

Exercício-Programa 2

Lucas Magno
7994983

1 Introdução

Este trabalho consiste em, a partir de um *dicionário*, um arquivo com sequência de palavras, determinar um maior conjunto de anagramas dentre tais palavras. Um dicionário tem a forma:

```
A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
:
zymotoxic
zymurgy
Zyrenian
Zyrian
Zyryan
zythem
Zythia
zythum
Zyzomys
Zyzzogeton
```

Listing 1: Exemplo de dicionário

Para tal foi desenvolvido um programa em C, cujos detalhes serão discutidos a seguir. Também foi implementado um gerador de dicionários, a fim de testar o programa, discutido mais adiante.

2 Algoritmo

O algoritmo para encontrar um maior conjunto de anagramas consiste em, para cada string de um dado dicionário, calcular uma *hash* (tal que todos os anagramas de uma palavra têm a mesma *hash*) e colocar a string na fila do nó correspondente a tal *hash* em uma árvore binária de busca. Ao longo do processo, basta manter um registro de uma maior fila encontrada até então e ao final teremos uma maior fila, um maior conjunto de anagramas, do dicionário.

Devido à necessidade de se verificar um grande número de strings contra determinadas chaves (*hashes*), algoritmo faz uso de uma árvore binária de busca B , que permite tal busca em tempo logarítmico, cujas chaves e valores de cada nó são

Chave: um vetor **letters** de 26 inteiros, onde cada elemento representa a contagem de uma determinada letra em uma palavra.

letters[0] é o número de vezes que a letra a aparece,

\vdots

letters[25] é o número de vezes que a letra z aparece.

Valor: uma fila de strings, tais que todas têm a mesma contagem de letras que a chave.

Onde a comparação entre duas chaves l_1 e l_2 é feita da seguinte forma:

$$\begin{cases} l_1 = l_2, & \text{se } l_1[i] = l_2[i], \text{ para todo } i = 0, \dots, 25 \\ l_1 < l_2, & \text{se } l_1[j] = l_2[j], j = 0, \dots, i-1 \\ & \text{e } l_1[i] < l_2[i] \text{ para algum } i = 0, \dots, 25 \\ l_1 > l_2, & \text{se } l_1[j] = l_2[j], j = 0, \dots, i-1 \\ & \text{e } l_1[i] > l_2[i] \text{ para algum } i = 0, \dots, 25 \end{cases}$$

Assim, ao irmos adicionando as strings lidas do dicionário D às filas de cada chave correspondente, garantimos que cada uma dessas filas é um conjunto de anagramas, pois contém strings com exatamente as mesmas letras. Portanto, após ler todo o dicionário, uma fila com maior tamanho é um maior conjunto de anagramas em D . A seguir o pseudocódigo do programa implementado.

Entrada: D dicionário

Saída: Fila com as strings que pertencem a um maior conjunto de anagramas em D

$B \leftarrow$ árvore binária de busca vazia;

$qmax \leftarrow NULL$;

enquanto não chegou ao final de D **faça**

 ler uma string s de D ;

$ls \leftarrow$ a forma *lowered* de s ;

// letras minúsculas

$letters \leftarrow$ um vetor com a contagem das letras em ls ;

se $letters$ está em B **então**

 | adiciona s à fila correspondente a esta chave;

senão

 | cria um nó correspondente a esta chave em B e coloca s na nova fila;

fim

$q \leftarrow$ a fila correspondente a $letters$ em B ;

se $\text{length}(q) > \text{length}(qmax)$ **então**

 | $qmax \leftarrow q$;

fim

fim

retorna $qmax$;

Algorithm 1: Busca por um maior conjunto de anagramas em um dicionário.

3 Gerador

Para testar o programa, também foi implementado um gerador de dicionários, baseado na função `npermutations`, que gera as n primeiras (ou todas, se não existirem mais que n) permutações únicas em ordem lexicográfica de um conjunto de letras, determinado por uma sequência `letters` de letras.¹

A função funciona recursivamente, gerando permutações únicas de subsequências de `letters` e as concatenando com cada letra única restante, como descrito no algoritmo a seguir.

Entrada:

`letters` uma sequência de letras;

`n` um inteiro.

Saída: As n primeiras permutações únicas de `letters` em ordem lexicográfica.

$P \leftarrow$ uma fila de strings vazia;

se `length(letters) = 1` **então**

$s \leftarrow$ a única letra em `letters`;

`enfileira(P, s)`;

retorna P ;

fim

para cada letra única c em `letters` em ordem alfabética **faça**

$Q \leftarrow \text{npermutations}(\text{letters} - c, n)$; // remove uma instância de c em `letters`

enquanto `length(Q) > 0` **faça**

$s \leftarrow \text{desenfileira}(Q)$;

`enfileira(P, c+s)`; // concatenação

se `length(P) = n` **então**

retorna P ;

fim

fim

fim

retorna P ;

Algorithm 2: Função `npermutations`.

A partir desta função podemos construir de fato o gerador de dicionários, que consiste em:

1. Gerar uma contagem de letras `letters` aleatória com um dado tamanho máximo `wordlen`;
2. Calcular um dado número `setlen` de permutações dessas letras e salvar como o maior conjunto de anagramas `qmax`;
3. Inserir `qmax` em uma árvore binária de busca `B` com a chave `letters`;
4. Enquanto ainda precisarmos gerar mais palavras sortear outra contagem de letras `letters` e, se esta não está em `B`, calcular um número menor que `setlen` de permutações e as salvar em `B` com a chave `letters`.

A seguir é dado o algoritmo mais detalhadamente.

¹A implementação é um pouco diferente, utilizando um vetor `letters` de contagem como definido anteriormente e mais argumentos, por questões de eficiência, mas a lógica é a mesma.

Entrada:

wordlen o tamanho máximo das palavras;
setlen o tamanho máximo dos conjuntos de anagramas;
dictlen o número de palavras no dicionário;
file o nome arquivo de saída.

Saída: Um dicionário com *dictlen* palavras de tamanho máximo *wordlen*, cujo único maior conjunto de anagramas *qmax* tem tamanho no máximo *setlen*, escrito no arquivo de saída *file*. Além disso, imprime *qmax* para a STDOUT.

```

B ← uma árvore binária de busca vazia;
letters ← lettersrandom(wordlen);
qmax ← npermutations(letters, setlen);
B[letters] ← qmax;                                // insere qmax em B com chave letters
setlen ← length(qmax);                             // o conjunto gerado pode ser menor que setlen
imprime(qmax);
enquanto length(qmax) > 0 faça
    | s ← desenfileira(qmax);
    | imprime(file, s);
fim
restantes ← dictlen − setlen;                        // número de palavras que ainda temos que gerar
enquanto restantes > 0 faça
    | len ← rand([1, wordlen]);                      // número aleatório no intervalo [1, wordlen]
    | letters ← lettersrandom(len);
    | se letters está em B então
    | | continua;
    | fim
    | m ← rand([1, min(restantes, setlen−1)]);
    | q ← npermutations(letters, m);
    | B[letters] ← q;
    | enquanto length(q) > 0 faça
    | | s ← desenfileira(q);
    | | imprime(file, s);
    | fim
    | restantes ← restantes − length(q);
fim

```

Algorithm 3: Gerador de dicionários.

Onde lettersrandom(*n*) é uma função que retorna uma contagem aleatória de letras com tamanho *n*.

4 Compilação e Uso

Para compilar os programas, basta executar os seguintes comandos:

Programa: `make`

Gerador: `make gen`

que criam os executáveis `EP2` e `gen`, respectivamente.

O programa `EP2` deve ser invocado da forma

```
./EP2 dicionario.txt
```

e o resultado é mostrado na tela.

Já o gerador é invocado na forma

```
./gen wordlen setlen dictlen output.txt
```

onde `wordlen`, `setlen` e `dictlen` são inteiros positivos como definido anteriormente e `output.txt` é o arquivo de saída onde será gravado o dicionário. O programa então imprime para a tela o maior conjunto de anagramas gerado.

5 Resultados

Para o dicionário `dicionario.txt` fornecido como exemplo, obtemos a seguinte saída

Biggest set of anagrams:

Queue (11 elements):

```
angor
argon
goran
grano
groan
nagor
Orang
orang
organ
rogan
Ronga
```

e cuja execução levou 0.27 s.

A seguir são dados outros exemplos gerados, que foram executados num computador com a seguinte configuração (saída do programa `neofetch`):

```

eeeeeeeeeeeeeeee
eeeeeeeeeeeeeeee
  eeeee eeeeeeeee eeeee
  eeee  eeeee      eee  eeee
  eeee  eeee       eee   eeee
eee   eee         eee    eee
eee  eee          eee     eee
ee   eee          eeee    eeee
ee   eee          eeeee   eeeee
ee   eee          eeeee  eeee ee
eee  eeee  eeeee  eeeee  eee
eee  eeeeeeeee  eeeee  eee
eeeeeeeeeeeeeeee eeeee
  eeeeeee eeeeeeeee  eeee
    eeeee                eeeee
      eeeeeee          eeeee
        eeeeeeeeeeeee
```

```
OS: elementary OS 0.4 Loki x86_64
Model: Z97-D3H
Kernel: 4.4.0-77-generic
Uptime: 14 hours, 21 mins
Packages: 2300
Shell: fish 2.2.0
Resolution: 1920x1080
DE: Pantheon
WM: Mutter(Gala)
Theme: Arc [GTK2/3]
Icons: Paper [GTK2/3]
Terminal: mc
CPU: Intel i5-4690K (4) @ 3.900GHz
GPU: NVIDIA GeForce GTX 750
Memory: 4915MiB / 15925MiB
```

5.1 `./gen 4 10 100000 output.txt`

Tempos de execução:

`gen:` 0.10 *s*

`EP2:` 0.04 *s*

Saída de `gen`:

Biggest set of anagrams:

Queue (10 elements):

dhsy

dhys

dshy

dsyh

dyhs

dysh

hdsy

hdys

hsdy

hsyd

Saída de `EP2`:

Biggest set of anagrams:

Queue (10 elements):

dhsy

dhys

dshy

dsyh

dyhs

dysh

hdsy

hdys

hsdy

hsyd

5.2 `./gen 8 10 1000000 output.txt`

Tempos de execução:

gen: 0.97 s

EP2: 0.63 s

Saída de gen:

Biggest set of anagrams:

Queue (10 elements):

aijlpuvz
aijlpuzv
aijlpvuz
aijlpvzu
aijlpzuv
aijlpzvu
aijlupvz
aijlupzv
aijlupvz
aijlupzv
aijlupzv

Saída de EP2:

Biggest set of anagrams:

Queue (10 elements):

aijlpuvz
aijlpuzv
aijlpvuz
aijlpvzu
aijlpzuv
aijlpzvu
aijlupvz
aijlupzv
aijlupvz
aijlupzv
aijlupzv

5.3 ./gen 20 20 1000000 output.txt

Tempos de execução:

gen: 0.98 s

EP2: 0.44 s

Saída de gen:

```
Biggest set of anagrams:
Queue (20 elements):
defggijjklmmorttwxyz
defggijjklmmorttwxzy
defggijjklmmorttwyxz
defggijjklmmorttwyzx
defggijjklmmorttwzxy
defggijjklmmorttwzyx
defggijjklmmorttxwyz
defggijjklmmorttxwzy
defggijjklmmorttxyzw
defggijjklmmorttxzwy
defggijjklmmorttxzyw
defggijjklmmorttywxz
defggijjklmmorttywzx
defggijjklmmorttyxwz
defggijjklmmorttyxzw
defggijjklmmorttyzwx
defggijjklmmorttyzxw
defggijjklmmorttzwxy
defggijjklmmorttzwyx
```

Saída de EP2:

```
Biggest set of anagrams:
Queue (20 elements):
defggijjklmmorttwxyz
defggijjklmmorttwxzy
defggijjklmmorttwyxz
defggijjklmmorttwyzx
defggijjklmmorttwzxy
defggijjklmmorttwzyx
defggijjklmmorttxwyz
defggijjklmmorttxwzy
defggijjklmmorttxyzw
defggijjklmmorttxzwy
defggijjklmmorttxzyw
defggijjklmmorttywxz
defggijjklmmorttywzx
defggijjklmmorttyxwz
defggijjklmmorttyxzw
defggijjklmmorttyzwx
defggijjklmmorttyzxw
defggijjklmmorttzwxy
defggijjklmmorttzwyx
```