UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE MATEMÁTICA E ESTATÍSTICA

RODRIGO DE CASTRO MICHELASSI
NUSP: 13672703

Exercício-Programa 3: Reconstrução de Filamentos Genéticos com Grafos

São Paulo
2023

Sumário

1 Introdução

Nesse EP, tínhamos o objetivo de utilizar os algoritmos em grafos trabalhados em sala de aula para resolver um problema clássico da Ciência da Computação: a reconstrução de trechos de código genético com base no uso de grafos.

Para isso, o EP nos dava como entrada diversos pedaços de código genético, cortados de forma aleatória, e, dessa forma, deveríamos implementar uma solução para conseguir reconstruir esse filamento.

Foram trabalhados diversos algoritmos vistos em sala, porém principalmente a construção de grafos e o uso do algoritmo recursivo dfs, que realiza uma busca em profundidade em um grafo, ou seja, percorre todos os vértices do grafo em questão. Os detalhes da implementação desse algoritmo serão discutidos a seguir.

OBS: O EP entregue foi feito em uma máquina rodando MACOS. Ao testar em uma máquina com Linux baseado em Debian, ocorreram alguns Warnings que não ocorrem no MAC, porém ao mudarmos o código para que atendesse às demandas do Ubuntu, o mesmo código apresentava erro no PC em MAC. Nesse sentido, a versão entregue foi feita inteiramente no MAC e deve funcionar corretamente, apesar dos Warnings que serão obtidos durante a compilação.

2 Estruturas de Dados

Primeiramente, foram implementadas duas estruturas de dados principais, e utilizadas outras funções pré-prontas da biblioteca STL do c++.

As estruturas implementadas foram uma lista de listas e um grafo, implementado por meio de uma lista de adjacências. Nesse caso, a lista de listas se torna uma estrutura auxiliar para a estrutura do grafo, e foi implementada como um *vector* do c++, onde cada posição possui um apontador para outro vector.

Dessa forma, em cada Nó, nome dado no código à estrutura que guarda a lista de listas, possuímos parâmetros como o tamanho, tamanho da interseção entre duas palavras, um atributo cor, que será utilizado no dfs, o fragmento de DNA a ser analisado e, claro, um vector.

Por outro lado, na estrutura geral do Grafo, possuímos bem menos atributos, são eles V, o número de Vértices, E, o número de arestas, e um vector, representando a lista base da lista de adjacências, com o tipo Nó.

3 Algoritmos Implementados

Antes de analisarmos os algoritmos, note que receberemos no nosso terminal três impressões diferentes. A primeira, conta com todos os vértices, com seus respectivos arcos, a segunda conta com todos os vértices e seus arcos após remoção de arestas, e a terceira conta com o sequencimento de DNA obtido.

3.1 Leitura de Grafo

Para a leitura de grafos, utilizamos a leitura de um arquivo de entrada, o qual será pedido no início da execução. Esse algoritmo será responsável por ler esse arquivo, que receberá um valor $V \in \mathbb{Z}^+$ e seguido por V strings, representando os trechos de DNA recortados manualmente.

Para fazermos isso, leremos cada uma das linhas com strings e, para cada palavra já adicionada na nossa lista principal, da lista de adjacências, comparamos com a nova palavra que será inserida, caractere a caractere, e, caso K (valor dado na entrada e definido no enunciado do EP) ou mais caracteres ao fim da primeira palavra sejam iguais a K ou mais caracteres na segunda palavra, definimos um arco apontando na mesma ordem.

Obviamente, esse algoritmo não está completo por si só dessa forma, porque se uma palavra U definida possui um arco para V, V também pode possuir um arco para U. Dessa forma, realizamos novamente o mesmo algoritmo porém levando sentido contrário, a fim de incluirmos todos os arcos possíveis no nosso grafo.

3.2 Verificar e Excluir Circuitos

A lógica por trás desse algoritmo se baseia na lógica básica utiliza em qualquer algoritmo básico de busca em profundidade em grafos. Obviamente, foi utilizado o dfs para realizarmos uma busca em profundidade. Antes, o algoritmo estava sendo feito com base em um atributo "cor"presente em cada vértice, todavia essa lógica foi substituída por um vetor $marked$, responsável pela mesma função.

Começamos mapeando nossos vértices entre 0 e V-1, na ordem que foram dados na entrada, onde cada posição do vetor $marked$[V] pode assumir os valores 0 e 1, os quais representam se um vértice específico já foi ou não visitado na dfs.

A elegância então do algoritmo consiste no uso de um outro vetor $onStack$, utilizado como uma fila, o qual é do tipo booleano e recebe True caso um vértice seja visitado, e False, caso, durante o dfs, paremos de visitar esse vértice em determinado instante. Dessa forma, ao detectarmos um ciclo, encontraríamos o valor True no vértice que estamos visitando, e, assim, podemos remover essa aresta.

Note que não houve um algoritmo mais complexo para remoção de arestas, apenas removemos aleatoriamente uma aresta quando encontramos um ciclo na nossa dfs. Isso ocorreu devido a uma falta de tempo para implementar um algoritmo mais sofisticado. A ideia pensada aqui seria implementar uma fila de prioridades, onde excluiríamos a aresta com menor interseção com o vértice original, porém essa ideia não foi implementada.

3.3 Encontrar caminho máximo e reconstruir

O algoritmo aplicado nessa função é um algoritmo clássico na teoria dos grafos, chamado de Ordenação Topológica, que consiste em uma permutação dos vértices de um grafo dirigido acíclico, de forma que, para qualquer aresta $(v_i, v_j), i < j$, caminhos orientados percorrem os vértices em ordem crescente, e qualquer caminho entre v_i e v_j não passa por v_k, para $k < i$ ou $k > j$. Esse algoritmo está disponível em materiais online produzido pelo ICMC-USP.

Brevemente explicando o código, o algoritmo consiste em realizar uma busca em profundidade por dfs com base em dois vetores de início e fim, utilizados para marcar a ordenação do caminho do nosso grafo. Esses vetores preenchem um array com o caminho percorrido no grafo, e assim podemos obter o caminho máximo desejado.

4 Testes e Execução

Para executar o programa, no diretório onde o código está localizado, podemos rodar no terminal o comando $ *make*, responsável por compilar o programa e as bibliotecas utilizadas. Dessa forma, basta rodar o comando $ *./ep3* e iniciar o programa.

Com o programa iniciado, é pedido inicialmente um arquivo para leitura, que deve conter o número de vértices V do nosso grafo em questão e V strings contendo o trecho de DNA. Logo em seguida é pedido um valor de K, tamanho mínimo da interseção entre duas palavras para leitura. Note que o algoritmo implementado não é 100% preciso, podendo apresentar algumas divergências esperadas na reconstrução dos algoritmos. Nesse sentido, segue que o valor de K deve ser escolhido de forma a aproximar melhor a sequência genética que queremos reconstruir, então, para filamentos genéticos muito grandes, é esperado que o valor de K seja mais alto, enquanto para filamentos genéticos menores, também é esperado uma menor interseção entre as strings, a fim de melhor ler o grafo e montar suas arestas.

Segue então alguns testes realizados e resultados obtidos. Para os primeiros três testes, utilizamos o exemplo do próprio enunciado, e a resposta esperada era **ACTCGTAAATACATAA**

Algoritmo 1 Teste 1

```
← input.txt                                  ▷ Arquivo de Entrada
← 3                                          ▷ Valor de K

→ ACTCGTAAATACATAACGATACATAAATAAC            ▷ Saída
```

Algoritmo 2 Teste 2

```
← input.txt                                  ▷ Arquivo de Entrada
← 2                                          ▷ Valor de K

→ ACTCGTAAATACATAACGATACATAACATAAATA         ▷ Saída
```

Algoritmo 3 Teste 3

```
← input.txt                                  ▷ Arquivo de Entrada
← 3                                          ▷ Valor de K

→ GATACGTAAATACTCGTATACATAAATAACGAT          ▷ Saída
```

Segue, do que foi explicado antes nesse enunciado, que o valor de K deve ser escolhido de maneira precisa. Dessa forma, é fácil ver que, para K = 4, a resposta obtida varia muito do esperado, agora para K = 2 e K = 3 a resposta se aproxima do esperado,

com erro apenas nas últimas posições da string. Vamos testar para um exemplo gerado pelo professor:

Algoritmo 4 Teste 4

← input2.txt ▷ Arquivo de Entrada
← 6 ▷ Valor de K

→ AAATCAATACAGTTGGTGTGATGCCTTTCTACACGGTTGGG
TAAGGATTTGCCGGGACCATGTGTCAAACTGGCTGTGTGA
TGCCTTTCTACACGGTTGGGTAAGGATTTGCCGGGACCAT
GTGTCAAACTGGCTGTACAGTTGGTGTGATGCCTTTCTAC
ACGGTTGGGTAAGGATTTGCCGGGACCATCAATACAGTTG
GTGTGATGCCTTTCTACACGGTTGGGTAAGGATTTGCCAC
AAATCAATACAGTTGGTGTGATGCCTTTCTACACGGTTGG
GTAAGGATTTGCCGGGACCATGTGTCAAACTGGCTGTTGG
GTAAGGATTTGCCGGGACCATGTGTCAAACTGGCTGT ▷ Saída

Nesse exemplo, não temos uma resposta esperada, todavia foi um exemplo útil para testar a funcionalidade do algoritmo para entradas mais longas.

O próximo exemplo testado foi gerado pelo ChatGPT, e iremos analisar a performance do nosso algoritmo nele:

Algoritmo 5 Teste 5

← input3.txt ▷ Arquivo de Entrada
← 3 ▷ Valor de K

→ TACGATCGACGGTACGTAGTCGACGTA ▷ Saída

Algoritmo 6 Teste 6

← input3.txt ▷ Arquivo de Entrada
← 2 ▷ Valor de K

→ ACGGTACGTAGTCGACGTACGATCGA ▷ Saída

Nesse caso, vimos uma resposta bem diferente para K = 2 e K = 3. Todavia, a resposta esperada era ACGGTACGTAGTCGACGTACGATCGATC, muito próxima do obtido com K = 2. Note que, como essa sequência é curta e foi separada manualmente, se utilizarmos K = 1 obteríamos exatamente a sequência desejada.

5 Considerações Finais

Com base nesse projeto, foi possível explorar e abrir portas para utilização de grafos e ferramentas matemáticas para solucionar problemas computacionais, além de ter a oportunidade de trabalhar com um problema clássico do nicho da ciência da computação e explorar mais a fundo algoritmos em grafos, como por exemplo o funcionamento recursivo do dfs.

Utilizando os resultados avaliados como referência, podemos dizer que nosso algoritmo é funcional e consegue aproximar o sequenciamento genético pedido, principalmente para sequências menores, com valores de interseção que não variam muito, como podem ser observados nos exemplos de entrada enviados junto a esse projeto. Além disso, o algoritmo se mostrou eficiente em mapear grafos em listas de adjacências e funcional para entradas grandes, embora seja mais preciso para entradas pequenas e sequenciamentos de DNA divididos em trechos menores.

Dessa forma, o projeto realiza o que é proposto de maneira eficiente e funcional, e pode ser utilizado para projetos futuros na área, apesar de ainda ter erros de precisão que necessitam de ajustes.