Universidade de Évora

Teoria da Informação

TRABALHO FINAL 2015/2016

Janeiro 8, 2016

Discentes:

27421 - Pedro Jacob
31626 - André Figueira

Docente:

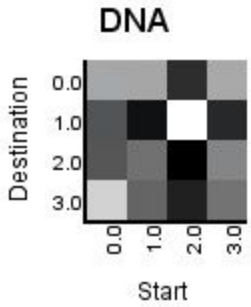
Luís Rato

1. Introdução

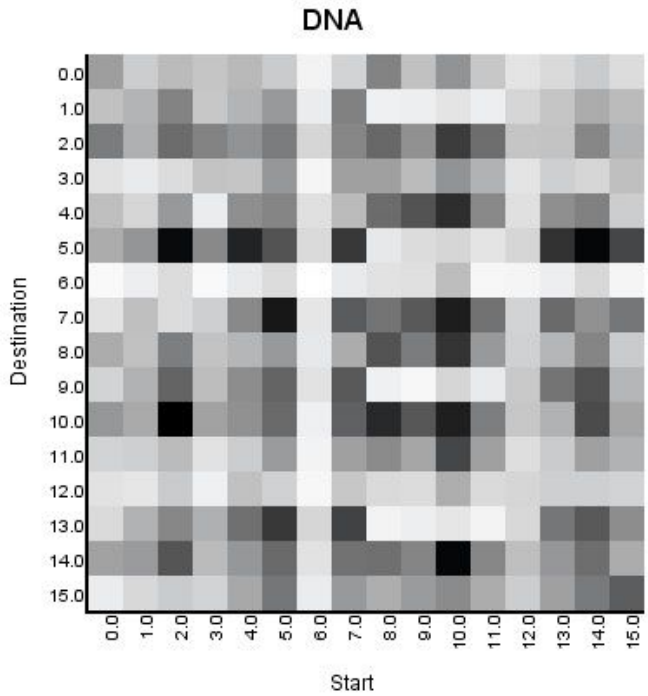
O Trabalho tem como objetivo, a aplicação de técnicas de compressão a uma sequência de ADN (sequências de pares base), o qual tem um alfabeto com apenas 4 símbolos - A, C, T, G. A aplicação de técnicas de compressão a sequências de ADN é uma das técnicas usadas para medição de complexidade e entropia e tem aplicação na deteção e comparação de genes. Neste trabalho será usado um conjunto de dados retirado dum conjunto denominado HMR-195. O conjunto de Dados HMR-195, é um conjunto de dados usado como bechmark, e tem 195 genes (humanos e não humanos). Este conjunto inclui o gene humano "SKIV2L" (com o ID AF059675), que é responsável pela codificação de uma enzima, a "Helicase SKI2W" e será este gene que será usado como modelo da fonte.

2. Descrição da Fonte (Tarefa 1)

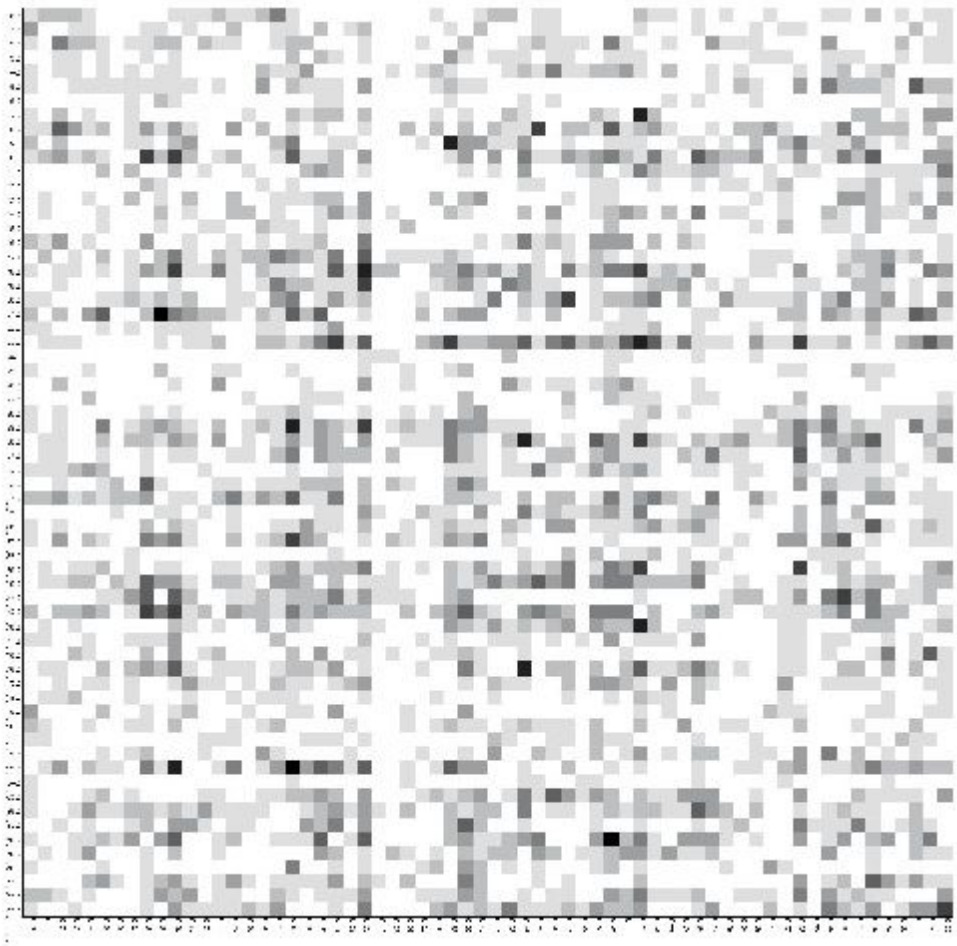
Visto que, a fonte tem como contexto uma sequencia de ADN, os símbolos desta não são certamente independentes. Como tal, decidimos estudar a mesma como um processo de markov. Começamos por considerar cada simbolo um a um, representado os resultados em forma de tabela de ocorrências e de transição (tal que a soma de uma coluna de transição = 1) Para melhor visualização, a tabela vem representada como um heatchart.



Quando começamos a considerar que combinações de símbolos 2 a 2, tal que formassem um total de 16 símbolos, decidimos criar uma codificação numérica para os símbolos individuais, Dando A C T G os valores 0 1 2 3 respectivamente, e calculando os seus valores na matriz como base 3. Desta maneira o código pode ser facilmente modelado para qualquer número de símbolos a considerar com um único simbolo.
Para Combinações 2 a 2, a matriz tem o seguinte aspecto em forma de heatchart.



Os resultados interessantes são confirmados quando olhamos para a matriz que representa combinações de símbolos 3 a 3. Isto deve-se, provavelmente, na área de microbiologia combinações 3 a 3 têm significado, seja codões de parada ou codões de inicio.



Podemos observar uma falta de ocorrências em certos valores, que correspondem a combinações de valores que contenham a combinação GC. Isto, não é aleatório. Visto que DNA tem de ter codões de parada obrigatoriamente e estes mesmos apenas tomam os formatos : TAG, TAA e TGA, então podemos concluir que sequencias com um numero alto de codões de parada reduzem o numero de ocorrência de conteúdo GC.

2.1. Entropia da Fonte

A entropia para as cadeias de markov com as combinações 1 para 1, 2 para 2 e 3 para 3, são as seguintes.

Entropia 1 a 1:

H(x)=1.9933420376124567

Entropia 2 a 2:

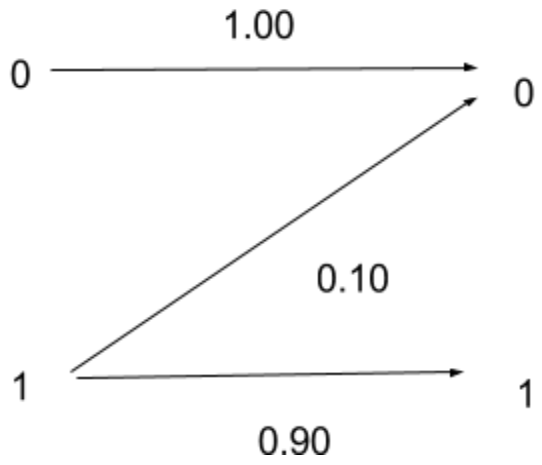
H(x)=3.90965597110501153

Entropia 3 a 3:

H(x)=5.807813202571877

3. Canal

O canal usado neste trabalho é um canal binário assimétrico em Z que tem uma entrada binaria, e uma saída binaria onde a transferência 1 para 0 ocorre com probabilidade não negativa P enquanto que a transferência 0 para 1 ocorre com probabilidade igual a 0.



Para implementação, o programa recebe um ficheiro binário , carregando-o em memoria. Corre, e aplica ao canal (Ou seja, corre o array de bytes que esta em memória, verificando bit a bit para se o bit está a 1, e , se tiver, aplica um random com 10 por cento de chance. Se o random se verificar, esse bit passa a 0) ao array de bytes, guardando-o em disco como ficheiro binário no final da operação.

3.1. Capacidade do canal

Visto que se trata de uma canal assimétrico com probabilidade de erro de 0.10 então:

$C = \max I(X,Y) = H(Y) - H(Y|X)$

$C = \log(1-p+p(p/(p-1))- (p / p-1)*\log p$

Neste caso como p = 0.10 então substituindo p, a capacidade do canal será aproximadamente C = 0.7

4. Codificação

A codificação da sequência é feita primeiramente em Huffman e depois aplicado Hamming.

4.1. Huffman Code

A implementação é composta por duas classes.

4.1.1. HuffmanNode

Representa um nó da arvore, isto é, simbolo, frequência, filho esquerdo e filho direito e código actual (\0 por default).

4.1.2. Huffman

Esta classe é composta por vários métodos. Mas para codificação só interessa os que criam a árvore e percorrem a árvore para fazer os códigos de Huffman.

Para criar a árvore, através da lista com as probabilidades de ocorrências dos símbolos, e os simbolos, usa-se para criar vários HuffmanNodes, que vão ser as folhas da árvore. Estes nós são introduzidos numa lista de prioridade. Em seguida percorre-se a lista, agrupando os nós mais pequenos (estes removidos da lista) e cria-se um nó pai (com o simbolo \0 e código \0 por default) que é introduzido na lista. Quando o tamanho desta lista for só 1 elemento. Significa que a arvore está concluída pois este único elemento representa o Root da árvore, ou seja, o nó principal.

Para determinar os códigos, percorre-se esta mesma árvore usando recursividade. Se o nó não for folha então realiza chamada recursiva, para o lado direito da arvore e lado esquerdo, fazendo update ao código (+1 para direito, +0 para esquerdo) de forma a que toda a arvore seja percorrida.

```
public void makeHuffmanCode(String codes[], HuffmanNode node, String currentCode)
{
    // runs the huffman tree top to bottom, going through all nodes -> makes code
    if (!node.isLeaf())
    {
        makeHuffmanCode(codes, node.child_ESQ, currentCode + "0");
        makeHuffmanCode(codes, node.child_DIR, currentCode + "1");
    }else
    // since its a leaf means there is a symbol and not a '\0', so add it and update the node.
    {
        node.setCode(currentCode);
        dictionary.put(node.getSymbol(), currentCode); // correspond code to symbol
    }
}
```

Se for folha significa que encontro um simbolo,(para todos os outros nós são representados por \0) então actualiza código do nó (que era ‘\0’ por default) para o código actual.

Depois basta percorrer a sequência, 3 símbolos de cada vez, substituindo esses símbolos pelo seu código de Huffman.
Após se ter a codificação do conteúdo criam-se dois ficheiros binários. 1 deles com a codificação de Huffman em binário e outro com a informação da árvore. Este ficheiro da árvore é importante para descodificar. Nesse segundo, o utilizador tem de introduzir também o ficheiro da árvore correspondente para descodificar. O nome do ficheiro é Huffman_Tree_<Nome>.bin, sendo <Nome> o nome do ficheiro original. por exemplo Huffman_Tree_AF059675.bin.

4.2. Hamming Code

Após a sequência ter sido transformada em Huffman, aplica-se então código de Hamming. Usou-se o código 7,4 visto que foi o mais falado na aula. O código é gerado através do uso de uma matriz, a Geradora.

Na parte da codificação, usando o código, lendo 4 a 4, transforma-se esses 4 numa matriz de 4 por 1 para ser multiplicada pela matriz Geradora , para gerar um código de 7 por 1, este então é desconvertido para deixar de ser uma matriz e adicionar o resultado á nova sequência codificada em Hamming. Repete-se para todos os conjuntos de 4.

4.2.1. Comparação com o ficheiro original e o ficheiro codificado

Usando como teste o ficheiro AF059675.txt, que tem 11kb.

4.2.2. Codificado usando Huffman

Obteve-se um cheiro binário de 4kb, que equivale a uma compressão de aproximadamente de 63% do tamanho original.

4.2.2. Codificado usando Huffman + Hamming

Obteve-se um ficheiro binário de 5kb, que equivale aproximadamente a uma compressão de 54% do tamanho original.

Nota:
Ao aplicar o programa code, gera-se três ficheiros. Dois deles já foram descritos na secção Huffman que nao são passados por argumento, o terceiro ficheiro é o de output passado por argumento que contem a codificação com Huffman e Hamming. Este é o utilizado na descodificação e canal.

4.3. Descodificação

Descodificação é feita em duas partes.. A primeira parte é feita usando Hamming e a segunda Huffman.

4.3.1. Error Detection

Para corrigir qualquer erro, o código é lido 7 a 7. A cada 7 estes são transformados numa matriz e multiplicados pela matriz Verificação de Erro. Esta gera uma nova matriz de 3 por 1 que contem a posição onde ocorreu tal erro. Ao converter esta matriz numa valor,por exemplo [1 1 0] obtemos 110 que corresponde em binário á posição dos 7 elementos onde há um erro. Se o valor inteiro for 0, então assume-se que não houve nenhum erro na matriz e passa para a descodificação. Caso contrário, na matriz que contém o bloco altera-se o bit errado para o oposto (usando um NOT), apos concluído passa para a descodificação.

Problemas: Hamming não distingue código com um erro de um código com dois erros porque pode assumir que e um código com um erro gerado por outra palavra.

4.3.2. Decode usando Hamming

Depois de aplicar a correção de erros, o bloco de 7 passa por uma descodicação. esta descodificação simplesmente multiplica pela matriz Descodificadora e transforma a matriz (4 por 1) num valor que é adicionado para uma String com o código completo. Quando se obtém o código todo completo descodificado falta aplicar Huffman.

4.3.3. Decode usando Huffman

Antes de descodificar o programa tem de receber a informação sobre a arvore. Isto provém do cheiro HuffmanTree<nome-original>.bin que e recebido como parâmetro.

Exemplo: java decode bits-output.txt seq-output.txt Huffman_Tree_AF059675.bin

Ao receber este parametro sabe então que a árvore corresponde a sequência de bits passada pelo parâmetro. Para descodificar usando Huffman o conteúdo do cheiro e passado para uma String que provém da descodicação de Hamming.

Como os códigos de huffman não têm comprimento fixo tem de se percorrer o código posição a posição.

Ao lendo cada posição vai se percorrendo a árvore, fazendo verificações se é uma folha ou nao. Se o nó da árvore não for uma folha, tem duas opções, ou a árvore vai para a esquerda porque leu um 0, ou direita porque leu um 1.

Se é uma folha, então adiciona-se a String que contém o código descodificado o simbolo correspondente á codificação que se leu até aquele ponto e o nó passa a ser o Root (inicia uma nova descida pela árvore)

4.4. Comparação com o ficheiro original e o ficheiro descodificado

Os testes foram realizados com o ficheiro AF059675.txt, e usando este cheiro após vários testes consecutivos viu-se que em comparação com os dois ficheiros o número de erros variava.

Erros em alguns testes com canal:

Número de Erros do teste 1 = 7973
Número de Erros do teste 2 = 8150
Número de Erros do teste 3 = 8098
Número de Erros do teste 4 = 8043

De acordo com estes testes em média dá 8064 por teste.
Num contexto mais geral podemos estimar que ronda em média os 8100 erros por teste.

Sem canal o número de erros é 0.

Nota:
Este número de erros deve-se a detecção de erros por parte de Hamming. Visto que nao consegue-se distinguir um código com um erros de um código com dois erros isto deve se a assumir que é um código com um erro gerado por outra palavra.

5. Grupo de testes

Em todos os testes se usou o ficheiro AF059675.txt, portanto têm resultados muito semelhantes ou iguais.

Em todos os testes se notou o seguinte:

> Em todos os testes realizados o número de caracteres total no ficheiro original e ficheiro final são diferentes como esperado tenha passado pelo canal ou não, isto também se deve a se ter usado o método de 3 a 3.
>
> O número de erros é enorme e reparou-se em todos os testes, cerca de 70% do ficheiro descodificado está errado, ficando completamente corrompido. Isto deve-se ao facto de correcção de erros aplicando Hamming, não conseguir distinguir códigos de 2 ou mais erros de um código com 1 erro. Pois esse código com 2 ou mais erros, usando Hamming, significa um código com 1 erro mas gerado por outra palavra. Logo pouco eficiente.
>
> A compressão é eficiente, visto que em todos os testes se comprimia para valores muito semelhantes, de 4kb para huffman e 5kb para huffman + hamming.
>
> Huffman é eficiente pois sem aplicar o canal, nota-se que fica igual, o que é esperado.
> Hamming também é eficiente, á excepção da correcção de erros, pois sem aplicar canal, nota-se que o ficheiro fica igual, o que é esperado.

5.1 Exemplo de Comandos para os testes:

javac code.java decode.java Channel.java General Functions.java
HammingCode.java Human.java HuffmanNode.java Triple Genome.java Genome.java

java code AF059675.txt bits-input.bin

java Channel bits-input.bin bits-output.bin

java decode bits-output.bin seq-output.txt Huffman_Tree_AF059675.bin

Nota:

Em todos os comandos, o primeiro argumento é um ficheiro já existente e o segundo argumento é o **nome** do ficheiro de output que depois vai ser criado. No caso do decode, o terceiro argumento é um ficheiro criado por code.

Ao usar **code AF059675.txt bits-input.bin** cria dois ficheiros adicionais. Um deles é a compressão usando só huffman, o outro é o ficheiro binário que se refere a árvore correspondente á sequência (por exemplo: AF059675.txt). Isto serve para em decode conseguir carregar a árvore correspondente, como o exemplo indica e corresponder o output a essa arvore por causa das codificações.

6. Conclusão

O nosso conhecimento certamente aumentou com a execução deste trabalho, em termos de codificação, descodificação, correcção de erros, funcionamento de um canal e especialmente no estudo da fonte, que nos interessou e nos fez procurar o porquê dos heatcharts terem o comportamento descrito na secção.