EP4 – Algoritmos e Estruturas de Dados II

Adriano Elias Andrade

13/07/2023

1. Introdução:

O programa principal “ep4.cpp” recebe uma entrada como descrita no enunciado, e para cada palavra, diz se é reconhecida pela expressão regular.

2. Compilação e entrada:

Compilação: Para compilar o ep, basta usar o comando “make”.

Entrada: A entrada do programa deve ser como a do enunciado:

```
<expressão regular>
<número de palavras, N >
<palavra 1>
...
<palavra N>
```

3. Implementação

Na implementação, foi feita a classe “grafo”, que faz algumas operações no grafo, além das funções do programa principal, “ep4.cpp”.

3.1 Corrigindo a expressão regular:

Tendo uma expressão regular de entrada, precisamos corrigir ela para que funcione. Isso é feito na função corrigeExpreg(), na seguinte ordem:

Corrigindo”[]”: para que alguma expressão que utilize colchete (feixe e conjunto), é necessário colocá-las entre parênteses. Isso serve para que seja possível utilizar * ou + após os colchetes. Para isso basta colocar “(” antes de todo “[“, e “)” após todo “]“.

Corrigindo"*" e "+": para que estes símbolos funcionem após alguma letra, colocamos a letra entre parênteses. Caso já haja ")" antes desses símbolos, nada é feito. O que acontece é que "X*" vira "(X)*", e "(ABC)*" continua igual.

Corrigindo"|": para que seja possível usar mais de duas alternativas em seguida, é preciso separá-las em casos binários. Para isso, transformamos algo do tipo "(A|B|C|D)" em "(((A|B)|C)|D)". Para fazer isso, empilhamos os "(" até encontrar seus pares, e dentro do intervalo entre "(" e ")", separamos os N "|" em casos binários, colocando N-1 "(" antes da primeira alternativa ("A" no exemplo dado), e ")" antes de cada "|", além do primeiro.

3.2 Montando o grafo não determinístico:

No grafo, teremos um vértice para cada caractere na expressão regular, mais o vértice de fim de palavra.

Alternativas: para montar as alternativas, empilhamos as posições dos "(" e os "|" até encontrar um")". Assim, desempilhamos o "|", e o ")", para construir um arco que vai de "(" até a primeira alternativa e de "|" até o ")", e de "(" até a segunda alternativa. Com isso em (A|B), formamos o caminho (-> A => | ->), e o caminho (-> B =>).

Fecho: para montar fechos, aproveitamos as posições já conhecidas de um par de parênteses (feito na montagem de alternativas), e com elas, colocamos um arco de "(" até uma posição depois do "*", e um arco de "*" até "(", além do arco de "*" até o seguinte dele. Então, em (A)*B, teremos (-> B ; * -> (; * -> B.

Mais: para montar o mais, fazemos o mesmo do que no fecho, mas sem o arco do "(" até um depois do ")".

Parênteses: para os parênteses, "(" ou ")", sempre botamos um arco para a próxima posição.

Intervalo e conjunto: quando achamos um "[" verificamos se é complemento ou não. Em seguida, verificamos se é intervalo ou conjunto. Para isso, verificamos se há um "-" onde deveria estar caso fosse intervalo. Dependendo do caso, criamos um vértice especial de leitura (explicado a seguir, em 3.3) na posição do "[", e independente do caso, um arco do seguinte de "[" até o "]".

Coringa: para um coringa, criamos um vértice especial de leitura na posição dele.

Barra invertida: para a "\", criamos um vértice especial de leitura na posição dela, e um arco do seguinte de "\" até o seguinte do seguinte de "\".

Caracteres: para os caracteres, criamos um vértice especial de leitura na posição deles.

3.3 Vértices de leitura:

Para as leituras, separamos os diferentes casos, e criamos vértices especiais de leitura. Para isso, existe uma struct noLeitura, com as variáveis:

tipo: guarda o tipo do nó de leitura. 0=letra, 1=conjunto, 2=complemento de conjunto, 3=intervalo, 4=complemento de intervalo, 5=coringa, 6=fim de palavra, -1=não é de leitura.

letra: guarda o caractere a ser lido, caso haja só um.

conjunto: em caso de conjunto guarda a lista de caracteres que podem ser lidos (em um vector).

ini e fim: guarda o início e fim de um intervalo.

Com essa estrutura, é feito um vetor de noLeitura do tamanho do grafo, inicializando os tipos de cada nó como -1 (não é de leitura), e o último nó como 6 (fim de palavra). Durante a montagem do grafo quando aparece algum vértice de leitura, o nó na posição desse vértice recebe o tipo e outros atributos necessários. Assim, é possível ver se um nó lê um caractere durante o reconhecimento de uma palavra.

3.4 Reconhecendo uma palavra:

Para o reconhecimento de uma palavra, precisamos fazer busca em profundidade no vértices do grafo várias vezes. Para poupar o trabalho de precisar fazer uma dfs toda vez que um caractere novo for lido, antes de começar a ler palavras é feita uma dfs em cada vértice do grafo, e guardamos, para cada vértice, todos os vértices de leitura que ele alcança, em vector<int> próximos[]. Assim, na hora de ler um caractere, é só verificar próximos[vértice atual], que terá a lista de vértices de leitura alcançados. Isso é feito na função inicializaProximos()

Para reconhecer uma palavra, na função reconhece(), guardamos em “alcançados” os vértices de leitura alcançados pelos vértices atuais, e em “atuais” guardamos os vértices seguintes dos “alcançados” que puderam ser lidos pelo caractere atual. Isso é feito utilizando o vetor de próximos já inicializado. No início, “alcançados” começa com os próximos do vértice 0.

Durante as iterações nos caracteres, caso “alcançados” esteja vazio, retornamos falso, pois a palavra não pode mais ser reconhecida. Caso a palavra tenha acabado, vemos se o vértice de fim de palavra (um vértice de leitura tipo 6) está em “alcançados”, e retornamos verdadeiro caso esteja.

4. Formato da expressão regular:

A seguir, algumas especificações para que a expressão regular seja aceita:

- São aceitos caracteres (incluindo feixes e conjuntos) com * ou + em seguida. Por exemplo:

  `a*|[a-z]*` é a mesma coisa que **`(a)*|([a-z])*`**

- Não são aceitos caracteres especificados com \ antes, com * ou + em seguida. Por exemplo:

\.* não funciona, precisa ser **(\.)***

- Não são aceitos espaços na expressão regular, até porque seria lida como duas entradas diferentes.

- Conjuntos de "|" precisam estar entre parênteses (menos os que compõem a expressão inteira). Por exemplo:

(**a*b|cd***) é ambíguo, e será lido como(**(a*b)|(cd*)**) ,e não (**a*(b|c)d***)

a|b|c compõe a expressão inteira, e será lido como **(a|b|c)**

5. Testes:

A seguir, alguns exemplos de testes.

5.1 exemplo com vários "| em conjunto":

Entrada:

```
(A*B|C|D)|[^123]|(F|G|H)
8
AAB
C
D
2
6
F
G
H
```

Saída:

```
S
S
S
N
S
```

```
S
S
S
```

5.2 email:

Entrada:

```
([a-z]|[0-9]|\.)*@(([a-z])+\.)+br
3
adri.aea.adri@gmail.br
cef1999@ime.usp.br
180.84.543.20
```

Saída:

```
S
S
N
```

5.3 senha com pelo menos 1 caractere maiúsculo e 1 especial:

Entrada:

```
(.*[!@#$%&={}?]+.*[A-Z]+.*)|(.*[A-Z]+.*[!@#$%&={}?]+.*)
6
abc
Abc
%bc
A&c
!bC
aB#
{O}
B=P
```

Saída:

```
N
N
N
```

```
S
S
S
```

5.4 coordenada geográfica:

Entrada:

```
[0-9]+\.[0-9]+[SN],[0-9]+\.[0-9]+[EW]
3
40.5216N,117.1195W
51.51723N,0.10235W
35.704074N,139.557732E
```

Saída:

```
S
S
S
```