UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

RODRIGO DE CASTRO MICHELASSI
NUSP: 13672703

Exercício-Programa 4: Expressões Regulares

São Paulo
2023

Sumário

1 Introdução

Nesse EP, tínhamos o objetivo de implementar um sistema capaz de identificar, dado uma expressão regular, se uma dada string poderia ser lida pela expressão.

Para isso, recebemos como entrada uma expressão regular, um número inteiro n e n palavras, as quais deveriam ser verificadas. A ideia central do EP estava na construção do autômato da expressão regular, com base em um grafo, e no algoritmo de verificação.

Nesse sentido, foram utilizados alguns algoritmos vistos anteriormente, como o DFS, e o próprio algoritmo visto em sala de verificação de expressões regulares, que foi profundamente modificado a fim de alcançar as demandas do programa que foi desenvolvido.

OBS: O EP entregue foi feito em uma máquina rodando MACOS. Ao testar em uma máquina com Linux baseado em Debian, ocorreram alguns Warnings que não ocorrem no MAC, porém ao mudarmos o código para que atendesse às demandas do Ubuntu, o mesmo código apresentava erro no PC em MAC. Nesse sentido, a versão entregue foi feita inteiramente no MAC e deve funcionar corretamente, apesar dos Warnings que serão obtidos durante a compilação.

2 Estruturas de Dados

Primeiramente, foram implementadas duas estruturas de dados principais, e utilizadas outras funções pré-prontas da biblioteca STL do C++.

As estruturas implementadas foram um grafo, baseado em uma lista de adjacências, feita a partir do vector do STL do C++. Nosso grafo contém uma lista de vértices, onde cada vértice armazena o dígito correspondente à ele, o tamanho da aresta em que pertence, o seu número, um booleano que diz se esse vértice realiza λ-Transição, uma string que guarda se possui uma operação especial e uma lista com suas arestas.

Além disso, na classe do grafo possuímos basicamente todas as funções que o programa executa, sendo elas: o construtor, construir o autômato, verifica se reconhece uma string, e as funções auxiliares.

3 Algoritmos Implementados

Antes de analisarmos os algoritmos, note que receberemos no nosso terminal duas impressões diferentes. A primeira, conta com todos os vértices, com seus respectivos arcos e, caso esse vértice realize alguma operação específica, como conjunto, intervalo ou complemento, também será mostrada. A segunda mostra a resposta para cada entrada.

3.1 Construtor

Diferentemente do EP3, aqui foi implementado um construtor que já inicializa o grafo com seus respectivos vértices, na ordem que foram recebidos pela expressão regular. A grande preocupação aqui, além de inicializar os vértices com os valores de cada um e sua posição na expressão regular, utilizamos duas operações auxiliares.

A primeira função utilizada define se um vértice deve ou não realizar uma λ-Transição. Para isso, verificamos se o dígito desse vértice é o representante de operações da expressão regular, ou seja, pertence ao conjunto $\{(,), |, [,], +, *, \backslash\}$, e o seu predecessor não é um \, dígito utilizado para mostrar que o dígito especial não é um operador.

Já na seugnda função, definimos se um vértice realiza uma operação de complemento, intervalo ou conjunto. Para isso, analisamos os vértices que seguem [, e marcamos o tipo de operação no primeiro vértice após a abertura do colchete. Caso o vértice não realize nenhuma dessas operações, retornamos uma string vazia.

3.2 Constrói Autômato

A lógica por trás desse algoritmo é relativamente sofisticada, porém pode ser facilmente explicada. A ideia principal é perceber que, no geral, para as expressões regulares vistas, todos os vértices tem um arco para o vértice seguinte, a exceção do vértice que contem o operador '|'. Dessa forma, marcamos todos os vértices a excessão desse com um arco para o seguinte.

Dessa forma, para os vértices que se conectam a outros, não a sua frente, utilizamos uma pilha, para guardarmos toda vez que abrimos um parênteses ou identificamos o operador *or*, de forma que, ao fecharmos um parênteses, apontamos o operador *or* para o

fim do parênteses, e a abertura de parênteses para o operador $*$, que pode aparecer ao fim. Nesse sentido, apontamos também os dois tipos de fecho $(*, +)$ para a abertura de parênteses.

Ao fim, nosso grafo possui todos os vértices e arestas para simularmos uma palavra na expressão regular.

3.3 Verificar se o autômato reconhece uma string

Esse algoritmo também possui algumas pequenas sofisticações que devem ser exploradas. Primeiramente, inicializamos um vetor de tamanho igual a quantidade de vértices. Para esse vetor, partindo de 0, rodamos o DFS no nosso grafo, e marcamos todas casas que podemos chegar do início.

Com esse vetor de vértices visitados em mão, para cada vez que chegamos em um vértice cujo dígito é o que queremos ler (ou que se encaixa em um conjunto, complemento ou intervalo), salvamos que chegamos nesses vértices em um outro vetor, chamado *next*. Para cada vértice marcado como verdadeiro no vetor *next*, executamos o DFS novamente. Se retornar um vetor totalmente falso, significa que nossa expressão regular não identificou uma palavra. Caso retorne verdadeiro em algumas posições, significa que conseguimos ler a letra que queríamos e agora temos um novo vetor, como o inicial, de vértices visitados a partir do dígito que chegamos. Esse algoritmo iterativo permite que visitemos toda a expressão regular, caso seja necessário.

Aqui também foi utilizado uma verificação para as operações definidas no construtor. Então, se rodamos o algoritmo e chegamos em um vértice que já foi visitado, e esse vértice for do tipo conjunto, complemento ou intervalo, verificamos se a letra que queremos ler se encaixa nessa operação, e marcamos como verdadeiro, no vetor *next*, a posição onde se fecha o], significando que conseguimos passar por essa operação.

4 Testes e Execução

Para executar o programa, basta rodar no terminal, no diretório onde ele está contido, o comando *make*, e depois utilizar *./ep4*.

Para os testes desse programa, foram utilizados os 4 exemplos do enunciado e alguns outros exemplos simples feitos à mão, e o programa se mostrou eficiente ao obter respostas corretas para todos. Vale lembrar um adendo, o professor indicou no fórum do Moodle que poderíamos considerar que sempre que houvesse uma operação *or*, esse viria seguida de parênteses. Assim, o exemplo 3 do enunciado foi modificado, para que a expressão regular dada fosse $((A*CG|A*TA)|AAG*T)*$, a fim de que o programa funcionasse corretamente.

Um outro ponto importante é que também foi implementado a operação de complemento de conjuntos, de forma que o programa identifique toda vez que se deparar com algo do tipo $[\hat{}\,0-9]$, por exemplo. Segue então alguns exemplos testados e os resultados obtidos com sucesso:

4.1 Exemplo 1

$[\hat{}\,AEIOU][AEIOU][\hat{}\,A-G][ABE]$

3

JAJA

AEZB

PULA

Resp: S N S

4.2 Exemplo 2

$((A*B|AC)D)$ 2 AABCD ABD

Resp: N S

Os demais exemplos do enunciado devem funcionar corretamente, com base nos testes feitos.

5 Considerações Finais

Com base nesse projeto foi possível explorar um pouco melhor o funcionamento e utilidade do uso das expressões regulares, estudadas em aula durante o semestre. Dessa forma, foi possível implementar em código também, com base em uma simulação o algoritmo de verificação de expressões regulares.

Além disso, o EP tornou possível a adaptação de algoritmos vistos anteriormente, como o DFS, além de rever o uso de pilhas para a construção do autômato, da forma que desejássemos.

Assim, podemos dizer que o desenvolvimento desse EP foi de grande proveito para o aprendizado e para a disciplina, além de mostrar-se de grande interesse para a introdução de outras matérias do curso, como Autômatos e Algoritmos em Grafos.