Universidade do Estado de Santa Catarina
Centro de Ciências Tecnológicas
Departamento de Ciência da Computação
Bacharelado em Ciência da Computação

Felipe Carlos Caetano Severino e Mariana dos Santos

Relatório: Simulador de autômatos com pilha

Linguagens Formais e Autômatos
Prof. Ricardo Martins

Joinville
Julho/ 2024

Sumário

1 Atividade proposta ……..
2 Código fonte ……..
2.1 Estrutura de Dados …….
2.2 Função de Reconhecimento ……..
2.3 Interface com usuário …….
2.4 Geração do Grafo …….
3 Exemplos de execução ……
3.1 Exemplo 1 ……
3.2 Exemplo 2 ……
4 Dificuldades encontradas ……

1 Atividade proposta

Um Autômato Finito Determinístico com Pilha é uma extensão do conceito de Autômato Finito Determinístico que inclui uma pilha como componente adicional. Este tipo de autômato é usado para trabalhar com linguagens livres do contexto.

Neste trabalho foi realizada a implementação de um simulador para o Autômato com pilha em linguagem C. O programa permite aos usuários definir estados, transições e verificar se a palavra é aceita ou recusada pelo Autômato. Além disso, é utilizado a biblioteca Graphviz para gerar visualizações gráficas do autômato.

2 Código fonte

2.1 Estrutura de Dados

O código utiliza estruturas de dados para representar os estados, as transições e o autômato finito, essas estruturas estão no arquivo “aflib.h”. Abaixo segue as principais estruturas e suas funções.

ESTADO: Representa um estado do autômato, com indicação de inicial e final.

```c
typedef struct estado {
    int inicial; // 0 = false, 1 = true
    int final; // 0 = false, 1 = true
    char *nome;

    struct estado * next;
} ESTADO;
```

TRANSICAO: Define uma transição entre os estados (q_from, q_to), com símbolo de transição, operações na pilha e próxima transição.

```c
typedef struct transicao {
    char symbol;
    char *insertStack;
    char popStack;

    char *q_from;
    char *q_to;

    struct transicao * next;
} TRANSICAO;
```

AF: Estrutura principal que mantém uma lista de estados e transições.

```c
typedef struct af {
    ESTADO *h_estado;
    TRANSICAO *h_transicao;

    size_t n_estados;
} AF;
```

Além das estruturas destacadas acima, o arquivo “datast.h” contém a estrutura da pilha e suas funções padrões.

2.2 Função de Reconhecimento

A função “reconhecePalavra” verifica se uma palavra é aceita ou não pelo autômato, levando em consideração suas transições e operações na pilha. A função é recursiva e desloca-se entre os estados e transições do autômato. Importante observar que durante a produção foram criadas trechos para Debugging na função, que foram omitidos na explicação abaixo:

```c
int reconhecerPalavra(char *palavra, int index, ESTADO *currentState, PILHA *stack, ESTADO **visitados, int nVisitados, AF *af, int nivel) {
```

A função reconhece os seguintes parâmetros: a palavra a ser reconhecida, o índice atual, o estado atual, o estado atual, a pilha, os estados visitados, o número de estados visitados, o autômato e o nível.

```c
if (palavra[index] == '?' || index == strlen(palavra)) {
    //printf("-Entrou if 1\n");
    if (currentState->final) {
        return 1;
    }
}
```

Parte da condição de aceitação, no qual verifica se a palavra foi totalmente processada ou se encontrou um caractere especial. Retorna 1 caso o estado atual seja um estado final.

```
    TRANSICAO *tr = af->h_transicao;
    strcat(identacao, " ");
    while (tr != NULL) {
```

Inicia a variável “tr” para percorrer todas as transições do autômato.

```
if (tr->symbol == palavra[index] && strcmp(tr->q_from, currentState->nome) == 0) { //transicao
    if (tr->popStack == '#' || tr->popStack == stack->topo->symbol) {
        //printf("Entrou caso 1\n");
        ESTADO **visitedArray = getArrayEstado(af->n_estados);

        if (tr->popStack != '#') {voidPop(stack);}
        if (strcmp(tr->insertStack, "#") != 0) {
            //printf("Entrou insere pilha\n");
            for (int j = 0; j < strlen(tr->insertStack); j++) {
                //printf("insere %c\n", tr->insertStack[j]);
                push(stack, tr->insertStack[j]);
            }
        }

        //printf("Depois insere pilha\n");

        ESTADO *q_to = getEstadoByName(tr->q_to, af);
        //printf("Antes reconhece palavra");

        char *stringPilha = getStringPilha(stack);
```

Primeiro a função verifica se o símbolo da transição corresponde ao símbolo atual da palavra e se o estado de origem da transição é o estado atual, ou seja verifica se a transição pode ser aplicada. Após isso, verifica se a transição pode ocorrer com base no estado atual da pilha, verificando se o símbolo a ser removido da pilha está no topo da pilha ou se a transição não requer remoção. Caso satisfeitas as condições, cria um novo array de estados visitados e caso necessário, realiza uma operação de pop na pilha. Além disso, se a transição requer a inserção de símbolos na pilha, realiza operações de push para cada símbolo. Em seguida, busca o próximo estado.

```
        int reconhece = reconhecerPalavra(palavra, index+1, q_to, stack, visitedArray, 0, af, nivel+1);
        //printf("Depois reconhece palavra\n");
        if (reconhece) {
            char s = tr->symbol;
            if (s == '?') {s = ' ';}
            printf("%d) %c  |  %s->%s  |  %s\n", nivel, s, tr->q_from, tr->q_to, stringPilha);

            return 1;
        }

        free(stringPilha);

        if (strcmp(tr->insertStack, "#") != 0) {
            for (int j = 0; j < strlen(tr->insertStack); j++) {
                voidPop(stack);
            }
        }
        if (tr->popStack != '#') {push(stack, tr->popStack);}

    }
}
```

A próxima etapa refere-se a chamada recursiva. Chama recursivamente a função para processar o próximo símbolo da palavra e o próximo estado. Se a palavra é aceita retorna 1, caso não seja aceita reverte as operações realizadas na pilha e libera a memória alocada para o array de estados visitados.

```
        if (tr->symbol == '#' && strcmp(tr->q_from, currentState->nome) == 0) { //transicao vazia
            if (tr->popStack == '#' || tr->popStack == stack->topo->symbol) {
                if (!estadoPercorrido(currentState, visitados, nVisitados)) {
                    nVisitados++;
                    visitados[nVisitados] = currentState;

                    if (tr->popStack != '#') {voidPop(stack);}
                    if (strcmp(tr->insertStack, "#") != 0) {
                        //printf("Entrou insere pilha\n");
                        for (int j = 0; j < strlen(tr->insertStack); j++) {
                            //printf("insere %c\n", tr->insertStack[j]);
                            push(stack, tr->insertStack[j]);
                        }
                    }
```

Para o processamento das transições vazias, elas são tratadas com o “symbol == #”. Nesta etapa é verificado se o estado atual já foi visitado para evitar loops infinitos, realizando as operações na pilha conforme necessário.

```
                char *stringPilha = getStringPilha(stack);

                ESTADO *q_to = getEstadoByName(tr->q_to, af);
                int reconhece = reconhecerPalavra(palavra, index, q_to, stack, visitados, nVisitados, af, nivel+1);
                if (reconhece) {
                    char s = tr->symbol;
                    if (s == '?') {s = ' ';}
                    printf("%d) %c  |  %s->%s  |  %s\n", nivel, s, tr->q_from, tr->q_to, stringPilha);

                    return 1;
                }

                if (strcmp(tr->insertStack, "#") != 0) {
                    for (int j = 0; j < strlen(tr->insertStack); j++) {
                        voidPop(stack);
                    }
                }
                if (tr->popStack != '#') {push(stack, tr->popStack);}

                visitados[nVisitados] = NULL;
                nVisitados--;
            }
        }
    }

    tr = tr->next;
```

Nesta parte, é gerada uma representação atual da pilha e obtido o próximo estado da transição atual. Após isso é realizada a chamada recursiva para continuar o reconhecimento da palavra. Se a palavra é reconhecida, são impressas suas informações. Em seguida, a pilha é restaurada, realizando as operações necessárias. No final marca o último estado visitado como NULL e decrementa o contador.

2.3 Interface com usuário

A interface com usuário foi projetada para ser interativa, permitindo a fácil criação e simulação do autômato. Para as entradas e saídas a interface utiliza o console, orientando o usuário por meio do menu nos seguintes passos: definição de estados, transições e verificação da palavra.

Menu principal

Ao iniciar o programa, o usuário é apresentado ao menu principal com duas opções:

a) Iniciar a simulação: Permite ao usuário começar a definir o autômato.
b) Encerrar o programa: Encerra a execução do programa.

O menu principal é exibido repetidamente até que o usuário escolha a opção de encerrar o programa. No momento que o usuário escolher iniciar a simulação, ele entra no modo de criação de estados. Este processo se repete até que o usuário opte por avançar para o próximo passo, nesta etapa o usuário poderá definir as transições entre os estados, da mesma forma que anteriormente esse processo se

repete até que o usuário decida avançar. Por fim, o programa solicita a entrada de uma palavra e verifica se ela é válida ou não.

2.4 Geração de Grafo

Para a visualização gráfica do autômato utilizou-se da biblioteca Graphiz, facilitando a criação e exibição dos grafos.

Estrutura de dados para o Grafo

Para gerar o grafo do autômato, foram utilizadas as seguintes estruturas:

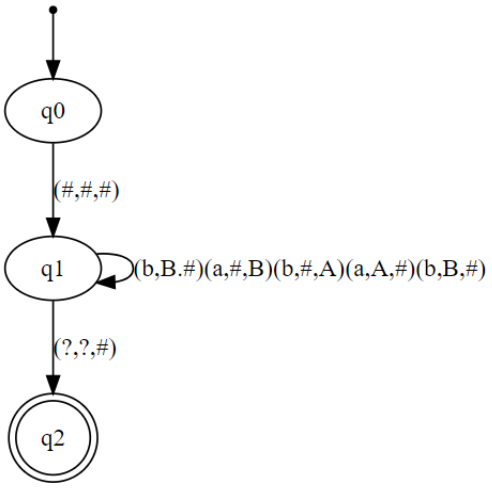
1. Estados (ESTADO): Cada estado do autômato é representado como um nó do grafo. Para a visualização dos nós finais e nó inicial, foram usadas as propriedades de inicial e final dos estados.
2. Transições(TRANSICAO): As transições entre estados são representadas como arestas. Cada aresta exibe o símbolo de transição e as operações sobre a pilha

O programa gera dinamicamente um link que contém todas as informações do grafo e abre automaticamente usando o comando “system”.

3 Exemplo de execuções

3.1 Exemplo 1

O primeiro exemplo foi realizado com o autômato {a^nb^n | n ≥ 0} onde n é um número natural, Abaixo segue a configuração do autômato (obs: imagem gerada pela aplicação ao final da execução):

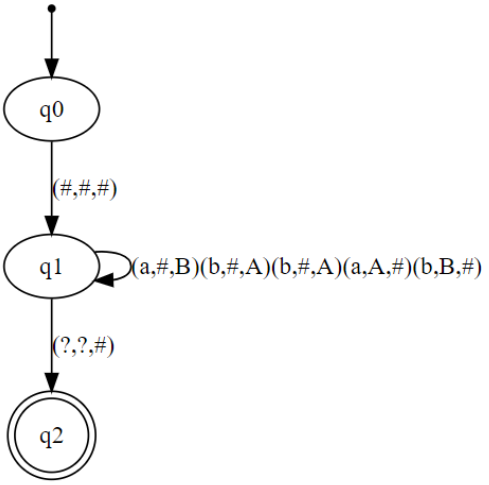


Palavras testadas

a) Palavra “aabb”
   i) transições e operações na pilha:
      1) Transições: q0-> q1-> q1-> q1-> q1-> q1-> q2
      2) Pilha: pilha vazia-> pilha vazia-> empilha B-> empilha B-> desempilha B-> desempilha B-> pilha vazia
   ii) Reconhecimento: Válida
b) Palavra “aab”
   i) transições e operações na pilha:
      1) Transições: q0-> q1-> q1-> q1-> q1-> qf
      2) Pilha: pilha vazia-> pilha vazia-> empilha B -> empilha B-> desempilha B-> (problema! pilha não está vazia).
   ii) Reconhecimento: Inválida

3.2 Exemplo 2

O Segundo exemplo foi realizado com o autômato no qual os números de “a”s são iguais aos número de “b”s, ou seja |a| = |b|. Abaixo segue a configuração do autômato (obs: imagem gerada pela aplicação ao final da execução):



Palavras testadas

c) Palavra “abab”
   i) transições e operações na pilha:
      1) Transições: q0-> q1-> q1-> q1-> q1-> q1-> q2
      2) Pilha: pilha vazia-> pilha vazia-> empilha B-> empilha A-> desempilha A-> desempilha B-> pilha vazia
   ii) Reconhecimento: Válida
d) Palavra “aab”
   i) transições e operações na pilha:
      1) Transições: q0-> q1-> q1-> q1-> q1-> q2

2) Pilha: pilha vazia-> pilha vazia-> empilha B-> desempilha B-> desempilha B-> (problema! pilha não está vazia).

ii) Reconhecimento: Inválida

4 Dificuldades encontradas

De todas as dificuldades encontradas, tivemos duas que verdadeiramente atrasaram o andamento do projeto.

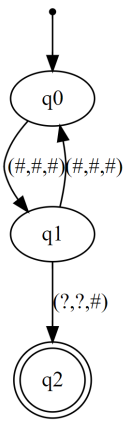
a) Visualização gráfica do autômato:

A estratégia inicial consistia em utilizar a biblioteca graphviz. Faríamos uso de algumas funções da biblioteca para que o código, ele próprio, fizesse a renderização da representação gráfica do autômato na forma de um dígrafo. Contudo, conforme avançamos com esta estratégia, encontramos inúmeras limitações e comportamentos inesperados da biblioteca graphviz. Um dos comportamentos inesperados foi a impossibilidade de definir um nó do grafo como “double circle”, o que seria fundamental, pois este formato indica que o estado é final.

Em virtude dos problemas supracitados, decidimos abandonar a biblioteca graphviz e fazer a representação gráfica a partir de uma aplicação online que renderiza códigos da linguagem dot. O sistema gera um link para esta aplicação (o link contém todas as informações do dígrafo) e abre este link com o comando “start” através da função system.

b) Função de reconhecimento

Encontramos dificuldade implementando a lógica de transições vazias do autômato. Por se tratar de uma transição que não consome nenhum símbolo da fita, ela poderia gerar um loop infinito na execução do programa.



No exemplo acima, o sistema poderia transitar infinitamente entre q0 e q1. Para evitar isso, criamos um vetor que armazena todos os estados que foram percorridos por transição vazia. Se uma transição vazia leva a execução a um estado que já foi percorrido por outra transição vazia, então esta transição não deve ser executada. Quando executamos uma transição que consome algum símbolo da fita, este vetor é zerado.