UNIVERSIDADE FEDERAL DE MINAS GERAIS

ALGORITMOS 2
DOCUMENTAÇÃO TP1

MATHEUS TIAGO PIMENTA DE SOUZA - 2018054893
PROFESSOR RENATO VIMIEIRO

Objetivo:

Este trabalho tem como objetivo fazer com que os alunos apliquem os conceitos de "manipulação de sequências" e de "árvores de sufixo" vistos em aula, além de praticarem e aperfeiçoarem técnicas de programação e fixar o conhecimento. Dessa forma, dado o problema de encontrar a maior substring que se repete em um texto, deve-se implementar uma árvore de sufixos a partir desse texto e identificar tal substring nele.

Ambiente de desenvolvimento, compilação e execução:

O projeto foi desenvolvido no ambiente de C++ e compilado utilizando o compilador G++. Para compilar, rodar *make* no diretório onde estão os arquivos. Em seguida, execute o executável gerado, passando por linha de comando o arquivo texto que será usado, via *arg*.

O problema: maior substring que se repete no texto:

Dado o genoma do Sars-Cov-2 (Coronavírus) como o texto, deve-se implementar uma árvore de sufixos no formato de uma Trie compacta, com complexidade máxima de $O(n^2)$.

Em seguida, deve-se implementar um algoritmo para realizar a busca da maior substring que se repete na árvore, de forma que ele deve retornar tal substring e o número de ocorrências dela no texto.

Análise do algoritmo:

- Montagem da árvore:

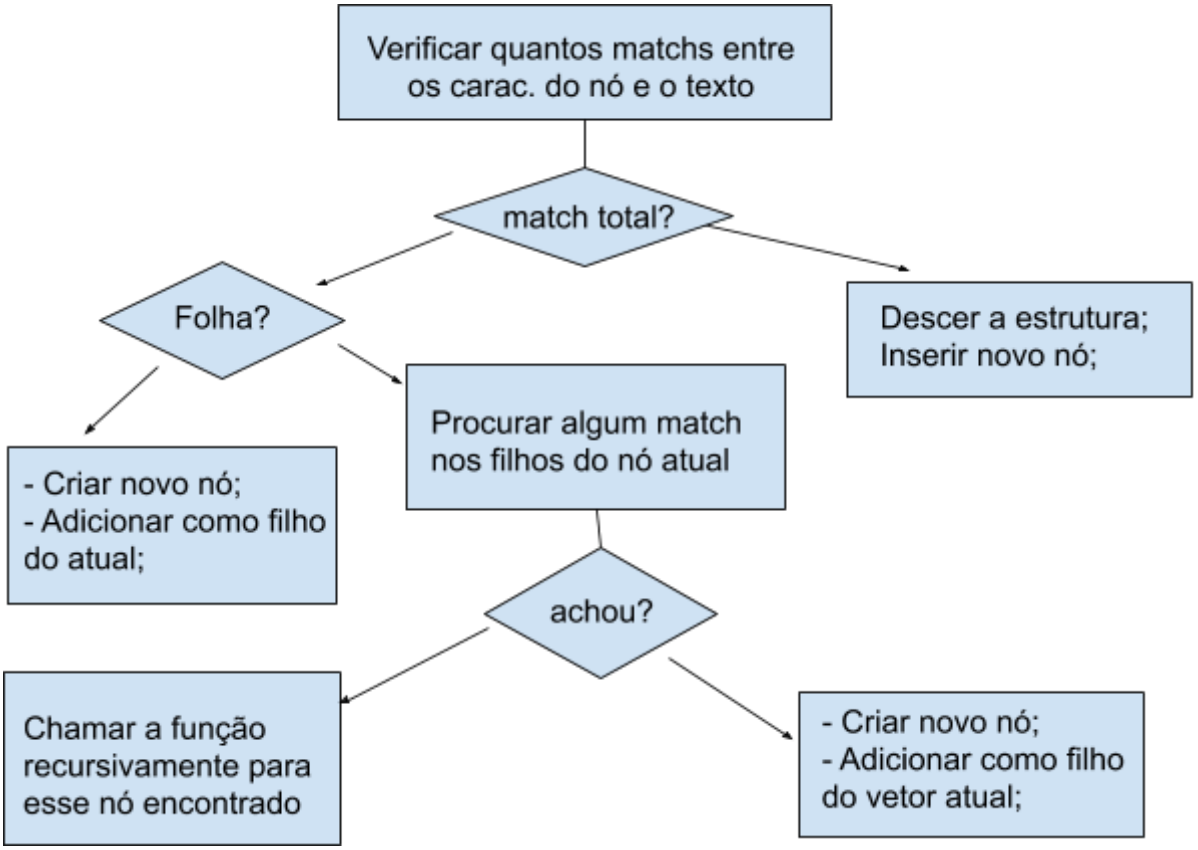
Para montar a árvore utilizamos apenas uma *struct no*, que contém as seguintes informações:

```
struct no{
   int sub[2];     // Coordenadas da substring que ele representa no texto
   int tipo_no = 0;        // 0 = nó interno; 1 = folha; 2 = folha-$
  vector<no *> filhos;    // vetor de filhos desse nó
};
```

A árvore portanto é composta por um conjunto desses nós, que estão ligados entre si de forma semelhante a uma lista simplesmente encadeada: criamos o nó raiz, e o programa vai criando outros nós e ligando-os entre si, montando a árvore.

Adicionamos o texto de trás pra frente, pois com isso temos a vantagem de nunca parar no meio da árvore, de forma que com certeza sempre vamos adicionar um novo nó.

A principal função para criação da mesma é a *add_no*. Ela assume que o nó passado por parâmetro tem pelo menos um caractere igual com o texto que estamos analisando, e faz tratamento para todos os casos possíveis. Abaixo temos o fluxograma do funcionamento da mesma:



O fato de partirmos do hipótese que existe pelo menos algum caracter igual entre o texto passado para função, e o nó que estamos analisando atualmente, facilita muito as decisões. Perceba que agora só é necessário analisar se temos um match completo ou não, e independente do caso, não é necessário voltar para o nó pai para fazer uma nova inserção (seria o caso de que não tivesse nenhum match). Isso permite não armazenar na struct uma referência ao pai.

A função faz uma chamada recursiva caso tenhamos um match completo e não seja folha, pois essa é a condição para descermos na árvore, e por isso chamamos a função para o nó filho. Entretanto, como nossa função pressupõe que o nó que chega por parâmetro tem um match, fazemos essa chamada recursiva apenas para nós filhos em que o primeiro caractere que ele representa seja o mesmo do próximo caractere do texto, e assim respeitamos a propriedade da função.

Porém, se a função fosse só assim não conseguiríamos tratar casos onde o penúltimo caracter seja diferente do último, porque não teríamos nenhum match. Tratamos esses casos no main, verificando antes de chamar se vai ocorrer algum match com o já adicionado. Caso nenhum match ocorra, adicionamos o nó manualmente.

- Identificando maior substring que se repete:

A ideia por trás dessa operação é simples: percorrer a árvore, procurando o maior ramo que não contém uma folha, e retornar a string que esse ramo forma. Isso garante encontrar a maior substring que se repete porque o último nó desse ramo (o mais perto da folha) aponta obrigatoriamente para pelo menos duas folhas, já que uma (ou mais) vai ser o nó (de tipo 1) que tornou nosso nó como não folha (tipo 0), e o outro vai ser o $ (tipo 2) gerado pela operação de inserir o novo nó. Dessa forma, garantimos que existem 2 substrings distintas que contém uma mesma substring em comum, que é o que desejamos.

A função *longest_r_sub* realiza esse processo. Partindo da raiz, realizamos uma busca recursiva em cada nó filho do atual que não é uma folha, somando o número de caracteres dos nós que percorremos. Quando encontramos a um cujos filhos são só folha, comparamos se a substring encontrada é a maior que já encontramos até o momento, trocando caso seja verdade. Ao final, retornamos o número de caracteres da substring, o índice do final dela, e os índices do começo de todas as folhas. Com essas informações, conseguimos retornar a substring e as ocorrências da mesma.

Avaliação experimental:

Para realizar a avaliação experimental, foram geradas 10 medidas do uso de memória e do tempo gasto pelas funções e pelo algoritmo, e tiradas a média, para melhor precisão. O resultado do algoritmo tendo o genoma do Sars-cov-2 como texto de entrada é:

A maior substring que se repete é a:
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

As ocorrencias delas são nas seguintes posições:
29897 29896

Abaixo a tabela com a avaliação empírica:

Consumo máximo de memória	15237.6 kB
Tempo p/ árvore de sufixos	257903.6 µs
Tempo localizar substring	34966.2 µs
Tempo total	289880.1 µs

Esses cálculos foram realizados a partir de funções do C++ disponibilizadas nas bibliotecas *chrono.h* e *sys/resource.h*. Os códigos estão comentados no final do programa.

Pela análise empírica podemos perceber que o tempo total condiz com a soma dos tempos de localizar a substring e o tempo para construir a árvore. Pelos valores, podemos

inferir que o código não ficou com uma complexidade muito absurda, o que condiz com o método de construção, que é descrito como $O(n^2)$ nos materiais de estudo.

Conclusões:

Com esse trabalho foi possível aprimorar os conhecimentos de árvores de sufixos e busca em árvores, entendendo conceitos como situações para criar novos nós e otimização de busca. Também foi possível compreender mais sobre analisar gasto de tempo e memória de um programa.