Fundamentos Teóricos da Computação: AFD e operações

Rafael Nepomuceno[1], **Kassio Rodrigues**[1]

[1]Instituto de Ciências Exatas e Aplicadas – Universidade Federal de Ouro Preto (UFOP)
João Monlevade – MG – Brasil

Resumo. *Este relatório descreve a implementação do programa proposto para o trabalho prático da disciplina Fundamentos Teóricos da Computação. O objetivo principal deste trabalho foi implementar um programa em linguagem de programação C capaz de cobrir um conjunto de funcionalidades que permitam a manipulação de linguagens formais através dos Autômatos Finitos Determinísticos (AFDs). O programa elaborado no desenvolvimento deste trabalho foi capaz de atingir boa parte dos requisitos especificados, incluindo funcionalidades como leitura e escrita de AFDs, assim como sub-programas responsáveis por determinar o reconhecimento de palavras e gerar o complemento e a minimização de AFDs.*

Contents

1. Introdução

Neste trabalho apresentamos a implementação do programa proposto, que tem como objetivo principal cobrir um conjunto de funcionalidades dentro do contexto da manipulação de linguagens formais através de Autômatos Finitos Determinísticos. Atendendo aos requisitos da proposta para o trabalho, o programa implementado foi em linguagem C e se atém às especificações definidas para as funcionalidades. O comportamento das funcionalidades implementadas, por sua vez, foi baseado nas definições propostas no livro *Introdução aos fundamentos da computação: linguagens e máquinas* [Vieira 2006].

Para documentar o processo de implementação este relatório é estruturado da seguinte forma: o capítulo [2] apresenta as estratégias utilizadas na implementação do programa, incluindo os resultados obtidos; o capítulo [3] traz o guia de uso do programa e, por fim, o capítulo [4] relata as considerações finais.

2. Solução do problema

Neste capítulo são apresentadas as estratégias utilizadas para a implementação dos requisitos funcionais propostos para o programa. É importante ressaltar que das 5 funcionalidades propostas, apenas a funcionalidade 3 (interseção e união) não foi implementada devida a falta de tempo e desistência de três membros do grupo.

2.1. Representação do AFD

Como o objetivo principal do programa desenvolvido é cobrir um conjunto de funcionalidades para manipulação de linguagens formais através de AFDs, é muito importante que a estrutura dos AFDs no programa seja consistente e de fácil manipulação.

Dada esta importância, optou-se por criar uma biblioteca específica definir a estrutura padrão para os AFDs. Inspirada no encapsulamento de classes, a biblioteca centraliza apenas os métodos de acesso e alteração das propriedades de um automato finito determinístico.

Por fim, a estrutura de dados que representa os AFDs no programa é baseada na definição formal $(E, \Sigma, \delta, i, F)$ adotada por [Vieira 2006]. A nível de implementação cada estado tem sua própria estrutura indicando seu nome, se é final e/ou inicial, ou ainda se é estado de erro. Desta forma, a representação a nível de implementação se resume da seguinte maneira:

E : vetor de estados;

Σ : vetor de símbolos, representado o alfabeto de entrada;

δ : matriz de transições;

F : quantidade de estados Finais;

Note que nesta estrutura, o objeto estado é que vai armazenar a referência se é inicial e/ou final. Além disso os atributos denotados como vetores torna o acesso simples. Por exemplo, é possível acessar uma transição diretamente com índices do estado e do simbolo da seguinte forma: $\delta[e][s] : e'$, onde e e e' são índices para um estado $\in E$, s é o índice para um símbolo $\in \Sigma$.

2.2. Leitura de um AFD de entrada

Outro requisito fundamental é permitir que o usuário forneça o AFD que deve ser considerado na execução das funcionalidades propostas. O método de leitura implementado é específico para arquivos `.txt` contendo a definição do AFD. É importante ressaltar que o método implementado só garante a leitura correta para arquivos com a estrutura que foi definida na proposta do trabalho prático.

2.3. Funcionalidades

Esta seção apresenta, para cada funcionalidade proposta, os objetivos considerados e as abordagens adotadas na implementação.

2.3.1. Visualização

O objetivo desta funcionalidade é que, a partir de um AFD de entrada, seja gerado o mesmo AFD porém no formato de saída `.dot` pronto para a visualização com o GraphViz.

O GraphViz possui uma notação específica para produzir representações visuais de grafos. E como vimos ao longo da disciplina, é uma prática comum representar AFDs por meio de grafos direcionados.

A Figura [1] mostra um arquivo `.dot` gerado pelo programa desenvolvido.

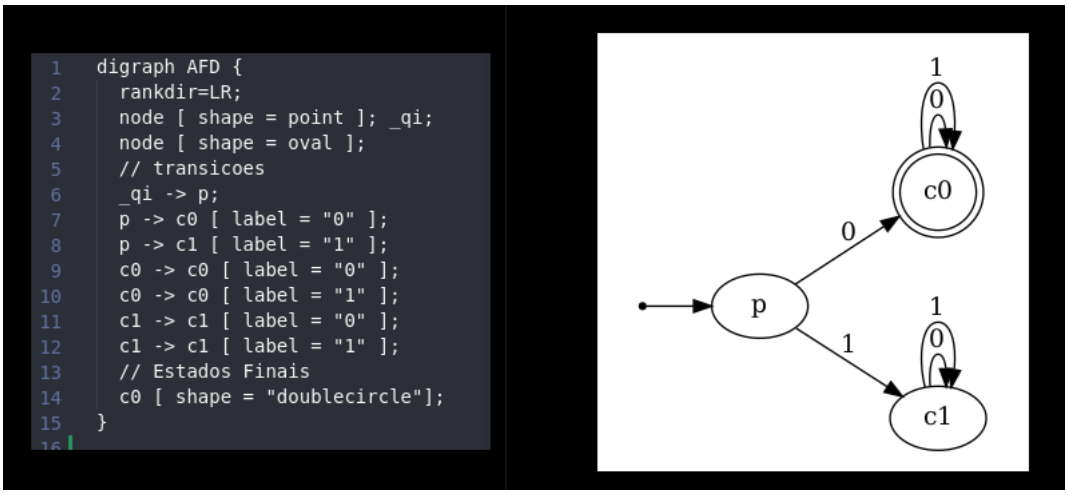

Figure 1. À esquerda o arquivo ".dot" gerado a partir da funcionalidade Visualizar. À direita o grafo gerado a partir deste ".dot".

2.3.2. Complemento

O objetivo desta funcionalidade é que, dado um AFD de entrada, seja gerado um arquivo `.txt` de um AFD' de saída que reconheça o complemento da linguagem do automato de entrada.

A implementação desta funcionalidade se atém a definição do complemento apresentada por [Vieira 2006]. Seja um AFD M, um AFD M' que aceita $\overline{L(M)}$ é dado por: $M' = (E, \Sigma, \delta, i, E - F)$, ou seja, os estados finais em M' são os não finais em M.

A Figura [2] mostra o resultado obtido partir esta funcionalidade no programa. Nesta figura o `.txt` gerado foi submetido ao método Visualizar (2.3.1) para ilustrar o resultado.

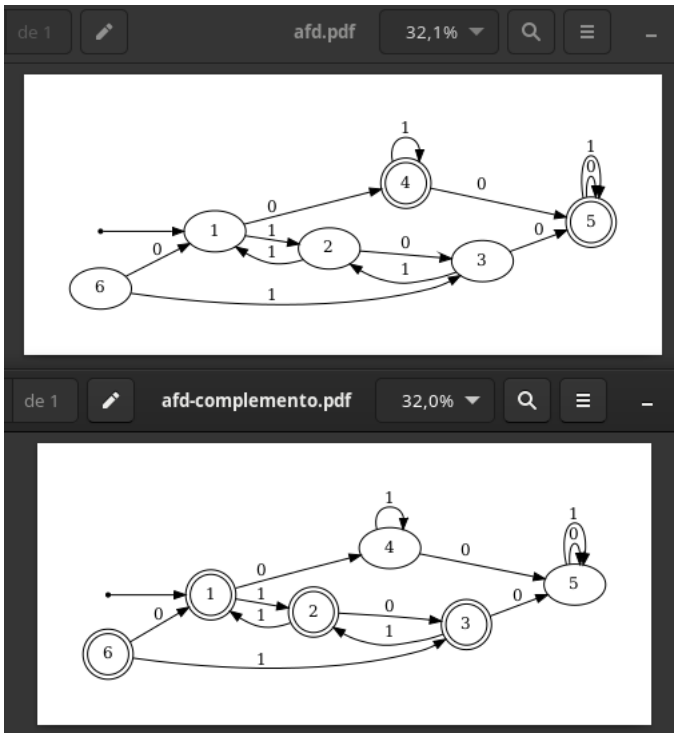

Figure 2. O grafo de cima mostra o afd original, o grafo de baixo mostra o resultado gerado pela funcionalidade Complemento.

2.3.3. Minimização

O objetivo desta funcionalidade é que, dado um AFD de entrada, seja gerado um arquivo `.txt` que seja mínimo, e que reconheça a mesma linguagem do autômato de entrada.

A implementação desta funcionalidade foi inspirada no algoritmo de minimização, da Figura [3], apresentado por [Vieira 2006]. A nível de implementação, optou-se por criar uma biblioteca específica para manipular o conjunto de classes de equivalência, assim como suas respectivas partições. A partir desta biblioteca foi possível representar o conjunto de classes de equivalência da seguinte forma:

$Particao$: contém um vetor de estados;

$VetorParticoes$: contém um vetor de $Particao$, representa uma classe de equivalência;

$VetorEquivalencias$: contém um vetor de $VetorParticoes$, representa o conjunto de classes de equivalência. Note que, para cada classe de equivalência i pode ser acessada por $VetorEquivalencias->VetorParticoes[i]$;

Observe que $VetorEquivalencias$ gera apenas uma camada de abstração a mais, e poderia ser dispensada utilizando apenas $VetorParticoes$. Mas com o intuito de melhorar a legibilidade do código esta abordagem não foi adotada.

A Figura [4] mostra o resultado obtido partir da minimização feita pelo programa. Nesta figura o arquivo (`.txt`) do AFD minimizado foi submetido ao método Visualizar (2.3.1) para ilustrar o resultado.

Entrada: um AFD $M = (E, \Sigma, \delta, i, F)$.
Saída: um AFD mínimo equivalente a M.

elimine de M todo estado não alcançável;
se $F = \emptyset$ **então**
    retorne $(\{i\}, \Sigma, \delta', i, \emptyset)$, sendo $\delta'(i, a) = i$ para todo $a \in \Sigma$
senãose $E - F = \emptyset$ **então**
    retorne $(\{i\}, \Sigma, \delta', i, \{i\})$, sendo $\delta'(i, a) = i$ para todo $a \in \Sigma$
fimse;
$n \leftarrow 0$; $S_0 \leftarrow \{E - F, F\}$;
repita
    $n \leftarrow n + 1$; $S_n \leftarrow \emptyset$;
    para cada $X \in S_{n-1}$ **faça** /* refina a classe X de $\approx_{n-1}$ */
        repita
            selecione um estado $e \in X$;
            /* $[\delta(e, a)]_{n-1}$ é o conjunto que contém $\delta(e, a)$ em S_{n-1}*/
            $Y \leftarrow \{e' \in X \mid \delta(e', a) \in [\delta(e, a)]_{n-1}$ para todo $a \in \Sigma\}$;
            $X \leftarrow X - Y$;
            $S_n \leftarrow S_n \cup \{Y\}$
        até $X = \emptyset$
    fimpara;
até $S_n = S_{n-1}$;
$i' \leftarrow$ conjunto em S_n que contém i;
$F' \leftarrow \{X \in S_n \mid X \subseteq F\}$;
para cada $X \in S_n$ e $a \in \Sigma$:
    $\delta'(X, a) =$ conjunto em S_n que contém $\delta(e, a)$, para qualquer $e \in X$;
retorne $(S_n, \Sigma, \delta', i', F')$

Figure 3. Pseudo-código do algoritmo de minimização de AFDs, por [Vieira 2006].

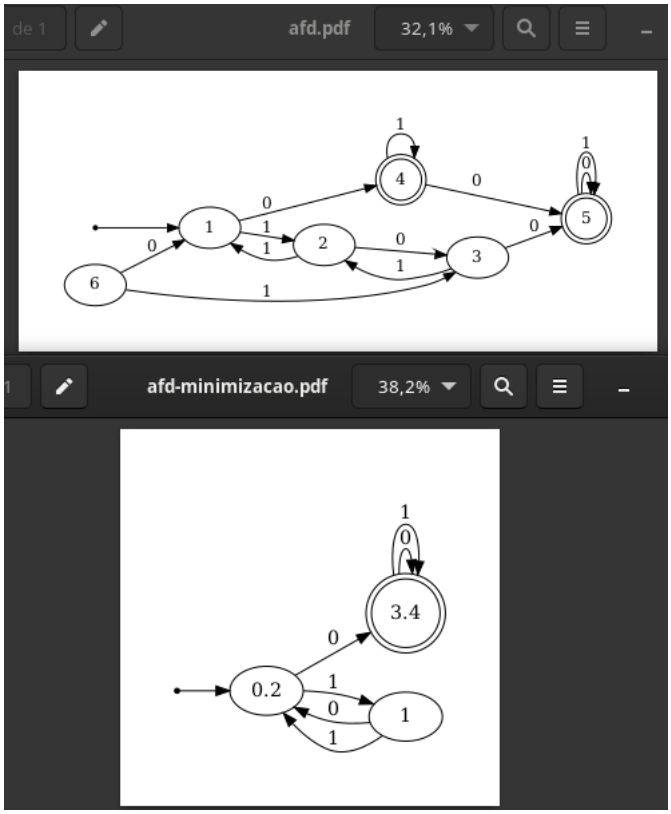

Figure 4. O grafo de cima mostra o afd original, o grafo de baixo mostra o resultado gerado pela funcionalidade de Minimização.

2.3.4. Reconhecer palavra

O objetivo desta funcionalidade é que, dado um AFD de entrada, e um arquivo contendo um conjunto de palavras (1 palavra por linha), seja gerado um arquivo de saída especificando quais as palavras foram reconhecidas, e quais não foram reconhecidas pelo AFD de entrada.

Como mencionado na seção 2.1 a estrutura definida para o AFD simplificou o processo de acessar as propriedades do autômato, incluindo as transições, portanto bastou efetuar a transição de estados a cada símbolo lido e, ao final da palavra atribuir o resultado no arquivo de saída, na respectiva linha da palavra.

A Figura [5] mostra o resultado de uma execução da funcionalidade no programa. Neste exemplo é possível notar que a verificação para palavras vazias também foi garantida.

3. Guia de uso

A aplicação foi desenvolvida e testada em ambientes Linux, e como não foram utilizadas bibliotecas externas, não é necessário pré-requisitos adicionais para compilar o programa. Seguindo as orientações exigidas para a construção do trabalho, foi elaborado um arquivo Makefile para automatizar a compilação do programa [Newhall].

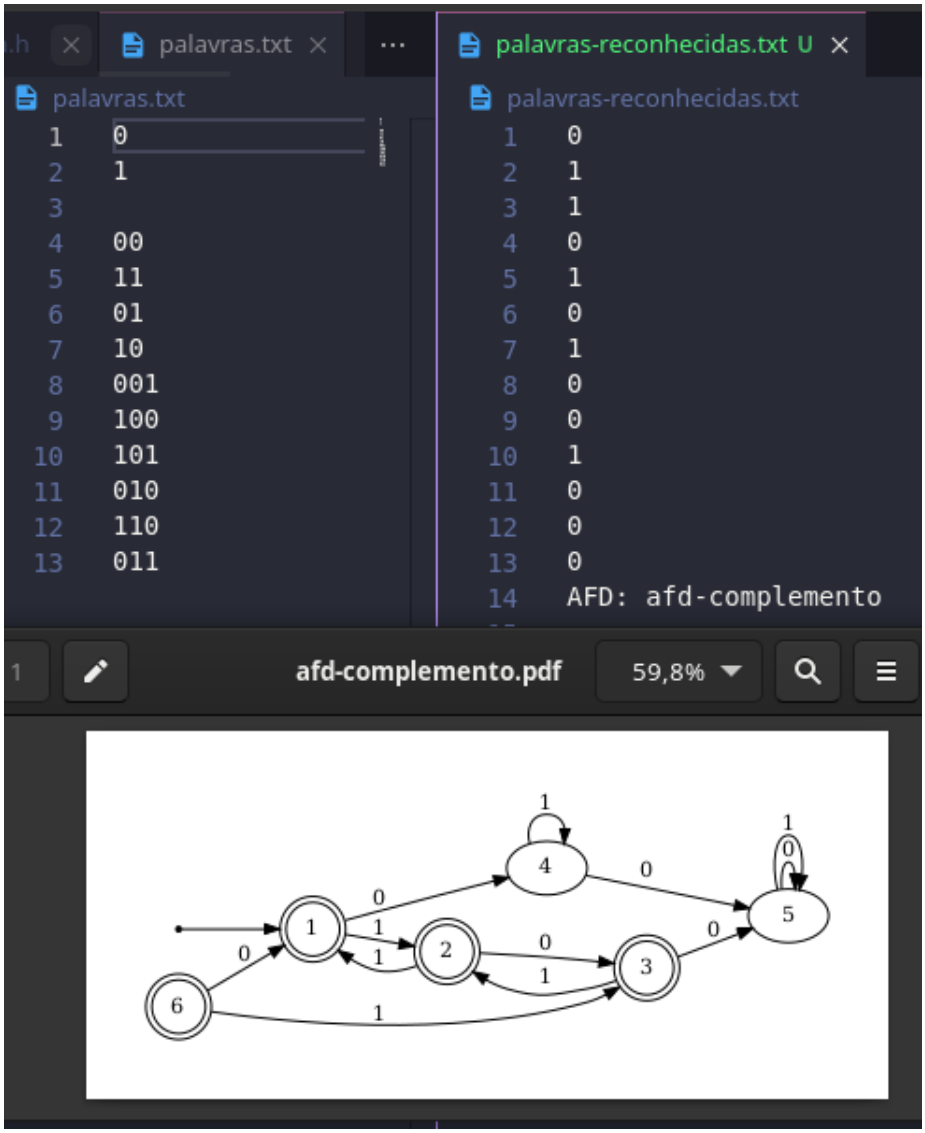

Figure 5. Resultado da funcionalidade Reconhecer Palavra para um AFD. O arquivo à direita mostra o resultado gerado a partir das palavras de entrada à esquerda.

3.1. Executando o Programa

Para compilar o programa e criar o executável `afdtool` na raiz do projeto, é necessário invocar o comando `make`. O programa é executado via linha de comando e é preciso especificar a funcionalidade desejada, juntamente com seus respectivos parâmetros de entrada.

No programa desenvolvido, as funcionalidades são executadas da seguinte maneira:

```
$ ./afdtool --dot afd.txt --output afd.dot
$ ./afdtool --complemento afd.txt --output afd1-complemento.txt
$ ./afdtool --minimizacao afd.txt --output afd1-minimizacao.txt
$ ./afdtool --reconhecer afd.txt palavras.txt \
                                --output palavras-reconhecidas.txt
```

Esta definição para os argumentos de execução mantém o formato proposto na especificação do trabalho. Uma observação é que cláusula `--dot` faz referência a funcionalidade de visualização (2.3.1).

4. Considerações Finais

O trabalho proposto engloba uma série de requisitos funcionais que exigem um conhecimento sólido em relação as propriedades dos AFDs, assim como domínio da linguagem C. Infelizmente, devido à desistência alguns membros do grupo, as funcionalidades de União e Interseção não puderam ser implementadas dentro do prazo. Mas de modo geral, o programa implementado apresenta o comportamento esperado para as funcionalidades implementadas e mantém as especificações definidas, tais como formatos e nomes de arquivos, linguagem de programação e argumentos para compilação. As estruturas utilizadas para representar os autômatos possuem boa flexibilidade para manipulação, e o algoritmo de minimização apresentou resultados consistentes. No entanto, acreditamos que é possível melhorar essas estruturas em termos de desempenho e consumo de memória.

References

[Newhall] Newhall, T. Using make and writing makefiles. `https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html`.

[Vieira 2006] Vieira, N. J. (2006). *Introdução aos fundamentos da computação: linguagens e máquinas*. Pioneira Thomson Learning.