Universidade de São Paulo
Instituto De Ciências Matemáticas e de Computação

Relatório Trabalho II

Alunos: Aline Martins N. Belchior (NUSP: 15611649)
João Victor Cosme Neres de Sousa (NUSP: 15600696)
Larissa Freire de Jesus Costa (NUSP: 11207731)
Disciplina: SCC5832 - Teoria da Computação
Docente: João Luís Garcia Rosa

Novembro
2024

Técnicas

Criamos uma classe em Python para representar uma Máquina de Turing (MT) genérica, mas necessariamente determinística. Essa classe, em primeiro lugar, inicia as informações que definem uma Máquina de Turing e informações relativas às entradas a serem processadas pelo simulador:

```
class TuringMachine:
    def __init__(self):
        self.num_states = 0
        self.states = []
        self.tape_alphabet = []
        self.extended_alphabet = []
        self.accept_state = ''
        self.transitions = {}
        self.input_strings = []
        self.blank_symbol = 'B'
```

- O número de estados da máquina `self.num_states` e quem eles são `self.states`;
- `self.tape_alphabet` ou Σ, o alfabeto da fita, juntamente com o `self.extended_alphabet` ou Σ', alfabeto estendido da fita, contendo símbolos terminais e também os demais símbolos importantes para o funcionamento da MT (como o símbolo `B`, por exemplo);
- `self.accept_state` ou q_a, o estado de aceitação armazenado como uma string;
- `self.transitions` ou δ, função de transição de estado, representada por um dicionário;
- `self.input_strings`, lista de cadeias a serem testadas;
- `self.blank_symbol` ou `B`, símbolo branco da fita.

Para representar as transições previstas para uma Máquina de Turing foi utilizado um dicionário de dicionários. Nesse dicionário, cada um dos estados possui um sub-dicionário para armazenar as transições da MT de acordo com o símbolo lido pela cabeça da fita.

Exemplo: se a MT possui o estado q_0 e que ao ler a a partir de q_0, o próximo estado será q_1, o símbolo escrito na fita será b e a direção de movimento da cabeça será R, segue a representação:

```
self.transitions = {
 q0: {a: ('q1', 'b','R')...},
 q1:{...},
 q2: {...},
...
}
```

Estruturação do código

O código foi estruturado de acordo com as seguintes funções da classe que definimos como TuringMachine:

`__init__(self)`
Função que inicia as listas ou dicionários para armazenar os estados, os símbolos do alfabeto terminal e estendido, o estado de aceitação,transições e cadeias a serem testadas.

`read_input(self, input_file)`
Função que lê o arquivo de entrada que contém as definições da MT presentes em um `input_file` e inicializa ele. O arquivo é lido e suas informações são armazenadas na lista `lines`.

Na primeira linha, a função lê e armazena o número de estados da MT, além disso uma mensagem de erro é emitida se esse número for superior a 10. Após essa validação os estados q_i's são criados (com i variando de 0 até o número de estados lidos menos 1).

A leitura e armazenamento dos símbolos terminais acontece em seguida, lendo em primeiro lugar a quantidade dos símbolos e depois os símbolos em si. Uma validação é realizada para que o número de símbolos terminais não ultrapasse o limite imposto pelo enunciado (10 símbolos). A leitura do alfabeto estendido, que acontece a seguir, ocorre de forma análoga. Em sequência, esses dois alfabetos são combinados para formar o `self.full_alphabet`, representando o conjunto completo de símbolos da MT.

Em seguida, ocorre a leitura do índice do estado de aceitação, bem como a validação do índice e conversão para uma string no formato "qX", onde X representa o índice lido e qX, consequentemente, representa o estado de aceitação.

Para as transições, após a leitura da quantidade delas, ocorre a validação para que verifiquemos se ele é menor ou igual 50. Em seguida as transições são lidas por linha onde `q` é o estado de origem, `symbol_read` é o símbolo lido na fita, `q_next` é o estado destino, `symbol_write` é o símbolo a ser escrito na fita e `direction` é a direção do movimento do cabeçote (podendo ser L,

R ou S). Em seguida, verificamos se a transição contém o número correto de elementos, isto é, 5. Após isso, validações são aplicadas para verificar a transição lida, tais validações serão comentadas posteriormente. Além disso, se para um estado qN um símbolo lido já estiver associado a uma transição, informa-se que há uma duplicidade ocorrendo na nossa MT determinística. Por outro lado, se não há duplicidades adicionamos a transição válida ao dicionário `self.transitions`.

Por último, o número de cadeia de entradas é lido e verificado. Se passar pelas verificações (deve haver no máximo 10 cadeias com no máximo 20 símbolos de comprimento cada), as cadeias são lidas e armazenadas em `self.input_strings`. Cadeias vazias são, em nosso caso, representadas por "_".

`_validate_max_number`
Garante que um número não ultrapasse o limite estabelecido. Utilizada em diversas validações ao longo do código.

`_validate_state_index`
Verifica se o índice de um estado está dentro de um intervalo válido para a MT. Ou seja, se pertence ao intervalo `[0,self.num_states-1]`. Usado para validar o index do estado de aceitação.

`_validate_state`
Verifica se um estado pertence ao conjunto de estados definidos.

`_validate_symbol`
Verifica se um símbolo pertence ao `full_alphabet` da MT.

`_validate_direction`
Verifica se a direção de movimento dada na transição é válida, ou seja, se é igual a R, L ou S.

A função `simulate` simula o comportamento de uma MT com o intuito de avaliar se uma cadeia é aceita ou rejeitada. Ao iniciar o loop while, a configuração inicial da MT é capturada em uma tupla do seguinte formato: `(estado atual, posição do cabeçote, conteúdo da fita)`, como uma espécie de descrição instantânea mais elaborada. Se essa configuração já foi alcançada no passado a cadeia é rejeitada. Se o estado atual for o de aceitação, então a cadeia é aceita. A infinitude da fita tanto pra direita quanto pra esquerda também é garantida nessa etapa. O símbolo atual é obtido através da posição da cabeça da MT. Se houver uma transição definida

no estado atual para esse símbolo lido então ela é executada, se não houver a cadeia é rejeitada. Caso haja a transição, o símbolo atual lido pelo cabeçote e o estado atual são atualizados. Além disso, a cabeça se move de acordo com a direção da transição.

Já em `evaluate_strings` as cadeias a serem avaliadas são enviadas para `self.simulate` e o seu resultado é armazenado em uma lista. Em `write_output`, então, os resultados são escritos em um arquivo de saída.

Em última instância, a função `main` conecta as partes supracitadas do nosso simulador de MT.Ou seja, recebe como argumentos os arquivos de entrada e saída, instancia a MT, faz a simulação com as cadeias a serem testadas e gera os resultados.

Qualidade da solução

A solução elaborada para codificação de um simulador universal para Máquinas de Turing determinísticas se adequa à sua definição formal, pois reflete corretamente a quíntupla dessas máquinas (conjunto de estados, alfabeto finito da fita [que pode ter uma extenção com os símbolos de marcação], transições, estado inicial e estado de aceitação). A classe foi testada com sucesso para simular tanto Máquinas de Turing determinísticas quanto Autômatos Linearmente Limitados (ALL), que processam linguagens do tipo 1, segundo a hierarquia de Chomsky. O simulador interpreta corretamente as transições de estado, os movimentos da cabeça de leitura/escrita e as alterações na fita, replicando o comportamento esperado de uma MT.

A implementação foi desenvolvida em Python devido à sua simplicidade sintática, vasta biblioteca de suporte e facilidade de leitura. Recursos como list comprehensions foram utilizados para otimizar o código. No entanto, embora Python seja excelente para prototipagem e implementações de pequeno porte (como autômatos com até 10 estados), ele apresenta limitações em escalabilidade e desempenho. Em aplicações mais complexas ou com grandes volumes de dados, sua menor eficiência e maior uso de memória, característicos de uma linguagem interpretada, podem ser fatores limitantes.

Sobre a estrutura de dados principal que representa as transições da MT, uma estruturas de dados dinâmica, o dicionário aninhado facilita a implementação da função de transição, permitindo que cada estado armazene suas regras de transição de forma organizada. A matriz de adjacência poderia ser uma alternativa eficiente em termos de acesso, na qual, ao converter os estados e os símbolos para índices inteiros, seria possível mapeá-los em arrays ou listas, sendo as linhas representativas dos estados, as colunas dos símbolos do alfabeto da fita e cada célula conteria o próximo estado, o símbolo a

ser escrito e a direção. Entretanto, poderia-se facilmente enfrentar o desperdício de memória em máquinas esparsas (muitos estados ou símbolos sem transição válida), além de ser uma forma de representação que pode apresentar dificuldade de interpretação, devido às conversões de símbolos e estados para inteiros. Já o uso de uma lista de adjacência seria uma forma eficiente de armazenar máquinas com transições esparsas. Em vez de usar uma matriz completa, seriam armazenadas apenas as transições existentes para cada estado e símbolo. Apesar de economizar espaço, é uma alternativa menos eficiente para busca, já que para encontrar a transição de um estado/símbolo, é necessário iterar sobre a lista de pares associados ao estado. Isso se torna menos eficiente que o dicionário aninhado principalmente quando o número de transições por estado é maior. Com uma limitação no número de estados a no máximo 10, a questão de armazenamento não se torna muito preocupante e a utilização do dicionário aninhado aparenta o cenário ideal, ainda mais devido ao rápido acesso determinístico: Como a máquina é determinística, sempre há apenas uma transição por estado/símbolo, e o dicionário lida bem com isso.

Por fim, a solução realiza várias validações ao longo do código como número de estados, número máximo de símbolos terminais, número de transições previstas, número de cadeias permitidas para testar como parte da entrada, se uma transição prevista parte e tem destino em um estado previamente definido, tamanho da cadeia de teste e entre outras. Ao violar essas transições há um tratamento de erro que finaliza a execução do código para aquela determinada entrada.

Eficiência da solução (tempo e espaço)

A solução apresentada possui, para a validação de cadeia, uma complexidade temporal de $O(|w|)$ para MT determinística, em que $|w|$ é o tamanho da cadeia de entrada e cada transição é executada em tempo constante $O(1)$ devido à eficiência do dicionário. Apresenta uma complexidade espacial de $O(n * |\Sigma|)$ para n estados e $|\Sigma|$ símbolos na fita. Considerando a estrutura da MT, sua construção é $O(n + m + |\Sigma|)$ tanto em tempo quanto espaço, sendo n o número de estados, m o número de transições e Σ o alfabeto de símbolos. O uso de dicionários para representar as transições permite um acesso rápido ($O(1)$) tanto para buscar quanto para inserir transições, o que é uma grande vantagem em termos de eficiência de tempo, mas cada transição é armazenada em um dicionário, e dicionários em Python ocupam mais memória do que outras estruturas mais simples (como listas ou matrizes).

Testes

Afim de testar a performance e corretude da codificação feita foram elaborados oito testes usando MT's de definições distintas para arquivos de entrada .txt que se encontram disponíveis no repositório do GitHub, mas segue abaixo a descrição de quatro desses testes que contemplam os requisitos previstos no trabalho:

`Entrada.txt`

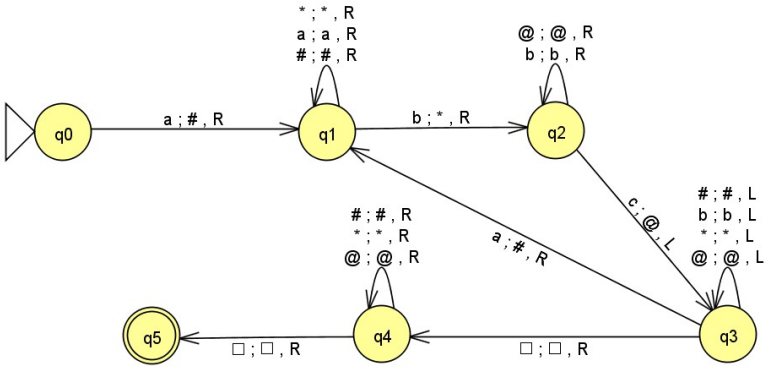

Figura 1: MT-1 definida pela Entrada.txt

Para essa primeira entrada foi usada uma MT conforme a Figura 1 que aceita cadeias definidas pela linguagem $a^n b^n c^n, n > 0$. A ideia é que ao ler um 'a' a máquina o substitua por '#' e procure um 'b' para substituir por '*' e um 'c' para substituir por '@' e volta ao início para refazer o ciclo. Se não encontra mais nenhum 'a', então checa se todos os 'b' e 'c' também foram substituídos para aceitar a cadeia.

As cadeias submetidas ao código proposto e suas respectivas saídas se encontram na Tabela 1.

Cadeia	JFLAP	Saída
abbcca	Reject	rejeita
aabbcc	Accept	aceita
bac	Reject	rejeita
aaabbbcccc	Reject	rejeita
-	Reject	rejeita
abcabc	Reject	rejeita
abc	Accept	aceita
abcc	Reject	rejeita
c	Reject	rejeita
aaabbbbccc	Reject	rejeita

Tabela 1: Tabela que mostra as cadeias submetidas, o resultado do processamento dessa cadeia, usando a mesma MT-1, no JFLAP e a saída produzida pelo código do trabalho.

`Entrada1.txt`

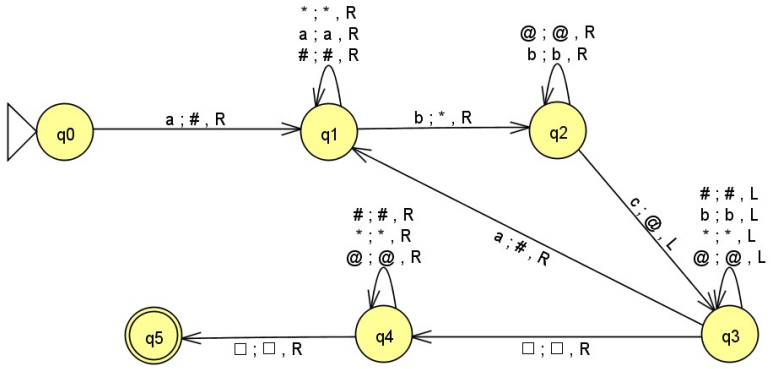

Figura 2: MT-2 definida pela Entrada1.txt

Na segunda entrada foi usada uma MT conforme a Figura 2 que aceita cadeias no formato $a^{2n}, n >= 0$, ou seja, cadeias de comprimento par de a's fazem a máquina parar no estado de aceitação, mas para cadeias de comprimento ímpar ela roda para infinitamente.

As cadeias testadas com o nosso código e suas respectivas saídas se encontram na Tabela 2

Cadeia	JFLAP	Saída
aa	Accept	aceita
a	Reject	rejeita
aaa	Reject	rejeita
aaaa	Accept	aceita
-	Accept	aceita
aaaaa	Reject	rejeita
aaaaaa	Accept	aceita
aaaaaaa	Reject	rejeita
aaaaaaaa	Accept	aceita
b	Reject	rejeita

Tabela 2: Tabela que mostra as cadeias submetidas, o resultado do processamento dessa cadeia, usando a mesma MT-2, no JFLAP e a saída produzida pelo código do trabalho.

`Entrada4.txt`

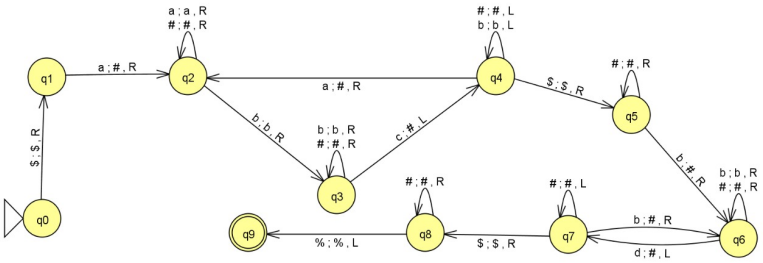

Figura 3: MT-5 (ALL) definida pela Entrada4.txt

Na quinta entrada foi usado um ALL conforme a Figura 3 que aceita cadeias da linguagem $a^n b^m c^n d^m, n, m > 0$. Os símbolos '$' e '%' são os delimitadores iniciais e finais da cadeia na fita, respectivamente.

As cadeias testadas com o nosso código e suas respectivas saídas se encontram na Tabela 3

Cadeia	JFLAP	Saída
$abbcdd%	Accept	aceita
$abcd%	Accept	aceita
$aabccd%	Accept	aceita
$a%	Reject	rejeita
$b%	Reject	rejeita
$c%	Reject	rejeita
$d%	Reject	rejeita
$%	Reject	rejeita
$aaabcccd%	Accept	aceita
$aabbcccdd%	Reject	rejeita

Tabela 3: Tabela que mostra as cadeias submetidas, o resultado do processamento dessa cadeia, usando o mesmo ALL (MT-5), no JFLAP e a saída produzida pelo código do trabalho.

`Entrada7.txt`

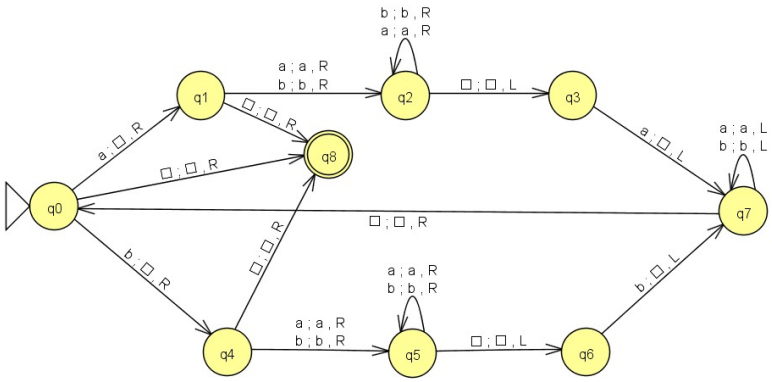

Figura 4: MT-8 definida pela Entrada7.txt

Na oitava entrada foi usada uma MT conforme a Figura 4 que processa a linguagem dos palíndromos sobre o alfabeto $\Sigma = \{a, b\}$.

As cadeias testadas com o nosso código e suas respectivas saídas se encontram na Tabela 4

Cadeia	JFLAP	Saída
-	Accept	aceita
a	Accept	aceita
aa	Accept	aceita
aba	Accept	aceita
abba	Accept	aceita
bab	Accept	aceita
ab	Reject	rejeita
aab	Reject	rejeita
abb	Reject	rejeita
abab	Reject	rejeita

Tabela 4: Tabela que mostra as cadeias submetidas, o resultado do processamento dessa cadeia, usando a mesma MT-8, no JFLAP e a saída produzida pelo código do trabalho.