Relatório do trabalho da disciplina de Processamento de Linguagens

TP01 - Linguagens Regulares

Hugo Filipe Nogueira Silva – a16368

Hugo Daniel Neves Poças - a26339

Pedro Ricardo Rocha Silva - a26342

ESIPL

Barcelos, 2024

ESCOLA SUPERIOR DE TECNOLOGIA IPCA

Afirmo por minha honra que não recebi qualquer apoio não autorizado na realização deste trabalho prático. Afirmo igualmente que não copiei qualquer material de livro, artigo, documento web ou de qualquer outra fonte exceto onde a origem estiver expressamente citada.

Hugo Filipe Nogueira Silva – a16368

Hugo Daniel Neves Poças - a26339

Pedro Ricardo Rocha Silva - a26342

Índice

Autómatos Finitos Deterministas (AFD) – Exercício A

Introdução

Este capítulo apresenta uma implementação em Python de um Algoritmo de Reconhecimento de Linguagens baseado em Autómatos Finitos Deterministas (AFD). O código fornece funcionalidades para ler a definição de um AFD a partir de um ficheiro JSON, gerar a representação gráfica do grafo associado ao AFD utilizando a biblioteca Graphviz e reconhecer palavras de acordo com as regras definidas pelo AFD.

Estrutura do Código

O código está estruturado em três partes principais: a leitura da definição do AFD a partir de um ficheiro JSON, a geração da representação gráfica do grafo do AFD e o reconhecimento de palavras.

Leitura da Definição do AFD

O método carregar_automato(ficheiro_definicao) é responsável por carregar a definição do AFD a partir de um ficheiro JSON. Este método utiliza a biblioteca padrão json do Python para carregar o conteúdo do ficheiro e retorná-lo como um dicionário Python.

```
def carregar_automato(ficheiro_definicao : str) -> dict:
    # Carrega o automato a partir do ficheiro JSON
    with open(ficheiro_definicao, 'r') as ficheiro:
        return json.load(ficheiro)
```

Validações do Autómato

A função validaAutomato tem como objetivo validar a definição de um autómato representado por um dicionário em Python. Esta função verifica diversos critérios importantes para garantir a correta definição do autómato, abrangendo desde a estrutura básica até detalhes específicos relacionados às transições e símbolos.

Verificação do Tipo de Dados:

A função inicia verificando se o argumento passado é de facto um dicionário. Isso é fundamental, pois a função espera receber um dicionário que represente um autómato. Caso contrário, o programa é encerrado com uma mensagem de erro indicando que o ficheiro JSON deve ser um dicionário.

```
    if not isinstance(automato, dict):
        exit("O ficheiro JSON deve ser um dicionário.")
```

Verificação dos Estados Finais:

Em seguida, verifica-se se o autómato possui estados finais definidos. Se não houver estados finais definidos, o programa é encerrado com uma mensagem de erro informando que o autómato não tem estados finais definidos.

```
    if automato['F'] == '':
        exit("O automato não tem estados finais definidos.")
```

Verificação dos Estados Finais no Autómato:

Posteriormente, a função verifica se todos os estados finais definidos no autómato realmente existem nos estados do autómato. Se algum estado final definido não existir, o programa é encerrado com uma mensagem de erro indicando que nem todos os estados finais estão definidos no autómato.

```
    if not all(estado in automato['delta'].keys() for estado in auto-
mato['F']):
        exit("O automato não tem todos os estados finais definidos.")
```

Verificação dos Estados do Autómato:

Após isso, verifica-se se o autómato possui estados definidos. Se não houver nenhum estado definido, a função adiciona os estados com base nas transições presentes. Por outro lado, se já houver estados definidos, a função verifica se todos os estados presentes nas transições também estão definidos como estados do autómato.

```
    if not automato['Q']:
        automato['Q'] = sorted(list(set(automato['delta'].keys())))
    else:
        # Verifica se Q tem todos os estados que estão em delta
        if not all(estado in automato['delta'].keys() for estado in auto-
mato['Q']):
            exit(f"O automato não tem todos os estados definidos.")
```

Verificação dos Símbolos do Autómato:

Em seguida, a função verifica se o autómato possui símbolos definidos. Se não houver nenhum símbolo definido, a função obtém todos os símbolos das transições, seleciona apenas os valores únicos e ordena a lista.

```
    if not automato['V']:
        automato['V'] = []
        for transitions in automato['delta'].values():
            for transition in transitions.keys():
                automato['V'].append(transition)
        automato['V'] = sorted(list(set(automato['V'])))
```

Verificação do Símbolo 'ε':

É realizada uma verificação adicional para garantir que o símbolo 'ε' não esteja presente nos símbolos do autómato. Se estiver presente, o programa é encerrado com uma mensagem de erro indicando que o autómato não é um AFD (Autómato Finito Determinista).

```
    if 'ε' in automato['V']:
        exit("O automato não é AFD, porque o simbolo 'ε' está em V.")#ve-
rificar se paramos o programa ou se apenas avisamos o user
        pass
```

Verificação de Transições Múltiplas:

Por fim, a função verifica se há mais de uma transição para o mesmo símbolo em alguma transição do autómato. Se essa condição for verificada, o programa é encerrado com uma mensagem de erro indicando que o autómato é um AFND (Autómato Finito Não-Determinista).

```
    for transitions in automato['delta'].values():
        for destinations in transitions.values():
            if isinstance(destinations, list) and len(destinations) > 1:
                exit("O automato é AFND, porque existe mais do que uma
transição para o mesmo simbolo.")
```

Gerar Representação Gráfica do Grafo

O método gerar_grafo(automato) utiliza a biblioteca Graphviz para gerar a representação gráfica do grafo associado ao AFD. Este método percorre os estados e as transições definidas no AFD e utiliza os métodos da biblioteca Graphviz para adicionar os nós e as arestas correspondentes ao grafo.

```
def gerar_grafo(automato: dict) -> graphviz.Digraph:

    # Cria o grafo
    grafo: graphviz.Digraph = graphviz.Digraph(format='png')
    # Adiciona os estados ao grafo
    for estado in automato['Q']:
        if estado in automato['F']:
            grafo.node(estado, shape='doublecircle')
        else:
            grafo.node(estado, shape='circle')

    # Adiciona as transições ao grafo
    for estado, trans in automato['delta'].items():
        for simbolo, prox_estado in trans.items():
            grafo.edge(estado, prox_estado, label=simbolo)

    return grafo
```

Reconhecimento de Palavras

O método reconhecer_palavra(automato, palavra) é responsável por reconhecer se uma palavra pertence à linguagem definida pelo AFD. Este método percorre a palavra, verificando se existe uma transição definida para cada símbolo da palavra. Se a palavra for reconhecida, o método retorna verdadeiro juntamente com o caminho percorrido. Caso contrário, retorna falso e indica a situação de erro encontrada.

```
def reconhecer_palavra(automato : dict, palavra : str) -> tuple[bool,
list, str]:
    estado_atual : str = automato['q0']
    caminho : list = [estado_atual]

    for char in palavra:
        try:
            prox_estado : str = automato['delta'][estado_atual][char]
            caminho.append(prox_estado)
            estado_atual = prox_estado
        except KeyError:
            return False, caminho, f"Não há transição do estado {es-
tado_atual} com o símbolo '{char}'."

    if estado_atual in automato['F']:
        return True, caminho, None
    else:
        return False, caminho, f"O estado {estado_atual} não é um estado
final."
```

Utilização do Código

Para utilizar o código, pode-se fornecer o caminho para um ficheiro JSON contendo a definição do AFD e opcionalmente especificar as operações desejadas, como a geração da representação gráfica do grafo ou o reconhecimento de uma palavra.

Exemplos de utilização:

```
python3 1-AFD.py automato.json -graphviz
```

Este comando irá gerar a representação gráfica do grafo associado ao AFD definido no ficheiro automato.json.

```
python3 1-AFD.py automato.json -rec 101
```

Este comando irá reconhecer a palavra "101" utilizando o AFD definido no ficheiro automato.json.

Conclusão

A implementação apresentada neste capítulo oferece uma forma simples e eficiente de trabalhar com Autómatos Finitos Deterministas, permitindo a definição, visualização e reconhecimento de linguagens através de AFDs.

Hugo Filipe Nogueira Silva – a16368, Hugo Daniel Neves Poças - a26339 e Pedro Ricardo Rocha Silva - a26342

Expressão Regular para AFND – Exercício B

Introdução

Este capítulo apresenta uma implementação em Python de um Algoritmo de Conversão de Expressões Regulares para Autómatos Finitos Não-Determinísticos (AFND). O código apresenta funcionalidades para converter uma expressão regular num AFND, permitindo representar linguagens regulares de forma não determinística.

Estrutura do Código

O código está estruturado em três partes principais: a definição das operações básicas nos AFNDs, a conversão da expressão regular num AFND e a execução do programa principal.

Operações Básicas nos AFNDs

As operações básicas nos AFNDs incluem a concatenação, a união e o fecho de Kleene. Cada operação é implementada como uma função que recebe dois AFNDs, um como entrada e retorna um novo AFND como saída.

Concatenação:

A função concatenação(afnd1, afnd2) recebe dois AFNDs como entrada e concatena-os, adicionando transições vazias do estado final de afnd1 para o estado inicial de afnd2.

```
    def concatenacao(afnd1, afnd2):

        # Adiciona transições vazias do estado final de afnd1 para o es-
tado inicial de afnd2
        for estado_final in afnd1["F"]:
            afnd1["delta"].setdefault(estado_final, {}).update({"" :
[afnd2["q0"]]})

        # Atualiza os estados finais do AFND resultante
        afnd1["F"] = afnd2["F"]

        # Atualiza as transições do AFND resultante
        afnd1["delta"].update(afnd2["delta"])

        return afnd1
```

União:

A função união(afnd1, afnd2) recebe dois AFNDs como entrada e realiza a união entre eles, criando um novo estado inicial e definindo transições vazias deste estado para os estados iniciais de afnd1 e afnd2.

```
    def uniao(afnd1, afnd2):

        # Cria um novo estado inicial para o AFND resultante
        novo_estado_inicial = criar_estado()

        # Define as transições vazias do novo estado inicial para os es-
tados iniciais de afnd1 e afnd2
        novo_delta = {"": [afnd1["q0"], afnd2["q0"]]}

        # Atualiza o conjunto de estados, alfabeto e conjunto de estados
finais do AFND resultante
        novo_afnd = {
            "Q": [novo_estado_inicial] + afnd1["Q"] + afnd2["Q"],
            "Sigma": afnd1["Sigma"] | afnd2["Sigma"],
            "delta": {**novo_delta, **afnd1["delta"], **afnd2["delta"]},
            "q0": novo_estado_inicial,
            "F": afnd1["F"] + afnd2["F"]
        }

        return novo_afnd
```

Fecho de Kleene:

A função kleene(afnd) recebe um AFND como entrada e aplica o fecho de Kleene criando um novo estado inicial e um novo estado final, e definindo transições vazias entre estes estados e os estados originais do AFND.

```
    def kleene(afnd):

        # Cria um novo estado inicial e um novo estado final para o AFND
resultante
        novo_estado_inicial = criar_estado()
        novo_estado_final = criar_estado()

        # Define as transições vazias do novo estado inicial para o es-
tado inicial do AFND original
        # e do estado final do AFND original para o novo estado final
        novo_delta = {"": [afnd["q0"], novo_estado_final]}

        # Atualiza o conjunto de estados, alfabeto e conjunto de estados
finais do AFND resultante
        novo_afnd = {
            "Q": [novo_estado_inicial, novo_estado_final] + afnd["Q"],
            "Sigma": afnd["Sigma"],
            "delta": {**novo_delta, **afnd["delta"]},
            "q0": novo_estado_inicial,
            "F": [novo_estado_final]
        }

        return novo_afnd
```

operadores: Este é o nome do dicionário que é usado para mapear as operações.

"seq": Esta chave está associada à operação de concatenação. Quando o código encontrar a operação "seq" numa expressão regular, ele chama a função concatenacao.

"alt": Esta chave está associada à operação de união. Quando o código encontrar a operação "alt" numa expressão regular, ele chama a função uniao.

"kle": Esta chave está associada à operação de fecho de Kleene. Quando o código encontrar a operação "kle" numa expressão regular, ele chama a função kleene.

```
    # Dicionário para mapear as operações
    operadores = {
        "seq": concatenacao,
        "alt": uniao,
        "kle": kleene
    }
```

Conversão da Expressão Regular num AFND

A função converter_afnd(expressao_regular) recebe uma expressão regular como entrada e converte-a num AFND utilizando as operações básicas nos AFNDs.

```
def converter_estadoER(estadoER):

        if "simb" in estadoER:  # Se o nó representa um símbolo então...

            # Cria dois novos estados para representar a transição com o
símbolo
            estado_inicial = criar_estado()
            estado_final = criar_estado()

            # Define a transição com o símbolo do estado inicial para o
estado final
            delta = {estado_inicial: {estadoER["simb"]: [estado_final]}}
            return {
                "Q": [estado_inicial, estado_final],
                "Sigma": {estadoER["simb"]},
                "delta": delta,
                "q0": estado_inicial,
                "F": [estado_final]
            }

        else:  # Se o nó representa um operador (seq, alt ou kle) en-
tão...

            # Converte recursivamente os argumentos do operador para
AFNDs
            sub_afnds = [converter_estadoER(arg) for arg in esta-
doER["args"]]

            # Aplica a operação correspondente ao operador aos AFNDs con-
vertidos
            return operadores[estadoER["op"]](*sub_afnds)

    # Converte a expressão regular num AFND
    afnd = converter_estadoER(expressao_regular)

    return afnd
```

Utilização do Código

Para utilizar o código, deve-se fornecer o caminho para um ficheiro JSON que contém a expressão regular. O programa converterá a expressão regular num AFND e imprimirá o AFND resultante no terminal.

Exemplo de Utilização:

```
python meu_programa.py expressao_regular.json
```

Conclusão

A implementação apresentada neste capítulo oferece uma forma simples e eficiente de converter expressões regulares em Autómatos Finitos Não-Determinísticos, permitindo a representação e manipulação de linguagens regulares de forma não determinística.

Conversão de AFND para AFD – Exercício C

Introdução

Este script Python é uma ferramenta projetada para converter autómatos finitos não-determinísticos (AFND) em autómatos finitos determinísticos (AFD). O processo de conversão é essencial para simplificar a análise e o reconhecimento de linguagens regulares, permitindo a implementação de reconhecedores mais eficientes. A conversão é realizada através da construção de conjuntos de estados alcançáveis e aplicação do fecho-ε.

Estrutura do Código

O código é dividido em várias funções principais que tratam da leitura, a conversão e a gravação de autómatos:

carrega_automato(caminho): Lê um AFND de um ficheiro JSON.

```
def carrega_automato(ficheiro_automato : str) -> dict:
    with open(ficheiro_automato, 'r', encoding='utf-8') as ficheiro:
        return json.load(ficheiro)
```

AFNDtoAFD(automatoAFND): Converte o AFND lido em AFD.

A função **AFNDtoAFD** é responsável por converter um autómato finito não-determinístico (AFND) num autómato finito determinístico (AFD). Ela utiliza conceitos de teoria de autómatos como o fecho-ε e a construção de subconjuntos.

Subfunções Utilizadas:

-encontrar_conjunto(estadoAtual, simbolo, transicoes): Esta função interna é usada para determinar todos os estados que podem ser alcançados a partir de um conjunto de estados atual dado um símbolo específico. Utiliza um conjunto (set) para garantir que os estados não sejam repetidos e retorna um frozenset imutável dos estados alcançáveis.

```python
def encontrar_conjunto(estadoAtual, simbolo, transicoes) -> frozenset:# 'Daqui, até onde posso ir?'
    # Criamos um set, para garantir que não existem estados repetidos
    conjuntoAlcancavel : set = set()
    for estado in estadoAtual:
        if estado in transicoes and simbolo in transicoes[estado]:
            conjuntoAlcancavel.update(transicoes[estado][simbolo])
    return frozenset(conjuntoAlcancavel)# Retorna um frozenset, que é imutavel
```

- fecho_epsilon(conjunto, transicoes): Calcula o fecho-ε de um conjunto de estados, que são todos os estados alcançáveis a partir do conjunto dado, considerando apenas transições ε (epsilon). A função usa uma stack para explorar todos os estados alcançáveis e adiciona novos estados ao fecho à medida que são encontrados.

```python
def fecho_epsilon(conjunto : set, transicoes : dict) -> frozenset:
    fecho : set = set(conjunto)

    # Expande recursivamente o fecho épsilon de um estado,
    #adicionando estados alcançáveis através de transições épsilon a um dado conjunto.
    Codeium: Refactor | Explain | Generate Docstring | ×
    def expandir_fecho(estado : str):
        if estado in transicoes and 'ε' in transicoes[estado]: # Se o estado tiver transições com o simbolo 'ε'
            for proximoEstado in transicoes[estado]['ε']: # Para cada estado alcançável por uma transição épsilon
                if proximoEstado not in fecho: # Se o estado ainda não foi visitado
                    fecho.add(proximoEstado) # Adiciona o estado ao fecho
                    expandir_fecho(proximoEstado) # Chama a função recursivamente

    for estado in conjunto:
        expandir_fecho(estado)
```

Processo de Conversão:

- Inicializações: A função começa por configurar as variáveis iniciais, como os estados e transições do AFND, além do estado inicial e os estados finais.

```python
transicoesAFND : dict = automatoAFND['delta']
estadoInicialAFND : dict = automatoAFND['q0']
estadosFinaisAFND : set = set(automatoAFND['F'])
estadosAFD : dict = {}
alfabetoAFD : list = [simbolo for simbolo in automatoAFND['V'] if simbolo != 'ε']
transicoesAFD : dict = {}
stack : list = []
```

- Construção dos Estados do AFD: A determinização inicia com o fecho-ε do estado inicial. Este conjunto de estados representa o primeiro estado do AFD. A função usa uma stack para verificar os conjuntos de estados que ainda precisam ser processados.

```
conjuntoEstadosInicial : frozenset = fecho_epsilon({estadoInicialAFND}, transicoesAFND)
estadosAFD[f"N{len(estadosAFD)}"] = conjuntoEstadosInicial # Cria o novo estado no AFD
stack.append(conjuntoEstadosInicial)
```

- construir_AFD(listaEstadosAtuais)

No loop principal, a função processa cada conjunto de estados na stack. Para cada símbolo do alfabeto, exceto ε, determina-se os estados alcançáveis e aplica-se o fecho-ε. Novos conjuntos encontrados são colocados na stack para análise subsequente.

```
def construir_AFD(listaEstadosAtuais):
    # Declara que as variáveis estão fora do escopo da função, mas podem ser modificadas
    nonlocal estadosAFD, stack, transicoesAFD
    for simbolo in alfabetoAFD:
        listaEstadosAlcancavel : frozenset = encontrar_conjunto(listaEstadosAtuais, simbolo, transicoesAFND)
        listaFechoEpsilon : frozenset = fecho_epsilon(listaEstadosAlcancavel, transicoesAFND)
        if listaFechoEpsilon:
            if listaFechoEpsilon not in estadosAFD.values():
                estadosAFD[f"N{len(estadosAFD)}"] = listaFechoEpsilon # Adiciona o novo conjunto de estados ao AFD
                stack.append(listaFechoEpsilon)
            for estado, valor in estadosAFD.items(): # Para cada estado nos estados AFD
                if valor == listaEstadosAtuais: # Verifica se o valor do conjunto atual é igual ao conjunto do estado
                    conjuntoAtual = estado
                if valor == listaFechoEpsilon:
                    conjuntoAlcancavel = estado
            transicoesAFD.setdefault(conjuntoAtual, {})[simbolo] = conjuntoAlcancavel

    if stack:
        construir_AFD(stack.pop())
```

- Identificação dos Estados Finais do AFD: Qualquer novo estado do AFD que contenha pelo menos um estado final do AFND é considerado um estado final do AFD.

```
estadosFinaisAFD = []
for estado, conjunto in estadosAFD.items():
    if conjunto.intersection(estadosFinaisAFND):
        estadosFinaisAFD.append(estado)
```

- Construção do Objeto AFD: O AFD é então construído como um dicionário contendo seus estados, alfabeto, estado inicial, estados finais, e transições.

```
afd = {
    "Q": list(estadosAFD.keys()),
    "V": alfabetoAFD,
    "q0": "N0",
    "F": estadosFinaisAFD,
    "delta": transicoesAFD
}
```

- **Retorno**: A função retorna o autómato finito determinístico(AFD) construído.

- **gravar_automato(afd, caminho):** Grava o AFD convertido num ficheiro JSON.

```
def gravar_automato(afd, ficheiro_automato : str) -> None:
    with open(ficheiro_automato, 'w' , encoding='utf-8') as ficheiro:
        json.dump(afd, ficheiro, ensure_ascii=False, indent=4)
```

Utilização do Código

Para utilizar o script, deve-se fornecer o caminho para um ficheiro JSON que contém a definição AFND e o nome para o output do script.

```
python 3-AFNDtoAFD.py <caminho_para_afnd.json> -output <caminho_para_afd.json>
```

Exemplo de Utilização:

```
python 3-AFNDtoAFD.py meu_afnd.json -output meu_afd.json
```

Conclusão:

Esta implementação fornece um método prático e eficiente para a conversão de Autómatos Finitos Não-Determinísticos em Determinísticos.