Relatório de Estrutura de Dados II - EP3

Pedro Cruz Martins - 11965859

Introducão:

Para esse programa, utilizei muito bibliotecas do C++, principalmente a biblioteca <vector> a qual eu utilizei para o armazenamento de vértices do grafo e seus vizinhos, além do largo uso de suas funções associadas de *push_back()* e *pop_back()*.

A ideia geral desse programa é: definir um número de fragmentos desejados para o método shotgun, ligar esses vértices com seus possiveis arcos, escrever da forma pedida no enunciado esses arcos em um arquivo txt, produzir o grafo a partir desse arquivo utilizando classes, eliminando os circuitos presentes no grafo e achando o caminho de comprimento máximo, que vai nos dar uma estimativa da fita de DNA.

Classes:

1. ***Vértice:***

   Essa classe, como sugere o nome, vai representar os vértices do grafo. Cada vértice possui 2 atributos: uma string que vai representar um fragmento de DNA que é produzido pela função shotgun, e um vetor (biblioteca <vector>) de ponteiros para os vizinhos desses vértices. É importante adicionar também que se X é vizinho de Y, significa que há uma aresta que liga Y a X nesse mesmo sentido, mas não necessariamente no sentido inverso.

   Além de seus atributos, a classe dos vértices também apresenta alguns métodos importantes para o programa, como o *void elimina_ciclos* e o *void acha_maior_caminho*, esses métodos, respectivamente, ao serem aplicados nos vértices, apagam as arestas que são as responsáveis por situar tal vértice em um circuito e preenchem um vetor caminho_MAX que guarda, como o nome sugere, o caminho de maior comprimento do grafo, agora acíclico.

2. ***Dígrafo:***

   Essa classe vai representar o grafo (dígrafo) em si. Seu único atributo é o conjunto de vértices que o compõem.

Além desse atributo, como o esperado, os métodos que associei a classe de dígrafos foram os de adicionar um vértice, adicionar uma aresta e printar o grafo.

Funções Notáveis:

1. ***shotgun:***

Essa função é a responsável pela "fragmentação" das fitas de DNA em fragmentos menores que posteriormente serão "grudados".

Utilizei para essa função as bibliotecas <chrono> e <random> para poder randomizar o tamanho do fragmento, que terá um mínimo igual ao parâmetro int *min_fragmento* e um máximo igual ao parâmetro int *max_fragmento*. Além disso, para efetivamente randomizar esses fragmentos, utilizei um gerador de números aleatórios para gerar números que funcionam como índices para os primeiros caracteres desses fragmentos.

Assim, apliquei tudo isso em um for loop em que cada fragmento produzido vai ser armazenado em um ponteiro de strings.

2. ***Produzir_Grafo:***

Essa função, como sugere o nome, constrói um grafo a partir de um arquivo txt, cujo formato é especificado no enunciado do EP.

Dessa forma, é simples, apenas apliquei um for loop nessas linhas do arquivo e utilizei os métodos de adicionar vértice e adicionar arcos.

3. ***Checagem:***

Essa função é a responsável por avaliar se dois vértices podem ser conectados por um arco ou não.

Como parâmetros, passamos duas strings, que são os fragmentos de DNA carregados pelos vértices, e um int k, uma constante explicada no enunciado do EP.

Assim, utilizamos o método da biblioteca <string> *substr* para fazermos um "splicing" nessas strings e comparando-as, retornando o valor booleano do resultado.

Função main():

Para a função main, primeiro, abrimos um arquivo .txt denominado “arquivo_da_string” que guarda uma fita de DNA composta por “A”, “C”, “G” e “T” concatenados. Além disso, também declaro logo no começo qual será a constante K do programa além da quantidade de fragmentos gerados pela função *shotgun.*

Em seguida, chamamos de fato a função *shotgun* e geramos um ponteiro de strings. Como um tratamento para eliminar os fragmentos possivelmente duplicados, passamos esse ponteiro de strings para um vetor de strings e utilizamos os métodos de sort e erase dos vetores para fazer essas eliminações.

Seguindo, abrimos um arquivo “arquivo_do_grafo” para edição (“ofstream”) e preenchemos tal arquivo da forma como foi especificado no enunciado do EP.

Assim, com o “arquivo_do_grafo” ja devidamente preenchido e formatado, utilizamos a função *produzir_grafo*, passando o arquivo em questão e, de fato, produzimos o grafo. Entretanto, há uma possibilidade muito grande que esse grafo conte com circuitos em sua estrutura, o que impossibilita achar os caminhos de maior comprimento, indo contra o objetivo de trabalho, então, como forma de tratamento, apliquei o método *elimina_ciclos* em cada vértice do grafo.

Após a eliminação dos ciclos, basta aplicar o método *acha_maior_caminho* para obter o vetor procurado e utilizar a função *decodifica_vetor_caminho_max* para, finalmente, atingir o objetivo do trabalho.

Testes:

Para os testes, adotei que o tamanho mínimo dos fragmentos era de 3 caracteres e o tamanho máximo era de 6. Ademais, o tamanho da string de DNA que vai ser repartida é de cerca de 50 caracteres.

Para k = 2: Em relação a velocidade de código, eu observei que em muitos testes, o tempo de execução acabou assumindo dois extremos: ou o código rodava em poucos segundos, cerca de 1s a 3s, ou o código demora cerca de 60s para a formação da fita de DNA, o que percebi é que parece que quanto maior as “gavetas” das listas de adjacência, o tempo de execução aumenta vertiginosamente, e como os fragmentos são randomizados, muitas vezes para os mesmos parâmetros temos resultados muito distintos. Para esse valor de K, cheguei que minha quantidade de fragmentos máxima é cerca de 50.

Para k = 3: Comparando com o valor anterior de k, temos um grande aumento de capacidade e velocidade. O problema dos extremos de tempo de execução permaneceu, porém agora cheguei em um limite de cerca de uma quantidade de fragmentos de cerca de 80 - 90.

Para k = 4: Comparando com o valor anterior de k, temos um grande aumento de capacidade e velocidade. O problema dos extremos de tempo de execução permaneceu, porém agora cheguei em um limite de cerca de uma quantidade de fragmentos de 150.

Em relação a precisão da fita de DNA formada ao ser comparada com a fita de DNA original, nenhuma delas igualou 100%, entretanto cheguei em um ponto em que havia uma boa semelhança com a fita original, às vezes até mesmo contendo quase inteiramente a fita original em com a adição de alguns caracteres, portanto, é sempre possível observar uma falta de precisão exata.