Relatório - Anticompressor de Textos

Giovane Bianchi Milani[1]

[1]Escola Politécnica - PUCRS

18 de abril de 2022

Resumo. *Este artigo descreve três alternativas para solucionar o problema proposto na disciplina de Algoritmos e Estrutura de Dados II, que consiste em encontrar o tamanho final de um arquivo a partir de uma tabela de substituições. As ideias das soluções são apresentadas, juntamente com os pseudocódigos dos algoritmos mais importantes e uma análise de desempenho. Por fim, serão apresentados os resultados obtidos nos dez casos de teste providos pelo professor.*

1. Introdução

No contexto de que estamos irritados com as tecnologias para comprimir textos, pois elas deixam-os muito apertados, o problema que investigaremos pode ser descrito da seguinte forma: desenvolver um anticompressor de textos baseado nas letras minúscula do alfabeto, em que cada letra descrita deve ser transformada em uma nova sequência, até que nada mais possa ser transformado, ou seja, até que todos os caracteres estejam em seu estado final.

O anticompressor recebe um arquivo de entrada, que contém uma tabela de substituições para cada letra, e a partir disso deve-se formar o arquivo de saída, que poderá ser muito grande, dependendo da complexidade da entrada.

Exemplo de arquivo de entrada:

Letra	**Substituição**
a	memimomu
e	mimomu
i	mooo
u	mimimi
o	
m	

Tabela 1. Exemplo tabela de substituições

Exemplo de saída com 47 caracteres:

a = mmmooomommmooommooommooommooomommmooommooommooo

Dada a tabela de substituições, deve-se seguir basicamente dois passos para gerar a saída, são eles:

1. Determinar qual é a letra inicial do arquivo de entrada (é aquela que não tem ocorrências no resto da tabela);

2. Informar quantas letras terá o arquivo final.

Uma observação importante, explícita no exemplo do enunciado, é que podem existir letras na tabela que não possuem substituição. Além disso, o enunciado ainda informa que devemos processar alguns casos de teste, em formato de arquivo, então para isso, devemos desenvolver uma forma de ler e organizar esses dados para posteriormente serem processados no anticompressor.

Para resolver o problema descrito, analisaremos as características de três soluções diferentes, apresentando as ideias usadas e no fim os resultados obtidos a partir da implementação delas.

2. Primeira Solução

Após analisar o problema, podemos perceber que o algoritmo precisará fazer muitas consultas na tabela de substituições, por esse motivo resolvemos organizar os dados um uma estrutura que use *Hash*. Além disso, estabelecemos que os dados estariam em formato de chave e valor, como por exemplo um *Map*, pois assim fica mais fácil de encontrar a substituição referente a cada letra. Segue um exemplo que representa a estrutura escolhida:

Chave	**Valor**
a	“memimomu”
e	“mimomu”
m	null
...	...

Tabela 2. Exemplo estrutura de dados usada

Para ler o arquivo e transformar na estrutura descrita, poderíamos iterar sobre cada linha do arquivo e separá-la no espaço em branco, assim, podemos inserir o primeiro item como chave e o segundo como valor.

Com os dados de entrada organizados, podemos encontrar a letra inicial. Analisando os casos de teste, percebemos alguns padrões e definimos dois requisitos que nos levam a achá-la, são eles:

1. A letra deve conter uma substituição;

2. A letra não pode aparecer na substituição de nenhuma outra letra da tabela.

Dadas as regras, podemos iterar sobre cada item da tabela, verificando se a letra, no caso a chave, possui um valor associado e se não há ocorrência dela no resto da tabela. A letra que cumprir com os dois requisitos é a letra inicial. O algoritmo que resolve esse problema é parecido com esse:

```
para cada (chave, valor) da tabela {
      se valor não é null {
          bool apareceu = falso;
          para cada (substituicao) da tabela {
              se a chave estiver contida na substituicao {
                  apareceu = verdadeiro;
                  break;
              }
          }
          se apareceu é falso { retorna chave }
      }
}
```

Agora que temos todos os dados organizados em uma tabela e já conseguimos encontrar a letra inicial, podemos começar a converter as sequências.

Para isso, pensamos em um algoritmo que recebe uma *string* e a *tabela de substituições*. Nele, a *string* seria iterada e para cada *letra*, uma consulta na *tabela* seria feita, encontrando assim a *sequência* que ela faz referência. Com isso, a *letra* seria substituída, na *string,* por essa *sequência*, que por sua vez seria parâmetro de uma chamada recursiva, deixando assim tudo convertido. Esse algoritmo seria parecido com esse:

```
funcao anticompressor(str, tabela) {
      para cada (letra) da str {
            subs = tabela.get(letra);
            se subs != null {
                  str.troca(letra, anticompressor(subs, tabela));
            }
      }
      return str;
}
```

Dessa forma, podemos fazer a primeira chamada desse algoritmo com a sequência referente a letra inicial, sendo passada como parâmetro. No final, teríamos essa sequência com todos os seus caracteres substituídos, inclusive aqueles que vieram das outras letras, com isso, podemos imprimir o tamanho dessa string, dando o resultado. Exemplo de primeira chamada com a letra inicial:

```
anticompressor("memimomu", tabela);
```

Esta ideia, após implementada, demonstrou-se muito ineficiente no decorrer dos casos de teste. O primeiro demostra-se relativamente demorado para ser resolvido e já a partir do segundo, apresenta erro por falta de memória.

Levantamos algumas possibilidades de o motivo desses problemas acontecerem. Um deles pode ser pelo número de substituições repetidas que o algoritmo faz, por exemplo se temos “eee” ele faz três vezes a conversão da sequência que representa o “e”, o que gera muito trabalho repetido. O outro levantamento foi que lidar com *strings* gigantescas, principalmente concatenando-as consome muita memória, isso se dá, pois, *strings* são imutáveis e ao concatená-las, uma nova alocação de memória tem de ser feita, onde vai ser copiado as duas *strings* concatenadas. Portanto, seguiremos com outra ideia para o problema.

3. Segunda Solução

Nessa proposta, tentaremos otimizar a ideia anterior, para que o algoritmo não precise fazer mais de uma transformação da mesma sequência. No decorrer das conversões iríamos salvando os valores já processados. Para que essa ideia ocorra bem, precisaríamos também encontrar a ordem certa para iterar sobre as letras, segue o exemplo dado no enunciado na ordem ideal, onde a letra inicial “a” está por último:

m
o
i mooo
u mimimi
e mimomu
a memimomu

Percebemos, que ter uma ordem de iteração na hora de fazer as conversões poderia nos ajudar a diminuir muito trabalho. A ideia é a seguinte: dada uma letra com sua substituição, colocamos acima dela todas as outras letras que necessita para ser convertida. Isso significa que quando fossemos decifrar uma sequência, todas as substituições necessárias já estariam em seu estado final, ou seja, com todas as conversões feitas. Dessa forma, podemos salvá-las a medida em que iteramos sobre a tabela.

A forma que encontramos para chegar nessa ordem descrita acima, foi de usar um algoritmo igual aquele que encontra a primeira letra na solução 1. Em outras palavras, aquele algoritmo encontra a letra que não tem ocorrências na tabela, o que significa que mais nenhuma outra sequência precisa dela para ser convertida. Então, pensamos no seguinte: a medida em que identificamos letras sem ocorrências nas substituições, retiramo-las da tabela original e inserimos em outra, que será a ordenada. O algoritmo que ordena a tabela seria parecido com o pseudocódigo abaixo:

```
funcao ordenaTabela(tabela) {
      tabelaOrdenada = [];

      prox = getLetraSemOcorrencia(tabela);
      enquanto (prox != null) {
            tabela.remove(prox);
            tabelaOrdenada.add(0, prox);
            prox = getLetraSemOcorrencia(tabela);
      }
      retorna tabelaOrdenada;
}
```

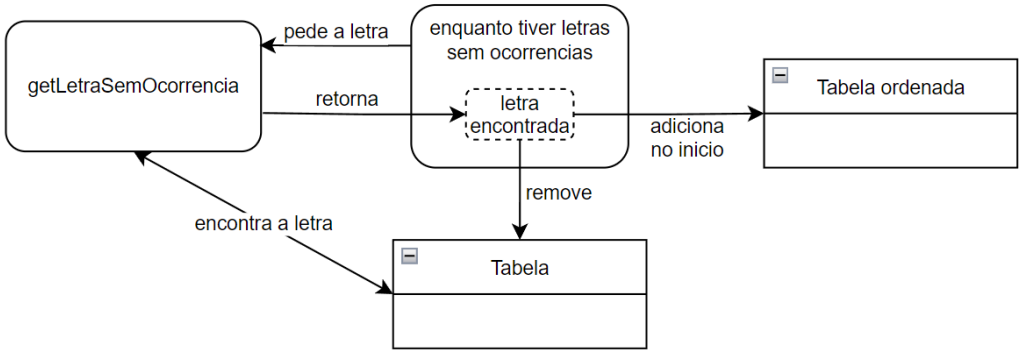

Figura 1. Diagrama do algoritmo de ordenação

Agora que podemos descobrir a ordem, já podemos começar a converter as sequências, para isso pensamos da seguinte forma: iterar sobre cada *item (chave e valor)* da tabela ordenada, se tiver *valor*, passamos por cada *letra* dele, verificando se existe referência para alguma *sequência*, se sim, a *letra* é substituída por essa *sequência* no *valor*. No fim da iteração do *valor*, armazenamos a nova *sequência* gerada para aquela *chave*, que será usado nas próximas conversões.

```
para cada (chave, valor) da tabela {
      se valor != null {
          para cada letra do valor {
               String subs = tabela.get(letra);
               se subs != null {
                   valor = valor.troca(letra, subs);
               }
          }
       }
      tabela.setValor(chave, valor);
}
```

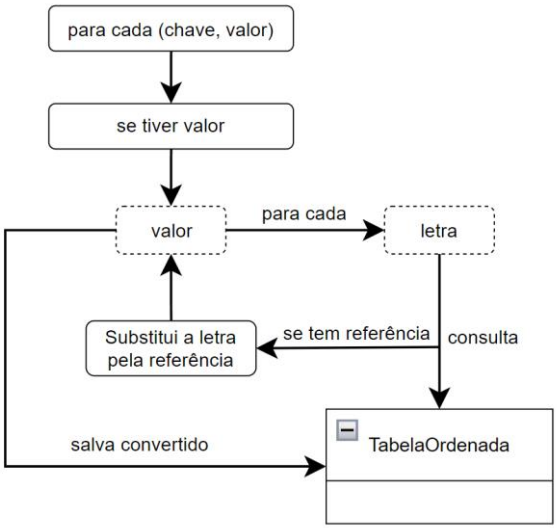

Figura 2. Diagrama algoritmo anticompressor da solução 2

Note que esse algoritmo, não precisa fazer chamadas recursivas como na solução anterior, mas por causa disso, a tabela fornecida deve estar na ordem que foi apresentada aqui, para que gere os resultados corretos.

No fim, optamos por não implementar esta solução, por motivos de não resolver todos os problemas da anterior, e inclusive piora um deles. Esta ideia pode até terminar com o processamento repetido de substituições, o que não deixa de ser ruim, porém acaba usando mais memória ainda, pois guarda todas as *strings* convertidas, enquanto a anterior, apenas a da letra inicial. Isso quer dizer, que se a solução 1 já apresentou problemas com memória, então prevemos que a solução 2 também terá esta ocorrência. Portanto, com o mesmo pensamento dessa ideia, proporemos a seguir uma solução com um detalhe diferente.

4. Terceira Solução

Como sabemos que manipular as *strings*, geradas pelos casos de teste, geram erros de memória, seguiremos com uma alternativa diferente. Analisando de novo e mais a fundo o enunciado, percebemos que na verdade não precisamos descobrir a string final em si, mas apenas o tamanho dela. Com isso, poderíamos apenas guardar o tamanho das conversões, em vez das *strings* gigantescas.

Com a mesma ideia da solução anterior, antes de começar a fazer as substituições, devemos ordenar a tabela. Dessa forma, quando fossemos contar o tamanho de uma sequência, as suas referências já estariam contadas, só precisando somar no tamanho e, após convertido, salvar esse valor para ser usado nas outras conversões.

Então, para descobrir o tamanho da letra inicial pensamos em ter uma outra tabela que guarda a letra e o tamanho da sua sequência, e ao decorrer do algoritmo, armazenamos as sequências já contadas nela. Como isso, seguiremos com a seguinte lógica: iterar sobre cada *item(chave, valor)* da *tabela ordenada*, se tiver um valor, passamos por cada *letra* dele, verificando se a *tabela de contados* tem o tamanho da letra referenciada, se sim é somado em uma *variável*, que será, no final da iteração, o tamanho *da sequência daquela letra*. Vale ressaltar, que no caso da *tabela de contados* não ter referência para alguma letra, quer dizer que ela não possui substituição, por isso, será somado 1 unidade no tamanho da letra atual.

```
tabContados;
para cada (chave, valor) da tabelaOrdenada {
    se valor != null {
        long tamanho = 0;
        para cada (letra) do valor {
            // se possui valor armazenado para a letra
            se tabContados.possui(letra) {
                tamanho += tabContados.getValor(letra);
            } senão {
                // senão é a própria letra, tamanho 1
                tamanho += 1;
            }
        }
        tabContados.adiciona(chave, tamanho);
    }
}
```

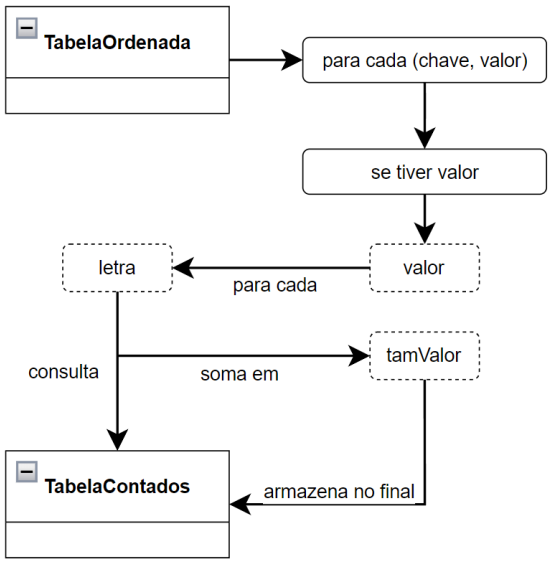

Figura 3. Diagrama do algoritmo anticompressor da solução 3.

Depois disso, teremos o tamanho de todas as sequências da tabela. Para identificar a letra inicial basta olhar para o maior número, ou mesmo, o último resultado, pois como a letra inicial não tem ocorrências na tabela, ela será convertida por último.

5. Resultados

Resultados obtidos a partir dos casos de teste, executados no algoritmo implementado a partir da ideia proposta na primeira solução:

Obs.: os casos sem resultados obtiveram erro por falta de memória.

Caso	Letra Inicial	Resultado	Tempo
1	f	17168630	7,1 s
2	w	-	-
3	r	-	-
4	m	-	-
5	k	-	-
6	v	-	-
7	x	-	-
8	d	-	-
9	k	-	-
10	o	-	-

Tabela 3. Resultados primeira solução

Resultados obtidos a partir dos casos de teste, executados no algoritmo implementado a partir da ideia proposta na terceira solução:

Caso	Letra Inicial	Resultado	Tempo
1	f	17168630	0,01 s
2	w	1625605381	0,02 s
3	r	364622462116	0,02 s
4	m	1026230712124	0,02 s
5	k	577049844147	0,02 s
6	v	4379846435596106	0,03 s
7	x	3855388134526119574	0,02 s
8	d	324439189762191	0,02 s
9	k	47528980167033010	0,02 s
10	o	2893870896418341711	0,02 s

Tabela 4. Resultados terceira solução

6. Conclusão

Apesar das duas primeiras soluções não terem sido eficientes para resolver o problema, elas foram de extrema importância para a terceira e última alternativa. Mais especificamente, a primeira solução nos mostrou alguns problemas da estratégia que até então estávamos usando, que era de concatenar as *strings*. Já a segunda, mesmo não implementada, foi onde gastamos um bom esforço para pensar em uma solução que ordenasse a tabela, e acabou sendo usada para a nossa solução final.

Portanto, acreditamos ter apresentado uma alternativa agradável ao problema proposto. Uma solução que se demonstrou eficiente mesmo no decorrer dos testes maiores, não apresentando diferenças nos tempos de execução.