Implementação do Reconhecedor De Cadeias Por Pilha Vazia

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

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

Abstract. *This paper intends to present the realization of the implementation of an empty stack string recognizer, given the initial transition rules defined, of a CFG if (p, x) that belongs to M (q, a, Z) then [q, aw , Zy] -¿ [p, w, xy]. Since the algorithm consists in identifying whether by these rules under the given chain it can be recognized or not, showing changes in the queue and stack in each step.*

Resumo. *Este artigo pretende apresentar a realização da implementação de um reconhecedor de cadeias por pilha vazia, dada as regras de transições iniciais definidas, de uma gramática livre de contexto (GLC) se (p,x) que pertence a M(q,a,Z) então* $[q, aw, Zy] \rightarrow [p, w, xy]$*. O algoritmo consiste em identificar se por estas regras sob a cadeia fornecida ela pode ser reconhecida ou não, mostrando mudanças na fila e na pilha em cada passo.*

1. Introdução

O objetivo deste trabalho é apresentar uma solução para o reconhecimento de uma cadeia por pilha vazia utilizando de uma gramática livre de contexto, no formato:

Se (p,x) que pertence a M(q,a,Z) então $[q, aw, Zy] \rightarrow [p, w, xy]$

Além disto precisa-se saber como é um autômato de pilha, que está explicado na próxima seção e como é o reconhecimento por pilha vazia, ou seja, até que a pilha e a fita estejam vazias.

Sendo que o algoritmo deve reconhecer as cadeias dadas por pilha vazia, tanto quanto deve mostrar a mudança na fita e na pilha conforme o avanço do reconhecimento da cadeia.

O trabalho foi implementado na linguagem C++, sendo usado suas funções e bibliotecas.

2. Referencial Teórico

Um breve resumo sobre a gramática livre de contexto (GLC) e pilha segundo [Ramose 2008], em teoria de linguagem formal, é uma gramática formal onde todas as regras de produções são da forma

A ::= a

Onde A é um símbolo não terminal, e a é uma cadeia de terminais e/ou não terminais (a pode ser vazia).

Já o Autômato de pilha é um autômato finito com uma memória auxiliar em forma de pilha. Atuando na pilha como:

1.Eles podem fazer uso da informação que está no topo da pilha para decidir qual transição deve ser efetuada;

2.Eles podem manipular a pilha ao efetuar uma transição.

Um autômato com pilha é formalmente definido por uma 6-tupla M = (Q, Σ, R, δ, Z, s, F) onde:

Q é o conjunto finito dos estados do autômato

Σ é o alfabeto de entrada

R é o alfabeto da pilha (não é requerido que $R \cap \Sigma$ =)

$Z \in R$ é o símbolo inicial da pilha (único elemento da pilha no momento inicial – iremos ver que este elemento é facultativo)

$s \in Q$ é o estado inicial do autômato

$F \subseteq Q$ é o conjunto dos estados finais

$delta \subseteq ((Q \times \Sigma * \times R*) \times (Q \times R*))$ há a relação (finita) de transição.

(Quando se isenta a utilização do símbolo inicial de pilha a definição de um autômato, restringe-se a um 6-tupla – $Z = \epsilon$)

3. Desenvolvimento

3.1. Definições

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

```
1 typedef pair<char,char> pc;
2 typedef vector<string> vs;
3 typedef vector<int> vi;
4 typedef vector<char> vc;
5
6 #define MAX 1123
```

Cada definição significa:
1- Pair of char nomeado pc.
2- Vector of string nomeado vs.
3- Vector of int nomeado vi.
4- Vector of char nomeado vc.
5- Definição do MAX como um valor constante.

3.2. Main

A função main terá algumas variáveis importantes, sendo elas:

```
1 char word[MAX];
2 char first_stack;
3 char wtp[MAX];
4 int terms_count;
5 int accepted;
6 map<pc,vs> transitionsMap;
```

```
7 map<char, int> map_terms;
8 vc terms_name;
9 FILE *inputFile;
```

Cada variável significa
1- String da palavra de tamanho MAX. 2- O primeiro elemento da pilha. 3- Uma string que representa o que deve ser colocado a seguir na pilha.
4- Contador de terminais. 5- Variável para dizer se foi aceita a entrada.
6- Map de pc para vs chamado transitionsMap. Mapeia um par de chars para um vetor de strings, contendo as possíveis transições.
7- Map de char para inteiro chamado map_terms. Mapeia o nome do terminal para seu id correspondente.
8- Vector de char dos nomes de terminais.
9- Ponteiro do arquivo de entrada da GLC.

3.3. Funções

3.3.1. Main

Função no arquivo main.cpp

Sendo chamada a função *loadTransitions* para leitura do arquivo de entrada, com as transições, sendo que a função *printTransitions* imprime na tela o que foi mapeado para o map *transitionsMap*.

Para a construção do map_terms é feita a função *build_map_terms*, onde ocorre a classificação dos terminais e sua contagem. Sendo lida a cadeia de tokens que deve ser reconhecida. Para isso são criadas as variáveis:

```
1 vi word_terms;
2 vi stack_terms;
3 stack<char> astack;
4 \textit{backtracking}
```

1- Vector dos terminais na cadeia lida. 2- Vector dos terminais na pilha. 3- Pilha feita para ser desempilhada.

Sendo que na função *calculate_word_terms* é calculado quais e quantos terminais existem.

3.3.2. função *printTransitions*

Função no arquivo *recognize.cpp*

Função responsável por mostrar o que foi lido no mesmo formato o qual foi passado como entrada ao programa.

3.3.3. Função *nextSymb*

Função no arquivo *recognize.cpp*

Função que procura pela primeira ocorrência de um caractere símbolo na string que está sendo lida, sendo passada á ela o símbolo procurado.

3.3.4. Função *loadTransitions*

Função no arquivo *recognize.cpp*

Função que faz a leitura do arquivo de entrada, linha por linha pegando cada linha com o que será usado para o reconhecimento por pilha vazia. É chamada a função *nextSymb* para percorrer a linha, guardando as transições no map, a pilha e a fita.

3.3.5. Função *build_map_terms*

Função no arquivo *recognize.cpp*

Função que constrói o map de terminais do conjunto de transições do autômato, sendo aumentado o contador de quantos terminais existem.

3.3.6. Função *calculate_word_terms*

Função no arquivo *recognize.cpp*

Função que cria um vetor de inteiros calculando quantos de cada terminal existem para fazer o controle deles.

3.3.7. Função *backtracking*

Função no arquivo *recognize.cpp*

Esta função trabalha da seguinte maneira: baseando-se no vetor de controle de *calculate_word_terms*, ela terá mais dois vetores locais de controle, um com a quantidade de terminais já usados até agora e um para simular caso ela escolha uma transição, para que esta não influencie no seu próprio vetor. Assim, é comparado o vetor dos terminais já usados mais os terminais simulados com a quantidade de terminais totais na palavra. Deste modo, caso a soma seja maior do que a quantidade que existe, ele não irá escolher este caminho e verá se é viável chamar a função *backtracking* para a próxima transição. Isso limitará o número de transições que ela fará, chegando em um reconhecimento. Caso seja viável, chama-se a função *backtracking* e ela criará novos vetores locais.

4. Conclusão

Implementar um reconhecedor por autômato de pilha exigiu uma grande análise de um algoritmo utilizando um *backtracking* para percorrer todos os caminhos que o indeterminismo do autômato de pilha não-determinístico gera. A maior dificuldade foi analisar a complexidade de espaço para alocação de uma pilha local para cada nó da árvore e decidir com qual estratégia atacar.

Como a função de *backtracking* consiste em uma abordagem gulosa com controle de terminais para determinar a parada, a recursão certamente acabará, e se a palavra pertencer a gramática do autômato, a função encontrará o ramo em que ela aceita a palavra. Se não for aceita, gera uma ambiguidade entre a pilha e a fita que não pode ser resolvida.

Com o presente trabalho se evidenciou o funcionamento do Autômato de Pilha Não-determinístico, lapidando a percepção das altas complexidades de tempo e espaço que reconhecer uma palavra por esse método pode ter.

Referências

Ramose, M. V. M. (2008). Linguagens formais e autômatos. pages 181–246.