Construção de uma aplicação para geração, determinização e minimização de AF

João Paulo Castilho[1]**,Felipe Chabatura Neto**[2]**, Leonardo Tironi**[3]**, Henrique Dalla Corte**[4]

[1]Curso de Ciência da Computação – Universidade Federal Fronteira Sul (UFFS)
Chapecó – SC – Brasil

Abstract. *This paper offers a project to implement a deterministic finite automaton (DFA) generator from regular grammars and/or tokens, as well as to determinizing and minimizing the NDFA, but without the application of the Equivalence Class. The application implementation was made using C ++ language and its libraries.*

Resumo. *Este artigo apresenta a implementação de um gerador de Autômatos Finitos Determinísticos (AFD) a partir de gramáticas regulares e/ou tokens, determinizando e minimizando o autômato finito não-determinístico (AFND) , mas sem utilizar da Classe de Equivalência. A implementação da aplicação foi feita usando linguagem C ++ e suas bibliotecas.*

1. Introdução

Este trabalho foi construído com o intuito de solucionar o problema da conversão para Autômato Finito Determinístico (AFD) de uma gramática regular e/ou tokens. Sendo pesquisado o que é e como se chega a um AFD, que pode ser visto no referencial teórico.

O tratamento das gramáticas regulares e tokens consiste em uma minimização explicada detalhadamente na próxima seção, mas neste trabalho não foi utilizada a minimização por classe de equivalência.

A tabela de transições, que também está na próxima parte do trabalho, foi implementada utilizando-se de variáveis de controle além de estruturas de dados como vetores dinâmicos tridimensionais e árvores rubro-negras, já que foi escolhida a linguagem de programação C++.

Além de utilizar as bibliotecas, para a criação de maps, vectors, entre outras funções exportadas, foram criadas muitas outras e acrescentadas variáveis que serão citadas e explicadas nas próximas seções deste artigo.

2. Referencial Teórico

O Autômato Finito Determinístico (AFD) segundo [Hopcroft et al. 2002] é uma Máquina de estados finita que aceita ou rejeita cadeias de símbolos gerando um único ramo de computação para cada cadeia de entrada, sendo que o não-determinístico aceita vários ramos para cada cadeia de entrada. AFDs reconhecem exatamente o conjunto de Linguagens Regulares que são, dentre outras coisas, úteis para a realização de Análise léxica e reconhecimento de padrões. Um AFD pode ser construído a partir de um Autômato finito não determinístico através de uma composição do conjunto das partes.

Um Autômato Finito Determinístico A é uma 5-tupla (ou quíntupla), (Q, Σ, δ, q0, F) consistindo de:

1- Um conjunto finito de símbolos de entrada chamado Alfabeto (δ)

2- Um conjunto finito de estados (Q)

3- Uma função de transição ($\delta : Q \times \Sigma \rightarrow Q$)

4- Um estado inicial ($q_0 \in Q$)

5- Um conjunto de estados de aceitação ($F \subseteq Q$)

Finito: O autômato é dito finito porque o conjunto de estados possíveis (Q) é finito.

Determinístico: O autômato é determinístico quando dado o estado atual de M, ao ler um determinado símbolo na fita de entrada existe apenas um próximo estado possível.

Minimização: É a retirada de não-terminais mortos (Não levam a nenhum não-terminal final), inalcançáveis (Que pelo não-terminal inicial e suas produções nunca será alcançado) e aplicada a classe de equivalência que não foi utilizada.

Tabela de transições é uma forma tabular de representar as transições do autômato entre os Não-Terminais pelos Terminais, tendo como valores os Não-terminais resultantes.

3. Desenvolvimento

3.1. Definições

Ao longo do projeto, foram feitas algumas definições. Elas são:

```
1 #define MAX_STATE 1123
2 typedef pair<string, bool>psb;
3 typedef vector<psb>vpsb;
4 typedef vector<string> vs;
5 typedef vector<int> vi;
6 typedef vector<vi> vii;
7 typedef vector<vii> viii;
```

Cada definição significa:
1- PSB: Pair of string-boolean. Utilizado para mapear não terminais, foi usado um pair onde a string é o nome do terminal e o bool é para saber se ele é um estado final ou não. 2- VPSB: Vector of pair of string-boolean. Um vetor utilizado para inserir os estados, sejam eles finais ou não.
3- VS: Vector of strings. Utilizado para mapear os terminais.
4- VI: Vector of integers. Utilizado durante a construção da AFND.
5- VII: Vector of vectors of integers. Utilizado durante a construção da AFND.
6- VIII: Vector of vectors of vectors of integers. Como é necessário construir um AFND, é usado um vector tridimensional, onde o AFND pode ser escrito da seguinte maneira: AFND[Estado][Terminal][Transição]. Ou seja, se ele estiver no estado x, houver um terminal y, ele transicionará para o estado z.

3.2. Main

A função main terá algumas variáveis importantes, sendo elas:

```
int states_cont;
int terms_cont;
int states_qtty;
int lim;
vpsb states_name;
vs terms_name;
map<string, int> map_states;
map<string, int> map_terms;
vi transition_cont;
char last_state_generated[MAX_STATE];
file *f;
```

Cada variável significa:
1- Um contador de estados, ele dirá quantos estados existem.
2- Um contador de terminais, ele dirá quantos terminais existem.
3- Um contador de terminais, ele dirá quantos terminais existem.
4- Limite que indica o identificador do último estado mapeado gerado a partir da gramática original.
5- Um vetor dos nomes dos estados, sejam eles finais ou não.
6- Um vetor dos nomes dos terminais.
7- Um map de string para int dos estados. Serve para mapear cada estado a um inteiro, sendo este inteiro o seu id.
8- Um map de string para int dos terminais. Serve para mapear cada terminal a um inteiro(id).
9- Um vector de inteiros para saber quantas transições tem cada não terminal.
10- Um char com tamanho definido MAX_STATE, para o último estado gerado.
11- Um ponteiro para ler o arquivo de entrada.

3.3. Funções

Para construir as funções foram criadas duas bibliotecas, uma para fazer o mapeamento de todos os símbolos da entrada e outra para verificar se o autômato gerado está determinizado ou não. Caso a resposta seja negativa, é chamada uma função para o determinizar.

Nesta seção serão detalhadas todas as funções.

3.3.1. Função *main*

Função no arquivo *main.cpp*.

Após abrir o arquivo de entrada, a *main* irá chamar a função *symbols_mape* para mapear a entrada. Para cada linha, a *symbols_mape* irá decidir se faz parte de uma gramática ou de um token. Após mapear o arquivo de entrada, o programa deverá construir o autômato finito não-determinístico, baseando-se no mapeamento feito anteriormente, e determinizá-lo.

Depois de executada a função *symbols_mape*, as dimensões do autômato serão conhecidas e portanto ele será criado como um *viii*. Logo após, será chamada a função de

create_afnd, que recebe todos os símbolos mapeados e o ponteiro para o autômato, para assim poder criá-lo efetivamente.

A função *minimize_afnd* é responsável por remover todos os estados inalcançáveis e mortos. Ela recebe um limite, para minimizar somente até os estados pertencentes a gramática. A seguir é feita a determinização, sendo invocada a função *automaton_determinize* que retira qualquer indeterminismo que o autômato possuir. Finalmente, é chamada a função *print_file* onde o autômato já determinizado é transferido para um arquivo de saída em formato *.csv* com o resultado final.

3.3.2. Função *symbols_mape*

Função no arquivo *grammar.cpp*.

É a função responsável por mapear os símbolos, tanto terminais quanto não terminais. Esta função é dividida em dois casos: um trata o caso da linha pertencer a uma gramática, o outro, dela pertencer a um *token*. No primeiro caso, a função *mape_grammar* é chamada, no segundo, a função *mape_token*.

3.3.3. Função *mape_grammar*

Função no arquivo *grammar.cpp*.

Esta função trabalha por partes, na primeira, mapeia o nome do estado pertencente a linha que está sendo tratada. Na outra, mapeia os terminais do afnd, sendo que podem existir várias possibilidades de terminais. É nesta função onde ocorre o controle do limite dos estados que pertencem a gramática, para que a função que minimiza o autômato seja otimizada.

3.3.4. Função *mape_token*

Função no arquivo *grammar.cpp*.

Esta função verifica se existe algum estado já criado, ou seja, se antes de criar algum *token*, foi gerado um estado inicial pertencente a uma gramatica. Caso a resposta seja negativa, será criado um estado inicial para compartilhar as transições do primeiro símbolo de cada *token*. Do contrário, os estados serão mapeados com seu início junto ao estado inicial.

3.3.5. Função *create_afnd*

Função no arquivo *grammar.cpp*.

É a função responsável por criar o autômato não-determinístico através do mapeamento feito anteriormente por *symbols_mape*. Como a aplicação pode receber tanto *tokens* quanto gramáticas, é utilizada uma função de criação de *tokens* caso a linha lida

seja referente a um *token*. Essa função é responsável por criar os novos estados para o reconhecimento de cada *token*. Caso a linha seja referente a uma gramática, os estados dela já estarão mapeados e, portanto, basta colocar suas respectivas transições no autômato.

A função *create_afnd* utiliza as funções *take_term_name* e *take_state_name* para buscar os nomes dos terminais e não-terminais, respectivamente.

3.3.6. Função *print_afnd*

Função no arquivo *grammar.cpp*. É a função responsável por percorrer o AFND e imprimí-lo na saída padrão.

3.3.7. Função *print_file*

Função no arquivo *gramar.cpp*.

É a função responsável por escrever em um arquivo .csv o autômato determinizado.

3.3.8. Função *minimize_afnd*

Função no arquivo *gramar.cpp*.

É a função que gerencia a minimização do autômato não-determinístico. Ela chama uma *DFS* para buscar por todos os estados alcançáveis a partir do estado inicial, assim eliminando todos os inalcançáveis. Para eliminar os que não atingem estado final, é mantido um controle para que, se naquele ramo gerado pela árvore da busca em profundidade alcançar um estado final, todos os estados pertencentes a esse ramo - antes desse estado - são marcados como vivos. Apenas os estados pertencentes a gramática precisam ser minimizados, portanto o limitante é usado para que a *DFS* só percorra o autômato até o limite da gramática.

Após a minimização, se algum terminal se tornar inválido, ou seja, nenhum estado tiver ao menos uma transição por ele, o terminal será removido do autômato.

3.3.9. Função *remove_invalid_terms*

Função no arquivo *grammar.cpp*.

Para cada estado não inalcançável, é chamada a função *terminal_verification*. Após verifica-se se a transição em sua posição é alcançável, caso não seja, ela será removida do vetor. Por conseguinte, se alguma transição foi removida, é comparado se o seu vetor está vazio, uma vez que esteja, é decrementado do contador de estados válidos que usam aquele terminal.

3.3.10. Função *terminal_verification*

Função no arquivo *grammar.cpp.*

Esta função varre o vetor de terminais de um estado inválido a procura de alguma transição não vazia daquele terminal. Caso encontre, será decrementado uma unidade do terminal que possui essa transição de um estado inválido.

3.3.11. Função *select_valids*

Função no arquivo *grammar.cpp*. Esta função é uma busca em profundidade (DFS), que é chamada inicialmente na função *minimize_afnd*. Esta busca é lançada a partir do estado inicial, para ver quais estados são alcançáveis e a partir de cada um dos estados, para saber quais deles alcançam um estado final (não são mortos). Desta forma, ficam marcados quais estados devem ser removidos no processo de minimização.

3.3.12. Função *tokens_mape*

Função no arquivo *grammar.cpp*.

Realiza o mapeamento dos tokens do arquivo criando estados para eles e os colocando no map dos estados.

3.3.13. Função *new_state_inc*

Função no arquivo *utilities.cpp*.

Esta função cria um novo nome para um estado que está sendo criado, seja devido ao mapeamento de tokens ou ao processo de determinização.

3.3.14. Função *next_simb*

Função no arquivo *grammar.cpp*. Esta função procura pela primeira ocorrência de um caractere em uma string. É utilizada durante a leitura do arquivo, para encontrar os locais onde estão os nomes dos estados e os terminais.

3.3.15. Função *take_state_name*

Função no arquivo *grammar.cpp*.

Função utilizada para extrair o nome do estado de uma linha do arquivo, previamente lida para uma string.

3.3.16. Função *take_term_name*

Função no arquivo *grammar.cpp*.

Função para colocar o nome do terminal que antecede uma produção ou não, será colocano no map do seu vector de terminais.

3.3.17. Função *create_new_state*

Função no arquivo *determinize.cpp*.

Está função cria um novo estado para os estados indeterminísticos, chamando a função *_new_state_inc*, a qual dará o nome para este novo estado, que será inserido no dicionário, mapeado, inserido no autômato, aumentado o contador de estados e seu nome é mapeado para o *states_name*.

Logo os estados que compõem a indeterminação são colocados no dicionário, sendo escrito no terminal qual estado esta sendo determinizado. É feita uma nova linha para ser colocado no autômato e na AFND. Caso alguma das transições desse novo estado tenha indeterminismo é chamada a função de *Sort* para os ordenar e *transition_determinize* para tratar este não determinismo.

3.3.18. Função *automaton_determinize*

Função no arquivo *determinize.cpp*.

Esta função é a responsável pela determinização do autômato. Ela chama a função *dictionary_initialize* que inicializará um dicionário, o qual passará por cada terminal de cada estado colocando o nome do estado e quais estados iniciais, inseridos via input o compõem.

Depois de criado o dicionário, as transições de um terminal são ordenadas e cada estado é verificado se existe alguma transição para outro estado. Caso haja e seja maior que um, há um indeterminismo, o qual será tratado na função *transition_determinize*.

3.3.19. Função *transition_determinize*

Função no arquivo *determinize.cpp*.

Função responsável por tratar os indeterminismos. Checa o dicionário para saber se este já é um indeterminismo conhecido, se não , cria um novo estado para abrigar o conjunto de transições e se este ainda for indeterminístico, o determiniza, de maneira recursiva. Caso o indeterminismo seja conhecido e já tenha sido mapeado antes, ocorre apenas a substituição do indeterminismo pelo nome do estado.

3.3.20. Função *dictionary_initialize*

Função no arquivo *determinize.cpp*.

Inicializa o dicionário com os estados já existentes no autômato e os estados que eles mapeiam, que necessariamente serão eles mesmos.

3.3.21. Função *print_dictionary*

Função no arquivo *determinize.cpp*.

Imprime o dicionário.

4. Conclusão

Durante a implementação percebeu-se a dificuldade em montar o AFND, pela necessidade de todo o mapeamento e controle de símbolos terminais e não-terminais, atentando para todos os casos que podem ocorrer para que não haja nenhum equívoco na montagem.

Como a presença de estados inalcançáveis e estados que não alcançam estado final aumentam gradativamente a complexidade do autômato, se viu necessária a minimização dele para a remoção desses estados. A partir disso, pensou-se um utilizar dos conceitos de busca em grafos para, através de uma busca em profundidade, compreender o autômato como um grafo para analisar seu conjunto de transições e remover os estados indesejáveis.

A determinização envolveu um processo de análise de ocorrências de múltiplas transições a partir de um mesmo terminal em um mesmo estado. Ao identificá-las, foi necessário tratá-las e iniciou-se novamente um processo de criação de novos estados (os chamados estados pertencentes a determinização). Como novos estados foram gerados e as transições dos indeterminismos foram alteradas, pode ser que novos estados inalcançáveis tenham aparecido e com isso foi necessário fazer uma nova busca para removê-los.

Com o presente trabalho se evidenciou o funcionamento dos Autômatos Finitos Determinísticos e Não-determinísticos, as diferenças de complexidade e porque se torna computacionalmente inviável validar palavras de uma linguagem em um AFND.

Referências

Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2002). *Introdução à Teoria de Autômatos, Linguagens e Computação*. Campus-RJ, 1st edition.