Trabalho Prático 1 - Relatório
Manipulação de Sequências
DCC207 - Algoritmos II

Professor: Renato Vimieiro
Aluno: Caio Guedes de Azevedo Mota (2018054990)

Introdução

O relatório a seguir se refere ao primeiro trabalho prático da disciplina de Algoritmos 2, que aborda estruturas de dados e algoritmos de localização de sequências em textos. Especificamente, foi entregue um arquivo .fasta com o genoma mapeado do Sars-CoV-2, com o objetivo de encontrar a maior subcadeia (substring) que se repetia dentro dele. Além disso, era necessário utilizar uma árvore de sufixos como estrutura de dados por trás da solução.

Instruções de Execução

O programa principal está em *main.py*, e o programa pode ser executado em terminal da forma:

> python3 main.py [caminho do arquivo]

O caminho do arquivo deve ser substituído pelo sarscov2.fasta. É importante notar que o programa foi feito com o formato .fasta em mente, então ele sempre elimina a primeira linha de descrição e elimina as quebras de linha.

Alternativamente, pode ser feita a execução apenas com:

> python3 main.py

E ele utiliza o arquivo sarscov2.fasta que é incluso na pasta junto com os outros arquivos.

A execução do arquivo imprime uma lista no formato [substring, início, fim, ocorrências], sendo substring a maior substring repetida no texto do arquivo, início e fim os índices da primeira ocorrência do texto e ocorrências o número de ocorrências da substring no texto.

Pré-processamento e Construção da Árvore

Ao receber o texto do genoma, era necessário armazená-lo numa árvore de sufixos. Para isso, além de realizar um tratamento no texto original (removendo primeira linha, tirando quebra de caractere, adicionando caractere terminal $, etc.) foi utilizado um algoritmo para a construção dessa árvore, que será descrito a seguir.

A árvore consiste em um conjunto de nós conectados entre si. Cada nó representa uma substring de um dos sufixos do texto, e são armazenados o índice de início e de fim da

ocorrência dessa substring no texto. Além disso, cada nó contém uma lista de filhos que compartilham dessa mesma substring acumulada como prefixo. Na implementação, essa lista de filhos é um dicionário de python, onde as *chaves* são os caracteres iniciais de cada substring (já que, numa árvore de sufixos, não há filhos de um nó com o mesmo caractere inicial) e os *valores* são os nós de fato.

O algoritmo é ingênuo e adiciona cada um dos possíveis sufixos um a um. Inicia adicionando um nó raiz para a árvore, com índices de início e fim (-1, -1) para representar a raiz, e adicionando também o primeiro sufixo, que é o texto inteiro. Após isso, em cada iteração ele adiciona cada um dos sufixos, do maior até o menor, percorrendo a árvore e decidindo onde e como será adicionado.

Suponha que S = comprimento do texto original. A busca é feita num nó *base*, e inicialmente base = raiz. Primeiro, o algoritmo identifica o índice do primeiro caractere do sufixo a adicionar que ainda não tenha sido consumido por outros nós na busca (suponha que seja j). Existem 3 possibilidades:

1. Caso texto[j] *não esteja* em uma das chaves dos filhos da base, significa que não há mais prefixo em comum, logo adiciona esse sufixo como um novo nó filho do nó de busca atual. Seu índice de início é j e o final é S - 1.

2. Caso texto[j] *esteja* em uma das chaves dos filhos da base, significa que há um prefixo em comum. Agora, precisa-se verificar se ele é inteiro ou parcial. Primeiro, descobre-se k tal que k é o índice onde está o último caractere do prefixo em comum entre o filho do nó base indicado por texto[j] (chamado de filhoBase) e o sufixo a adicionar.
    2.1. Se o comprimento (k - j) for igual ao comprimento da substring dos índices de filhoBase, então a substring de filhoBase é um prefixo por inteiro do sufixo a adicionar. Logo, a busca deve reiniciar com base = filhoBase.
    2.2. Se o comprimento (k - j) não for igual ao comprimento da substring dos índices de filhoBase, então a substring de filhoBase é parcialmente um prefixo do sufixo a adicionar. Nesse caso, se faz um terceiro nó, com o prefixo compartilhado encontrado. Esse nó será filho da base, e terá dois filhos: filhoBase, com o prefixo comum retirado dos índices, e um novo nó com os índices do sufixo a adicionar, também sem o prefixo em comum. Agora, o terceiro nó representa o prefixo compartilhado entre os dois, e entra no lugar de filhoBase como filho da base.

Assim, ao terminar esse procedimento, a árvore de sufixos para o texto é construída com êxito.

Busca da Maior Substring Repetida

Tendo a árvore de sufixos pronta, a busca da substring é bem mais simples. Em uma árvore de sufixos, cada nó interno tem pelo menos dois filhos, já que os nós internos são prefixos compartilhados pelos sufixos. Logo, cada nó interno representa uma substring acumulada até então que é compartilhada por pelo menos dois sufixos do texto.

O algoritmo para buscar a maior substring explora disso e usa a noção de profundidade, nesse caso definida como o tamanho da substring acumulada até então em um dado nó interno. Basta encontrar o nó interno com maior profundidade na árvore.

É feita uma Depth-First Search (DFS) percorrendo todos os nós internos desde a raiz, mantendo o valor do nó com a maior profundidade e atualizando-o à medida que são encontrados nós internos com maior profundidade na busca. No final da busca, se tem o nó interno com a maior profundidade, e com ele é descoberto sua substring acumulada (caminhando até a raiz e adicionando os prefixos), e é calculado seu índice de ínicio baseado no índice de fim já armazenado no nó (detalhe: é obtido o índice de primeira ocorrência) e o número de ocorrências baseado no número de filhos do nó.

Resultados e Análise Empírica

Os testes foram feitos em uma máquina com as seguintes especificações técnicas:

- ➔ **Processador:** Intel Core i7-7500U @ 2.7 Ghz
- ➔ **Memória RAM:** 8GB
- ➔ **Memória Secundária:** SSD Kingston A400 120GB
- ➔ **Sistema Operacional:** Linux

Os testes foram feitos em um arquivo separado (main_testing.py) onde foram importadas as funções do módulo *time* do python para testes de tempo de execução e a biblioteca *guppy3* para testes de memória utilizada.

O resultado ao executar o programa com o arquivo “sarscov2.fasta” fornecido pelo enunciado do trabalho é o seguinte:

```
Antes de ler e armazenar o arquivo:
+++ Tempo total:      0.000 segundos
+++ Memória utilizada:      4.131 megabytes
Após construção da árvore de sufixos do texto:
+++ Tempo total:    125.394 segundos
+++ Memória utilizada:     20.610 megabytes
Após encontrar a maior substring repetida:
+++ Tempo total:    125.586 segundos
+++ Memória utilizada:     20.611 megabytes
['AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA', 29870, 29901, 2]
```

A linha final é a substring desejada, o índice no texto onde sua primeira ocorrência inicia, onde ela termina e o número de ocorrências dela.

Adicionalmente, percebe-se que o processo demorou cerca de 125.6 segundos para encontrar a solução com o genoma do sarscov2, sendo que 125.4 desses segundos foram na construção da árvore de sufixos. Não só isso, como a memória gasta durante a criação da árvore aumentou em aproximadamente 16.5MB (do valor inicial de 4.1 para o final de 20.6MB). Essa solução, por ter sido feita em cima de um algoritmo ingênuo, pode ser melhorada usando o algoritmo de Ukkonen para construção da árvore, por exemplo.

Foram feitos mais testes usando partes menores do texto completo do genoma do sarscov, todas anexadas junto ao trabalho na pasta “tests”, e com resultados na pasta “tests/out”.