Relatório EP3 - MAC0323

Anahí Coimbra Maciel | N°USP: 11809127

Junho de 2023

Esse relatório é sobre o EP3 da disciplina Algoritmos e Estruturas de Dados II, lecionada pelo professor Carlos Ferreira no primeiro semestre de 2023. O programa foi escrito em C++. O programa respeita todas as restrições apresentadas na proposta.

1 Como compilar e executar o EP

Para compilar o EP simplesmente rode no diretório do EP o comando:

```
g++ EP3.cpp -o EP3
```

Em seguida, para executá-lo, utilize:

```
./EP3
```

2 Estrutura e funções implementadas

Para esse EP foi implementada a estrutura *Graph*, que corresponde a um grafo dirigido. Essa estrutura foi implementada a partir de uma lista de adjacência,

```
unordered_map<string, vector<Edge>> AdjList
```

que é uma hashtable em que cada string (vértice do grafo) corresponde a um vetor de Edge, struct

```
struct Edge {
string start;
string end;
int k;

Edge(string s, string e, int k_) : start(s), end(e), k(k_) {}};
```

que representa as arestas, onde k é o valor máximo de k para as strings *start* e *end* (i.e, o overlap entre os últimos caracteres de *start* e os primeiros de *end*).

Cada grafo também contém um parâmetro K, que corresponde ao valor K a partir do qual o grafo foi feito, conforme o enunciado do EP, e o valor $edges$, que é número de arestas do grafo.

As funções (além do construtor) implementadas na classe *Graph* foram:

```
private:

void _findMaxPath(string vertex, unordered_map<string,
int>& lengths, unordered_map<string, string>& previous);

bool _hasCircuit(string u, unordered_map<string,int>& pre,
unordered_map<string,int>& pos,unordered_map<string,
string>& pred,int& time);

public:

void buildGraph(string filename);

bool isInCircuit(string u, string v);

bool hasCircuit();

void removeEdges();

void findMaxPath();

void printGraph();
```

Respectivamente, as funções:

1. Função helper para a função findMaxPath: acha o maior caminho a partir de um dado vértice.
2. Função helper para a função hasCircuit.
3. Função que constrói o grafo a partir do arquivo designado. O arquivo consiste em uma primeira linha que contém um número V correspondente à quantidade de vértices do grafo e V linhas seguintes, cada uma com um vértice. Para cada um dos vértices, a função verifica quais são seus vizinhos, adicionando-os na lista de adjacência.
4. Função que verifica se uma dada aresta está em um circuito usando busca em profundidade. Essa função acabou não sendo utilizada nos testes, mas foi implementada conforme o enunciado.
5. Função que retorna se o grafo tem circuitos.
6. Função que torna o grafo cíclico em um grafo acíclico. A função retira arestas até o grafo não ser mais cíclico, de acordo com o valor de k. Ou seja, as arestas com menor valor de k são retiradas primeiro.
7. Função que acha o maior caminho no grafo acíclico e o imprime.
8. Função que imprime a lista de adjacência do grafo.

Além disso, duas funções auxiliares foram implementadas:

```
bool isValid(string u, string v,int k);
int check_K(string u ,string v);
```

A primeira função retorna se as k últimas letras de u são iguais às k primeiras letras de v, dados duas strings u e v e um inteiro k. Já a segunda retorna qual o overlap de caracteres entre as últimas letras de u e as primeiras de v.

3 Notas sobre o funcionamento do EP

O EP recebe duas entradas pela linha de comando, o nome do arquivo com os dados necessários para a construção do grafo e o valor de k desejado. O arquivo, como já dito antes, consiste em uma primeira linha que contém um número V correspondente à quantidade de vértices do grafo e V linhas seguintes, cada uma com um vértice. O programa imprime a lista de adjacência do grafo montado, a lista de adjacência do grafo após ser transformado em acíclico (se o grafo já era acíclico, ele não imprime a mesma lista novamente) e a reconstrução da sequência original, que corresponde ao maior caminho no grafo. Note que para que a sequência seja reconstruída corretamente, o programa imprime o caminho cortando os k primeiros caracteres do segundo até o último fragmento, para que não haja repetições desnecessárias na sequência.

3.1 Notas sobre a solução encontrada

A solução para tornar o grafo acíclico foi simplesmente retirar arestas até o grafo não ser mais cíclico. Isso foi feito de acordo com o valor de k das arestas, que corresponde ao overlap entre as últimas letras de u e as primeiras de v. Ou seja, as arestas com k menor são retiradas primeiro, porque entende-se que é menos provável que esse overlap seja significativo. Dessa forma, é possível reconstruir a sequência original de forma satisfatória.

4 Testes e análise

A partir das impressões das listas de adjacência e da reconstrução da sequência (maior caminho no grafo), é possível perceber que as funções implementadas *(buildGraph(), hasCircuit(), removeEdges(), findMaxPath()*) estão funcionando perfeitamente, já que o EP constrói o grafo corretamente, decide corretamente se ele é cíclico ou não, remove as arestas de forma a tornar um grafo cíclico em um acíclico corretamente e acha o maior caminho no grafo (ou um dos). O EP também reconstrói a sequência corretamente, como veremos a seguir.

Utilizei o programa *inputCreator.cpp*, também disponibilizado na entrega, para criar mais arquivos de teste (dna2.txt, dna3.txt, dna4.txt, dna5.txt e dna6.txt), os quais também disponibilizei. Esse programa cria uma sequência aleatória com um dado tamanho, depois separa-a em fragmentos, dados tamanho do fragmento mínimo e máximo, overlap mínimo e máximo e número de fragmentos, e depois salva os fragmentos em um arquivo conforme o padrão do input do EP.

Os resultados dos testes que mais se aproximaram da sequência original e parâmetros utilizados no programa *inputCreator.cpp* são mostrados abaixo. O primeiro teste foi feito com o exemplo dado no enunciado do EP, enquanto os outros foram criados, como já mencionado com o programa *inputCreator.cpp*, variando os parâmetros de forma a observar o comportamento do programa em vários casos.

```
Arquivo: dna1.txt (exemplo do enunciado do EP)
Sequência original: ACTCGTAAATACATAACGATAC
K=3:                  TCGTAAATACATAACGATAC

Arquivo: dna2.txt
sequenceLength = 22;
desiredNumFragments = 12;
minFragmentLength = 5;
maxFragmentLength = 8;
minOverlap = 2;
maxOverlap = 5;

Sequência original:CGGCCTTCCACAGGTAAGCGTC
K=3:               CGGCCTTCCACAGGTAAGC

Arquivo: dna3.txt
sequenceLength = 40;
desiredNumFragments = 21;
minFragmentLength = 11;
maxFragmentLength = 15;
minOverlap = 2;
maxOverlap = 5;

Sequência original/reconstrução com K=4:

AGACGGCTAGAGACTCAACCGTAGTTCCATGACTCCCTAC
AGACGGCTAGAGACTCAACCGTAGTTCCATGACTCCCT

Arquivo: dna4.txt
sequenceLength = 10;
desiredNumFragments = 6;
minFragmentLength = 3;
maxFragmentLength = 4;
minOverlap = 1;
maxOverlap = 2;

Sequência original:AGAATATAGG
K=1:               AGAATATAGG

Arquivo: dna5.txt;
sequenceLength = 30;
desiredNumFragments = 16;
minFragmentLength = 9;
maxFragmentLength = 11;
minOverlap = 2;
maxOverlap = 5;

Sequência original/reconstrução com K=4:
```

```
GAGGTGATCCAAGATAGCGCGCTGGTTGCT
GAGGTGATCCAAGATAGCGCGCTGGTTG

Arquivo: dna6.txt;
sequenceLength = 50;
desiredNumFragments = 27;
minFragmentLength = 15;
maxFragmentLength = 17;
minOverlap = 2;
maxOverlap = 5;

Sequência original/reconstrução com K=4:

TTTGTCCCTTTGATCTACAGGTCTCCTTAGTTACTCACGTCATCTGGCGC
TTTGTCCCTTTGATCTACAGGTCTCCTTAGTTACTCACGTCATCTGGCG
```

Nota-se que o programa não reproduziu as sequências inteiras na maior parte dos casos, mas como a diferença foi muito pequena (de no máximo 3 letras nos exemplos anteriores), considerei que o objetivo do EP foi alcançado satisfatoriamente. Creio que essa diferença seja devido aos parâmetros utilizados no *inputCreator.cpp*, principalmente os condizentes ao tamanho dos fragmentos e ao tamanho da sequência original.

Além disso, percebe-se a partir dos resultados dos testes que o valor ótimo de k para a criação do grafo depende do arquivo base, na maior parte dos casos anteriores está entre 3 e 4. Analisando os resultados, esse valor parece aumentar conforme o tamanho da sequência aumenta.