para a especificação e reconhecimento de linguagens, podendo assim classificá-las, e diferenciar suas estruturas e suas propriedades. As aplicações da área dentro da computação são importantes para conceitos teóricos, como por exemplo: computabilidade, decidibilidade e complexidade computacional. Os reconhecedores são dispositivos que servem para verificar se uma frase pertence a uma certa linguagem. Os geradores são dispositivos que permitem a geração sistemática de todas as frases de uma linguagem. Exemplos de reconhecedores: autômatos finitos, autômatos de pilha e máquina de Turing. Para classificar os geradores existe o que é chamado de Hierarquia de Chomsky, que classifica as gramáticas em quatro níveis: gramáticas regulares, gramáticas livres de contexto, gramáticas sensíveis ao contexto, e gramáticas irrestritas.

A proposta deste relatório é a explicação do processo de implementação de um gerador de autômato finito determinístico mínimo, livre de épsilon transição, porém sem a utilização de classes de equivalência.

Neste artigo será apresentado o referencial teórico necessário para entendimento completo da execução e resultados da aplicação, bem como todo o processo de desenvolvimento da mesma.

2. Referencial Teórico

Para melhor compreensão do projeto é necessário conhecer os seguintes conceitos:

- Símbolo: representações indivisíveis das cadeias;
- Alfabeto: conjunto finito de símbolos (Menezes, 2008);
- Palavra: sequência finita de símbolos do alfabeto (Menezes, 2008);
- Cadeia: palavras formadas por símbolos do alfabeto;
- Linguagem: conjunto de palavras sobre um alfabeto (Menezes, 2008);
- Sentença: uma cadeia da linguagem.

Para desenvolver o projeto foi utilizado a gramática regular, e dentro dela um Autômato Finito é um sistema de estados finitos.

Uma expressão regular é um conjunto de operações de concatenação. Além disso, é a classe de linguagens mais simples. Os Autômatos Finitos são a classe de algoritmos mais eficiente em relação ao tempo de processamento.

Segundo Menezes, as condições para geração das palavras da linguagem são definidas pelas regras de produção e "a aplicação de uma regra de produção é denominada derivação de uma palavra" e que aplicando-se as regras de produção sucessivamente geram-se as palavras da linguagem que é representada pela gramática. A linguagem gerada pela gramática é "composta por todas as palavras de símbolos terminais deriváveis a partir do símbolo inicial S" (Menezes 2008).

Um Autômato Finito possui um número finito e predefinido de estados, constitui de um modelo computacional sequencial, e o próximo passo somente é executado após o fim do passo anterior. Segundo Menezes, um Autômato Finito, além disso, é uma quíntupla composta por um alfabeto, um conjunto dos possíveis estados, função de transição, um estado inicial e um conjunto de estados finais que pertence ao conjunto de estados do autômato.

O Autômato será não-determinístico (AFND) quando um estado possui, para um mesmo símbolo, mais de uma transição de estado. Menezes resume um AFND como: "para o estado corrente e o símbolo lido da entrada, determina aleatoriamente um estado de um conjunto de estados alternativos". Ou seja, a cada transição é gerada uma árvore de opções, sendo que se algum dos caminhos aceite a entrada, então ela é aceita.

O Autômato será determinístico (AFD) se em cada estado, para um símbolo, há apenas uma transição de estado. Ou seja, para uma certa cadeia, em um AFD existirá apenas uma sequência de estados que aceitará a cadeia, enquanto em um AFND pode existir mais de uma sequência diferente de estados para aceitá-la.

A determinização de um AFND é o processo de conversão de um AFND para AFD, segundo Ricarte tem como base "criar novos estados que representam todas as possibilidades de estados originais em um dado momento da análise da sentença em processo de de reconhecimento". Para garantir a equivalência dos dois autômatos, deve-se copiar, para cada novo estado do AFD, todas as transições que partem de estados que seriam atingidos pelas transições do AFND original. Durante a execução dessa conversão, pode surgir novos indeterminismos, portanto o mesmo deve ser feito de forma iterativa até não serem mais encontrados novos indeterminismos. A eliminação das epsilon transições, necessária para a

determinização do autômato, que se dá ao incluir nas regras que as possuem epsilon transição as produções alcançadas por esta.

Muitas vezes, nem todos os estados de um autômato são úteis ou alcançáveis, então outro procedimento adotado para a otimização do mesmo é a eliminação dos estados inúteis e inalcançáveis. Os estados inalcançáveis são aqueles que não possuem caminho até ele, com transições válidas a partir do estado inicial. Uma forma de eliminá-los é marcando os estados que foram alcançados a partir do estado inicial. Os estados inúteis são alcançados a partir do estado inicial, porém não conseguem atingir os estados finais da gramática. E com isso, esses estados não contribuem para a aceitação das sentenças, e a sua remoção não causará prejuízos. Uma varredura no autômato, selecionando os estados inúteis, já é um método simples de eliminá-los.

3. Desenvolvimento

Para a implementação do trabalho foi utilizado a linguagem Python na versão 3. A escolha da linguagem se deu principalmente por sua legibilidade de código e facilidade no tratamento de cadeias de caracteres.

O código está estruturado em funções, das quais cada uma executa um dos processos exigidos pelo projeto. As funções utilizadas são:

```
def insList(lista, vlr):
def decodificaEntrada():
def criaAutomatoGramatica():
def criaProducoesGramatica(producoes):
def criaProducoesToken():
def ultimoEstado(terminal):
def determiniza():
```

```
def determinizaLinha(estado):
def minimiza():
def eliMortos(estado):
def eliProdMortos(estado, qtd):
def eliInalc(estado):
def eliProdInal(estado, qtd):
def printAFD(tipo, eMorto, aut, listNTerm, mensagem)
```

Figuras 1 e 2: Assinaturas das funções do código.

Abaixo estão expostas as principais partes do código, divididas nos processos de construção, determinização e minimização do autômato finito.

Primeiramente, as funções de criação do autômato a partir de gramática e também de tokens. A primeira está dividida em duas funções, uma que cria os estados do autômato e a

outra as produções de cada estado. Para a criação do autômato por tokens, é necessário somente uma função, pois não é necessário separar os nomes das regras delas próprias.

```
def criaAutomatoGramatica():
    global aut
    global gramaticas
    estTemp = []

    linhas = gramaticas.split('\n')
    indice = 0
    for linha in linhas:
        if linha != '':# se a linha não for vazia
            partes = linha.split('::=') # separa o nome da r
            estado = partes[0][1:len(partes[0])-1] # pega o
            if estado in estTemp: # estado repetido, concate
                linhas[0] = str(linhas[0])+'|'+partes[1]
                linhas.remove(linhas[indice])
            estTemp.append(estado)
            indice += 1

    for linha in linhas:
        print(linha) #mostra cada linha da gramatica
        if linha != '':
            partes = linha.split('::=') #faz a separação em
            estado = partes[0][1:len(partes[0])-1] #pega som
            aut[estado] = criaProducoesGramatica(partes[1])
```

Figura 3: Função de criação dos estados do autômato por GR.

```
def criaProducoesGramatica(producoes):
    regra = {}
    final = 0
    listTermTmp = set()
    producoes = producoes.split('|')
    for producao in producoes:
        if len(producao) == 1: # seja produção unitaria
            if producao != 'ε': # se não for epsilon
                insList(listTerm, producao) # chama funçao para ins
                if producao not in listTermTmp:
                    regra[producao] = 'X' # estado de erro
                else:
                    regra[producao] = str(str(regra[producao])+'X')
                listTermTmp.add(producao)
                aut['X'] = [1, {'':''}]
                insList(listNTerm, 'X') #insere estado de erro
            else: #epsilon
                final = 1 # estado sera final
        else: # terminal nao esta sozinho
            aux      = producao.split('<') #separação dos terminais
            terminal = aux[0] #pega o terminal de cada produção
            estado   = aux[1].replace('>', '') # pega e tira os > d
            listTermTmp.add(terminal) # coloca no listunto de termi
            insList(listTerm, terminal)
            if str(regra).find(terminal) != -1: # encontrou
                regra[terminal] = str(regra[terminal])+str(estado)
            else:
                regra[terminal] = estado

    return [final, regra]
```

Figura 4: Função de criação das produções do autômato por GR.

```
def criaProducoesToken():
    global listNTerm
    global tokens
    global aut

    linha = tokens.split("\n")
    for i in linha:
        if i != '':     # cada token
            cont = 0
            for j in i: # cada caractere do token
                if cont == 0: # primeiro caracterer do token
                    if aut == {}: # caso o automato esteja vazio
                        aut['S'] = [0, {j:''}]
                        listNTerm.append('S')
                    if j in aut['S'][1]: # se já tiver aquele token no automat
                        estadoFinal = ultimoEstado(j)
                        aut['S'][1][j] = str(aut['S'][1][j])+str(estadoFinal)
                    else: # caso padrão, o automato não está vazio e não tem n
                        estadoFinal = ultimoEstado(j)
                        aut['S'][1][j] = estadoFinal
                else: # caso não seja o primeiro caractere, cria uma regra pra
                    proximoEstado = listNTerm[len(listNTerm)-1]
                    estadoFinal = ultimoEstado(j)
                    aut[proximoEstado] = [0, {j:estadoFinal}]
                    aut[proximoEstado][1][j] = listNTerm[len(listNTerm)-1]

                # valia de se o estado é final ou não, se for
                if cont == len(i)-1: # ultima letra do token... gera um estado
                    aut[estadoFinal] = [1, {'':''}]

                cont += 1
```

Figura 5: Função de criação das produções do autômato através dos tokens.

Após os passos descritos anteriormente, já é possível obter o autômato finito não determinístico.

A determinização é realizada em duas funções, a principal é chamada e dentro dela é chamada a função de determinização por linha, onde é tratado os indeterminismos restantes. Estas funções estão implementadas da seguinte forma:

```
def determiniza():
    global aut
    global autDet
    global listNTerm
    global listTerm
    global autAux
    global estVisit
    global estVisitOrd

    autAux = aut

    ordemDeterm = []
    ordemDetermX = set()
    insList(ordemDeterm, listNTerm[0])

    autDet[listNTerm[0]] = aut[listNTerm[0]] # aut de S

    while 1:
        if ordemDeterm == '' or ordemDeterm == []:
            return

        if ordemDeterm[0] not in estVisit:
            estVisitOrd.append(ordemDeterm[0])
        estVisit.add(ordemDeterm[0])

        print("Estados visitados")
        print(estVisit)
        print(estVisitOrd , '\n')
```

```
    for terminal in listTerm:
        if str(autAux[ordemDeterm[0]][1]).find(terminal) != -1: # este terminal existe neste estado
            if autAux[ordemDeterm[0]][1][terminal] not in ordemDetermX and autAux[ordemDeterm[0]][1][terminal] != '':
                ordemDeterm.append(autAux[ordemDeterm[0]][1][terminal])
            ordemDetermX.add(autAux[ordemDeterm[0]][1][terminal])

    # passa o primeiro valor da lista de ordem (S) para determinizar a linha no autDet
    ordemDeterm.pop(0)

    if len(ordemDeterm) > 0 and ordemDeterm[0] not in estVisit:

        if len(ordemDeterm) > 0 and len(ordemDeterm[0]) > 1: #lista não esta vazia e há indeterminismo
            determinizaLinha(ordemDeterm[0])
        if ordemDeterm[0] != '':
            autDet[ordemDeterm[0]] = autAux[ordemDeterm[0]]
```

Figuras 6 e 7: Função principal de determinização.

```
def determinizaLinha(estado):
    global listTerm
    global autAux
    flagFinal = 0
    regra = {}
    # concatena
    for a in listTerm:
        regra[a] = ''

    for letra in estado:
        for terminal in listTerm:
            # Verifica se concatenacao nao repete nenhuma letra
            if str(autAux[letra][1]).find(terminal) != -1:
                #  valida para não concatenar as mesmas letras
                if str(regra[terminal]).find(autAux[letra][1][terminal]) == -1: # não encontrou
                    regra[terminal] = str(regra[terminal])+str(autAux[letra][1][terminal])

        # Verifica se este estado eh
        if autAux[letra][0] == 1:
            flagFinal = 1

    autAux[estado] = [flagFinal, regr
```

References:
AFD_Generator.py:24
AFD_Generator.py:38
AFD_Generator.py:38
AFD_Generator.py:42

Figura 8: Função de determinização por linha.

Posteriormente a determinização do código é realizado o processo de minimização sem classe de equivalência, que é composto basicamente pelas funções de eliminação de inalcançáveis e eliminação de mortos.

```
def eliMortos(estado):
    global listTerm
    global autDet

    for terminal in listTerm:
        if str(autDet[estado][1]).find(terminal) != -1: # existe ess
            aux = ''
            for producoes in autDet[estado][1][terminal]: # varre to
                aux = str(aux)+str(producoes)
            producao = aux

            eliProdMortos(producao, 0) # funcao que busca epsilon so
```

```
def eliProdMortos(estado, qtd):
    global listTerm
    global listMortos
    global autDet
    qtd += 1
    for terminal in listTerm:
        if estado != '' and autDet[estado][0] == 0: # nao eh final
            if str(autDet[estado][1]).find(terminal) != -1:
                if qtd < 20:
                    eliProdMortos(autDet[estado][1][terminal], qtd)
                else:
                    # MORTO
                    if autDet[estado][0] == 0: # nao eh final
                        listMortos.add(estado) # primeiro estado do
                    if autDet[autDet[estado][1][terminal]][0] == 0:
                        listMortos.add(autDet[estado][1][terminal])
```

Figuras 9 e 10: Funções de eliminação de mortos.

No processo de eliminação de mortos é feita a busca de produções que não seja final ou não alcancem algum estado que seja final. é feita a busca por epsilon nas produções, ou seja, que ele é final, se ele não for o código analisa através de loops e marca como mortos os que forem.

```
def eliInalc(estado):
    global listAlcancaveis
    global listTerm
    global aut
    global listVisitInalc

    listVisitInalc.add(estado)

    listAlcancaveis.add(estado) # insere o primeiro estado

    for terminal in listTerm:
        if str(aut[estado][1]).find(terminal) != -1: # existe esse termi
            aux = ''

            for producoes in aut[estado][1][terminal]:
                aux = str(aux)+str(producoes)
            producao = aux
            # estado ainda nao criado, anal. separadamente
            if len(producao) > 1:
                for cadaCaractEst in producao:
                    # funcao que busca ebsolon somente nas producoes dest
                    eliProdInal(cadaCaractEst, 0)
                    listAlcancaveis.add(cadaCaractEst)
            else:
                eliProdInal(producao, 0)
                listAlcancaveis.add(producao)
```

```
def eliProdInal(estado, qtd):
    global listTerm
    global listAlcancaveis
    global autDet
    global listVisitInalc
    qtd += 1
    for terminal in listTerm:
        if estado != '' and str(autDet[estado][1]).find(terminal) != -1:

            listVisitInalc.add(autDet[estado][1][terminal])

            if proximoEstado in listVisitInalc and qtd < 20: # Ok
                print('estado')
                print(estado)
                proximoEstado = autDet[estado][1][terminal]
                # Encontrou um alcançado
                listAlcancaveis.add(proximoEstado)
                #
                eliProdInal(proximoEstado, qtd)
            else: # deu loop
                return
```

Figuras 11 e 12: Funções de eliminação de inalcançáveis.

A última função do código é a função que monta o autômato em forma de matriz para fazer a exibição. Na função está contido apenas algoritmos que tratam os autômatos para exibição. A função serve para exibir todos os autômatos, tanto o determinístico como o não determinístico, com um conjunto de variáveis.

```
matriz = str(mensagem)+'\n\n'
matriz = str(matriz)+' |  δ\t'
for i in listTerm: # exibe todas as letrinhas
    matriz = str(matriz)+' | '+i+'\t'
matriz = str(matriz)+' |\n'
matriz = str(matriz)+' ---------'
for i in listTerm: # exibe todas as letrinhas
    matriz = str(matriz)+'--------'
matriz = str(matriz)+'\n'

for i in listNTerm:
    if eMorto == 1 or (eMorto == 0 and i not in listMortos and tipo == 1):
        #
        if autPrint[i][0] == 1:
            final = '*'
        else:
            final = ' '
        matriz = str(matriz)+' | '+final+i+'\t'

        for j in listTerm:
            matriz = str(matriz)+' | '+autPrint[i][1][j]+"\t"

        # Verifica se é morto
        morto = ' | '
        if i in listMortos and tipo == 1: # é morto
            morto = ' | Morto'

        matriz = str(matriz)+morto+'\n'

print(matriz)
```

Figura 13: Função de exibição dos autômatos.

4. Testes e Resultados

Para validar a implementação da aplicação são feitos testes diretamente no terminal, testando as saídas em cada execução. Para isso os casos de teste foram colocados em um arquivo (entrada.in) e testados através do código.

Um exemplos dos testes realizados é o seguinte:

```
1  se
2  entao
3  senao
4  <S> ::= a<A> | e<A> | i<A> | o<A> | u<A> | s | i
5  <A> ::= a<A> | e<A> | i<A> | o<A> | u | ε
```

Figura 14: Entrada de exemplo dos casos de teste.

Autômato Finito Minimo Determinístico

δ	a	e	i	o	u	s	n	t
S	A	AD	AX	A	A	BI	X	X
*A	A	A	A	A	X	X	X	X
*AD	A	A	A	A	X	X	E	X
*AX	A	A	A	A	X	X	X	X
BI	X	CJ	X	X	X	X	X	X
*X	X	X	X	X	X	X	X	X
E	X	X	X	X	X	X	X	F
*CJ	X	X	X	X	X	X	K	X
F	G	X	X	X	X	X	X	X
K	L	X	X	X	X	X	X	X
G	X	X	X	H	X	X	X	X
L	X	X	X	M	X	X	X	X
*H	X	X	X	X	X	X	X	X
*M	X	X	X	X	X	X	X	X

Figura 15: Exemplo de saída dos casos de teste.

5. Conclusão

Podemos concluir que foi possível atingir os objetivos solicitados no projeto. Foram encontradas algumas dificuldades durante o processo de implementação, como por exemplo o uso da linguagem de programação e suas propriedades, o que pode ser explicado pela falta de conhecimento aprofundado da linguagem, logo não foi possível obter o máximo que a mesma tinha a oferecer, mas isso não foi um impedimento para a finalização do projeto.

É possível afirmar que este projeto foi de grande utilidade para compreensão sobre o conteúdo da disciplina e como eles podem ser utilizados em áreas práticas. Todos estes fatores contribuíram e proporcionaram grande embasamento quanto a compreensão teórica do

curso de ciência da computação, o que possibilita a melhor percepção sobre a importância das linguagens formais e autômatos como um todo.

6. Referencial bibliográfico

RAMOS, M. V. M. **Linguagens Formais e Autômatos**: Universidade Federal do Vale do São Francisco, 2008. Disponível em: <http://www2.fct.unesp.br/docentes/dmec/olivete/lfa/arquivos/Apostila.pdf> . Acesso em: 23 jun. 2019.

MENEZES, P. **Linguagens formais e autômatos.** Serie livros didáticos. Bookman, 2008

MENEZES, P. B. **Linguagens Formais e Autômatos.** Ed. 6. Editora Bookman, 2011.

RICARTE, I. L. M. **Conversão para autômato finito determinístico.** 2014. Disponível em: http://www.dca.fee.unicamp.br/cursos/EA876/apostila/HTML/node48.html. Acesso em: 23 jun. 2019.