



INSTITUTO POLITÉCNICO DO CÁVADO E AVE

ESCOLA SUPERIOR DE TECNOLOGIA

LICENCIATURA EM ENGENHARIA DE SISTEMAS INFORMÁTICOS

TRABALHO PRÁTICO 1

PROCESSAMENTO DE LINGUAGENS

Arthur Fellipe Cerqueira Gomes – 24200

Júlia Dória Rodrigues – 24204

Barcelos

2024

Sumário

| | |
|--|----|
| Parte A..... | 4 |
| Estrutura do Código | 4 |
| Bibliotecas/Módulos Importados | 4 |
| Função ‘ler_automato(nome_arquivo)’ | 4 |
| Função ‘gerar_diagrama(automato)’ | 5 |
| Função ‘reconhecedor(entrada, estado_inicial, transicoes, estados_finais)’ | 6 |
| Função ‘main()’ | 7 |
| Exemplos de Utilização | 8 |
| Gerar Diagrama do Autômato..... | 8 |
| Reconhecer Palavras | 10 |
| Conclusão..... | 10 |
| Parte B..... | 11 |
| Estrutura do Código | 11 |
| Bibliotecas/Módulos Importados | 11 |
| Variável global ‘contador_estados’ | 11 |
| Função ‘novo_estado()’ | 11 |
| Função ‘construir_arvore(er)’ | 11 |
| Função ‘converter_para_afnd(arvore)’ | 12 |
| Função ‘ler_er(nome_arquivo)’ | 14 |
| Função ‘main()’ | 14 |
| Exemplo de Utilização | 15 |
| Conclusão..... | 18 |
| Parte C..... | 18 |

| | |
|--|----|
| Estrutura do Código | 18 |
| Biblioteca/Módulos Importados..... | 18 |
| Função ‘converter_afnd_para_afd’ | 18 |
| Função ‘encontrar_alcancaveis’ | 19 |
| Função ‘fecho_epsilon’ | 19 |
| Função ‘construir_afd’ | 20 |
| Função ‘carregar_automato’ | 21 |
| Função ‘salvar_automato’ | 21 |
| Função ‘main’ | 22 |
| Exemplo de utilização | 22 |
| Conclusão | 25 |

Índice de Figuras

| | |
|---|----|
| Figura 1 - Diagrama gerado a partir de autômato | 9 |
| Figura 2 - Reconhecimento de palavra com erro de aceitação | 10 |
| Figura 3 - Reconhecimento de palavra com aceitação e caminho percorrido | 10 |

Parte A

Em atenção ao enunciado proposto na 'Parte A' deste Trabalho Prático 1, implementámos um algoritmo de reconhecimento de línguas baseado num Autómato Finito Determinista (AFD) em *Python*. O AFD é uma estrutura de dados essencial para o reconhecimento de línguas regulares, permitindo a validação de palavras pertencentes a uma determinada língua.

O código foi estruturado para realizar três funcionalidades principais:

1. Leitura da definição do autómato a partir de um ficheiro JSON.
2. Geração da representação gráfica do autómato usando a biblioteca *Graphviz*.
3. Reconhecimento de uma palavra, indicando se ela é aceita pelo autómato e mostrando o caminho percorrido.

Estrutura do Código

Bibliotecas/Módulos Importados

As três principais importações utilizadas para o desenvolvimento deste trabalho são:

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

- **json:** A biblioteca JSON é utilizada para ler e manipular arquivos JSON em *Python*, permitindo a definição do autómato em formato estruturado.
- **argparse:** Este módulo é uma biblioteca do *Python* utilizada para facilitar a interpretação dos argumentos da linha de comando, facilitando a interação com o usuário. Neste trabalho, é utilizado para interpretar os argumentos da linha de comando fornecidos pelo usuário, como o nome do arquivo JSON de entrada e opções adicionais.
- **Digraph de graphviz:** Utilizado para gerar a representação gráfica do autómato em formato de grafo.

Função ‘ler_automato(nome_arquivo)’

```
def ler_automato(nome_arquivo):
    with open(nome_arquivo, "r", encoding="utf-8") as f:
```

```
return json.load(f)
```

Esta função é responsável por ler a definição do autômato a partir de um arquivo JSON e retornar a definição como um dicionário. O ficheiro JSON deve conter as seguintes informações:

- **‘Q’** - Conjunto de estados do autômato: este é o conjunto de todos os estados possíveis do autômato
- **‘V’** - Alfabeto da linguagem reconhecida pelo autômato: é o conjunto de símbolos que compõem a linguagem reconhecida pelo autômato.
- **‘q0’** - Estado inicial do autômato: informa qual é o estado inicial a partir do qual o autômato começa seu processamento.
- **‘F’** - Conjunto de estados finais: contém os estados nos quais o autômato pode terminar sua execução, indicando que uma palavra de entrada foi aceite.
- **‘delta’** - Função de transição do autômato: é a função que define quais as transições possíveis entre os estados do autômato de acordo com os símbolos de entrada do alfabeto.

Função ‘gerar_diagrama(automato)’

A função ‘gerar_diagrama(automato)’ é responsável por criar o diagrama de grafos que representa visualmente o autômato.

```
def gerar_diagrama(automato):
    # Gerar o diagrama de grafos do automato
    dot = Digraph(comment='Automato')

    # 'none' para um node invisível
    dot.node('start', shape='none', Label='')
    # criar transição inicial
    dot.edge('start', automato["q0"], Label='')

    # Criar os estados
    for estado in automato["F"]:
        # caso o estado seja final
        if estado in automato["F"]:
            dot.node(estado, estado, shape="doublecircle")
        else:
            dot.node(estado, estado, shape="circle")

    # Criar as transições
    for estado_inicial, transitions in automato["delta"].items():
        for simbolo, estado_final in transitions.items():
```

```
dot.edge(estado_inicial, estado_final, label = simbolo)

return dot
```

Esta função gera o diagrama de grafos do autômato usando a biblioteca Graphviz. Começando com um nó invisível de partida, ela cria os nós (estados) e as arestas (transições) do grafo, estabelecendo transições do estado inicial para os outros estados do autômato. Cada estado é representado por um nó no grafo, sendo marcados como estados finais se necessário. As transições entre os estados são ilustradas por arestas direcionadas, cada uma etiquetada com o símbolo de entrada correspondente. Os estados finais são representados como círculos duplos e o estado inicial é um ponto de partida.

Função ‘reconhecedor(entrada, estado_inicial, transicoes, estados_finais)’

A ‘função reconhecedor(entrada, estado_inicial, transicoes, estados_finais)’ é responsável por verificar se uma determinada entrada (palavra) é reconhecida pelo autômato finito.

```
def reconhecedor(entrada, estado_inicial, transicoes, estados_finais):
    entrada = entrada.replace("ε", "")

    estado_atual = estado_inicial
    caminho = [estado_atual]

    for char in entrada:
        # caso haja uma transição
        if char in transicoes[estado_atual]:
            estado_atual = transicoes[estado_atual][char]
            caminho.append(estado_atual)

        # caso haja uma transição com a palavra vazia
        elif "ε" in transicoes[estado_atual]:
            estado_atual_aux = transicoes[estado_atual]["ε"]
            if char in transicoes[estado_atual_aux]:
                estado_atual = transicoes[estado_atual_aux][char]
                caminho.append(estado_atual)
            else:
                # se não houver transição definida para este caracter de entrada, a
                # palavra não é aceite
                return False, caminho, f"símbolo '{char}' não é acessível a partir
                do estado {estado_atual}"
        else:
            return False, caminho, f"símbolo '{char}' não pertence ao alfabeto"
```

```

if estado_atual in estados_finais:
    return True, caminho, f"estado {estado_atual} é final"
else:
    return False, caminho, f"estado {estado_atual} não é final"

```

A função inicia o reconhecimento a partir do estado inicial fornecido e percorre a entrada, verificando cada caractere em relação às transições do autômato. Durante o processo, mantém um registro do caminho percorrido. Se a palavra for aceita, a função retorna verdadeiro, junto com o caminho percorrido e uma mensagem indicando que o estado atual é final. Caso contrário, retorna falso, o caminho percorrido até o momento e uma mensagem explicando por que a palavra não é reconhecida, como um caractere que não pertence ao alfabeto ou um estado não final.

Além disso, a função também lida com transições que contêm a palavra vazia ‘ ϵ ’, permitindo o reconhecimento de palavras que podem ter transições diretas ou passar pelo estado vazio.

Função ‘main()’

A função ‘main()’ é responsável por coordenar a interação do usuário com o programa.

```

def main():
    parser = argparse.ArgumentParser(description='Automato Finito Determinista')
    parser.add_argument('ficheiro', type=str, help='Ficheiro JSON do automato')
    parser.add_argument('-graphviz', action='store_true', help='Gerar diagrama do automato')
    parser.add_argument('-reconhecer', type=str, help='Reconhecer palavra')

    args = parser.parse_args()

    automato = ler_automato(args.ficheiro)

    if args.graphviz:
        dot = gerar_diagrama(automato)
        dot.render('automaton_graph', view=True, format='png')

    if args.reconhecer:
        palavra = args.reconhecer
        resultado, caminho, mensagem = reconhecedor(palavra, automato["q0"],
        automato["delta"], automato["F"])
        if resultado:
            print(f"A palavra '{palavra}' é aceite pelo automato.")
            print(f"Caminho: {caminho}")
        else:
            print(f"A palavra '{palavra}' não é aceite pelo automato.")

```

```

        print(f"Erro: {mensagem}")

if __name__ == "__main__":
    main()

```

Assim, ela gerencia a entrada do usuário e chama as funções auxiliares conforme necessário. A função utiliza o módulo ‘argparse’ para interpretar os argumentos da linha de comando, que podem ser:

- O arquivo JSON do autômato.
- A opção ‘-graphviz’ para gerar o diagrama do autômato.
- A opção ‘-reconhecer’ seguida de uma palavra a ser reconhecida.

Após analisar os argumentos, a função deverá ler o autômato do arquivo JSON, gerar o diagrama do autômato se a opção -graphviz for especificada e reconhecer a palavra fornecida se a opção -reconhecer for utilizada. Em seguida, deverá imprimir na saída padrão se a palavra é aceita ou não pelo autômato, juntamente com o caminho percorrido (se for o caso) e uma mensagem explicando se existiu algum erro durante o processo.

Exemplos de Utilização

Gerar Diagrama do Autômato

Ao utilizar o comando *parteA.py automato.json -graphviz* no terminal o programa gera e exibe em formato .png o diagrama do autômato definido no arquivo ‘automato.json’. Neste exemplo foi utilizado o seguinte autômato:

```

{
  "Q" : "",
  "V" : "",
  "q0" : "q1",
  "F" : ["q5", "q6"],
  "delta" : {
    "q1" : {
      "a" : "q2",
      "b" : "q3"
    },
    "q2" : {
      "a" : "q1",
      "ε" : "q4"
    }
  }
}

```



```

    },
    "q3" : {
        "b" : "q1",
        "ε" : "q4"
    },
    "q4" : {
        "c" : "q5",
        "d" : "q6"
    },
    "q5" : {
        "d" : "q5"
    },
    "q6" : {
        "c" : "q6"
    }
}
}

```

Por sua vez, a imagem gerada após o tratamento deste autômato foi a seguinte:

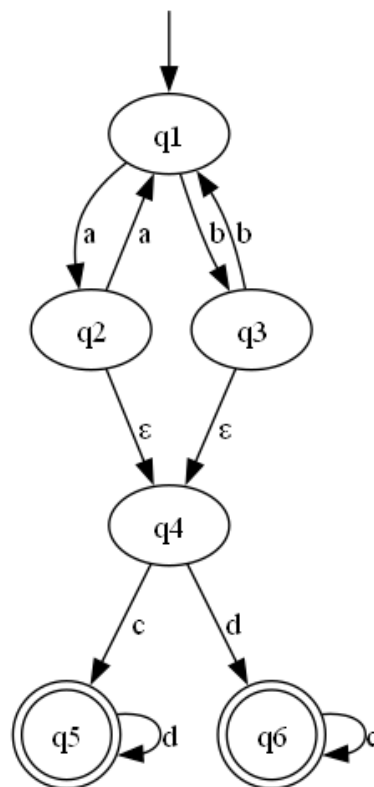
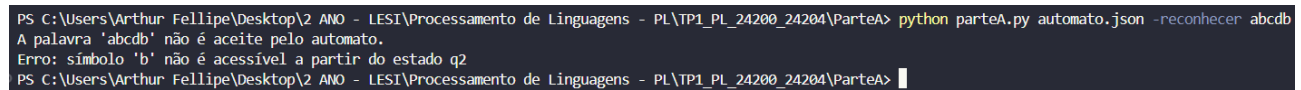


Figura 1 - Diagrama gerado a partir de autômato

Reconhecer Palavras

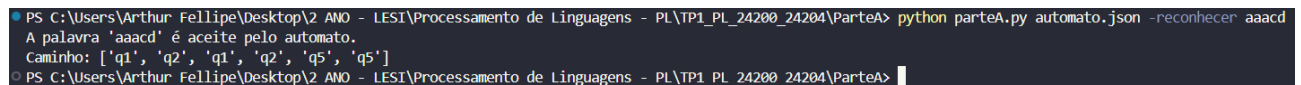
Ao utilizar o comando `python parteA.py automato.json -reconhecer abcdb`, o programa reconhece a palavra ‘abcdb’ usando o autômato definido no arquivo ‘automato.json’ e indica que esta não é aceite pelo autômato, pois o símbolo ‘b’ não é acessível a partir do estado ‘q2’.



```
PS C:\Users\Arthur_Fellipe\Desktop\2 ANO - LEST\Processamento de Linguagens - PL\TP1_PL_24200_24204\ParteA> python parteA.py automato.json -reconhecer abcdb
A palavra 'abcdb' não é aceite pelo automato.
Erro: símbolo 'b' não é acessível a partir do estado q2
PS C:\Users\Arthur_Fellipe\Desktop\2 ANO - LEST\Processamento de Linguagens - PL\TP1_PL_24200_24204\ParteA>
```

Figura 2 - Reconhecimento de palavra com erro de aceitação

De outra forma, ao passar a palavra ‘aaacd’, o programa indica que esta é aceite e exibe o caminho percorrido no autômato.



```
PS C:\Users\Arthur_Fellipe\Desktop\2 ANO - LEST\Processamento de Linguagens - PL\TP1_PL_24200_24204\ParteA> python parteA.py automato.json -reconhecer aaacd
A palavra 'aaacd' é aceite pelo automato.
Caminho: ['q1', 'q2', 'q1', 'q2', 'q5', 'q5']
PS C:\Users\Arthur_Fellipe\Desktop\2 ANO - LEST\Processamento de Linguagens - PL\TP1_PL_24200_24204\ParteA>
```

Figura 3 - Reconhecimento de palavra com aceitação e caminho percorrido

Conclusão

A implementação do algoritmo de reconhecimento de linguagens baseado em AFD em Python permite a validação eficiente de palavras em uma linguagem regular. O código apresentado oferece uma solução clara e concisa para as funcionalidades propostas, facilitando o entendimento e a utilização do autômato determinista. A estrutura modular do código permite fácil manutenção e extensão para funcionalidades adicionais, como a conversão entre expressões regulares, AFND e AFD.

Parte B

Estrutura do Código

Bibliotecas/Módulos Importados

Variável global ‘contador_estados’

Função ‘novo_estado()’

```
def novo_estado():  
    global contador_estados  
    estado = f"q{contador_estados}"  
    contador_estados += 1  
    return estado
```

Esta função é responsável por gerar um novo estado único para o AFND, garantindo que cada estado tenha um identificador único. Ao utilizar uma variável global ‘contador_estados’, a função gera um novo identificador de estado único. Após a geração do identificador único, o contador de estados é incrementado para garantir que o próximo estado gerado seja único.

Função ‘construir_arvore(er)’

```
def construir_arvore(er):  
    if isinstance(er, str):  
        return {"op": "simb", "args": er}  
    elif "simb" in er:  
        return {"op": "simb", "args": er["simb"]}  
    elif "ε" in er:  
        return {"op": "ε", "args": None}  
    elif "plus" in er:  
        return {"op": "plus", "args": [construir_arvore(arg) for arg in er["args"]]}  
    else:  
        op = er["op"]  
        args = [construir_arvore(arg) for arg in er["args"]]  
        return {"op": op, "args": args}
```

Esta função é responsável por construir uma árvore de expressão regular a partir do objeto JSON fornecido. A árvore é representada como um dicionário onde cada nó tem um operador (op) e seus argumentos (args).

A função verifica o tipo de dado da entrada *er* e constrói a árvore de acordo com as seguintes regras:

- Se 'er' for uma string, ela retorna um nó folha representando um símbolo do alfabeto.
- Se 'er' contiver a chave "simb", ela retorna um nó folha com o símbolo especificado.
- Se 'er' contiver a chave "ε", ela retorna um nó folha representando uma transição vazia.
- Se 'er' contiver a chave "plus", ela retorna um nó representando o operador "plus" e seus argumentos são processados recursivamente.

Nos demais casos, ela retorna um nó com o operador e seus argumentos processados recursivamente.

Função 'converter_para_afnd(arvore)'

```
def converter_para_afnd(arvore):
    estados = set()
    transicoes = {}

    def adicionar_transicao(origem, simbolo, destino):
        if origem not in transicoes:
            transicoes[origem] = {}
        transicoes[origem][simbolo] = destino

    def percorrer_arvore(no, estado_inicial, estado_final):
        if no["op"] == "simb":
            estado_atual = novo_estado()
            estados.add(estado_atual)
            adicionar_transicao(estado_inicial, no["args"], estado_atual)
            adicionar_transicao(estado_atual, "ε", estado_final)
        elif no["op"] == "ε":
            adicionar_transicao(estado_inicial, "ε", estado_final)
        elif no["op"] == "plus":
            estado_atual = novo_estado()
            estados.add(estado_atual)
            adicionar_transicao(estado_inicial, "ε", estado_atual)
            for arg in no["args"]:
                percorrer_arvore(arg, estado_atual, estado_final)
        else:
            estado_atual = novo_estado()
```

```

    estados.add(estado_atual)
    for arg in no["args"]:
        percorrer_arvore(arg, estado_inicial, estado_atual)
        adicionar_transicao(estado_atual, "ε", estado_final)

estado_inicial = novo_estado()
estado_final = novo_estado()
estados.add(estado_inicial)
estados.add(estado_final)
percorrer_arvore(arvore, estado_inicial, estado_final)

# Extrair os símbolos do alfabeto (V) do dicionário de transições
alfabeto = set()
for origem, trans in transicoes.items():
    for simbolo in trans.keys():
        alfabeto.add(simbolo)

afnd = {
    "Q": list(estados), # Estados
    "V": list(alfabeto), # Alfabeto
    "q0": estado_inicial, # Estado inicial
    "F": [estado_final], # Estados finais
    "delta": transicoes # Transições
}

return afnd

```

A função ‘converter_para_afnd(arvore)’ converte uma árvore de expressão regular em um Autômato Finito Não Determinista (AFND) equivalente.

Inicialmente, são definidos os conjuntos de estados (estados) e as transições (transicoes). A função ‘adicionar_transicao()’ é utilizada para adicionar uma transição ao dicionário de transições. A função ‘percorrer_arvore()’ é uma função interna que utiliza uma abordagem recursiva para percorrer a árvore de expressão regular e construir o AFND correspondente. Durante a conversão, são considerados os seguintes operadores:

- ‘simb’: Representa um símbolo do alfabeto.
- ‘ε’: Representa uma transição vazia.
- ‘seq’: Representa a concatenação de duas expressões.
- ‘alt’: Representa a alternativa entre duas expressões.
- ‘kle’: Representa o fecho de Kleene de uma expressão.
- ‘plus’: Representa uma ou mais repetições de uma expressão.

Se o operador for "simb", um novo estado é criado para representar o símbolo e são adicionadas transições correspondentes. Se o operador for "ε", é adicionada uma transição vazia. Nos demais casos, um novo estado é criado para representar o operador e os argumentos são processados recursivamente. Ao final da conversão, o AFND é montado com os estados, alfabeto, estado inicial, estados finais e transições.

Função 'ler_er(nome_arquivo)'

```
def ler_er(nome_arquivo):
    with open(nome_arquivo, "r", encoding="utf-8") as f:
        er = json.load(f)
        if isinstance(er, str):
            raise ValueError("O arquivo JSON deve representar um objeto, não uma string.")
        return er
```

Esta função lê o arquivo JSON que contém a expressão regular e retorna o objeto JSON correspondente. Se o arquivo contiver uma string em vez de um objeto, um erro será lançado.

Função 'main()'

```
def main():
    parser = argparse.ArgumentParser(description='Conversor de Expressão Regular para AFND')
    parser.add_argument('ficheiro', type=str, help='Ficheiro JSON da expressão regular')
    parser.add_argument('--output', type=str, help='Ficheiro JSON de saída para o AFND')

    args = parser.parse_args()

    er = ler_er(args.ficheiro)

    arvore = construir_arvore(er)
    afnd = converter_para_afnd(arvore)

    with open(args.output, "w", encoding="utf-8") as f:
        json.dump(afnd, f, ensure_ascii=False, indent=4)

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

A função principal ‘main()’ gerencia a execução do programa. Ela utiliza o ‘argparse’ para obter os argumentos da linha de comando, ler a expressão regular do arquivo JSON, converter a expressão para um AFND e salvar o AFND resultante em um arquivo JSON especificado pelo usuário.

Exemplo de Utilização

Para exemplificar o funcionamento do programa, foi utilizada a seguinte expressão regular representada no arquivo er01.json:

```
er01.json
{
  "op": "alt",
  "args": [
    {
      "simb": "a"
    },
    {
      "op": "seq",
      "args": [
        {
          "simb": "a"
        },
        {
          "op": "kle",
          "args": [
            {
              "simb": "b"
            }
          ]
        }
      ]
    },
    {
      "simb": "c"
    },
    {
      "op": "plus",
      "args": [
        {
          "simb": "d"
        },
        {
          "simb": "e"
        }
      ]
    }
  ]
}
```

```

    ]
  },
  {
    "op": "kle",
    "args": [
      {
        "simb": "f"
      }
    ]
  }
]
}

```

Após executar o comando `python parteB.py er01.json --output afnd.json`, o programa converte essa expressão regular em um Autômato Finito Não Determinista (AFND) e salva o resultado no ficheiro `afnd.json`.

O conteúdo do ficheiro `afnd.json` gerado é o seguinte:

```

afnd.json
{
  "Q": [
    "q4",
    "q14",
    "q9",
    "q1",
    "q13",
    "q3",
    "q10",
    "q5",
    "q8",
    "q11",
    "q12",
    "q7",
    "q2",
    "q6"
  ],
  "v": [
    "a",
    "b",
    "c",
    "e",
    "d",
    "ε",
    "f"
  ],
}

```



```

"q0": "q1",
"F": [
  "q2"
],
"delta": {
  "q1": {
    "a": "q6",
    "b": "q8",
    "c": "q9",
    "ε": "q10",
    "f": "q14"
  },
  "q4": {
    "ε": "q3"
  },
  "q6": {
    "ε": "q5"
  },
  "q8": {
    "ε": "q7"
  },
  "q7": {
    "ε": "q5"
  },
  "q9": {
    "ε": "q5"
  },
  "q10": {
    "d": "q11",
    "e": "q12"
  },
  "q11": {
    "ε": "q5"
  },
  "q12": {
    "ε": "q5"
  },
  "q5": {
    "ε": "q3"
  },
  "q14": {
    "ε": "q13"
  },
  "q13": {
    "ε": "q3"
  },
  "q3": {
    "ε": "q2"
  }
}

```

```

    }
  }
}

```

Conclusão

O código apresentado utiliza uma abordagem recursiva para percorrer a árvore de expressão regular e construir o AFND correspondente, seguindo as regras de construção de AFND a partir dos operadores da expressão regular. Ademais, a sua estrutura modular do código facilita a manutenção e a extensão para suportar mais operadores ou funcionalidades no futuro. Ao executar o programa com o exemplo fornecido, o AFND resultante é gerado e salvo no arquivo ‘afnd.json’, representando de forma equivalente a expressão regular definida no arquivo ‘er01.json’.

Parte C

Na Parte C do trabalho prático, desenvolvemos um programa em Python para converter um Autômato Finito Não Determinista (AFND) em um Autômato Finito Determinista (AFD).

Estrutura do Código

Biblioteca/Módulos Importados

Para esta etapa subsequente do projeto, continuamos a fazer uso das bibliotecas *json* e *argparse* para garantir a funcionalidade e interatividade do programa.

```

import json
import argparse

```

Função ‘converter_afnd_para_afd’

```

def converter_afnd_para_afd(afnd: dict) -> dict:

```

A função `converter_afnd_para_afd` recebe um autômato finito não determinista (AFND) representado como um dicionário em Python e retorna um autômato finito determinista (AFD) também representado como um dicionário. Essa função é responsável por realizar a conversão do AFND para o AFD, aplicando uma série de etapas e algoritmos para isso.

Função ‘encontrar_alcancaveis’

Esta função recebe como entrada o estado atual, um símbolo do alfabeto e as transições do autômato. Ela é responsável por encontrar os estados alcançáveis a partir do estado atual quando é lido o símbolo especificado. Assim, ela itera sobre os estados atuais e verifica se há transições para o símbolo dado. Se houver, adiciona os estados alcançáveis ao conjunto de alcance. Por fim, retorna um conjunto imutável (frozenset) contendo os estados alcançáveis.

```
def encontrar_alcancaveis(estado_atual, simbolo, transicoes):
    alcancaveis = set()
    for estado in estado_atual:
        if estado in transicoes and simbolo in transicoes[estado]:
            alcancaveis.update(transicoes[estado][simbolo])
    return frozenset(alcancaveis)
```

Função ‘fecho_epsilon’

A função ‘fecho_epsilon’ calcula o fecho- ϵ de um conjunto de estados. Recebe como entrada um conjunto de estados e as transições do autômato. Ela utiliza uma função interna chamada `aumentar_fecho` para expandir recursivamente o fecho- ϵ de cada estado no conjunto. A função `aumentar_fecho` verifica se há transições ϵ para o estado atual e, se houver, adiciona os estados alcançáveis ao fecho. Ao final, retorna um conjunto imutável contendo o fecho- ϵ do conjunto de estados dado.

```
def fecho_epsilon(conjunto, transicoes):
    fecho = set(conjunto)

    def aumentar_fecho(estado):
        if estado in transicoes and '\epsilon' in transicoes[estado]:
            for proximo_estado in transicoes[estado]['\epsilon']:
                if proximo_estado not in fecho:
                    fecho.add(proximo_estado)
```

```

        aumentar_fecho(proximo_estado)

    for estado in conjunto:
        aumentar_fecho(estado)

    return frozenset(fecho)

```

Função ‘construir_afd’

A função ‘construir_afd’ é responsável por construir o autômato finito determinista (AFD) a partir do autômato finito não determinista (AFND). Recebe como entrada os estados atuais do AFND. Itera sobre os símbolos do alfabeto do AFD e encontra os estados alcançáveis para cada símbolo. Em seguida, calcula o fecho- ϵ desses estados e adiciona ao AFD, se ainda não estiver presente. Por fim, atualiza as transições do AFD com os estados alcançáveis e retorna o AFD resultante.

```

def construir_afd(estados_atuais):
    nonlocal estados_afd, pilha, transicoes_afd
    for simbolo in alfabeto_afd:
        alcancaveis = encontrar_alcancaveis(estados_atuais, simbolo,
transicoes_afnd)
        fecho_epsilon_alcancaveis = fecho_epsilon(alcancaveis, transicoes_afnd)
        if fecho_epsilon_alcancaveis:
            if fecho_epsilon_alcancaveis not in estados_afd.values():
                estados_afd[f"N{len(estados_afd)}"] = fecho_epsilon_alcancaveis
                pilha.append(fecho_epsilon_alcancaveis)
            for estado, valor in estados_afd.items():
                if valor == estados_atuais:
                    estado_atual = estado
                if valor == fecho_epsilon_alcancaveis:
                    proximo_estado = estado
            transicoes_afd.setdefault(estado_atual, {})[simbolo] =
proximo_estado

        if pilha:
            construir_afd(pilha.pop())

    transicoes_afnd = afnd['delta']
    estado_inicial_afnd = afnd['q0']
    estados_finais_afnd = set(afnd['F'])
    estados_afd = {}
    alfabeto_afd = [simbolo for simbolo in afnd['V'] if simbolo != '\epsilon']
    transicoes_afd = {}
    pilha = []

```

```

# Inicializa o AFD com o fecho-ε do estado inicial do AFND
fecho_inicial = fecho_epsilon({estado_inicial_afnd}, transicoes_afnd)
estados_afd[f"N{len(estados_afd)}"] = fecho_inicial
pilha.append(fecho_inicial)

construir_afd(pilha.pop())

estados_finais_afd = []
for estado, fecho in estados_afd.items():
    if fecho.intersection(estados_finais_afd):
        estados_finais_afd.append(estado)

afd = {
    "Q": list(estados_afd.keys()),
    "V": alfabeto_afd,
    "q0": "N0",
    "F": estados_finais_afd,
    "delta": transicoes_afd
}
return afd

```

Função ‘carregar_automato’

Esta função é responsável por carregar um autômato salvo em um ficheiro JSON para a memória, convertendo-o em um dicionário Python. Recebe como entrada o caminho para o arquivo JSON contendo a definição do autômato. Utiliza a função `json.load()` para ler o arquivo JSON e retornar o autômato carregado como um dicionário.

```

def carregar_automato(json_file: str) -> dict:
    with open(json_file, 'r', encoding='utf-8') as file:
        return json.load(file)

```

Função ‘salvar_automato’

A função `salvar_automato` tem como objetivo salvar um autômato em um arquivo JSON. Recebe como entrada o autômato a ser salvo (no formato de dicionário) e o caminho para o arquivo JSON de saída. Utiliza a função `json.dump()` para escrever o autômato no arquivo JSON especificado, garantindo que os caracteres sejam codificados corretamente e que o arquivo seja formatado com indentação para facilitar a leitura.

```

def salvar_automato(afd, json_file: str) -> None:

```

```
with open(json_file, 'w' , encoding='utf-8') as file:
    json.dump(afd, file, ensure_ascii=False, indent=4)
```

Função ‘main’

A função ‘main’ como já visto nas outras duas etapas deste relatório, é o ponto de entrada do programa. Ela inicia analisando os argumentos fornecidos pela linha de comando, que devem incluir o caminho para o ficheiro JSON contendo a definição do AFND e o caminho para o ficheiro de saída onde o AFD resultante será armazenado.

Após a análise dos argumentos, o programa carrega o autômato definido no arquivo JSON usando a função ‘carregar_automato’. Em seguida, utiliza a função ‘converter_afnd_para_afd’ para converter o AFND carregado em um AFD. Por fim, o AFD resultante é salvo em um ficheiro JSON utilizando a função ‘salvar_automato’.

```
def main():
    parser = argparse.ArgumentParser(description="Conversão de AFND para AFD")
    parser.add_argument("arquivo_json",
                        help="Caminho para o arquivo JSON contendo a definição do AFND")
    parser.add_argument('-output', metavar='output_file', type=str,
                        help='Arquivo JSON de saída para o AFD')

    args = parser.parse_args()

    afnd = carregar_automato(args.arquivo_json)
    afd = converter_afnd_para_afd(afnd)
    salvar_automato(afd, args.output)

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

Exemplo de utilização

Como teste do código gerado foi utilizado o seguinte ficheiro *afnd.json*, e através do comando *python parteC.py afnd.json -output afd.json* foi obtido o posterior *afd.json*:

```
{
  "Q": ["q0", "q1", "q2", "q3", "q4"],
  "V": ["a", "b", "c"],
  "q0": "q0",
```

```

    "F": ["q2", "q4"],
    "delta": {
        "q0": {"a": ["q1", "q2"], "b": ["q3"], "c": ["q4"]},
        "q1": {"b": ["q2", "q3"], "c": ["q1"]},
        "q2": {"a": ["q4"], "b": ["q0"]},
        "q3": {"c": ["q0"]},
        "q4": {"a": ["q2"], "c": ["q4"]}
    }
}

```

```

{
    "Q": [
        "N0",
        "N1",
        "N2",
        "N3",
        "N4",
        "N5",
        "N6",
        "N7",
        "N8",
        "N9",
        "N10",
        "N11",
        "N12"
    ],
    "V": [
        "a",
        "b",
        "c"
    ],
    "q0": "N0",
    "F": [
        "N1",
        "N3",
        "N4",
        "N5",
        "N7",
        "N8",
        "N10",
        "N11",
        "N12"
    ],
    "delta": {
        "N0": {
            "a": "N1",
            "b": "N2",

```

```

        "c": "N3"
    },
    "N3": {
        "a": "N4",
        "c": "N3"
    },
    "N4": {
        "a": "N3",
        "b": "N0"
    },
    "N2": {
        "c": "N0"
    },
    "N1": {
        "a": "N3",
        "b": "N5",
        "c": "N6"
    },
    "N6": {
        "b": "N7",
        "c": "N6"
    },
    "N7": {
        "a": "N3",
        "b": "N0",
        "c": "N0"
    },
    "N5": {
        "a": "N8",
        "b": "N9",
        "c": "N10"
    },
    "N10": {
        "a": "N1",
        "b": "N2",
        "c": "N3"
    },
    "N9": {
        "a": "N1",
        "b": "N2",
        "c": "N10"
    },
    "N8": {
        "a": "N11",
        "b": "N5",
        "c": "N12"
    },
    "N12": {

```



```
        "a": "N4",  
        "b": "N7",  
        "c": "N12"  
    },  
    "N11": {  
        "a": "N11",  
        "b": "N0",  
        "c": "N3"  
    }  
}  
}
```

Conclusão

Nesta parte final do projeto, desenvolvemos um código em Python para converter um Autômato Finito Não Determinístico (AFND) em um Autômato Finito Determinístico (AFD). Implementamos funções que identificam os estados alcançáveis a partir de um estado e um símbolo, calculam o fecho- ϵ de um conjunto de estados e constroem o AFD a partir do fecho- ϵ do estado inicial do AFND. Apesar de o processo de conversão poder aumentar o número de estados no autômato resultante, o objetivo é que o AFD gerado seja equivalente ao AFND original, mantendo a mesma linguagem reconhecida.