Teoria da computação
Universidade Federal de Santa Catarina - UFSC
Relatório do trabalho 1

Luiz Carlos Ferreira Jr - 13100761
Joaquim B. Belo - 14100838

1

Resumo. *Este relatório descreve detalhes referentes ao primeiro trabalho da cadeira de Teoria da Computação no primeiro semestre de 2016 do curso de ciências da computação na Universidade Federal de Santa Catarina, ministrado pela prof. Jerusa Marchi. Este trabalho se refere a uma implementação de um programa que realiza a transformação de automatos finitos NÃO DETERMINISTICOS em automatos finitos DETERMINISTICOS.*
Palavras-chave*: computação, determinização, automatos.*

1. Introdução

O trabalho tem como objetivo o desenvolvimento (implementação) dos algoritmos básicos para tratamento de linguagens regulares. Trata-se da implementação das transformações de mecanismos não deterministicos em mecanismos determinísticos. Por opção nosssa resolvemos adicionar funcionalidades extras como o teste e execução da maquina.

Todas as entradas e saidas (Até mesmo as configurações) são via arquivos. Aquivos que possuem a extenção ".in"são entradas e ".out"são gerados pelo programa e são onde ficam as saidas. Existem saidas verbosas no programa, mas podem ser ignoradas utilizando apenas os arquivos de entrada e saida.

Todo o referencial teorico utilizado é referente so visto em sala e no livro texto [Sipser 2006]

2. O programa

O programa em si é de facil uso, ele determiniza AFND, testa AFD e AFND, e futuramente ele irá minimizar automatos. O mesmo funciona utilizando um confunto de arquivos de entrada e saida que serão detalhados a seguir.

Teste1AfndAfn.in Input machine file Teste1AfndAfn.out Determinized output machine file Teste.in List input words to test machine Teste.out Test output Config Configuration file

2.1. Arquivos de entrada:

O programa possui os seguintes arquivos de entrada:

- Teste1AfndAfn.in
- Teste.in

Onde, Teste1AfndAfn.in deve ser onde a maquina de entrada deve ser descrita em forma de tabela de transição e Teste.in deve ser onde as entradas devem ser listadas para serem usadas como entradas para a maquina, existe um exemplo nos arquivos.

2.2. Arquivos de saída:

O programa possui os seguintes arquivos de saída:

- Teste1AfndAfn.out
- Teste.out

Estes arquivos são gerados automaticamente pelo programa quando ele tiver a necessidade de utiliza-los para colocar seus outputs. No caso do Teste1AfnsAfn.out, quando uma maquina é determinizada ela é escrita neste arquivo em forma de tabela de transição. Para o arquivo Teste.out, ele lista todas as entradas existentes em Teste.in e coloca ao lado se a palavra pertence a linguagem que o automato reconhece ou não.

2.3. Configuração:

Config é o arquivo responsável pela configuração do programa, nele contem alguns parametros que mudam algumas coisas no funcionamento do programa. os parametros são:

EpsilonIdentifier: Usado para definir qual é o simbolodo na tabela de transição que representa as transições por epsilon. Por padrão ele já possui "&"como seu valor.

FinalStateIdentifier: Usado para definir qual é o simbolo que identifica um estado final na tabela de tansição, por padrão ele já possui "*"como seu valor.

InitialStateIdentifier: Um dos dois unicos parametros que podem conter mais de um caractér como seu valor, usado para definir um estado inicial. Por padrão ele possui "$->$"como seu valor.

FirstTableNullSlotId: Usado apenas para ser um slot nulo na primeira posição da tabela, por padrão ele é "#".

NotTransitionIdentifier: Outro parametro que pode ter mais de um caractér como seu valor, define uma "Não-transição", ou seja, simbolo que define que o estado não transita por uma certa entrada. Por padrão possui "$--$"como seu valor.

OutTableColumnLength: Valor numérico, representa o espaço que deve ser usado para tabular a tabela de transição da maquina de saida, por padão possui 18 caractéres para cada setor da tabela.

2.4. Detalhes técnicos do programa:

O programa tem a capacidade de executar a maquina e reconhecer linguagens regulares, uma caracteristica importante do software é que ele consegue simular o não determinismo a partir de uma estrutura de dados chamada "multimap". O programa foi implementado em c++, utilizando o standard definido em 2011 conhecido como c++11 onde ambos alunos que desenvolveram em conjunto possuem maior afinidade. Futuramente tentaremos implementar threads para ser mais fiel ao não determinismo.

Um automato não deterministico possui duas caracteristicas que o torna diferente do deterministico:

- Epsilon transições
- Transições de um mesmo estado por uma mesma estrada para estados diferentes

Partindo do principio que um automato finito é um tipo de grafo orientado, podemos definilo como tal onde cada vertice é um estado e cada transição é uma aresta. Um grafo é na verdade dois conjuntos, um sendo o conjunto de vertices e outro o de arestas. Um vétice possui um conjunto de estados a qual ele está conectado, um automato também possui tais caracteristicas, tanto que, pode ser definido como uma função $(estadoAtualX simbolo) -> proxEstado$ chegamos as seguintes soluções para implementação:

Conjunto extra para epsilon transições

O programa possui uma abstração de automato finito em sua implementação, nossas trasições são feitas usando mapeamento, onde cada estado possui um atributo que armazena esses mapeamentos(será tratado em breve), mas para as e-transições usamos um atributo exclusivo pois não se tem um simbolo por qual transitar. Tal atributo é uma instancia da estrutura std::set(conjunto do c++) em que todos os estados que esse estado atige por epsilon são adicionados. Ao transitar a maquina primeiro realiza os ramos de computação inciando pelos estados contidos no conjunto citado.[O algoritmo de transição será mostrado na seção correta deste relatório]

Multimap: Para multiplas transições por um mesmo simbolo.

Multimap foi a saida que achamos para mapear um estado a mutiplos estados utilizando uma mesma chave(simbolo), Nosso automato não segue a risca a função $(estadoAtualX simbolo) -> proxEstado$ mas sim algo parecido $(estadoAtualX simbolo) -> (ConjDeEstados)$ onde conseguimos mapear um estado a mutiplos por um unico simbolo.

3. Algoritimos usados

O programa em sí é extenso demais para se detalhar passo a passo de seus algoritimos aqui. Esboçaremos apenas o que achamos ser de suma importância e relevância para os objetivos do relatório e da avaliação.

3.1. Determinização de automatos

OBS : Levando em consideração que cada estado da maquina determinizada possua um ou mais estados que o compoem, podemos ve-los como cada estado na AFD fosse um conjunto de 1 ou mais estados da AFND. Abstraimos tal transformação no algoritmo, porém será avisado em forma de cometário.

Parte1

```
1
2  Automato determinizar(Automato AFND){
3
4      Estado antigoEstadoInicial = AFND.pegaEstadoInicial();
5
6      //pega o antigo estado inicial junto com seu epsilon fecho e
           cria um estado novo com eles
7      Estado novoEstadoInicial = antigoEstadoInicial.
           estadoMaisEpsilonFecho();
8
9      Automato AFD;
10
11     AFD.adicionaEstado(novoEstadoInicial);
```

```
12      AFD.atualizarEstadoInicial(novoEstadoInicial);
13
14      //Se pelo menos um dos estados usados para fazer "
            novoEstadoInicial"
15      //ent o ele ser  um estado final
16      se ( ContemEstadoFinal(novoEstadoInicial) ){
17              AFD.addNoConjDeEstadosFinais(novoEstadoInicial);
18          }
19
20      //Agora adicionamos os proximos estados recursivamente
21      //Adicionando os estados na AFD conforme vamos alcan ando
22      atingeProximoEstado(AFND, AFD, novoEstadoInicial);
23
24      retorna AFD;
25  }
```

Parte2

```
1   void atingeProximoEstado(Automato AFND, Automato AFD, Estado
        estadoAtual)
2   {
3       Simbolo[] sigma = AFND.pegaVetorSigma();//vetor com o
            alfabeto de entrada
4
5       para cada estado em estadoAtual  f a  a{
6
7           para cada simbolo em sigma  f a  a{
8
9               Conjunto<Estado> estadosAtingidos = estado.
                    pegaAtingidosPor(simbolo);
10
11              //abstraindo novamente a  transforma  o de conj de
12              //Estados para um unico estado.
13              Estado proximoEstado = estado.epsilonFecho() +
                    estadosAtingidos;
14
15              se ( AFD.estadoN oExistente(proximoEstado) ){
16                  AFD.adicionaEstado(proximoEstado);
17                  se (ContemEstadoFinal(proximoEstado)){
18                      AFD.addNoConjDeEstadosFinais(proximoEstado);
19                  }
20                  atingeProximoEstado(AFND, AFN, proximoEstado);
21              }
22
23                              AFD.addTransi  o(simbolo, estadoAtual,
                                    proximoEstado);
24              }
25          }
26      }
27  }
```

3.2. Transições e validação de palavras

Metodo recursivo que retorna os estados das folhas dos ramos de computação para se validar a palavra ou não.

Parte do automato:

```
1  bool Automato::validaPalavra(Entrada palavra){
2
3      //palavra    transformada em pilha para simplificar o consumo
4      //durante a computa  o
5      pillha<simbolo> pilha = palavra.transformaEmPilha();
6
7      conjunto<Estado> estadosAtingidos = estadoInicial.
           realizaTransi es(pilha);
8
9      para cada estado em estadosAtingidos fa a{
10         se (estado. final ()){
11             retorna true;//pelo menos um estado final atingido,
                   palavra aceita
12         }
13     }
14
15     retorna false;//nenhum estado final atingido, nega a palavra
16 }
```

Parte do estado:

```
1
2  conjunto<Estado> Estado::realizaTransi es(pilha<simbolo>
       palavra){
3      conjunto<Estados> estadosAtingidos;
4
5      //conjunto epsilon    o conjunto de estados atingidos por
           epsilon
6      //a partir de um estado
7      se (conjuntoEpsilon.tamanho() > 0){
8          para cada estado em conjuntoEpsilon fa a{
9              conjunto<Estado> resultado = estado.
                   realizaTransi es(palavra);
10             estadosAtingidos.uni o( resultado );
11         }
12     }
13
14     se ( palavra.estaVazia() ){
15         estadosAtingidos.insere(esteEstado);
16         return estadosAtingidos;
17     }else{
18
19         Simbolo s =  palavra.pegaERetiraDoTopo();
20
21         //transi  es    um objeto do tipo multimap, estrutura
               usada
22         //para representar as transi  es por algum simbolo.
23         se (transi  es.tamanho() > 0){
24             para cada estado em transi  es.estadosMapedosPor(s)
                   fa a {
25                 conjunto<Estado> resultado = estado.
                       realizaTransi es(palavra);
26                 estadosAtingidos.uni o( resultado );
27             }
28         }
29      }
```

```
30
31      retorna estadosAtingidos;
32  }
```

Referências

Sipser, M. (2006). *Introduction to the computing theory*. Thomson Course Technology, 2nd edition.