MAC 0323

RELATÓRIO EP3

Algortimos e Estrutura de Dados II

Odair Gonçalves de Oliveira
13671581

SUMÁRIO

INTRODUÇÃO

O Exercício Programa III (EP3) envolve a implementação de um algoritmo que visa simular a montagem de um DNA a partir de diferentes pedaços do mesmo.

Tal tarefa é executada com auxílio de algoritmos para manipulação de grafos, a estrutura utilizada, que é o tema principal de estudo desse EP.

A ideia é receber um arquivo com vários pedaços/fitas de um DNA maior e, a partir das conexões que podemos estabelecer entre essas partes, tentar encontrar uma resposta final que aproxime-se do DNA.

Para isso, cada pedaço do DNA torna-se um vértice do grafo montado. Para a montagem das arestas é feito o seguinte: Se o sufixo de tamanho k do vértice u é igual ao prefixo de tamanho k do vértice v, então temos uma aresta que sai de u em direção a v com peso k.

Assim, é passado pelo usuário um parâmetro k mínimo, que é o menor peso aceito para uma aresta.

O desafio então é encontrar o caminho de maior peso dentro desse grafo quando ele é acíclico. Caso ele não seja acíclico, será necessário ir removendo arestas até que ele não tenha mais ciclos.

Como a remoção de arestas que estão em ciclo pode prejudicar e inviabilizar o resultado final de ser igual ao DNA original, o algoritmo trabalha com uma eurística, isso é, uma estratégia que tenta produzir uma boa estimativa de resposta certa, pois é difícil garantir que a resposta esteja totalmente correta.

Veja a seguir detalhes da implementação realizada e da estratégia adotada para remoção das arestas em ciclos.

CÓDIGO & IMPLEMENTAÇÃO

Todos os programas e algoritmos estão em C++ e divididos da seguinte forma:

- EP3.cpp - Contém a função main principal para execução das simulações e dos testes;
- make_teste.cpp - Contém a função main principal utilizada para gerar os arquivos de testes para verificação do funcionamento do EP;
- operacoes.h - Contém algumas funções auxiliares construídas para o bom funcionamento do EP que não estão diretamente relacionadas a grafos;
- grafo.h - Contém as funções mais relacionadas a grafos utilizadas principalmente no EP3.cpp;

TESTES

Para a geração dos testes, foi realizado o seguinte:

- O usuário escolhe o tamanho do DNA original que deseja gerar;
- O usuário indica qual o tamanho mínimo e máximo para os pedaços/fitas que serão geradas a partir da string original.
- O usuário escolhe o número de cópias que serão feitas do DNA original para serem realizados os cortes;
- O tamanho das fitas é escolhido aleatoriamente e as cópias são cortadas nesse tamanho que varia entre os parâmetros mínimo e máximo escolhidos anteriomente;
- As strings são ordenadas e o usuário pode escolher o nome do arquivo em que deseja salvar o teste gerado;

Os testes tem a seguinte estrutura:

- 1 - A string DNA original;
- 2 - O número de cópias feitas;
- 3 - O número de fitas geradas;
- A lista ordenada de fitas geradas;

FUNÇÕES & ESTRUTUTURAS

Para o EP3 foram utilizadas as estruturas e funções que estão presentes no arquivo "grafo.h". A seguir, estão expostas essas estruturas e funções (No código, usei ll como abreviação de long long):

ESTRUTURAS:

```
struct NODE {
  • long long id;
  • string info;
  • long long int g_in;
  • long long int g_out;
}

typedef struct vector<NODE> VERTICES;
```

O parâmetro **id** guarda um valor inteiro que representa o vértice, o parâmetro **info** guarda a string da fita representada por aquele vértice. Já os parâmetros **g_in** e **g_out** guardam os graus de entrada e saída de cada vértice respectivamente.

Já **vertices** é um vetor de nodes que guarda todos os vértices, que são recebidos já ordenados a partir do arquivo de entrada.

```
struct ARESTA {
  • long long vertice;
  • long long peso;
}

typedef struct vector<vector<ARESTA>> ARESTAS;
```

A estrutura de **aresta** e o vetor **arestas** serve para acessar as arestas da seguinte forma:

arestas[i][j].vertice = O id do j-ésimo node que está conectado com o o node de id = i;
arestas[i][j].peso = O peso da aresta entre o node de id = i e o node de id = arestas[i][j].vertice;

FUNÇÕES & ESTRUTUTURAS

```
struct ARCO {
  • aresta a;
  • long long v;
}

typedef struct vector<ARCO> ARCOS;
```

A estrutura de arcos foi criada como auxiliar apenas para podermos ter as arestas ordenadas por peso no vetor de arcos. Os parâmetros são os seguintes: A aresta a tem como um dos parâmetros o 'vertice'. Assim, arcos[i] guarda o i-ésimo arco na ordenação por pesos, que é o arco de ligação entre os vértices arcos[i].a.vertice e arcos[i].vertice;

FUNÇÕES:

As funções estão documentadas diretamente no arquivo grafo.h, por tal fato, só irei citar o protótipo delas a seguir sem detalhar muito:

- bool compara_arco(const arco& arc1, const arco& arc2);
- void add_aresta_simples(arcos& arc, arestas &are, node &u, node &v, ll k);
- void add_aresta(arcos& arc, arestas& are, node& u, node& v, ll k);
- bool circ_r(ll u, ll * pre, ll * pos, ll& tempo, const arestas& adj);
- bool tem_circuito(ll V, const arestas& are);
- void dfs(ll u, const arestas& are, bool * vis);
- bool aresta_em_circuito(ll V, const arestas& are, const node &u, const node &v);
- void remove_aresta(arestas& are, node &u, node&v);
- void dfs_ordem_topologica(ll u, bool * vis, stack<ll>& order, const arestas& adj);
- vector<ll> ordenacao_topologica(ll V, const arestas& adj);
- vector<ll> caminho_maximo(const arestas& adj, ll source, ll V, ll& dest);
- vector<ll> o_caminho_maximo(vector<ll>& pred, ll dest);
- void printa_biggest_way(vector<ll>& big, vertices& verts, arestas& adj);
- string resp_final(vector<ll>& big, vertices& verts, arestas& adj);
- void printa_caminho_maximo(vector<ll>& pred, ll dest);

FUNÇÕES & ESTRUTUTURAS

FUNÇÕES SOLICITADAS:

O enunciado solicitava algumas funções específicas, porém elas não foram feitas da maneira exata que estava especificado por motivos que serão explicados.

Para começar, a primeira função solicitada pedia que fosse feita uma função que lesse um arquivo que começasse com o número de vértices e arestas e depois vários pares de arestas, e após a leitura construisse um grafo com essas arestas. Porém, no fórum, o professor deu outra sugestão de entrada que faria mais sentido, então não senti necessidade de fazer essa função, já que já tinha uma que adicionava arestas dado dois vértices, sendo ela a função a abaixo:

void add_aresta_simples(arcos& arc, arestas &are, node &u, node &v, ll k);

Essa função está no arquivo grafo.h na linha 119. Ela recebe dois nodes, u e v, e adiciona uma aresta entre esses nodes com peso k - apenas se não já existir uma aresta de peso maior entre v e u. Os vetores are e arc servem para guardarem as arestas para operações futuras.

A próxima função solicitada no enunciado é a função que verifica se um arco u -> v está em um ciclo ou não.

bool aresta_em_circuito(ll V, const arestas& are, const node &u, const node &v)

Essa função recebe dois nodes, u e v, e verifica se a aresta entre eles está em um ciclo ou não. Para execução desta tarefa, a função recebe o vetor que tem as arestas e o número de V de vértices. Ela está no arquivo grafo.h na linha 237.

FUNÇÕES & ESTRUTUTURAS

A última função solicitada no enunciado é uma função que encontra em um grafo aciclíco um caminho de comprimento máximo. Assim como as anteriores, essa função acaba fazendo uso de outras, por exemplo, a função de ordenação topológica.

vector<ll> caminho_maximo(const arestas& adj, ll source, ll V, ll& dest, const vector<ll>& order);

vector<ll> o_caminho_maximo(vector<ll>& *pred*, ll *dest*);

Essas são as duas funções utilizadas para construir o caminho máximo, o vetor produzido pela primeira função é o vetor pred que é passado como parâmetro da segunda função, ou seja, dava para ter feito apenas uma função, mas a escolha de dividir foi pelo fato de o pred ser parâmetro de outras funções que eu estava utilizando.

A primeira função gera um vetor de predecessores utilizando a ordenação topológica guardada no vetor order e salva em dest o vértice que para ser atingido teve o maior custo de caminho. Já a segunda, usa esse dest e o vetor de predecessores para construir esse caminho. A primeira está na linha 294 e a segunda na linha 323 do arquivo "grafo.h".

GERAÇÃO DE TESTES E DISCLAIMER

Para a geração de testes, basta utilizar/verificar o programa "make_teste.cpp" e executá-lo. Para fazer o DNA original faço um rand para escolher entre as 4 letras disponíveis até dar o tamanho que desejo. Depois vou pegando substrings dessa string com um tamanho que é sorteado entre o intervalo de tamanho máximo e mínimo passado pelo usuário. Para isso utilizo várias funções da biblioteca de strings.

Um importante disclaimer é que forneci alguns arquivos de testes para colegas, então talvez tenham arquivos de teste gerados por mim com eles, como o professor não proibiu isso e é comum compartilhar arquivos de teste nos EPS não achei que teria algum problema compartilhar.

Além disso, mais um disclaimer, várias das funções para achar os padrões e excluir arestas foram feitas utilizadas funções próprias da biblioteca de strings e vector do C++, novamente, como o foco do EP não era esse, optei por utilizar o que já estava pronto.

ESTRATÉGIA

A estratégia que adotei segue os seguintes passos:

- É executada a leitura do arquivo de testes e as strings são salvas no vetor de vértices;
- O usuário escolhe um k mínimo para estabelecer a ligação entre os vértices;
- Começa o processo de adição das arestas. Para todos os vértices do vetor, faço o seguinte:
  - Começo vendo o sufixo de tamanho máximo (o tamanho da string) e vou até o k mínimo passado pelo usuário adicionando as arestas comparando com o prefixo dos outros vértices;
    - Caso o vértice tenha um tamanho inferior ao k escolhido, tento adicionar arestas com o peso igual ao tamanho total da aresta para evitar ter muitas componentes conexas de 1 vértice;
  - Ao tentar adicionar uma nova aresta entre u e v com peso k, primeiramente verifico se existe uma aresta de v para u.
    - Se não exitir, coloco a aresta u - v com peso k;
    - Se existir, verifico se o peso é menor que k. Se for, excluo a antiga e adiciono a nova. Caso contrário, não adiciono a nova.
  - Para decidir entre quais arestas irei adicionar uma aresta, utilizo funções que devolvem a primeira e última aparição do sufixo buscado nos prefixos das strings da lista ordenada de vértices;
  - Também já atualizo os parâmetros de grau de entrada e saída e coloco as arestas no grau de arcos que serão ordenados durante a inserção;
- Após a adição de arestas verifico se o grafo tem ciclos;
  - Se tiver ciclo, preciso ir removendo as arestas. Para isso, vou percorrendo as arestas ordenadas por peso no vetor de arcos;
    - Verifico se a aresta atual está em um ciclo;
      - Se estiver, a removo e verifico se o grafo ainda possui ciclo. Se ainda tiver, continuo percorrendo o vetor de arestas e fazendo o mesmo processo.
      - Se não tiver mais ciclo, vou para o próximo passo;
- Se o grafo não tem mais ciclos, realizo uma ordenação topológica.
- Depois disso, salvo em um vetor de inteiros todos os índices de vértices que possuem grau de entrada 0;

ESTRATÉGIA

- Itero por todos os vetores de graus zero já salvos anteriormente e verifico o caminho máximo que consigo a partir deles utilizando a ordenação topológica já feita gerando um vetor de predecessores;
- Com esse vetor de predecessores consigo construir um caminho de peso máximo e calcular o tamanho da string gerada por esse caminho.
- Avalio o tamanho dessa string e deixo salvo o caminho que mais se aproxima da minha estimativa de tamanho do DNA original;
  - Para essa estimativa faço o seguinte: (Como estou supondo que não tenho o DNA original) somo o tamanho de todas as fitas de DNA e divido pelo número de cópias do DNA original;
  - Como os testes são feitos sem descartar nenhum pedaço, esse valor é sempre igual ao tamanho do DNA original;
- Assim, imprimo esse caminho como resposta final e comparo o tamanho da minha resposta final com a estimativa de tamanho feita para avaliar o quão bem ele está funcionando;

JUSTIFICA DA EURÍSTICA:

A justificativa para como decidi realizar a remoção das arestas em ciclos é a seguinte: Pensei que remover as arestas sem verificar seu peso seria arriscado, pois enquanto maior o peso, maior a certeza de que aquela aresta deveria existir, então decidi privilegiar essas arestas com maior peso ordenando-as. Assim, vou removendo arestas com peso baixo (que tem mais chance de serem acasos) que estão em ciclos.

Além disso, na própria adição de arestas já evito ciclos do tipo vai-e-vêm, isso é, u -> v e v -> u. Pois mantenho apenas a aresta entre essas duas com maior peso.

TESTES

Para executar compilar o EP usei o seguinte comando no terminal:

g++ EP3.cpp -Wall -o EP3

Recebi alguns Warnings pela comparação de valor do tipo long long int com variáveis do tipo de tamanho (o size dos vetores), mas isso não gerou nenhum problema na execução do programa.

No geral, as execuções não demoraram muito, mas admito que não me preocupei muito com eficiência nesse EP dada a complexidade e dificuldade que tive com algumas funções de grafos.

Entretanto, entre os arquivos de teste que gerei o que pude observar no geral é que:

1 - Quando tenho um DNA original muito grande que é picotado em pedaços muito pequenos em comparação ao tamanho original, exemplo: tamanho original 1000 e maior pedaço de tamanho 16 e mínimo 4, que é o caso do teste14.txt, a resposta não fica muito boa;

2 - Enquanto maior o k, menor a incidência de ciclos, assim, mais rápida é a execução, porém pior é a resposta, pois o grafo começa a se tornar desconexo, com várias componentes isoladas. É o caso do teste1.txt que é bem pequeno e com várias fitas repetidas. Com um k = 2, ele apresenta duas componentes conexas. Uma delas gera uma resposta de tamanho 6 a o outra gera uma string de tamanho 5. A única forma de juntar essas componentes é se k for igual a 1. Esse teste exemplifica bem esse problema.

3 - Para testes super grandes, como é o caso do 9, 10 e 11, a execução foi, considerando o tamanho dos testes, eficiente e as respostas foram muito boas, já que o intervalo de tamanho das strings por ser bem maior permitia usar um k de valor bem maior e ainda sim obter uma resposta satisfatória. Para valores de k pequeno para esses testes enormes, a execução demorava mais, mas não afetava tanto a resposta final, considerando que sempre tento estabelecer arestas de peso máximo até chegar ao k mínimo.

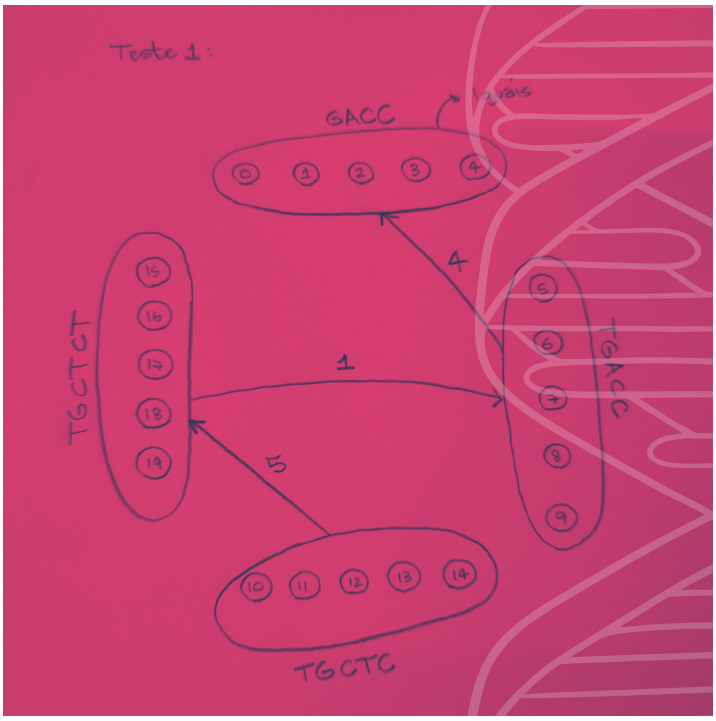
TESTES SUGERIDOS

Veja abaixo a lista dos testes sugeridos e seus parâmetros além dos resultados obtidos:

TESTE 1:

- Arquivo.....teste1.txt;
- Tamanho DNA Original.....10;
- Número de Cópias.....10;
- Número de Fitas.....20;
- Tamanho fita MIN/MAX.....4/6;

Para esse sugiro testar com k = 1 e k = 2 para ver o que foi ilustrado no ponto 2 exposto anteriormente. Para k = 1, obtive resultado de 100% e para k = 2, consegui apenas uma string de tamanho 6. Veja um desenho abaixo ilustrando o que ocorreu nesse grafo:



Desenho ilustrando o grafo produzido no EP usando o arquivo teste1.txt.

Note que sem k = 1, não é possível obter a aresta do meio de peso 1, o que faz a resposta ser apenas TGCTCT quando deveria ser TGCTCTGACC.

TESTES SUGERIDOS

TESTE 14:

- Arquivo.....teste14.txt;
- Tamanho DNA Original.....1000;
- Número de Cópias.....25;
- Número de Fitas.....2801;
- Tamanho fita MIN/MAX.....4/16;

Esse teste ilustra bem o ponto 1 levantado anteriormente, para k = 3, ocorre uma demora considerável para remoção de arestas e mesmo assim, com um k baixo, a resposta ainda não é satisfatória e fica próxima da metade do tamanho previsto. Para k maiores que o valor mínimo, a resposta torna-se ainda pior.

Vale destacar que mesmo que a fita tenha um tamanho menor que o k passado, eu decidi, após realizar alguns testes, que seria melhor para melhorar as respostas, adicionar arestas partindo desta fita com peso igual ao tamanho dessa string. Ou seja, no exemplo acima, mesmo para k = 7, as fitas com tamanho 4 ainda teriam arestas. Tal decisão foi feita para evitar que houvessem muitas componentes conexas de 1 só vértice.

Os outros testes não possuem muita peculiaridade, ou seja, não apresentam nada muito relevante sobre a implementação, então qualquer um deles serve para verificar o funcionamento para um caso mais geral. É claro, eles variam em tamanho, número de cópias e tamanho máximo e mínimo, mas no geral, apresentam respostas satisfatórias. No terminal é possível verificar por meio de escolhas de impressão as informações do teste, como imprimir a lista de arestas, a ordenação topológica e verificar o tamanho mínimo e máximo das fitas no arquivo antes de escolher o k. Além de que é possível criar testes personalizados com o make_teste.

Para testar com o arquivo que o professor mandou use o teste12.txt:

TESTE 12 (PROFESSOR):

- Arquivo.....teste12.txt;
- Tamanho DNA Original.....100;
- Número de Cópias.....20;
- Número de Fitas.....62;
- Tamanho fita MIN/MAX.....4/50;

Para o teste do professor, testei com k = 3, k = 10 e k = 25 e em todas as tentativas obtive a resposta de tamanho 100, para k = 30, obtive uma resposta de tamanho 78, o que ilustra a questão de enquanto maior o k, pior pode ser a resposta, mas enquanto menor o k, mais ciclos o grafo possui o que exige mais operações.

CONCLUSÃO

Após a realização dos testes, pude verificar que o código funcionou bem para a maioria dos casos, e pude acompanhar como o tamanho máximo e mínimo das fitas em conjunto com o k escolhido podem influenciar na eficiência e precisão /confiabilidade da resposta.

Não pude concluir muito sobre o quanto o número de cópias do DNA orginal influenciam, mas acredito que um maior número de cópias dentro de um intervalo maior de tamanhos mínimos e máximos (com um mínimo não muito baixo) permite que existam mais pedaços diferentes e essa diversidade dos pedaços pode ajudar.

Porém, como não fiz testes para verificar isso, não posso afirmar com toda certeza que um intervalo maior de tamanhos de fita e um número de cópias maior e mais diverso auxilia mais do que atrapalha, mas acredito que sim.

Foi interessante ver a aplicação das funções e teoria de grafos em um problema concreto da biologia. Confesso que tive dificuldade em pensar em formas mais eficientes de implementação e não me preocupei muito com isso ao desenvolver a solução e minha eurística.

Enfim, concluindo, foi interessante aplicar o que foi aprendido sobre a estrutura de grafos em um problema real, o que permitiu visualizar com clareza funções como a verificação de ciclos, ordenação topológica e a utilizada para construção do caminho máximo, o que me permitiu entender melhor o funcionamento dessas funções e suas utilidades.