Relatório do trabalho da disciplina de Processamento de Linguagens

Trabalho Prático 01

Grupo - 28

Rodrigo Pilar - 26536

Rúben Ribeiro - 25994

Engenharia de Sistemas Informáticos

abril de 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.

Rodrigo Pilar - 26536

Rúben Ribeiro - 25994

Índice

Rodrigo Pilar Rúben Ribeiro

Introdução

Este trabalho foi realizado com o intuito de explorar, aplicar e desenvolver os conceitos relacionados com a U.C. Processamento de Linguagens. Aplicamos e desenvolvemos conceitos relacionados com a construção de um Autómato Finito Não-Determinístico (AFND) a partir de uma expressão regular (ER), bem como a conversão desse AFND em um Autómato Finito Determinístico (AFD), além de outros exercícios relacionados.

Inicialmente, é realizada uma análise das necessidades do problema, definindo os objetivos a serem alcançados. Em seguida, são elaboradas as estruturas de dados necessárias para representar os autómatos, incluindo estados, alfabeto, transições e estados finais.

A implementação inclui a interpretação das expressões regulares e a construção dos autómatos correspondentes. Para isso, são desenvolvidas funções que avaliam as expressões regulares e reconhecem palavras de autómatos gerados. Além disso, é implementada a conversão de autómatos não determinísticos em determinísticos, visando otimizar o processo de reconhecimento de palavras.

Adicionalmente, o trabalho inclui a geração de grafos para visualização dos autómatos, proporcionando uma representação gráfica que auxilia na compreensão e análise das estruturas.

Por fim, são discutidas possíveis melhorias e extensões para o trabalho, visando aprimorar a eficiência e a funcionalidade das soluções implementadas.

Exercício A

```
import json
import argparse
from graphviz import Digraph
```

O programa importa as bibliotecas necessárias para funcionar, **json** para trabalhar com arquivos JSON, **argparse** para analisar argumentos de linha de comando e **Digraph** da biblioteca **graphviz** para criar e visualizar grafos.

```
def ler_automato(arquivo):
    # Abre o ficheiro JSON e carrega a definição do autómato
    with open(arquivo, "r", encoding="utf-8") as f:
        return json.load(f)

def gerar_grafo(automato):
    # Inicializa um grafo utilizando a biblioteca Graphviz
    dot = Digraph(comment='Automato')

    # Adiciona um nó de início vazio ao grafo
    dot.node('start', shape='none', label='')

    # Percorre os estados do autómato
    for estado in automato["delta"].keys():
        # Define a forma do nó como "duplo círculo" se o estado for final,
senão como "círculo"
        shape = "doublecircle" if estado in automato["F"] else "circle"
        # Adiciona o estado como um nó ao grafo
        dot.node(estado, estado, shape=shape)

    # Percorre as transições do autómato
    for estado_inicial, transitions in automato["delta"].items():
        # Percorre as transições a partir de cada estado inicial
        for simbolo, estado_final in transitions.items():
            # Define o rótulo da transição como o símbolo, exceto se for "ε"
(epsilon), que é representado como "ε"
            label = simbolo if simbolo != "ε" else "ε"
            # Adiciona uma aresta ao grafo representando a transição
            dot.edge(estado_inicial, estado_final, label=label)

    return dot
```

A função **"ler_automato"** abre um ficheiro JSON que contém a definição de um autómato e carrega essas definições num formato que o Python pode entender.

A função **"gerar_grafo"** cria um diagrama de um autómato, utilizando a biblioteca Graphviz. Percorre os estados e transições do autómato e adiciona os nós e arestas correspondentes ao diagrama. Os estados finais são representados por círculos duplos, enquanto os outros estados são representados por círculos simples.

```
def reconhecer_palavra(automato, palavra):
    # Inicializa o estado atual com o estado inicial do autómato
    estado_atual = automato["q0"]
    # Inicializa uma lista para armazenar o caminho percorrido no autómato
    caminho = ["q0"]

    # Percorre cada símbolo na palavra a ser reconhecida
    for simbolo in palavra:
        # Verifica se o símbolo não pertence ao alfabeto do autómato
        if simbolo not in automato["delta"][estado_atual]:
            # Retorna uma mensagem indicando que a palavra não é reconhecida
devido a um símbolo inválido
            return f"'{palavra}' não é reconhecida\n[símbolo '{simbolo}' não
pertence ao alfabeto]"

        # Atualiza o estado atual de acordo com a transição para o próximo
estado com base no símbolo atual
        estado_atual = automato["delta"][estado_atual].get(simbolo, None)
        # Adiciona o estado atual ao caminho percorrido
        caminho.append(estado_atual)

    # Verifica se o estado atual é um estado final do autómato
    if estado_atual in automato["F"]:
        # Retorna uma mensagem indicando que a palavra é reconhecida e o
caminho percorrido
        return f"'{palavra}' é reconhecida\n[caminho {'->'.join(caminho)}]"
    else:
        # Retorna uma mensagem indicando que a palavra não é reconhecida
porque o estado atual não é final
        return f"'{palavra}' não é reconhecida\n[caminho {'-
>'.join(caminho)}, {estado_atual} não é final]"
```

A função **"reconhecer_palavra"** verifica se uma palavra é reconhecida pelo autómato, percorrendo os símbolos da palavra para verificar a transição de estado até atingir um estado final.

```
def main():
    # Configura o parser de argumentos da linha de comando
    parser = argparse.ArgumentParser(description="Reconhecedor de linguagens baseado em um Autómato Finito Determinístico (AFD)")
    parser.add_argument('arquivo', help="Caminho para o ficheiro JSON que contém a definição do AFD")
    parser.add_argument('-graphviz', '--grafo', action='store_true', help="Gerar o grafo do autómato")
    parser.add_argument('-rec', '--palavra', help="Palavra a ser reconhecida pelo AFD")
    # Analisa os argumentos da linha de comando
    args = parser.parse_args()
    # Lê a definição do autómato a partir do ficheiro JSON
    automato = ler_automato(args.arquivo)
    # Gera o grafo do autómato se a opção correspondente for ativada
    if args.grafo:
        dot = gerar_grafo(automato)
        dot.render('automato_grafo', view="True", format='png')
    # Reconhece a palavra especificada pelo utilizador, se fornecida
    if args.palavra:
        resultado = reconhecer_palavra(automato, args.palavra)
        print(resultado)
if __name__ == "__main__":
    main()
```

Este código implementa uma função principal **"main()"** que utiliza a biblioteca **argparse** para processar argumentos da linha de comando. Os argumentos permitidos são o **caminho para um arquivo** JSON que contém a definição de um Autómato Finito Determinístico (AFD), a opção **-graphviz** para gerar o grafo do autómato e a opção **-rec** para especificar uma palavra a ser reconhecida pelo AFD.

Exercício B

Este código implementa a conversão de uma expressão regular em formato JSON para um Autómato Finito Não Determinístico (AFND). Ele utiliza a árvore de expressão regular para processar os operadores e símbolos, construindo as transições do AFND de acordo com as regras específicas de cada operador (alternância, concatenação e fecho de Kleene). O resultado é um AFND representado em formato JSON, pronto para ser utilizado em análises posteriores.

```
def construir_afnd_de_er(er_json, alfabeto=None, estados=None,
transicoes=None, estados_finais=None, contador_estado=0):

    # Conjuntos para armazenar informações sobre o AFND

    alfabeto = set()  # Símbolos do alfabeto

    estados = set()  # Estados do AFND

    transicoes = {}  # Transições do AFND

    estados_finais = set()  # Estados finais do AFND

    contador_estado = 0  # Contador para gerar nomes de estados únicos

    # Função para gerar um novo nome de estado

    def gerar_estado():

        nonlocal contador_estado

        nome_estado = f"q{contador_estado}"

        contador_estado += 1

        return nome_estado
```

Esta função **“construir_afnd_de_er”** serve para construir um Autómato Finito Não Determinístico (AFND) a partir de uma expressão regular representada em formato JSON. Também definimos a função **“gerar_estado”** que é um contador para gerar nomes únicos de estados.

```
# Função para processar um nó da árvore de expressão regular
    def processar_no(no, estado_atual):
        nonlocal alfabeto, estados, transicoes, estados_finais
        if isinstance(no, dict):
            if "simb" in no:
                # Se o nó for um símbolo, adiciona-o ao alfabeto e cria uma
transição para um novo estado
                alfabeto.add(no["simb"])
                novo_estado = gerar_estado()
                estados.add(novo_estado)
                transicoes.setdefault(estado_atual,
{}).setdefault(no["simb"], []).append(novo_estado)
                return novo_estado
            if "op" in no:
                if no["op"] == "seq":
                    # Se for uma sequência de símbolos, conecta cada
elemento da sequência
                    estado_atual_seq = estado_atual
                    for arg in no["args"]:
                        novo_estado_seq = processar_no(arg,
estado_atual_seq)
                        if novo_estado_seq:
                            estado_atual_seq = novo_estado_seq
                    estados_finais.add(estado_atual_seq)
                elif no["op"] == "alt":
                    # Se houver alternância entre expressões, são criados
novos estados para cada opção.
```

```
                    novo_estado_esquerdo = processar_no(no["args"][0],
estado_atual)

                    novo_estado_direito = processar_no(no["args"][1],
estado_atual)

                    if novo_estado_esquerdo:

                        estados_finais.add(novo_estado_esquerdo)

                    if novo_estado_direito:

                        estados_finais.add(novo_estado_direito)

                    transicoes.setdefault(estado_atual, {}).setdefault("ε",
[]).extend([novo_estado_esquerdo, novo_estado_direito])

                elif no["op"] == "kle":

                    # Se for um fecho de Kleene, cria um loop com transições
epsilon

                    novo_estado_kle = gerar_estado()

                    estados.add(novo_estado_kle)

                    estados_finais.add(novo_estado_kle)  # Torna o novo
estado um estado final

                    estado_interno = processar_no(no["args"][0],
novo_estado_kle)

                    if estado_interno:

                        transicoes.setdefault(estado_interno,
{}).setdefault("ε", []).append(novo_estado_kle)

                    transicoes.setdefault(estado_atual, {}).setdefault("ε",
[]).append(novo_estado_kle)

                    transicoes.setdefault(novo_estado_kle,
{}).setdefault("ε", []).append(estado_atual)


    estado_inicial = gerar_estado()  # Estado inicial

    estados.add(estado_inicial)
```

```
    processar_no(er_json, estado_inicial)  # Processa a expressão regular

    estados_finais.add(estado_inicial)  # Adiciona o estado inicial aos
finais para permitir reconhecimento de palavra vazia

    # Retorna a estrutura do AFND

    return {

        "V": sorted(list(alfabeto)),

        "Q": sorted(list(estados)),

        "delta": transicoes,

        "q0": estado_inicial,

        "F": sorted(list(estados_finais))

    }
```

A função **"processar_no"** é responsável por percorrer recursivamente a árvore de uma expressão regular, representada em formato JSON. Ele examina um nó da árvore de expressão regular e executa diferentes ações com base no tipo de nó e no seu conteúdo. Faz as seguintes verificações:

- **Verificação de Símbolo:**

Nesta parte do código, é verificado se o nó da árvore da expressão regular é um símbolo. Se for, o símbolo é adicionado ao alfabeto do autómato, um novo estado é criado e é estabelecida uma transição do estado atual para o novo estado. Essa lógica garante que cada símbolo da expressão regular seja incluído no alfabeto do autómato e que exista uma transição correspondente para cada símbolo.

- **Verificação de Operador Sequência:**

Nesta parte, é verificado se o nó representa uma sequência de símbolos na expressão regular. Se for o caso, cada símbolo da sequência é processado individualmente, chamando recursivamente a função **"processar_no"** para cada símbolo. Após processar todos os símbolos, o último estado da sequência é adicionado ao conjunto de estados finais do autómato.

- **Verificação de Operador Alternância:**

Aqui, é verificado se o nó representa uma alternância entre expressões na expressão regular. Se houver alternância, são criados estados para cada opção de expressão. Cada expressão é processada separadamente, e os estados finais de cada expressão são adicionados ao conjunto de estados finais do autómato. Transições epsilon ("ε") são estabelecidas entre os estados atuais e os estados finais das expressões.

- **Verificação de Operador Fecho de Kleene:**

Por último, é verificado se o nó representa um fecho de Kleene na expressão regular. Se for o caso, um novo estado é criado para representar o loop do fecho de Kleene. A expressão interna do fecho de Kleene é processada recursivamente, e o novo estado é adicionado ao conjunto de estados finais do autómato. Transições epsilon ("ε") são estabelecidas entre os estados atual, interno e novo estado, permitindo a repetição zero ou mais vezes da expressão.

Finalmente, a estrutura completa do AFND é retornada como um dicionário, contendo o alfabeto (V), os estados (Q), as transições (delta), o estado inicial (q0) e os estados finais (F), todos ordenados e convertidos de conjuntos para listas.

```python
def ler_er_de_arquivo(caminho_arquivo):
    # Lê a expressão regular de um arquivo JSON
    with open(caminho_arquivo, "r") as arquivo:
        er_json = json.load(arquivo)
    return er_json

def guardar_afnd_em_arquivo(afnd_json, caminho_arquivo):
    # Guarda a estrutura do AFND em um arquivo JSON
    with open(caminho_arquivo, "w") as arquivo:
        json.dump(afnd_json, arquivo, indent=4)
```

A função **“ler_er_de_arquivo”** tem como objetivo ler uma expressão regular armazenada em um arquivo JSON. Ela abre o arquivo especificado pelo caminho **“caminho_arquivo”**, lê o conteúdo e o converte em um objeto Python usando a função **“json.load”**, retornando a expressão regular em formato de dicionário.

Já a função **“guardar_afnd_em_arquivo”** tem a finalidade de guardar a estrutura de um autómato finito não determinístico (AFND) em um arquivo JSON. Recebendo como entrada o dicionário **“afnd_json”** que representa a estrutura do AFND e o **“caminho_arquivo”** onde será guardado o arquivo.

```
def main():
    parser = argparse.ArgumentParser(description='Converte uma expressão
regular em um autômato finito não determinístico (AFND).')
    parser.add_argument('input', help='Arquivo JSON contendo a expressão
regular')
    parser.add_argument('--output', help='Nome do arquivo de saída para o
AFND (padrão: afnd.json)', default='afnd.json')
    args = parser.parse_args()

    er_json = ler_er_de_arquivo(args.input)  # Lê a expressão regular do
arquivo
    afnd_json = construir_afnd_de_er(er_json, alfabeto=set(), estados=set(),
transicoes={}, estados_finais=set(), contador_estado=0) # Constrói o AFND a
partir da expressão regular
    guardar_afnd_em_arquivo(afnd_json, args.output)  # Salva o AFND em um
arquivo JSON

    print(f"A estrutura do AFND foi guardada no arquivo '{args.output}'.")

if __name__ == "__main__":
    main()
```

O objetivo da função **"main"** é permitir que o utilizador forneça os parâmetros necessários para a execução do programa diretamente no terminal. Para isso, ela utiliza o módulo **"argparse"** para criar um analisador de argumentos da linha de comando. Primeiramente, temos descrição do programa usando o parâmetro **"description"**. Em seguida, são definidos os argumentos que o programa aceitará: o argumento **'input'** que especifica o arquivo JSON contendo a expressão regular a ser convertida em AFND (obrigatório), e o argumento **'--output'** que permite ao utilizador escolher o nome do arquivo de saída para o AFND.

Após a definição dos argumentos, a função lê a expressão regular do arquivo JSON especificado utilizando a função **"ler_er_de_arquivo"**. Depois, chama a função **"construir_afnd_de_er"** para construir o AFND a partir da expressão regular lida, criando a estrutura do AFND em formato de dicionário.

Por fim, a função chama a função **"guardar_afnd_em_arquivo"** para guardar a estrutura do AFND no arquivo JSON de saída escolhido. Após a conclusão do processo, imprime uma mensagem indicando o nome do arquivo onde a estrutura do AFND foi guardada.

Exercício C

```python
def fecho_epsilon(estados, delta):
    # Calcula o fecho epsilon de um conjunto de estados em um AFN.
    fecho = set(estados)
    pilha = list(estados)
    while pilha:
        estado = pilha.pop()
        novos_estados = delta.get(estado, {}).get("ε", [])
        for novo_estado in novos_estados:
            if novo_estado not in fecho:
                fecho.add(novo_estado)
                pilha.append(novo_estado)
    return fecho
```

A função **"fecho_epsilon**" calcula o fecho épsilon de um conjunto de estados em um autómato finito não determinístico (AFND). Utilizando um algoritmo de busca em profundidade, explora as transições épsilon a partir de cada estado do conjunto inicial, adicionando os estados alcançáveis ao fecho. O resultado é o conjunto de todos os estados que podem ser alcançados por transições épsilon a partir dos estados fornecidos.

```
def transicao(estados, simbolo, delta):
    # Calcula o conjunto de estados alcançáveis a partir de um conjunto de
estados e um símbolo de entrada em um AFN.
    resultado = set()
    for estado in estados:
        transicoes = delta.get(estado, {})
        resultado.update(transicoes.get(simbolo, []))
    return resultado
```

A função **"transicao",** percorre os estados de entrada, consulta as transições possíveis para cada estado com o símbolo fornecido e atualiza o conjunto de estados alcançáveis com esses estados. O resultado final é o conjunto de todos os estados alcançáveis a partir do conjunto inicial de estados e do símbolo de entrada dado.

```
def nfa_para_dfa(nfa):
    # Converte um autômato finito não determinístico (AFN) em um autômato
finito determinístico (AFD).
    V, q0, F = nfa["V"], nfa["q0"], nfa["F"]
    delta_nfa = nfa["delta"]
    Q_dfa, delta_dfa = [], {}
    fila = [fecho_epsilon({q0}, delta_nfa)]
    mapa_estados_dfa = {frozenset(fila[0]): "q0"}

    while fila:
        estados_atuais = fila.pop(0)
        estado_dfa_atual = mapa_estados_dfa[frozenset(estados_atuais)]
        Q_dfa.append(estado_dfa_atual)

        for simbolo in V:
            proximos_estados = fecho_epsilon(transicao(estados_atuais,
simbolo, delta_nfa), delta_nfa)
            if proximos_estados:
                if frozenset(proximos_estados) not in mapa_estados_dfa:
                    fila.append(proximos_estados)
                    novo_estado = "q" + str(len(mapa_estados_dfa))
                    mapa_estados_dfa[frozenset(proximos_estados)] =
novo_estado
                delta_dfa.setdefault(estado_dfa_atual, {})[simbolo] =
mapa_estados_dfa[frozenset(proximos_estados)]

        if any(estado in F for estado in estados_atuais):
            F.append(estado_dfa_atual)

    return {"V": V, "Q": Q_dfa, "delta": delta_dfa, "q0": q0, "F": F}
```

A função **“nfa_para_dfa”** converte um autómato finito não determinístico (AFND) em um autômato finito determinístico (AFD). Ela faz isso explorando as transições do AFND para determinar os estados e as transições do AFD. Ao percorrer os conjuntos de estados do AFND, chamados de "estados atuais", a função calcula o fecho épsilon desses estados. Em seguida, para cada símbolo do alfabeto, determina os próximos estados alcançáveis e seus fechos épsilon. Com base nisso, constrói as transições do AFD, atribuindo novos estados conforme necessário. O resultado é um AFD equivalente ao AFND original, permitindo o reconhecimento determinístico de linguagens descritas pela expressão regular.

```
def main():
    # Configura o parser de argumentos da linha de comando
    parser = argparse.ArgumentParser(description='Converter AFND para AFD')
    parser.add_argument('input', help='Arquivo de entrada do AFND no formato
JSON')
    parser.add_argument('--output', help='Arquivo de saída do AFD no formato
JSON')
    args = parser.parse_args()

    # Lê a definição do AFND a partir do arquivo JSON
    with open(args.input, "r") as f:
        nfa = json.load(f)

    # Converte o AFND em um AFD
    dfa = nfa_para_dfa(nfa)

    # Define o nome do arquivo de saída ou usa o padrão "AFD.json"
    output_file = args.output if args.output else "AFD.json"
    # Escreve a definição do AFD no arquivo JSON de saída
    with open(output_file, "w") as f:
        json.dump(dfa, f, indent=4)

if __name__ == "__main__":
    main()
```

A função **“main”** tem como principal objetivo facilitar a interação do utulizador com o programa diretamente a partir da linha de comando. Para isso, utiliza o módulo **“argparse”**, o qual permite definir e analisar os argumentos fornecidos pelo utilizador. Inicialmente, é especificada uma descrição para o programa por meio do parâmetro **“description”**. Em seguida, são definidos os argumentos aceites pelo programa: o argumento **'input'** indica o arquivo JSON contendo a definição do autómato finito não determinístico (AFND) a ser convertido em autómato finito determinístico (AFD), e o argumento **‘--output'** permite que o usuário especifique o nome do arquivo de saída para o AFD.

Após a definição dos argumentos, a função lê a definição do AFND a partir do arquivo JSON fornecido pelo utilizador usando a função **“ler_er_de_arquivo”**. Em seguida, chama a função **“nfa_para_dfa”** para converter o AFND em um AFD. O nome do arquivo de saída para o AFD é determinado com base no argumento fornecido pelo utulizador ou, caso não seja especificado, utiliza-se o nome padrão **'AFD.json'**. Por fim, a função escreve a definição do AFD no arquivo JSON de saída e imprime uma mensagem indicando onde a estrutura do AFD foi guardada.

Conclusão

O desenvolvimento deste trabalho permitiu adquirir conhecimentos aprofundados sobre conceitos essenciais de Processamento de Linguagens, destacando a ligação entre expressões regulares e autómatos finitos.

A construção dos autómatos envolveu a definição de estados, alfabeto, transições e estados finais, oferecendo uma compreensão completa dos elementos essenciais que compõem estas estruturas. Além disso, a implementação de algoritmos para avaliar expressões regulares, reconhecer palavras e converter autómatos não determinísticos em determinísticos demonstrou a aplicação prática destes conceitos.

A criação de representações gráficas dos autómatos através da biblioteca “Graphviz” acrescentou uma dimensão visual ao trabalho, tornando mais fácil a visualização e compreensão das estruturas de forma intuitiva.

No final, este trabalho permitiu uma reflexão sobre as possibilidades de melhorias e expansões, sublinhando a importância contínua da aprendizagem e do aperfeiçoamento das técnicas e algoritmos apresentados. A aplicação prática dos conceitos de Processamento de Linguagem neste trabalho serviu como uma base sólida para futuras explorações e projetos na área.