Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação

Aluno: Lucas Fidelis Pereira - NUSP: 12625628

Disciplina: Teoria da Computação - SCC5832

Professor: João Luís Garcia Rosa

Relatório do Trabalho

O trabalho consiste em um simulador de autômatos finitos. A princípio é realizada a leitura das variáveis necessárias e requisitadas na especificação do trabalho. Sendo elas número de estados, número de terminais, os terminais, o número de estados iniciais, transições, estados de aceitação e entrada. O código fonte está disponível na pasta "*src*" nos arquivos *FiniteAutomaton.java* e *State.java*. Os demais arquivos da pasta são obtidos ao compilar e gerar o arquivo executável ".jar".

```
public class State {
    private ArrayList<Map> transitions;
    private boolean acceptance;
    private int label;
    private String paths;
```

Figura 1: Classe *State.java* e atributos

O trabalho possui duas classes Java a *FiniteAutomaton.java* e *State.java*, a primeira classe consiste na função *main* que é a função executável do trabalho. Nessa classe são realizadas as leituras e a "montagem" do autómato. A classe *State.java* (representada na Figura 1) consiste na estrutura de um estado, no trabalho em questão o estado possui uma lista das possíveis transições do estado em questão, uma variável lógica que define se o estado é ou não um estado de aceitação, um valor inteiro para representar o rótulo do estado (valor de 1 a n) e uma *string* com todas as chaves presentes nas transições do mesmo (essa última adicionada para facilitar as comparações).

```
//Classe privada para mapear o objeto Map
private class Map{
    private State state;
    private String key;

    public Map(State state, String key){
        this.state = state;
        this.key = key;
    }

    public String getKey(){
        return key;
    }

    public State getState(){
        return state;
    }
}
```

Figura 2: Classe *Map* e atributos

O autômato é estruturado de modo que percorrer o mesmo se assemelha a percorrer uma árvore e cada estado (nó) aponta para os possíveis caminhos a partir do mesmo. O que define o caminho a ser percorrido é a produção inserida como entrada para ser testada. Para simular essa estrutura foi criado um objeto chamado de *Map* como uma classe interna do objeto *State*. A os objetos do tipo *Map* possuem dois atributos: o estado e a chave necessária para realizar a transição para tal estado. A estrutura da classe é apresentada na figura 2.

```
//Estado "inicial"
State startState = new State(-1);
//Adicionando o estado inicial ao falso inicio
for(int i = 0; i < starter; i++)
    startState.addTransition(".", q[i]);
```

Figura 3: Declaração do estado “-1” e adição das transições para os estados iniciais definidos pelo usuário.

Para permitir que um autómato possua mais de um estado inicial é utilizado um novo estado inicial que sempre precede os estados iniciais definidos pelo usuário. Esse estado é rotulado como “-1” e sempre aponta para os estados iniciais indicado pelo usuário. Para percorrer o caminho de “-1” até o estado inicial uma nova chave representada por “.” é inserida no início de todas as produções. A definição do estado “-1” é apresentada na Figura 3.

```
public State goTo(String input){
    State next = null;
    for (Map transition: transitions ){
        if(transition.getKey().equals(input.substring(0,1))){
            if(input.length() <= 1){
                if(transition.getState().isAcceptance()){
                    next = transition.getState();
                }
            }else if(verifyNext(transition.getState(), input.substring(1))){
                next = transition.getState();
            }else if(next == null){
                next = transition.getState();
            }
        }
    }

    return next;
}
```

Figura 4: Função *goTo* que indica o próximo estado a ser percorrido.

Para percorrer o autômato todo estado possui a função *goTo* (Figura 4) que indica, com base na produção de teste entrada, qual transição deve ser realizada. A função recebe a entrada de testes e percorre a lista de transições possíveis para encontrar a adequada a ser realizada, dando prioridade a estados terminais caso a entrada possua tamanho 1, caso contrário é realizada uma busca nos estados seguintes para verificar estados “promissores” (estados ou sequência de estados que possuem transições com as chaves indicadas na entrada). Caso não encontre um estado adequado retornará nulo.

A solução proposta não foi testada em relação a eficiência em termos de espaço e tempo. No entanto, como a solução proposta foi desenvolvida em java é importante ressaltar que a máquina virtual do Java (JVM) faz o controle em relação ao custo de memória. Em relação ao tempo, a solução proposta utiliza de vários laços de repetição para percorrer todos os caminhos disponíveis em cada estado, mas devido às limitações de entrada já impostas na especificação do trabalho não ocorreram problemas relacionados a tempo durante a execução dos testes.