EP3 – Algoritmos e Estruturas de Dados II

Adriano Elias Andrade

23/06/2023

1. Introdução:

O programa principal “ep3.cpp” recebe uma entrada com os fragmentos do dna original e um k, e com isso tenta reconstruir o dna.

2. Compilação e entrada:

Compilação: Para compilar o ep, basta usar o comando “make”.

Entrada: A entrada do programa deve seguir a seguinte estrutura:

```
<Número de fragmentos do dna> <k (o mesmo do enunciado)>
<Fragmento 1>
...
<Fragmento N>
```

3. Implementação:

Na implementação, foi feita a classe “grafo”, que faz as operações no grafo, e a biblioteca de funções “palavra”, que é usada na montagem do arquivo da lista de adjacências, além do programa principal, “ep3.cpp”, que recebe a entrada, e devolve o dna reconstruído.

3.1 Montagem das conexões entre fragmentos:

Para montar a lista de adjacência inicial, é feito para cada fragmento “u”, o seguinte processo: vemos se seu começo encaixa com o fim de cada outro fragmento “v” dado. Caso encaixe, anotamos uma aresta na lista de adjacências “v -> u”, indicando que é possível encaixar v e u, com v na esquerda e u na direita. Esse processo é feito respeitando k, o encaixe de tamanho mínimo dado.

Para cada encaixe feito, também é verificado o tamanho do encaixe, ou seu “overlap”. Por exemplo, “AAACTC” e “CTCGGG” têm um overlap de tamanho 3, “CTC”. Isso também é anotado na lista de adjacências, como o peso de uma aresta.

A lista de adjacências é salva em um arquivo “lista”, que têm na primeira linha, a quantidade de arestas nela, e em cada linha seguinte, as arestas na forma “<v> <u> <overlap>”, indicando que u encaixa no final de v, com um overlap. O arquivo “lista” é removido assim que não for mais ser utilizado.

3.2 Estrutura do grafo:

Na estrutura do grafo, temos:

- **nVertices:** O número de vértices do grafo;

- **adj[ ][ ]:** A lista de adjacência (diferente da lista montada no arquivo);

- **fragmentos[ ]:** onde fragmentos[i] corresponde à string do fragmento do vértice i;

- **rep[ ]:** representantes dos vértices;

3.3 Montagem do grafo:

Para montar o grafo, seria melhor ter as arestas mais pesadas nele, pois como indicam que um fragmento tem mais em comum com outro, é mais provável que essas conexões estejam corretas.

Para isso, poderíamos remover as arestas mais leves, até que não reste mais ciclos. Porém é mais simples resolver esse problema ao contrário: montar o grafo, inserindo primeiro as arestas mais pesadas, caso não formem um ciclo. Para isso, ordenamos a lista de adjacência, de maior a menor peso. Então colocamos nessa ordem, cada aresta, antes verificando se ela não forma um ciclo. As arestas são colocadas na lista de adjacência do grafo, adj.

Para verificar se uma aresta forma um ciclo ao ser colocada, marcamos o vértice de onde a aresta sai como “pai”, e então é feita uma dfs no vértice para onde essa aresta aponta. Caso esse dfs volte para o pai em algum momento, então há um ciclo sendo formado, e não devemos colocar a aresta. Neste processo, também marcamos os novos representantes dos vértices durante a dfs num vetor auxiliar, e caso a aresta seja colocada, o vetor de representantes da estrutura é atualizado.

Na lista de adjacência (adj) do grafo, há um vetor de vectors de pares, onde adj[u][i] = (v,p), indica que há uma aresta “u -> v”, com peso p. “i” é apenas o índice de um elemento em adj[u].

3.4 Caminho mais longo:

Para achar o caminho mais longo no grafo, inicialmente separamos os vértices de onde faz sentido começar este caminho, isto é, os vértices que não têm nenhuma aresta apontando para eles. Para isso, como já temos um vetor de representantes construído, basta separar os diferentes representantes de cada vértice.

Um caminho é representado por uma lista vector<int>, onde o primeiro elemento da lista é o vértice inicial, e cada próximo elemento da lista é o próximo vértice no caminho.

Para encontrar o maior caminho iniciando de um vértice “R”, um dos representantes, começamos marcando um valor para os vértices que não apontam para nenhum outro como 0. Então, para cada vértice, o valor deles será o máximo do valor entre seus filhos, + 1. Assim, o vértice R terá o tamanho do maior caminho, e cada seguinte vértice no maior caminho terá valor igual ao valor do pai menos 1.

Então, para cada um dos representantes, achamos o maior caminho neles, e caso seja achado um novo maior caminho, este é atualizado.

Agora com o maior caminho conhecido, basta percorrê-lo montando o dna, o que é trivial já que o overlap entre dois vértices já é conhecido como o peso das arestas.

4. Parâmetros bons, e limitações:

Tamanho dos fragmentos: Quanto maior o tamanho dos fragmentos dados, maior será o tamanho dos overlaps entre eles, e portanto, melhor é a reconstrução do dna.

k: com um k muito grande, acabamos desconsiderando muitas conexões corretas, e o dna fica menor do que o original. Com um k muito pequeno, acabamos criando conexões falsas, e o dna fica maior do que deveria ser. Portanto um bom valor para k, é algo maior ou igual a 10, o que já deixa a probabilidade de erro baixa. Porém, para sequências de dna grandes (maiores que 1000), não há problema em utilizar k como 1% do tamanho do dna original. Não é bom utilizar um k maior que metade do tamanho dos fragmentos.

Quantidade de fragmentos: Quanto mais fragmentos utilizados, melhor a aproximação, porém isso também afeta o tempo de execução. A quantidade de fragmentos utilizados depende bastante do k. Com valores altos de k, é possível utilizar vários fragmentos. Mas com valores baixos (por exemplo 10), não é possível utilizar muito mais de 30 fragmentos, pois o programa passa a demorar demais.

Tamanho do dna original: O tamanho do dna afeta pouco o tempo de execução do programa, só por volta de 10 milhões o programa passa a demorar mais de 10 segundos. Desde que o tamanho dos fragmentos seja aumentado proporcionalmente, sequências de dna maiores são mais bem reconstruídas, já que o overlap entre seus fragmentos é maior.

5. Testes:

A seguir, alguns exemplos de testes, mostrando diferentes casos bons, ou ruins. Os testes foram feitos sorteando pedaços aleatórios do dna, com um tamanho mínimo e máximo. É garantido que o início e o fim do dna original estão na lista.

5.1 Fragmentos grandes com k alto (bom): String original de tamanho 50, com tamanho dos fragmentos entre [30, 40], k=10.

String original: `CGTTTTCATACTAGAGACCTCATGACTTAAGGGCTTAAAAATCTCCCGTA`

Entrada:

```
10 10
CGTTTTCATACTAGAGACCTCATGACTTAAGG
GTTTTCATACTAGAGACCTCATGACTTAAGGGCTTA
TACTAGAGACCTCATGACTTAAGGGCTTAAA
ACTAGAGACCTCATGACTTAAGGGCTTAAAAAT
CATACTAGAGACCTCATGACTTAAGGGCTTAAAAAT
CATACTAGAGACCTCATGACTTAAGGGCTTAAAAATCTCC
TTTTCATACTAGAGACCTCATGACTTAAGGGCTTAA
GTTTTCATACTAGAGACCTCATGACTTAAGGGCTTAAAAA
AGACCTCATGACTTAAGGGCTTAAAAATCTC
CTCATGACTTAAGGGCTTAAAAATCTCCCGTA
```

String reconstruída: `CGTTTTCATACTAGAGACCTCATGACTTAAGGGCTTAAAAATCTCCCGTA`

(igual à original)

5.2 k muito baixo e fragmentos pequenos (ruim): String original de tamanho 20, com tamanho dos fragmentos entre [3, 8], k=2.

String original: `CTCGTCGGCGGTTTGTAATC`

Entrada:

```
15 2
CTCG
TCGTC
GTA
GTA
GTTTGTAA
GGTTT
TTTGT
GTTTGTA
CGTCGG
GGCGGTTT
GGTTTG
CTCGTCGG
GGCGG
TCGGCGG
TGTAATC
```

String reconstruída: `CTCGTCGGCGGCGGCGGTTTGTTTGTAATC`

(10 letras a mais do que a original)

5.3 k muito alto (ruim): String original de tamanho 50, com tamanho dos fragmentos entre [20, 30], k=20.

String original: `CGTCAGAGAAGATTTATAGACCAGAGTCTTCCACTAAGTAGATCCGTTGC`

Entrada:

```
10 20
CGTCAGAGAAGATTTATAGACCAG
AGATTTATAGACCAGAGTCTTCCACTAA
ATTTATAGACCAGAGTCTTCCA
AGATTTATAGACCAGAGTCTTCCA
GAAGATTTATAGACCAGAGTCTTCCACTA
ATAGACCAGAGTCTTCCACTAAG
```

GATTTATAGACCAGAGTCTTCCACTAAG

GAGAAGATTTATAGACCAGAG

AAGATTTATAGACCAGAGTCTTCC

GTCTTCCACTAAGTAGATCCGTTGC

String reconstruída: GAAGATTTATAGACCAGAGTCTTCCACTAAG

(19 letras a menos do que a original)

5.4 dna "grande" com k pequeno em comparação (bom, mas o k poderia ser maior): String original de tamanho 1000 (é possível utilizar tamanhos bem maiores, como dito anteriormente, mas não caberia no pdf). Tamanho dos fragmentos entre [200, 300], k=10 (1% do tamanho da string original).

String original:
ACGGCAATGAAGTTAGACCAAAATTGGCTCTGGCTCGCCGAGAATACAGAGTGCTAACGCTAGG
CACTTTAGGTCAACATGACGCAAACCATCAAACAGGGGTTGCAGGGCGAATGGGAAGAGAAATT
TAACCCCAGGAGCGAAAACTAACCTTGCATAGAAGCGACAGGAAGGCGTGAATTAGGTGACGTA
GCGATCGGCTCGATAGGGTTCAGGGCCGCTTCGGTACGCCCCGTGGTAGATAATCTTGCTATCA
TCGCCTACGTCCTTGGCACAGTTAAGGACTCCCTCTAACCTGAGGTGGCATGAGACAGACTACT
GTGCTTTTGACGGACGCATCTCCTGTGCGAGACGGCCGAGTATTTCGCCCATAGACTATGACCA
ATTCGAAATCTAGGCCAACCTAGTTCCATTACTTATTTCCCTAAAATGGTATGAGGTAGCCAAC
ATGTAAACAGGGATGATGATGCCCCTTGAGCATATGATCATCGTCATTATACCTTCCCCAGGAC
TCAGGCATACGCAGACAATGTGACGGGAGCTGCGGCTAGCCGTGCTCTTGCTTGACGTGGAAAT
AAGTCGGTGCATGCAGTAATAAGAGATGCGTACATATACCCGTTATAAATACATCTAGTTAGTG
AATAAAGTTTTAATGTGGAAAGTAAGTGGAGCTCACAGCTTGGCCAAACCAGACACGGAGCTCT
CAGGCTTGTCCTCGCGGATAATGTGTCAGAGGTTCTCTTGAGACCACGCCAAATGTACCTGTTG
AAAGTAACGCGGCGTGGGCGGAGTAGTGCTCCTCTGCTGTCCATCTTCCCCCTGAACGAACGCA
TTGGAATTTCGGATCTTTCCCAAGCAAGCTCGCTCTTGCGACGCCCGTGTGTGCGTTGCGATCC
TTTATGTAGTACCGCTCTTTTAGCTAGCGCCCGATGCGGGCTGTCTGTTCAGACAGGATCGGGC
GTGACGTTAGAAGTTGAGCTGCCACAGCTTATTTTCCCTT

Entrada:

10 10

ACGGCAATGAAGTTAGACCAAAATTGGCTCTGGCTCGCCGAGAATACAGAGTGCTAACGCTAGG
CACTTTAGGTCAACATGACGCAAACCATCAAACAGGGGTTGCAGGGCGAATGGGAAGAGAAATT
TAACCCCAGGAGCGAAAACTAACCTTGCATAGAAGCGACAGGAAGGCGTGAATTAGGTGACGTA
GCGATCGGCTCGATAGGGTTCAGGGCCGCTTCGGTACGCCCCGTGGTAGATAATCTTGC

GGTGCATGCAGTAATAAGAGATGCGTACATATACCCGTTATAAATACATCTAGTTAGTGAATAA
AGTTTTAATGTGGAAAGTAAGTGGAGCTCACAGCTTGGCCAAACCAGACACGGAGCTCTCAGGC
TTGTCCTCGCGGATAATGTGTCAGAGGTTCTCTTGAGACCACGCCAAATGTACCTGTTGAAAGT
AACGCGGCGTGGGCGGAGTAGTGCTCCTCTGCTGTCCATCTTCCCCCTGAACGAAC

CAGAGTGCTAACGCTAGGCACTTTAGGTCAACATGACGCAAACCATCAAACAGGGGTTGCAGGG
CGAATGGGAAGAGAAATTTAACCCCAGGAGCGAAAACTAACCTTGCATAGAAGCGACAGGAAGG
CGTGAATTAGGTGACGTAGCGATCGGCTCGATAGGGTTCAGGGCCGCTTCGGTACGCCCCGTGG
TAGATAATCTTGCTATCATCGCCTACGTCCTTGGCACAGTTAAGGA

TAGTGAATAAAGTTTTAATGTGGAAAGTAAGTGGAGCTCACAGCTTGGCCAAACCAGACACGGA
GCTCTCAGGCTTGTCCTCGCGGATAATGTGTCAGAGGTTCTCTTGAGACCACGCCAAATGTACC
TGTTGAAAGTAACGCGGCGTGGGCGGAGTAGTGCTCCTCTGCTGTCCATCTTCCCCCTGAACGA
ACGCATTGGAATTTCGGATCTTTCCCAAGCAAGCTCGCTCTTGCGACGCCCGTGTGTGCGTTGC
GATCCTTTATGTAGTACCGCTC

CATGCAGTAATAAGAGATGCGTACATATACCCGTTATAAATACATCTAGTTAGTGAATAAAGTT
TTAATGTGGAAAGTAAGTGGAGCTCACAGCTTGGCCAAACCAGACACGGAGCTCTCAGGCTTGT
CCTCGCGGATAATGTGTCAGAGGTTCTCTTGAGACCACGCCAAATGTACCTGTTGAAAGTAACG
CGGCGTGGGCGG

GGCTTGTCCTCGCGGATAATGTGTCAGAGGTTCTCTTGAGACCACGCCAAATGTACCTGTTGAA
AGTAACGCGGCGTGGGCGGAGTAGTGCTCCTCTGCTGTCCATCTTCCCCCTGAACGAACGCATT
GGAATTTCGGATCTTTCCCAAGCAAGCTCGCTCTTGCGACGCCCGTGTGTGCGTTGCGATCCTT
TATGTAGTACCGCTCTTTTAGCTAGCGCCCGATGCGG

CAGGGCCGCTTCGGTACGCCCCGTGGTAGATAATCTTGCTATCATCGCCTACGTCCTTGGCACA
GTTAAGGACTCCCTCTAACCTGAGGTGGCATGAGACAGACTACTGTGCTTTTGACGGACGCATC
TCCTGTGCGAGACGGCCGAGTATTTCGCCCATAGACTATGACCAATTCGAAATCTAGGCCAACC
TAGTTCCATTACTTATTTCCCTAAAATGGTATGAGGTAGCCAACATGTAAACAGGGATGATGAT
GCCCCTTGAGCATATGATCATCGTCATTATACCTTCCCC

TTTCCCTAAAATGGTATGAGGTAGCCAACATGTAAACAGGGATGATGATGCCCCTTGAGCATAT
GATCATCGTCATTATACCTTCCCCAGGACTCAGGCATACGCAGACAATGTGACGGGAGCTGCGG
CTAGCCGTGCTCTTGCTTGACGTGGAAATAAGTCGGTGCATGCAGTAATAAGAGATGCGTACAT
ATACCCGTTATAAATACATCTAGTTAGTGAATAAAGTTTTAATGTGGAAAGTAAGTGGAGCTCA
CAGCTTGGCCAAAC

TAGGGTTCAGGGCCGCTTCGGTACGCCCCGTGGTAGATAATCTTGCTATCATCGCCTACGTCCT
TGGCACAGTTAAGGACTCCCTCTAACCTGAGGTGGCATGAGACAGACTACTGTGCTTTTGACGG
ACGCATCTCCTGTGCGAGACGGCCGAGTATTTCGCCCATAGACTATGACCAATTCGAAATCTAG
GCCAACCTAGTTCCATTACTTATTTC

TCCTCGCGGATAATGTGTCAGAGGTTCTCTTGAGACCACGCCAAATGTACCTGTTGAAAGTAAC
GCGGCGTGGGCGGAGTAGTGCTCCTCTGCTGTCCATCTTCCCCCTGAACGAACGCATTGGAATT
TCGGATCTTTCCCAAGCAAGCTCGCTCTTGCGACGCCCGTGTGTGCGTTGCGATCCTTTATGTA
GTACCGCTCTTTTAGCTAGCGCCCGATGCGGGCTGTCTGTTCAGACAGGATCGGGCGTGACGTT
AGAAGTTGAGCTGCCACAGCTTATTTTCCCTT

String reconstruída: (igual à original)

5.5 k pequeno com muitos fragmentos (ruim): String original de tamanho 10, com tamanho dos fragmentos entre [2, 4], k=2, 50 fragmentos.

String original: GAGACTAGGA

Entrada:

50 2

GAGA AGG GAGA TAG GAC ACTA GAG GAG GAGA AC TAG GACT ACT GAG AC
TA TA AG TAGG CTAG ACTA CTA CTAG GA AG AG TAGG GAC AGA AGA AG

AGAC GAGA GA AG AG GA AGAC GAC GA TAGG GACT GAC TAG ACTA TAGG ACT AGA GAGA AGGA

String reconstruída: AGAGAGAGAGAGACTAGGACT

(11 letras a mais do que a original)