Prova 1 Algoritmos II

Rita Rezende Borges de Lima
Matrícula: 2020065317

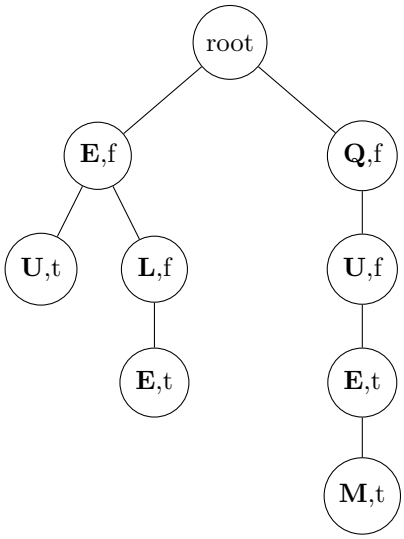
Julho 2021

1. Para o seguinte problema assumi que o objetivo era armazenar e verificar se uma dada string é uma stopword visto que o limite assintótico é da ordem do tamanho desta. Uma maneira de armazenar as stopwords é com uma trie, ou árvore de prefixos. Primeiro adicionaremos todas as stopwords em nossa estrutura, de modo que os vértices desta contenham tuplas onde o primeiro elemento é um caracter e o segundo um booleano dizendo se ali é o fim de alguma stopword. Cada nó possuirá a princípio A filhos apontando para null onde A é o tamanho do alfabeto. Dessa forma, tendo mapeado cada filho para um simbolo do alfabeto, na hora de verificarmos se um certo nó já contém um filho com a transição desejada, esta operação será de complexidade constante.

   Se uma certa palavra de tamanho k é uma stopword precisamos apenas percorrer a árvore de prefixo de acordo com cada caracter da palavra lido da esquerda para a direita onde cada caracter será pareado com um nível da árvore. Três casos podem ocorrer:

   - Caso cheguemos em um nó da stopword onde não existe uma transição para um nó filho que contenha o caracter buscado (o nó mapeado aponte para null) a palavra não é uma stopword e a execução é terminada.
   - Caso a leitura da palavra termine em um nó que não tenha o valor verdadeiro no segundo valor da tupla (que determina se este é o fim de uma stopword) a palavra lida também não é uma stopword.
   - Caso a palavra seja lida até o final e a leitura termine em um nó marcado como verdadeiro no segundo elemento da tupla, a palavra lida é uma stopword.

   Árvore de prefixo de stopwords com alguns pronomes adicionados: *que, quem, eu, ele.* (Não foram adicionadas transições apontando para null com o intuito de proporcionar uma melhor visualização).

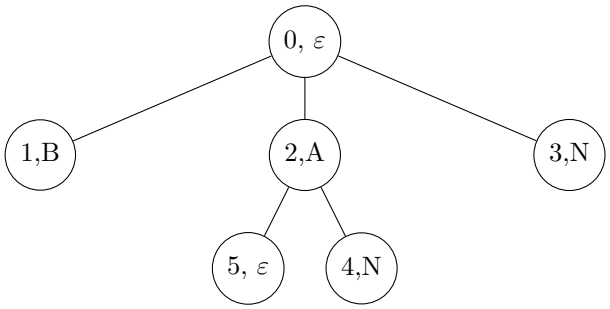


   Considerando apenas o tempo de leitura e verificação se a palavra lida é uma stopword o algoritmo descrito tem complexidade O(k) onde k é o tamanho da stopword. É fácil observar esse limite assintótico pois durante a leitura dos k caracteres iremos percorrer cada nível da árvore apenas uma vez e o programa será encerrado após essa leitura.

2. O algoritmo para realizar a compressão descrita pode ser feito por intermédio de uma árvore de tuplas. A string de entrada é percorrida no sentido esquerda direita, de $i = 0...i < len(string)$, utilizaremos uma variável auxiliar k inicializada com 1 no início que irá ser incrementada cada vez que um nó for adicionado na árvore. Para cada cada caracter é verificado se este é um dos nós filhos de nosso nó referência, R, este inicialmente é a raiz da árvore, que é inicializada com a tupla $< 0, \varepsilon >$, os seguintes casos podem ocorrer:

   - Caso o caracter seja o valor de algum dos nós filhos de R: O ponteiro R passa a ser este nó filho. Se a cadeia de entrada tiver chegado ao fim, adicionamos um novo nó filho a R que conterá a tupla $< k, \varepsilon >$.
   - Caso o caracter não seja o valor de nenhum dos nós filhos de R é adicionado um nó filho de valor $< i, string[i] >$, também é adicionado no final do texto comprimido a tupla $< indexR, string[i] >$, por fim o nó de referência passa a ser a raíz.

   Exemplo de Compressão para a string BANANA:



   Texto comprimido: $\{< 0, B >, < 0, A >, < 0, N >, < 2, N >, < 2, \varepsilon >\}$

   Para a descompressão basta indexar cada uma das tuplas em uma tabela hash e percorrer esta do fim ao início adicionando sempre o elemento textual da célula e o elemento textual da célula pai (célula que tem como index o número armazenado no primeiro elemento da tupla). Esta operação é feita para cada uma das células da hash recursivamente até que o caso base, index = 0, seja atingido.

   No caso do exemplo dado previamente iniciamos em i = k, que no caso é 5. Encontramos ε, depois seguimos para a posição 2 na tabela que contém A, vamos para a posição 0 que contem ε e terminamos a recursão pois chegamos no caso base. i = 5 - 1 = 4, encontramos N, seguimos para a posição 2 na tabela que contém A, vamos para a posição 0 que contem ε e terminamos a recursão pois chegamos no caso base. i = i - 1, e assim sucessivamente até i = 0. Por fim encontraremos a string εAεNAεNεAεBε. Para armazenarmos a string sem esta estar invertida basta armazena-la em uma pilha durante o procedimento.

Index	Tuple
0	$< 0, \varepsilon >$
1	$< 0, B >$
2	$< 0, A >$
3	$< 0, N >$
4	$< 2, N$
5	$< 2, \varepsilon >$

   A complexidade do algoritmo descrito é da ordem de O(n) na compressão, afinal uma iteração acontece para cada caracter. Já a descompressão itera uma vez para cada um dos k itens da tabela hash e para cada uma dessas retrocede em até l onde l é o tamanho do diâmetro da árvore gerada na etapa de compressão tornando a complexidade de orden $O(k * l)$.

3. Uma maneira de verificar se existe uma reta que divida os dois tipos de ponto é encontrar uma envoltória convexa para cada um dos dois grupos de pontos de rótulos distintos. Depois basta verificar se algum dos segmentos da primeira envoltória faz interseção com um segmento da segunda envoltória, caso exista interseções não é possível separar os conjuntos de pontos com uma reta. Para gerar ambas as envoltórias podemos utilizar o algoritmo incremental v2 cujo custo total é de ordem $O(n * log(n))$. Para verificar se existem interseções em um conjunto de segmentos podemos utilizar o sweep line que também tem complexidade $O(n * log(n))$. Desta forma a complexidade total do algoritmo também é de forma $O(n * log(n))$.

4. a) Uma estrutura que nos permite encontrar tais intervalos em $O(log(n) + k)$ é uma árvore de intervalos. Para a construção dessa estrutura inicialmente selecionamos o ponto médio dos limites dos intervalos, M, depois encontramos todos os intervalos que contenham M e armazenamos estes no vértice em dois vetores, um ordenado pelo limite inferior de cada intervalo e o outro ordenado pelo limite superior. Para os intervalos que tem limite inferior maior que o ponto médio, M, adicionamos um vértice a direita do nó atual e repetimos o processo descrito previamente. Para intervalos que possuem limite superior menor que M adicionamos um vértices a esquerda do nó atual e repetimos o processo recursivamente até que tenhamos n folhas para os n intervalos.

   b) Para encontrar os intervalos que contém o valor X inicialmente percorremos a árvore para cada um de seus $log(n)$ níveis. Para cada nível comparamos o valor de X com o valor armazenado ali (o ponto médio M calculado no preprocessamento). Temos os seguintes casos:

      i. X é igual a M: todos os intervalos do nó são adicionados a resposta e a execução é finalizada.
      ii. X é maior que M: verificamos os primeiros valores do vetor de intervalos do nó (que foi ordenado pelo limite superior dos intervalos) até que este seja maior que X, isto resultará em k_i intervalos que contém X e estes serão adicionados a resposta.
      iii. X é maior que M: verificamos os últimos valores do vetor de intervalos do nó (que foi ordenado pelo limite inferior dos intervalos) até que este seja menor que X, isto resultará em k_i intervalos que contém X e estes serão adicionados a resposta.

   Por fim, depois de cada nível ter sido percorrido, teremos os k intervalos da resposta, afinal $\mathrm{k} = \sum_{i=0}^{log(n)} \mathrm{k}_i$. Como só percorremos um nó por nível e percorremos cada intervalo apenas uma vez dentro das operações de cada vértice a complexidade final será de forma $O(log(n) + k)$.