O anticompressor de textos

Anderson R. P. Sprenger

Escola Politécnica – Pontifícia Universidade Católica do Rio Grande do Sul
Av. Ipiranga, 6681 – 90.619-900 – Porto Alegre – RS – Brasil

anderson.sprenger@edu.pucrs.br

Resumo: Este artigo apresenta uma solução para a busca do tamanho final de sequências de letras obtidas por um *anticompressor* de textos, que recebe como entrada uma tabela de substituições, e gera uma sequência de letras resultante das substituições da tabela em uma letra inicial a ser encontrada. É incluso uma descrição detalhada do problema, sua análise e uma proposta de solução. A proposta traz uma descrição e apresentação dos algoritmos da implementação de uma busca recursiva do tamanho final da substituição em um vetor de objetos, representando cada linha da tabela. Para tal, foram testados 11 casos fornecidos e analisados os resultados obtidos. Este artigo é referente ao trabalho 1 da disciplina de Algoritmos e Estrutura de Dados 2 da Pontifícia Universidade Católica do Rio Grande do Sul.

Problemática

O enunciado do trabalho [1] descreve a frustração de uma pessoa com a tecnologia de compressão de textos, e que em decorrência disto ela resolve desenvolver um anticompressor de textos. Para tal, o anticompressor substitui uma letra do alfabeto por uma sequência de letras, repetindo o processo, em cada letra da nova sequência, até não ser mais possível substituir nenhuma.

Existem diversos cenários que podem ser aplicados no anticompressor. Para cada cenário é apresentada como entrada uma tabela de substituições, havendo em cada linha uma letra e na maioria dos casos, uma sequência para substitui-la. A Tabela 1 demonstra uma possível entrada, onde a letra “a” é substituída por “memimomu”, e as letras “o” e “m” não são substituídas por nenhuma sequência.

a	memimomu
e	mimomu
i	mooo
u	mimimi
o	
m	

Tabela 1: Exemplo de tabela de substituições apresentada no enunciado.

Como saída, é solicitado: a letra que iniciará as substituições; a quantidade de letras que haverá na sequência final; e o tempo de execução para cada caso de teste. Segundo o problema [1], no exemplo acima, a primeira letra é “a”, e a sequência final tem 47 caracteres.

mmmoooomommmoooommoooommoooommoooomommmoooommoooommooo

Sequência final resultante da tabela 1.

Análise

Analisando o problema, é possível deduzir a existência de uma relação de dependência entre as letras para a formação da sequência final. Analisando a letra “a”, é necessária, entre outras, da sequência da letra “e”, que também precisa da sequência de outras letras para formar sua própria.

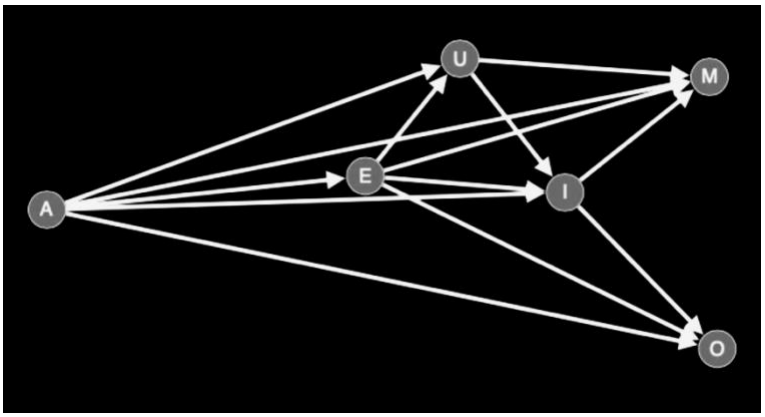

Imagem 1: Representação gráfica das letras do caso do enunciado e suas dependências.

Este cenário é conhecido como um grafo de dependências, um grafo acíclico direcionado, representando as dependências de vários objetos entre si. Este tipo de grafo é direcionado, porque as relações representadas pelas arestas seguem o sentido de sua dependência. Ele é acíclico, porque caso haja um ciclo – por exemplo, a letra “a” depende de “b”, e a letra “b” depende de “a” – ao tentar juntar suas sequencias resultaria em um *loop* infinito, sendo assim impossível determinar uma sequência de operações para a resolução da dependência.

Solucionática

No grafo representado na Imagem 1, observa-se que não existe nenhuma aresta que aponte para letra inicial do caso. Isso decorre na seguinte hipótese: se isso é a condição, ou uma das condições para encontrar a letra inicial.

De modo a validar a hipótese, foi desenvolvido um programa que carrega a tabela de substituições, e armazena ela em um objeto denominado Letra, que contêm:

- A letra a ser substituída;
- A sequência de caracteres que substituirão a letra;
- Um conjunto das letras que contêm a letra do objeto em sua sequência;
- E um conjunto das letras contidas na sequência de substituição do objeto.

```
public class Letra {
    public let caractere: Character
    public let substituto: String?

    public var contem: Set<Letra>
    public var contidaEm: Set<Letra>

    public init(caractere: Character, substituto: String? = nil) {
        self.caractere = caractere
        self.substituto = substituto

        self.contem = []
        self.contidaEm = []
    }
}
```

Imagem 2: Implementação do objeto descrito, que representa uma letra da tabela de substituições.

Para carregar as letras do arquivo caso, o programa lê cada linha do arquivo, verifica se há um caractere e a sequência. Caso haja só o caractere, é criado uma instância de *Letra*, com a variável *substituto* como nula. Caso a linha tenha também a sequência, a variável *substituto* armazena a sequencia.

Após isso, são carregadas as relações de dependência de cada letra. Verifica-se para cada elemento a no conjunto das letras, se existe um elemento b do conjunto de sua substituição, neste caso:

- a letra b contida na sequência é inserida no conjunto das letras que contêm a sequência da letra a;
- e a letra a que contêm a sequência é inserida no conjunto das letras em que a letra b está contida dentro da sequência.

```
func encontraRaiz() -> Letra {
    for letra in letras {
        if letra.contidaEm.isEmpty && !letra.contem.isEmpty {
            return letra
        }
    }

    fatalError("raiz não encontrada!")
}
```

Imagem 3: Algoritmo que carrega as relações de dependência entre as letras do caso.

A complexidade do algoritmo acima é: $n^2 * l$, onde n é o número de letras do caso de teste, e l é o tamanho da sequência de substituição. Isto porque há dois comandos *for* encadeados percorrendo o conjunto de letras, e a execução da função *contains* que, segundo a documentação da linguagem utilizada [2], possui complexidade O(n), onde n é o tamanho da *String* de entrada. Logo, a complexidade é O(n^2), sendo n o tamanho do conjunto de letras.

De modo a testar os algoritmos acima, foi carregado o caso do enunciado e impresso o conteúdo do conjunto das letras na área de *debug*. A saída, mostrada na Imagem 4, foi conforme o esperado, validando o teste de execução.

```
a --> Substituto: memimomu
Contem as letras: i,o,e,m,u
Esta contido em:
e --> Substituto: mimomu
Contem as letras: u,m,i,o
Esta contido em:  a
i --> Substituto: mooo
Contem as letras: o,m
Esta contido em:  a,e,u
u --> Substituto: mimimi
Contem as letras: m,i
Esta contido em:  a,e
o --> Substituto: nil
Contem as letras:
Esta contido em:  a,i,e
m --> Substituto: nil
Contem as letras:
Esta contido em:  u,e,a,i
```

Imagem 4: Saída da área de debug, com o conteúdo dos elementos do conjunto de letras carregado a partir do caso do enunciado, note que a letra "a" não está contida em nenhuma outra letra.

Após o teste, foi procurado no caso de teste 1 por letras cujo conjunto de letras *contidaEm* era vazio, sendo encontrado duas letras: "f" e "s". Porém, a letra "s" não tem uma sequência de substituição, logo se ela for a letra inicial não haveria nenhuma substituição. Consequentemente, é deduzível que letra inicial deve ter uma sequência de substituição atrelada ao seu caractere.

Ao filtrar as letras de todos os casos por letras com substituição que não estejam na substituição de nenhuma outra letra, é encontrada uma única letra para cada caso, conforme na tabela a seguir.

Caso	Letra
0	a
1	f
2	w
3	r
4	m
5	k
6	v
7	x
8	d
9	k
10	o

Tabela 2: Letras iniciais encontradas para cada caso disponibilizado com o trabalho, foi incluso o caso do enunciado como caso 0.

Agora, resta testar a aciclicidade do grafo de dependências formado pela substituição das letras da Tabela 2. Para tal, foram encontradas duas soluções:

1. construir uma estrutura de dados em grafo, e executar algum algoritmo que identifique ciclos dentro do grafo, dentre eles o algoritmo de Tarjan para componentes fortemente conectados [3], ou;
2. assumir *a priori* que o grafo é acíclico e tentar encontrar o tamanho da sequência final, analisando se há algum *loop* infinito na execução, o que caracterizaria a ciclicidade.

Por conveniência, e pela simplificação do problema, foi escolhida a segunda opção. Para tal, foi criada uma variável chamada *tamanho* no objeto *Letra*, para salvar o resultado do cálculo, não precisando recalculá-lo em cada ocorrência da letra nas sequencias de um caso de teste.

No algoritmo da solução, primeiro é verificado se a variável de tamanho da letra é ou não nula, pois caso não seja, o algoritmo já foi executado e esta variável é retornada. Depois, é verificado se a variável do substituto é nula, caso seja é retornado 1, porque este é o espaço que a letra ocupara na sequência final. Caso contrário, é calculado o tamanho final da sequência.

Para tal, primeiro é carregado o tamanho das sequencias de cada caractere da sequência da letra. Para armazenar isso, é criado um dicionário com um caractere como chave, e o tamanho final da sequência de uma letra como valor.

Então, o conjunto *contem*, cujo é referente as letras que contêm a sequência da letra do objeto, é iterado, e o tamanho das substituições de cada caractere é obtido por chamadas recursivas deste algoritmo, tendo seu valor inserido no dicionário, com o caractere do elemento por chave. Então é calculada a soma do tamanho da sequência, através do somatório dos valores do dicionário para cada caractere da sequência.

```
public func encontraTamanho() -> Int {
    ///Se já encontrou o tamanho, return tamanho.
    if let tamanho = self.tamanho {
        return tamanho
    }

    ///Se não houver substituto, return 1.
    guard let substituto = substituto else {
        return 1
    }

    var tamanhoLetras = [Character: Int]()
    var tamanho = 0

    for letra in contem {
        tamanhoLetras[letra.caractere] = letra.encontraTamanho()
    }

    for caractere in substituto {
        tamanho += tamanhoLetras[caractere]!
    }
    self.tamanho = tamanho
    return tamanho
}
```

Imagem 5: Implementação do algoritmo que busca o tamanho final da sequência da letra.

Considerando que o calculo é executado uma vez para cada letra do conjunto de substituições, a complexidade deste algoritmo é O(n), onde n é o numero de letras no conjunto de letras que o substituto do objeto contem.

Ao testar 11 diferentes casos de teste, incluindo o presente no enunciado, foi possível obter o tamanho da sequência final das letras em todos os casos, provando também que eles são acíclicos.

Caso	Letra	Tamanho	Tempo
0	a	47	0.12s
1	f	17.168.630	1.10s
2	w	1.625.605.381	0.94s
3	r	364.622.462.116	1.34s
4	m	1.026.230.712.124	1.57s
5	k	577.049.844.147	1.18s
6	v	4.379.846.435.596.106	1.72s
7	x	3.855.388.134.526.119.574	1.60s
8	d	324.439.189.762.191	1.42s
9	k	47.528.980.167.033.010	1.14s
10	o	2.893.870.896.418.341.711	1.59s

Tabela 3: Tamanho final da sequência de substituição para cada letra inicial dos casos de teste, com o tempo de execução da solução em cada caso.

Conclusões:

Por consequência da solucionática, é possível concluir que a hipótese em que a letra inicial de um caso não deve estar contida na substituição de outra letra, e a dedução que a letra inicial também deve conter alguma letra, não contradizem nenhuma premissa do universo discurso da problemática.

Ademais, foi encontrada uma única letra nestas condições para cada caso, sendo possível também calcular o valor de sua substituição sem a ocorrência de um *loop* infinito para todos os casos disponíveis com o trabalho,

provando que todos os grafos derivados das substituições das letras iniciais encontradas se enquadram como um grafo de dependências. Outro fato que corrobora a hipótese, é que ao analisar o caso do enunciado, os dados obtidos são iguais aos esperados para o caso de teste do enunciado.

O tempo de execução dos algoritmos são sujeitos a interferências externas, como sistema operacional, aplicativos abertos, entre outros. Contudo, ao considerar que: duas das sequencias finais tem um tamanho final na ordem dos quintilhões (10^{18}); que uma representação desta sequência de letras codificada em UTF-8, representando uma letra em 1 byte, ocuparia cerca de alguns exabytes (10^{18} bytes); e que o tempo de execução da solução para estes casos, foi menor que 2 segundos; é possível considerar adequados os algoritmos implementados nesta solução.

Destarte, por não haver contradições ao universo discurso, e por terem sido encontrados resultados em todos os casos do enunciado, é sensato considerar validos os resultados obtidos na Tabela 3. Sendo assim, atingindo os objetivos deste trabalho.

Referencias Bibliográficas:

1. OLIVEIRA, João Batista Souza de. O anticompressor de textos, 2022;
2. APPLE (Cupertino, CA). String.contains(_:). In: Apple Developer Documentation. [S. l.], 2022. Disponível em: https://developer.apple.com/documentation/swift/string/2893238-contains. Acesso em: 26 abr. 2022.
3. TARJAN, Robert Endre. Depth-first seach and linear graph algorithms, 1972. Disponível em: https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.327.8418. Acesso em: 26. Abr. 2022.