Alguns métodos comparativos para seqüências de DNA

Augusto Fernandes Vellozo

Relatório de estudo para MAC 5701

25 de junho de 2002

Alguns métodos comparativos para seqüências de DNA

Para a extração de informações de uma seqüência de DNA, comparações com seqüências mais conhecidas sempre foram utilizadas.

Os principais métodos para comparar seqüências de DNA, utilizam o alinhamento de seqüências como forma de comparação, e o alinhamento de seqüências pode ser visto como uma generalização de uma LCS *(Longest Commom Subsequence)*. Ademais, os principais algoritmos para obtenção de uma LCS utilizam técnicas de programação dinâmica, e não é diferente com o algoritmo do Smith-Waterman para alinhar duas seqüências. Estes algoritmos são quadráticos no comprimento da seqüência, o que faz com que o alinhamento de seqüências do tamanho de um genoma fique inviável.

Para tornar a comparação de seqüências do tamanho de um genoma viável, é necessário que sejam desenvolvidos algoritmos assistidos por técnicas de comportamento linear. Um exemplo de sistema que pode ser utilizado para comparação de genomas de espécies próximas é o MUMmer. Este sistema utiliza árvore de sufixos como uma técnica chave, já que a construção e a busca em uma árvore de sufixos é linear no comprimento da seqüência, o que torna o MUMmer viável para a comparação entre genomas.

A seguir, apresentaremos:

1. O Problema da LCS
2. Árvore de sufixos
3. O sistema MUMmer

1. O Problema da LCS

1.1 - Introdução

Seja $A= a_1a_2a_3 \ldots a_m$ e $B=b_1b_2b_3 \ldots b_n$ duas seqüências sobre o alfabeto $\Sigma = \{\sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_s\}$. Uma seqüência $C = c_1c_2c_3 \ldots c_l$ é chamada uma subseqüência de A, se C puder ser obtida de A deletando um certo número de símbolos (não necessariamente consecutivos) de A.

C é uma subseqüência comum de A e de B, se e somente se C é uma subseqüência de A e C é uma subseqüência de B.

O problema da maior subseqüência comum (*Longest Common Subsequence* ou LCS) é determinar uma subseqüência comum C entre duas seqüências A e B, tal que nenhuma outra subseqüência comum de A e B tenha comprimento maior que C. É importante deixar claro que C pode não ser única. Definimos comprimento de C, ou $|C|$, como o número de símbolos que compõem C. Muitas vezes estamos apenas interessados na determinação do comprimento da LCS.

Um par ordenado (i, j) de posições em A e B é chamado um *match* se e somente se $a_i = b_j$. Seja $A_i = a_1a_2 \ldots a_i$ e $B_j = b_1b_2 \ldots b_j$, definimos $L_{i,j} = \max\{ |C| : C$ é uma subseqüência comum de A_i e $B_j\}$.

1.2 - Método da programação dinâmica

Um método simples para resolver o problema, pode ser obtido a partir da matriz de programação dinâmica obtida das recorrências:

$L_{0,j} = 0$

$L_{i,0} = 0$

$L_{i,j} = L_{i-1,j-1} + 1$, se $a_i = b_j$

$L_{i,j} = \max \{L_{i-1,j}, L_{i,j-1}\}$, se $a_i \neq b_j$

Neste método todos os valores de $L_{i,j}$ podem ser calculados em tempo e espaço $\boldsymbol{O(mn)}$. É fácil perceber que somente os valores $L_{i-1,j-1}$, $L_{i-1,j}$ e $L_{i,j-1}$ são utilizados no cálculo de $L_{i,j.}$ Uma pequena melhora possibilita que seja utilizado espaço linear se quisermos obter somente o valor de $L_{m,n}$, que é o comprimento da LCS.

Vários outros algoritmos para a solução do problema da LCS foram propostos. Não temos a pretensão de relatá-los ou descrever os mais eficientes, mas faremos agora uma breve descrição de dois destes algoritmos. Estes são particularmente vantajosos em alguns subcasos, como por exemplo, quando o número de *matches* for pequeno.

1.3 - Método Hunt/Szymanski

Se (i, j) é um *match* e $L_{i,j} = k$, então dizemos que o ***rank*** de (i, j) é igual a k. Dizemos que o *match* (i, j) é ***k*-dominante,** se e somente se ele tem *rank* k e para qualquer outro *match* (i', j') de *rank* k, ou $i' > i$ e $j' \leq j$, ou $i' \leq i$ e $j' > j$. O número total de *matches* dominantes será denotado por d.

Para toda classe C_k (os *matches* com *rank* k) existe pelo menos um *match* (i, j) que é k-dominante, pois se não existisse, isto implicaria que existiria algum *match* (i', j') de *rank* k, tal que $i' > i$ e $j' > j$. Mas se isto acontecesse, o *match* $(i', ,j')$ não poderia ter *rank* k, pois bastaria acrescentarmos à subseqüência comum de A_i e B_j de comprimento k o símbolo $a_{i'}$ (que é igual a $b_{j'}$) e obteríamos uma subseqüência comum de $A_{i'}$ e $B_{j'}$ de comprimento k+1.

Se $L_{m,n} = l$ então existe algum *match* (i, j) tal que o *rank* de (i, j) é igual a l e $i \leq m$ e $j \leq n$, desta forma temos pelo menos um *match* l-dominante.

O método do Hunt/Szymanski se baseia no fato que existe uma maior subseqüência comum de A e B formada somente por *matches* dominantes.

Abaixo temos o algoritmo de **Hunt/Szymanski** :

Entrada: Seqüências $A= a_1a_2a_3 \ldots a_m$ e $B=b_1b_2b_3 \ldots b_n$, $m \leq n$.

Saída: Uma LCS $C = c_1c_2c_3 \ldots c_l$

Início

Thresh[0] := 0

para *k* := 1 até *m* faça

Thresh[*k*] := *n*+1

para *i* := 1 até *m* faça

para *todo j decrescendo com* $b_j = a_i$ faça

encontre k tal que Thresh[k-1] $<$ *j* $\leq$ *Thresh[k]*

se *j* $<$ *Thresh*[*k*] então

Thresh[*k*] := *j*

Link[*k*] := *new node* (*i*, *j*, *Link*[*k*-1])

k := *m*

Enquanto *Thresh*[*k*] = *n*+1 faça

k := *k*-1

PTR := *Link*[*k*]

Enquanto PTR $\neq$ nil faça

c_k := símbolo do *match* apontado por PTR

k := *k*-1

avança PTR

Fim

Seja r o número de *matches* entre *A* e *B*, então o algoritmo será executado em tempo $\boldsymbol{O(m + r \log l)}$.

1.4 - Método Apostolico/Guerra

O método do Hunt/Szymanski possui 2 pontos que podem originar problemas de performance. Um é no momento em que é feita uma busca binária para identificar o *rank* de cada *match*. O outro é o fato dele percorrer por todos os *matches* à procura dos *matches* dominantes.

O método de Apostolico/Guerra utiliza uma matriz como descrita baixo:

$CLOSEST[\sigma_i , n + 1] = n + 1$

$CLOSEST[\sigma_i, j] = \min\{\{j' \geq j \mid b_{j'} = \sigma_i\} \cup \{n+1\}\}$, para $j = 1, 2, .., n$

Algoritmo **Apostolico/Guerra**

Entrada: Seqüências $A = a_1a_2a_3 \ldots a_m$ e $B = b_1b_2b_3 \ldots b_n$, $m \leq n$.

Saída: Uma LCS $C = c_1c_2c_3 \ldots c_l$

Início

```
Thresh[0] := 0
para k := 1 até m faça                    // loop 1
        Thresh[k] := n+1
para i := 1 até m faça                    // loop 2
        j := CLOSEST[a_i,1]
        k := 1
        Enquanto j ≠ n + 1 faça           // loop 3
                se j < Thresh[k] então
                        temp:= Thresh[k]
                        Thresh[k] := j
                        Link[k] := new node (i, j, Link[k-1])
                        j:= CLOSEST[a_i, temp + 1]
                se j = Thresh[k] então
                        j := CLOSEST[a_i, j + 1]
                k := k + 1
k := m
Enquanto Thresh[k] = n+1 faça             // loop 4
        k := k-1
PTR := Link[k]
Enquanto PTR ≠ nil faça                   // loop 5
        c_k := símbolo do match apontado por PTR
        k := k-1
        avança PTR
```

Fim

Para cada iteração do *loop* 2 iremos analisar o j no máximo l vezes, ou seja o *loop* 3 terá no máximo lm iterações. O *loop* 1 terá no máximo m iterações, o *loop* 4 m-l iterações e

o loop 5 l iterações. Logo o algoritmo executa com tempo $O(lm)$. Acrescentando-se o pré-processamento para calcular *CLOSEST* teremos um tempo total de $\boldsymbol{O(sn + lm)}$. O espaço é de $\boldsymbol{O(sn + d)}$, onde d é o número de *matches* dominantes. Isto ocorre pois cada *match* dominante é armazenado e $O(sn)$ é o espaço ocupado pela estrutura *CLOSEST*.

Uma melhoria para $O(n)$ no espaço ocupado pela estrutura *CLOSEST* pode ser conseguida, ao custo de um tempo de acesso à estrutura de log s.

1.5 - Comparação entre os métodos

Podemos verificar que a ordem do tempo de execução dos três métodos vistos anteriormente, Programação Dinâmica (PD), Hunt/Szymanski (HS) e Apostolico/Guerra (AG), podem ser similares, dependendo das condições dos dados de entrada e de saída, ou seja, no pior caso nenhum deles tem tempo de execução menor que $O(mn)$.

No método da PD sempre teremos a ordem de execução dependendo dos comprimentos dos dados de entrada: $O(mn)$.

O método do HS, além de m, depende do número de *matches* da entrada e do comprimento da LCS, $O(m + r \log l)$, ou seja, se tivermos os dados de entrada com poucos *matches* e a LCS de comprimento pequeno, podemos ter uma ordem de tempo de execução melhor que a PD. Mas no caso em que o número de *matches* é $O(mn)$, e o comprimento da LCS é $O(m)$, temos um método com ordem de tempo de execução pior que o método da PD, ou seja, $O(mn \log m)$.

O método do AG, além de m e n, depende do comprimento da LCS e do tamanho do alfabeto, $O(sn + lm)$, ou seja, nos casos em que o alfabeto é pequeno e o comprimento da LCS é pequeno, este método pode ter boas vantagens. Tanto no tempo de execução como no espaço ocupado. Mas no caso em que l é $O(n)$ e o alfabeto é $O(m)$, o algoritmo tem tempo de execução igual a $O(mn)$.

2. Árvore de sufixos

2.1 - Introdução

Árvore de sufixos é uma estrutura de dados que armazena todos os sufixos de uma determinada seqüência. O seu valor vem da possibilidade de resolver em tempo linear problemas complexos em seqüências.

Um dos problemas clássicos resolvidos em tempo linear pela árvore de sufixos é o *problema do substring*. Neste problema é dado um texto T de comprimento m e um texto P (chamado padrão de busca) de comprimento n. Deve-se procurar uma ocorrência de P em T. Se o texto P ocorrer em T deve-se retornar a localização de P em T, e senão estiver contido, deve-se retornar que não está contido. Com um tempo de pré-processamento de $O(m)$ pode-se criar uma árvore de sufixos associada a T. De posse da árvore de sufixos é

possível em tempo *O(n)* saber se *P* é prefixo de algum sufixo de *T*, o que resolve o problema em tempo *O(m + n)*.

Muitas vezes deseja-se procurar vários padrões *P* em *T*. Após criada a árvore de sufixos o tempo de execução para qualquer busca somente depende do comprimento do texto que se quer pesquisar. Isto não é conseguido por outros algoritmos que não utilizam árvore de sufixos, os quais a cada busca necessitam de um tempo de acesso de *O(m)*, ou seja dependem do comprimento de *T*, que pode ser bem maior que o comprimento de *P*.

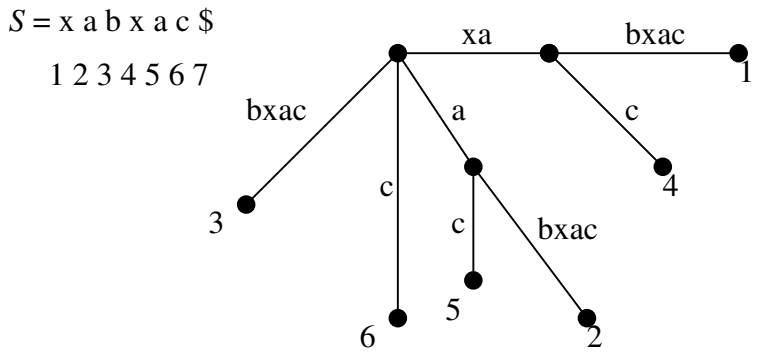
2.2 - Definições

Uma **árvore de sufixos** *T*, de um *string S* de comprimento *m*, é uma árvore dirigida com raiz e exatamente *m* folhas numeradas de 1 a *m*. Cada nó interno, exceto a raiz, tem no mínimo dois filhos e cada aresta é rotulada com um *string* não vazio de *S*. Se duas arestas saem do mesmo nó, então elas não podem ter rótulos que comecem pelo mesmo caracter. A característica chave de uma árvore de sufixos é que para uma folha *i*, a concatenação de todos os rótulos das arestas de um caminho da raiz até *i*, é igual ao sufixo de *S* que começa na posição *i*, ou seja é igual a *S[i..m]*.

Uma observação sobre a definição de árvore de sufixos é que esta definição não garante que dado um *string S*, uma árvore de sufixos de *S* existe. Não existe uma árvore de sufixos de *S*, no caso de um sufixo aparecer em um prefixo de qualquer outro sufixo de *S*. Por esta razão para construir uma árvore de sufixos de um *string S*, iremos considerar o *string S$*, sendo *$* a representação de um símbolo que não pertence ao alfabeto de *S*. Desta forma, um sufixo não aparecerá em um prefixo de qualquer outro sufixo de *S.*

O ***path-label*** de um nó é a concatenação, em ordem, de todos os rótulos das arestas do caminho da raiz até o nó.

Exemplo de uma árvore de sufixos:



2.3 - Método ingênuo para construir uma árvore de sufixos

Um método simples para construir um árvore de sufixos da seqüência S é construir a árvore, colocando cada um dos sufixos, partindo do sufixo *S[1..m]$* e então sucessivamente inserindo o sufixo *S[i .. m]$* para *i* de 2 até *m*. Denotaremos a árvore de sufixos N_i como a árvore intermediária construída até a inserção do sufixo *S[i .. m]$*.

A árvore N_1 tem somente uma aresta que vai da raiz até a folha 1. Esta aresta tem rótulo *S[1..m]$*. A árvore N_{i+1} é construída a partir da árvore N_i da seguinte forma: seja *P* o prefixo mais comprido de *S[i + 1 .. m]$* que é também prefixo de algum sufixo armazenado em N_i. Seja *S[k .. m]$* tal sufixo e seja *k* a folha correspondente a tal sufixo. Seja $\alpha_1\alpha_2.\alpha_l$ o caminho da raiz até *k*. Seja $p \leq l$ máximo, tal que a concatenação dos rótulos de $\alpha_1\alpha_2.\alpha_p$ tenha comprimento *c* menor ou igual ao comprimento *d* de *P*. Se $c < d$ então tome a aresta $\alpha_{p+1} = (u, v)$. Seja *r[1 .. q]* o rótulo de α_{p+1}. Crie o nó *w* e quebre a aresta α_{p+1} nas arestas *(u, w)* e *(w, v)* com rótulos *r[1 .. d - c]* e *r[d - c + 1 .. q]* respectivamente. Crie a folha *i + 1* e a aresta *(w, i + 1)* com rótulo *S[d + 1 .. m]$*. Se $c = d$ então tome a aresta $\alpha_p = (u, w)$ e crie a folha *i + 1* e a aresta *(w, i + 1)* com rótulo *S[d + 1 .. m]$*.

Um algoritmo que utilize diretamente este método terá um tempo de execução de $O(m^2)$. Existem algoritmos mais complexos, que constróem a árvore de sufixos com tempo de execução linear no tamanho da seqüência, ou seja, *O(m)*.

3. O sistema MUMmer

3.1 - Introdução

Desde que o primeiro genoma foi sequenciado, a cada ano o número de genomas sequenciados tem aumentado rapidamente. É muito comum após concluído o sequenciamento de uma espécie, sequenciar outra espécie bem próxima. Quando torna-se disponível o sequenciamento de dois organismos relativamente próximos, uma das primeiras questões que surge é como as duas seqüências estão alinhadas.

Existem inúmeros métodos para o alinhamento de duas seqüências, mas a maior parte deles tem o objetivo de alinhar apenas dois fragmentos (pedaços) das seqüências, usualmente contendo dois genes, ou alinhar duas proteínas. Para estes casos estes métodos trabalham muito bem, mas para o alinhamento de genomas completos, normalmente, estes métodos não são eficientes. Isto se deve unicamente ao fato do tamanho dos genomas completos serem muito maiores que os de genes, por exemplo. Este tamanho muito grande leva, normalmente, estes métodos a executarem em tempo inaceitável ou ultrapassarem os limites de memória. Além disto, normalmente, estes métodos levam em consideração somente inserções, exclusões e mutações, mas em genomas completos deve-se levar em consideração também outros fatos, como *tandem repeats* e grandes reversões.

Os algoritmos tradicionais e mais comuns de alinhamento utilizam os métodos de programação dinâmica ou técnicas de *hashing*. Existem inúmeros algoritmos e não é o objetivo deste texto descrevê-los. Um algoritmo ingênuo para a programação dinâmica executa em tempo e espaço $O(n^2)$, onde n é o comprimento da menor seqüência a ser alinhada. Penalizando um pouco mais o tempo pode-se conseguir que o espaço utilizado seja de $O(n)$, o que resolveria o problema de memória, mas ainda teria o problema do tempo. Nos métodos de *hashing* também teremos o tempo de execução em $O(n^2)$. Isto se torna ruim quando estamos analisando milhões ou centenas de milhões de bases, como nos genomas completos.

3.2 - Algoritmo do MUMmer

O método a ser descrito (do sistema conhecido como MUMmer) tem o objetivo de alinhar grandes seqüências de DNA, até mesmo do tamanho de um genoma completo, utilizando um tempo e espaço razoáveis e aceitáveis. Este método irá considerar alguns casos como os SNPs (*single nucleotide polymorphisms*), grandes inserções, repetições significativas, *tandem repeats* e reversões. O termo SNP é usado quando um segmento da seqüência ocorre nos dois genomas com um único nucleotídeo diferente.

O algoritmo do MUMmer utiliza uma estrutura de dados conhecida como árvore de sufixos. Esta estrutura permite encontrar de maneira eficiente todas os segmentos distintos de uma dada seqüência. Para a tarefa de alinhar duas seqüências de DNA, a árvore de sufixos permite que seja encontrada de forma eficiente, todos os segmentos comuns às duas seqüências sendo analisadas. A partir daí, um alinhamento é feito baseado nestes segmentos comuns.

O sistema do MUMmer usa uma combinação de três idéias:

1) Árvore de sufixos,
2) *Longest Increasing Subsequence* (LIS) e
3) Alinhamento de Smith-Waterman

Estas três idéias serão detalhadas mais a frente.

As entradas do sistema serão seqüências de DNA, que denotaremos por genoma *A* e genoma *B* (existe uma versão do sistema que trata de seqüências de proteínas).

Os principais elementos biológicos que são identificados pelo sistema são:

a) SNPs (*single nucleotide polymorphisms*), definidos como um segmento comum às duas seqüências *A* e *B,* a menos de uma única mutação interna ao segmento
b) Regiões do DNA onde as seqüências sofreram mutações em mais de um ponto.
c) Grandes regiões de DNA que foram inseridas em um dos genomas.
d) Repetições que ocorreram em um dos genoma de entrada, mas não no outro. Estas regiões repetidas podem ocorrer em locais bem espalhados.

e) *Tandem repeats*, regiões de contiguamente repetidos segmentos de DNA, mas com diferente número de cópias nos dois genomas. Os números de cópias, não necessariamente devem ser inteiros.

O processo de alinhamento no MUMmer pode ser dividido em 4 passos:

i) Executa-se uma decomposição dos dois genomas em MUMs (*maximal unique match*). Um MUM é um segmento maximal que ocorre exatamente uma vez em *A* e exatamente uma vez em *B*. O princípio crucial que está por trás deste passo é o seguinte: se um segmento razoavelmente comprido ocorre exatamente uma única vez em cada genoma, então é bem provável que a segmento ocorrido em *A* e o segmento ocorrido em *B*, devem estar emparelhadas no alinhamento final dos dois genomas. Se *A* e *B* são genomas próximos, então deveremos ter uma grande ocorrência de MUMs e podemos construir o alinhamento final a partir do alinhamento dos MUMs.

ii) Ordena-se os MUMs de acordo com os seus emparelhamentos, afim de obter a maior seqüência de MUMs que se emparelham na mesma ordem. Isto é feito utilizando uma variação do conhecido algoritmo para encontrar o LIS.

iii) Completa-se o que falta dos alinhamentos. Este passo identifica: grandes inserções , repetições, pequenas regiões que se modificaram, *tandem repeats* e SNPs.

iv) Mostra-se o alinhamento, incluindo o alinhamentos dos MUMs e os detalhes das regiões que não emparelham exatamente.

A seguir serão descritos com mais detalhes os passos de i) a iii).

3.2.1 - Identificação dos MUMs

Como descrito acima a identificação dos MUMs é um passo chave no processo de alinhamento. O problema de encontrar um conjunto de MUMs em seqüências de tamanhos grandes, não é algo trivial computacionalmente.

Para resolver este problema o MUMmer utiliza a estrutura de árvore de sufixos, cuja construção é feita baseada no algoritmo de McCreight. A implementação utilizada usa 12 bytes por nó do tipo folha, 24 bytes por nó interno e 1 byte para cada base no genoma. Um limite superior para o espaço utilizado neste passo do processo é de 37 bytes por base das seqüências de entrada. Para duas seqüências de 100M bases, este passo utilizaria menos de 8 Gigabytes. Verifica-se, na prática, que o espaço utilizado fica longe deste limite. Também o tempo de execução é linear.

A árvore de sufixos é construída com os sufixos de *A* e depois com os sufixos de *B*, de tal modo que cada folha é rotulada para identificar de qual genoma é a folha.

Com a árvore de sufixos construída é necessário identificar os MUMs. Para isto basta varrer toda a árvore de sufixos buscando os nós internos com duas folhas diretas como filhas e cada uma das folhas de um genoma. O *path-label* deste nó interno é um segmento candidato a MUM. Para que este segmento seja um MUM, é necessário que os símbolos imediatamente vizinhos deste segmento, em cada um dos genomas, sejam

distintos. Como o número de nós na árvore de sufixos é $O(n)$, o tempo necessário para executar esta busca é $O(n)$.

Normalmente, nem todos os MUMs encontrados nesta busca devem ser considerados no processo de alinhamento. Os MUMs muito pequenos encontrados devem ser ignorados. Para isto, o sistema MUMmer tem um parâmetro de entrada, com o comprimento mínimo dos MUMs a serem considerados para o alinhamento.

3.2.2 - Ordenando os MUMs

Após encontrar os MUMs mais significativos, eles são colocados em uma seqüência em ordem crescente de acordo com a posição no genoma *A*. Esta seqüência pode não estar em ordem crescente de acordo com as posições dos MUMs no genoma *B*, pois podem ter ocorrido reversões, transposições ou emparelhamentos "casuais" entre as duas seqüências. Desta forma, precisamos encontrar a maior subseqüência crescente das posições em *B*. Por exemplo, se a seqüência das posições em *B* é (1, 2, 10, 4, 5, 8, 6, 7, 9, 3) então a maior subseqüência em ordem crescente desta seqüência é (1, 2, 4, 5, 6, 7, 9). O MUMmer usa uma variante de um algoritmo que encontra uma LIS. Nesta variante, ele considera os comprimentos dos MUMs e o fato de poder haver sobreposição nos MUMs. A partir desta LIS, o MUMmer faz o alinhamento dos MUMs.

Um algoritmo simples do LIS através da programação dinâmica pode ser executado em tempo $O(k^2)$, onde k é número de MUMs. Este algoritmo pode ser aperfeiçoado para executar em tempo $O(k \log k)$. Como k, normalmente, é muito menor que $N/\log N$, então este passo no processo do alinhamento irá executar em tempo $O(N)$.

3.2.3 - Fechando os *gaps*

Após os MUMs estarem alinhados, é necessário alinhar o restante dos genomas.

Os *gaps*, ou interrupções no alinhamento dos MUMs, serão neste passo resolvidos, ou seja, serão alinhados. Os *gaps* são divididos em quatro classes:

i) SNP

ii) Inserção

iii) Região altamente polimórfica

iv) Repetição

A seguir iremos analisar cada uma destas classes:

i) SNP

A identificação de SNPs está se tornando uma tarefa importante na análise de seqüências de DNA, principalmente com o sequenciamento de espécies próximas.

Um caso simples de SNP é quando o símbolo que diverge nas duas seqüências está circundado por MUMs. Neste caso o *gap* é de apenas um símbolo, o que facilmente identifica um SNP.

O SNP pode aparecer algumas vezes entre segmentos não únicos, ou seja, pelo menos um dos segmentos aparece em outro lugar no genoma. Este caso específico é tratado no caso iv), onde são tratadas as repetições.

ii) Inserção

Gaps de inserção podem ser por transposição ou *gaps* simples.

Gaps de inserção por transposição ocorrem quando um segmento é removido de um lugar do genoma e inserido em outro lugar. Neste caso, o segmento poderia aparecer em um MUM, mas provavelmente não seria alinhado no passo de ordenação dos MUMs.

No caso de *gaps* de inserção simples um segmento aparece em um dos genomas e não aparece no outro. Neste caso este segmento não aparece no MUMs.

iii) Região altamente polimórfica

São regiões que tem um grande número de diferenças entre os dois genomas. Se estas regiões forem relativamente pequenas, um algoritmo do Smith-Waterman é utilizado para resolver o alinhamento destas duas regiões, senão o MUMmer é aplicado recursivamente, mas agora com um parâmetro menor para a identificação dos MUMs relevantes (passo 1 do processo).

iv) Análise de MUMs que se sobrepõe

Após a ordenação dos MUMs, dois MUMs podem se sobrepor em *A* e *B*. Este tipo de ocorrência indica a presença de *tandem repeats* nas áreas sobrepostas.

Seja *a* o número de símbolos que são sobrepostos pelos dois MUMs em *A,* e *b* o número de símbolos que são sobrepostos pelos dois MUMs em *B*. Pela definição de MUM, *a* e *b* são diferentes. Para $a < b$, o número de *tandem repeats* em *B* é menor que em *A*. Sendo assim, no alinhamento de *A* e *B* deve-se inserir $b - a$ espaços em *B*. Para $a > b$, deve-se inserir $a - b$ espaços em *A*. Para manter a consistência, o MUMmer sempre mostra os espaços no final dos *tandem repeats*.

3.3 - Resultados obtidos

3.3.1 Comparando duas espécies de *tuberculosis* - H37Rv e CDC1551

Estas espécies tem mais de 99% do genoma idênticos (exceto pelos grandes segmentos de repetições que possuem número diferente de cópias). O tamanho dos genomas é em torno de 4,4 milhões de bases.

Entender as diferenças dos genomas é muito importante para entender as grandes diferenças biológicas entre as espécies.

Executando o MUMmer possibilitou um alinhamento de cada base dos genomas. Este alinhamento revelou milhares de diferenças individuais entre os genomas, que pode ser visualizada na ferramenta de visualização do alinhamento. O que se pode ver é que a maior parte (75%) das diferenças entre genes ocorreram em genes hipotéticos, o que leva a crer que muitos dos genes hipotéticos não são reais.

O sistema levou 55 segundos para executar este alinhamento: 5 segundos para a construção da árvore de sufixos, 45 segundos para ordenar os MUMs e 5 segundos para alinhar os *gaps* entre os MUMs (algoritmo do Smith-Waterman). Esta execução foi em um DEC Alpha 4100.

3.3.2 - Comparando dois genomas de *Mycoplasma*

Na análise anterior as duas espécies eram bem próximas, mas para testar o sistema foi feito um alinhamento entre duas espécies "primas": a *M.genitalium* e a *M.pneumoniae*, cujos genomas tem 580.074 bases e 816.394 bases respectivamente. Apesar desta diferença de tamanho dos genomas, sabe-se que a maior parte das proteínas da *M.genitalium* também aparecem na *M.pneumoniae*. O que possibilitou o alinhamento através do MUMmer.

Este alinhamento no MUMmer foi feito de maneira fácil e rápida, o que foi surpreendente, pela dificuldade encontrada nos outros métodos. Executando em um DEC Alpha 4100 o MUMmer levou pouco mais de dois minutos: 6,5 segundos para a construção da árvore de sufixos, 0,02 segundos para a ordenação dos MUMs e 116 segundos para o alinhamento dos *gaps*. Este alinhamento foi muito mais demorado do que no caso anterior, devido ao fato dos *gaps* serem segmentos de comprimentos muito grandes.

Este alinhamento foi comparado com dois outros alinhamentos. Foi gerado um alinhamento com o FASTA, dividindo o genoma em segmentos de 1000 bases, e comparando todos os segmentos contra todos os segmentos. Isto levou horas. Um gráfico das regiões que eram de 50% a 80% idênticas foi gerado, e percebeu-se um grande ruído neste gráfico, o que dificultou a análise. Foi também aplicado um método que colocava em um gráfico as regiões que tinham emparelhamentos únicos de 25 bases. Isto é uma maneira parecida com o que o MUMmer faz, mas usa um tamanho de MUM fixo, o que fez gerar um gráfico, apesar de menos ruidoso que o do FASTA, mais grosseiro do que o gerado pelo MUMmer. O MUMmer gerou um gráfico dos MUMs do alinhamento e possibilitou uma análise mais clara e fácil deste alinhamento.

3.3.3 - Comparando uma seqüência do cromossomo humano e do rato

Para testar o MUMmer em seqüências mais distantes foi escolhido um segmento de 222.930 pares de bases do cromossomo humano e um fragmento de 227.538 pares de bases de um cromossomo do rato.

Um estudo realizado anteriormente alinhou estas seqüências e mostrou a similaridade entre elas. Neste estudo foram utilizadas inúmeras ferramentas para se chegar ao alinhamento final.

O MUMmer necessitou de apenas 29 segundos de tempo de CPU, sendo que a construção da árvore de sufixos precisou de apenas 1,6 segundos. O MUMmer usou para o comprimento mínimo dos MUMs o valor 15 e obteve vários *gaps* que correspondem as regiões intergênicas. O MUMmer gerou um resultado não tão detalhado como pelo estudo realizado, mas pode-se visualizar claramente que o alinhamento é similar ao gerado pelo estudo.

4. *Conclusão*

Vimos algumas técnicas utilizadas para a comparação de seqüências de DNA.

Primeiro, apresentamos alguns dos algoritmos usados para a resolução do problema da obtenção de uma subseqüência comum mais comprida (LCS), que pode ser visto como uma simplificação do que normalmente é feito para a obtenção de alinhamentos entre seqüências de DNA. Os algoritmos vistos têm comportamento ao menos quadrático no pior caso. Tais algoritmos são demasiadamente ineficientes face aos comprimentos de seqüências de genomas completos.

A seguir, apresentamos um conceito bastante utilizado, a árvore de sufixos, e mostramos como tal conceito é central dentro de um sistema como o MUMmer, que também foi apresentado. O MUMer foi especificamente desenvolvido para a obtenção de alinhamentos entre seqüências genômicas.

5. *Referência*

A. L. Detcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, S. L. Salzberg: Alignment of whole genomes, Nucleic Acids Research, 1999, Vol. 27, No. 11, 2369-2376.

A. Apostolico, C. Guerra: The Longest Commom Subsequence Problem Revisited, Algorithmica, 1987, 315-336.

Claus Rick : New Algorithm for the Longest Common Subsequence Problem, Institute Für Informatik der Universitãt Bonn, Report no. 85123-CS, Outubro 1994.

J. W. Hunt, T. G. Szymanski. A fast algorithm for computing longest commom subsequences. *Comm. ACM*, 20(5):350-353, 1977.

D. Gusfield, Algorithms on strings, trees and sequences: Computer Science and Computacional Biology, 1997, Cambridge University Press, New York.