

Alinhamentos de Múltiplas Sequências

Rogério Theodoro de Brito
Orientador: José Augusto Ramos Soares.

26 de setembro de 2001

Sumário

1	Introdução	2
2	Definições Básicas e Notação	3
2.1	Alfabetos, Símbolos e Seqüências	3
2.2	Alinhamentos	4
3	O Problema de Otimização	5
4	Alinhamentos de 2 Seqüências	7
4.1	Solução via Programação Dinâmica	9
4.1.1	Método de Programação Dinâmica para $k = 2$	10
4.1.2	Calculando um Alinhamento via Matriz de Programação Dinâmica	11
4.1.3	Complexidade do Método	12
5	Alinhamentos Múltiplos Exatos	14
5.1	Programação dinâmica para k seqüências	14
5.2	O Método de Carrillo-Lipman	14
6	O Problema k-ALIN é NP-difícil	15
7	Usando Menos Recursos para Calcular Alinhamentos	16
7.1	Heurísticas	16
7.1.1	Divisão e Conquista	16
7.2	Algoritmos de Aproximação	16
7.2.1	Algoritmo de Alinhamento Estrela	17
7.2.2	Algoritmo de l -estrelas	17
8	Alinhamentos de Múltiplas Seqüências na Prática	18
8.1	ClustalW	18
8.2	MSA	19
9	Apêndice	20
9.1	Algoritmo de Ukkonen para o Cálculo de Distância de Edição	20
9.2	Alinhamentos Ótimos de 3 seqüências	20
10	Cronograma de Trabalho	21

Capítulo 1

Introdução

Um problema de importância em Biologia Molecular é realizar um estudo comparativo de um conjunto de seqüências de bases nitrogenadas ou de aminoácidos. Uma maneira de efetuar a comparação destas seqüências é calcular um alinhamento entre elas. Intuitivamente falando, um alinhamento é uma maneira de inserir espaços nas seqüências de forma que elas fiquem com o mesmo comprimento e, possam, desta maneira, ser facilmente comparadas.

Além da aplicação para análises de padrões comuns nas seqüências alinhadas, o problema de encontrar alinhamentos de múltiplas seqüências ocupa uma posição de destaque em Biologia Molecular Computacional por causa de sua relação com vários outros problemas: alinhamentos múltiplos são úteis para mapeamento/reconstrução de genomas a partir de fragmentos de DNA, para construção de árvores filogenéticas e para predição de estrutura de proteínas (por exemplo, predição de estrutura secundária).

A despeito de o caso particular de um alinhamento ótimo de duas seqüências poder ser encontrado por meio de programação dinâmica em tempo polinomial, a extensão mais natural desta estratégia não parece fornecer algoritmos práticos ou viáveis para o problema de encontrar um alinhamento ótimo para um número arbitrário de seqüências. Na realidade, sabe-se que o problema de encontrar alinhamentos múltiplos ótimos é um problema NP-difícil [18, 2, 9].

Por causa da importância de alinhamentos múltiplos, da dificuldade mencionada acima em obter alinhamentos múltiplos ótimos, por algumas aplicações não serem muito sensíveis aos alinhamentos utilizados (i.e., alinhamentos ótimos não serem absolutamente necessários) e mesmo pelos alinhamentos ótimos nem sempre serem os alinhamentos mais relevantes do ponto de vista biológico¹, diversos algoritmos de aproximação [7, 10, 1] e heurísticas [13] foram propostos para o problema.

Neste texto, procuraremos expor as principais características conhecidas sobre alinhamentos de múltiplas seqüências e os principais métodos utilizados para obter soluções para o problema.

Grande parte deste texto foi elaborada com uso extensivo das referências [12] e [5].

¹Um motivo para nos preocuparmos com alinhamentos ótimos, bem como com alinhamentos que *não* sejam ótimos reside no fato de que alinhamentos que minimizam a distância de edição (uma função objetivo normalmente adotada para o problema) entre as seqüências alinhadas não necessariamente correspondem a alinhamentos “corretos” biologicamente falando. Isto é uma consequência de os modelos matemáticos propostos até o momento não capturarem o problema biológico de maneira exata, embora reflitam sua essência e sejam, por isto, empregados. Evidentemente, o problema de otimização combinatória que surge a partir destes modelos é por si só de interesse para a Ciência da Computação e possui aplicações — um exemplo disto é o utilitário `diff` do ambiente Unix.

Capítulo 2

Definições Básicas e Notação

Nesta seção, daremos algumas definições a fim de estabelecer a linguagem utilizada e de fixar a notação. Outras definições serão dadas no momento em que forem necessárias.

2.1 Alfabetos, Símbolos e Seqüências

Um *alfabeto* Σ é um conjunto finito e não vazio. Um elemento $\sigma \in \Sigma$ é chamado *caractere* ou *símbolo*. Uma *seqüência* s sobre o alfabeto Σ é uma seqüência $s = (\sigma_1, \sigma_2, \dots, \sigma_n) \in \Sigma^n$, onde $n \geq 0$ é um inteiro. Por conveniência e simplicidade, denotaremos s por $\sigma_1\sigma_2 \cdots \sigma_n$ apenas. Se $s = \sigma_1\sigma_2 \cdots \sigma_n$, definimos o *comprimento* $|s|$ de s por $|s| = n$. Se $n = 0$ (e, portanto, $|s| = 0$), então s é chamada *seqüência vazia* e é denotada por ε .

Um *segmento* $s[i..j]$ de uma seqüência s é a seqüência definida por $s[i..j] = \sigma_i\sigma_{i+1} \cdots \sigma_j$, para $i \leq j$. Se $i > j$, definimos $s[i..j] = \varepsilon$. Por comodidade, denotaremos o caso particular em que $j = i$ por $s[i]$ simplesmente, isto é, $s[i] = s[i..i]$. Neste caso, claramente $s[i] = \sigma_i$. Frequentemente usaremos esta notação para fazer referência aos símbolos de s em descrições de algoritmos.

Uma *subseqüência* s' de s é uma seqüência $s' = \sigma_{i_1}\sigma_{i_2} \cdots \sigma_{i_m}$ em que $\{i_1, i_2, \dots, i_m\} \subseteq \{1, 2, \dots, n\}$ e $i_1 < i_2 < \cdots < i_m$. Neste caso, também dizemos que s é uma *superseqüência* de s' . É importante observar que, enquanto um segmento de s é uma série de caracteres consecutivos de s , uma subseqüência pode ser constituída de caracteres que não necessariamente estejam consecutivos em s .

Se Σ é um alfabeto tal que $\sqcup \notin \Sigma$, denotamos por Σ' o alfabeto $\Sigma' = \Sigma \cup \{\sqcup\}$. O símbolo \sqcup é chamado *espaço em branco*, *branco* ou, simplesmente, *espaço*.

Em nosso texto, estaremos frequentemente supondo que $\Sigma = \{\mathbf{A}, \mathbf{C}, \mathbf{G}, \mathbf{T}\}$ é o alfabeto correspondente a bases nitrogenadas (ou simplesmente bases) presentes em seqüências de DNA e que seqüências sobre Σ são fragmentos (ou trechos) de DNA de alguma espécie sendo analisada ou que Σ é o alfabeto correspondente ao conjunto de aminoácidos e que uma seqüência sobre Σ é uma proteína (ou fragmento de uma proteína).

Embora estes dois alfabetos sejam usados com grande freqüência, eles não são os únicos alfabetos usados em Biologia Molecular. Um terceiro alfabeto, chamado *alfabeto genético estendido IUPAC*, é uma extensão do alfabeto de 4 símbolos usado para representar bases de DNA. Sua utilidade é expressar o fato de uma determinada posição de uma seqüência ser ocupada por alguma base que não é completamente conhecida. Esta necessidade surge durante o seqüenciamento de bases, processo em que pode haver ambigüidade de leitura de uma base que ocupa uma determinada posição. O alfabeto IUPAC possui um caractere

correspondente a cada possibilidade de leitura para a base: ele consiste de um símbolo para cada subconjunto não vazio de $\{A, C, G, T\}$ (e, portanto, possui $15 = 2^4 - 1$ símbolos).

Em todo caso, os algoritmos que estudaremos são genéricos e independem dos alfabetos utilizados.

2.2 Alinhamentos

Sejam k um inteiro positivo e s_1, s_2, \dots, s_k seqüências sobre um alfabeto Σ , com $|s_i| = n_i$, $i = 1, \dots, k$. Um *alinhamento* A de s_1, s_2, \dots, s_k é uma matriz $A = (a_{ij})$ de dimensões $k \times n$ com entradas de $\Sigma' = \Sigma \cup \{\sqcup\}$ (e $n \geq \max_{i=1}^k \{n_i\}$) tal que a linha a_i do alinhamento A consiste exatamente da seqüência s_i com possíveis espaços em branco (e apenas espaços em branco) inseridos entre os caracteres de s_i .

Podemos deixar a definição acima mais precisa reescrevendo-a como: um alinhamento A de s_1, s_2, \dots, s_k é uma matriz $A = (a_{ij})$ de dimensões $k \times n$ tal que, para cada i , existe um conjunto $J_i = \{j_1, j_2, \dots, j_{n_i}\} \subseteq \{1, 2, \dots, n\}$, com $j_1 < j_2 < \dots < j_{n_i}$ e tal que $a_{ij_1} a_{ij_2} \dots a_{ij_{n_i}} = s_i$ e tal que para todo $j \in \{1, 2, \dots, n\} - J_i$, temos $a_{ij} = \sqcup$.

Dizemos que dois caracteres $s_i[j]$ e $s_{i'}[j']$ estão *alinhados* em A se $s_i[j]$ e $s_{i'}[j']$ estão na mesma coluna de A .

Seja $A = (a_{ij})$ um alinhamento de s_1, s_2, \dots, s_k . Uma *lacuna* de s_i no alinhamento A é qualquer segmento não vazio maximal de a_i composto apenas de espaços em branco (\sqcup), isto é, uma lacuna de s_i em A é um segmento $a_i[j..j']$ de a_i tal que:

- $j \leq j'$;
- todos os símbolos de $a_i[j..j']$ são \sqcup ;
- não é possível estender $a_i[j..j']$ em a_i por meio de espaços em branco, ou seja, $j = 1$ ou $a_i[j - 1] \neq \sqcup$ e $j' = n$ ou $a_i[j' + 1] \neq \sqcup$.

O termo em inglês encontrado mais comumente na literatura para referir-se ao que definimos como lacunas é *gap*.

Sejam s_1, s_2, \dots, s_k seqüências sobre um alfabeto Σ . Denotaremos por $\mathcal{A}_{s_1, s_2, \dots, s_k}$ o conjunto de alinhamentos entre as seqüências s_1, s_2, \dots, s_k .

Capítulo 3

O Problema de Otimização

Em Biologia Molecular, observa-se empiricamente que há uma forte relação entre a forma de uma determinada molécula e a função que a molécula desempenha. Por exemplo, moléculas de proteína que possuem estruturas secundárias semelhantes geralmente possuem funções parecidas. Ademais, tanto a forma quanto a função de moléculas de DNA e de proteínas são determinadas, em grande parte, por sua seqüência de bases e de aminoácidos, quando observadas como cadeias unidimensionais apenas.

É, portanto, de grande interesse o estudo comparativo entre seqüências biológicas, principalmente entre seqüências de DNA e entre seqüências de aminoácidos.

Dado um conjunto de seqüências como, por exemplo, de proteínas que poderiam corresponder a diferentes etapas evolutivas de uma dada proteína, tal comparação teria interesse em verificar quais regiões das seqüências são semelhantes, quais regiões são diferentes e quais foram as possíveis mutações sofridas pelas seqüências durante seu processo evolutivo.

Uma maneira relativamente simples de realizar tais comparações é computar um alinhamento entre estas seqüências. Um alinhamento é construído por meio da inserção de espaços nas seqüências a serem comparadas de modo que elas fiquem todas com mesmo comprimento. Um alinhamento, é, intuitivamente falando, uma forma de dispor as seqüências a serem comparadas em uma tabela com uma seqüência por linha. Por exemplo, um alinhamento entre as seqüências TAGGTAC e TAGCTA é:

```
TAGGTAC_ _ _ _ _
_ _ _ _ _TAGCTA
```

Figura 3.1: Um alinhamento entre TAGGTAC e TAGCTA.

Uma breve inspeção das seqüências logo nos convence de que elas são “parecidas”, embora o alinhamento da Figura 3.1 não ressalte este fato. Um alinhamento “melhor” entre as seqüências poderia ser:

```
TAGGTAC
TAGCTA_
```

Figura 3.2: Outro alinhamento entre TAGGTAC e TAGCTA.

Este segundo alinhamento deixa mais que as seqüências têm muito em comum e que suas únicas diferenças são uma base G na primeira seqüência ocupando o lugar de uma base

C na segunda seqüência e a última base (um C) na primeira seqüência, que não está presente na segunda.

Naturalmente, estamos interessados em alinhamentos que exponham da melhor forma possível a estrutura das seqüências consideradas, evidenciando suas similaridades e diferenças e permitindo uma fácil comparação das seqüências dadas. Um tal alinhamento será chamado de alinhamento ótimo. Isto traduz nossa impressão de que o segundo alinhamento é “melhor” do que o primeiro.

De um modo geral, para comparar alinhamentos, atribuímos um “conceito” de qualidade ou “pontuação” para os alinhamentos. Baseando-nos neste critério, estaremos interessados em escolher o melhor alinhamento entre as seqüências dadas. É claro que o problema de encontrar alinhamentos entre seqüências, da forma como descrevemos acima é um problema de otimização.

Capítulo 4

Alinhamentos de 2 Seqüências

Por ora, estaremos focalizando nossa atenção apenas a alinhamentos entre duas seqüências s e t sobre um alfabeto Σ , com $|s| = m$ e $|t| = n$. Neste caso, todo alinhamento A entre s e t possui duas linhas e, para simplificar a notação, denotaremos a primeira linha do alinhamento A por s' (que é obtida de s por adição de espaços) e, a segunda, por t' (correspondente à seqüência t). O número de colunas de A será denotado por l .

Trataremos do caso geral de alinhar k seqüências em um capítulo posterior.

Para comparar alinhamentos, várias são as funções objetivo utilizadas, apesar de serem definidas de maneira semelhante: utilizamos uma função auxiliar $p : \Sigma' \times \Sigma' \rightarrow \mathbb{Z}$ que, a cada par (σ, ρ) de símbolos de $\Sigma' = \Sigma \cup \{\sqcup\}$, nos dá a pontuação $p(\sigma, \rho)$ de alinharmos os símbolos σ e ρ . Assumimos que $p(\sqcup, \sqcup) = 0$.

Usando a função p acima, dado um alinhamento A entre s e t , definimos a pontuação ou custo $c_p(A)$ do alinhamento A por:

$$c_p(A) = \sum_{j=1}^l p(s'[j], t'[j]).$$

Observe-se que a pontuação de um alinhamento A é definida como a soma das pontuações de suas colunas. Além disso, a função c_p pode ser interpretada como uma extensão da função p a alinhamentos de um número arbitrário de colunas se considerarmos que (σ, ρ) é um alinhamento de comprimento 1. Quando a função p estiver clara pelo contexto, escreveremos $c(A)$ para denotar a pontuação do alinhamento A . As diferentes formas de pontuar alinhamentos dependem, em essência, da função p utilizada.

Uma importante situação ocorre se p satisfaz os axiomas de métrica. Neste caso, se $\mathcal{A}_{s,t}$ é o conjunto dos alinhamentos entre s e t , dizemos que $d(s, t) = \min_{A \in \mathcal{A}_{s,t}} \{c(A)\}$ é a distância entre as seqüências s e t . É uma tarefa rotineira verificar que a função d definida acima satisfaz aos axiomas de métrica.

No caso particular em que p é a métrica zero-ou-um, isto é, quando $p(\sigma, \rho) = 0$, se $\sigma = \rho$, e $p(\sigma, \rho) = 1$, se $\sigma \neq \rho$, a distância $d(s, t)$ definida acima é chamada a distância de Levenshtein ou distância de edição entre s e t .

A distância de edição $d(s, t)$ pode ser interpretada como o número mínimo de operações de inserção, remoção ou troca de caracteres necessárias para transformar uma seqüência na outra. Por exemplo, para as seqüências $s = \text{TAGGTAC}$ e $t = \text{TAGCTA}$, a distância de edição $d(s, t)$ é 2 e um alinhamento entre s e t com esta pontuação determina quais são as operações necessárias para transformar s em t : o alinhamento da figura 3.2 nos diz que,

para transformar s em t , basta mantermos os caracteres TAG sem modificações, trocar o G seguinte por C, manter os caracteres TA e apagar o último caractere C de s .

De maneira análoga, este alinhamento também pode ser lido como uma maneira de transformar a seqüência t na seqüência s , mantendo-se os símbolos TAG, substituindo o caractere C por G, mantendo-se os caracteres TA e inserindo o caractere C ao final da seqüência¹.

Intuitivamente falando, no caso em que p é uma distância, $d(s, t)$ é uma “quantia pequena” se s e t são seqüências “parecidas” e é uma “quantia grande” quando s e t são seqüências “muito diferentes”.

Antes de prosseguirmos com nossa discussão, gostaríamos de comentar a respeito de um outro tipo de pontuação bastante utilizado na prática, chamado similaridade. Se a função p de pontuação entre caracteres de Σ' atribuir pontuação alta a caracteres “semelhantes” ou, digamos, que possuem boa probabilidade de serem trocados durante o processo evolutivo e atribuir pontuação baixa a pares de caracteres com baixa probabilidade de serem trocados durante o processo evolutivo, podemos estar interessados em alinhamentos entre s e t que tenham grande pontuação, isto é, que maximizem $c_p(A)$. Neste caso, definimos a similaridade $sim(s, t)$ entre s e t como $sim(s, t) = \max_{A \in \mathcal{A}_{s,t}} \{c_p(A)\}$.

Neste texto, estaremos tratando principalmente com distâncias em vez de similaridades, pois distâncias são matematicamente mais fáceis de serem tratadas. Entretanto, a maior parte dos algoritmos que apresentaremos pode ser minimamente alterada para trabalhar com similaridades em vez de distâncias.

O problema de otimização em que estaremos interessados quando p é uma distância é, portanto:

Problema 1 (Problema 2-ALIN) *Dadas duas seqüências s e t sobre um alfabeto Σ , encontrar um alinhamento A entre s e t de modo que o custo $c(A)$ seja mínimo.*

É claro que quando estamos lidando com distâncias como função objetivo, o problema de encontrar alinhamentos ótimos é obviamente limitado: todo alinhamento A possui custo $c(A) \geq 0$.

Pela forma como c é definida e pelo fato de que $p(\sqcup, \sqcup) = 0$, podemos restringir nossa atenção no Problema 2-ALIN a alinhamentos A que não possuem colunas compostas apenas por espaços alinhados. De fato, se A é um alinhamento ótimo que possui uma coluna, digamos l' , que seja composta apenas por caracteres \sqcup , então o alinhamento A' dado por

$$A' = \begin{pmatrix} s'[1] & \cdots & s'[l' - 1] & s'[l' + 1] & \cdots & s'[l] \\ t'[1] & \cdots & t'[l' - 1] & t'[l' + 1] & \cdots & t'[l] \end{pmatrix}$$

possui pontuação

$$\begin{aligned} c(A') &= \sum_{j \neq l'} p(s'[j], t'[j]) = 0 + \sum_{j \neq l'} p(s'[j], t'[j]) = p(s'[l'], t'[l']) + \sum_{j \neq l'} p(s'[j], t'[j]) \\ &= \sum_j p(s'[j], t'[j]) = c(A) \end{aligned}$$

e, como A é ótimo, A' também é ótimo. Naturalmente, o procedimento descrito acima para transformar A em A' de remover uma coluna com espaços em branco apenas pode ser

¹Notemos que dependendo do sentido da transformação, o caractere \sqcup pode ser interpretado como a remoção ou a inserção de um caractere. Por este motivo é comum na literatura a denominação *indel* para \sqcup , uma contração proveniente de *insert-delete*, do inglês.

repetido até que tenhamos um alinhamento que não possua colunas consistindo apenas de espaços.

Logo, se existe um alinhamento ótimo com colunas contendo espaços em branco apenas, existe um alinhamento ótimo que não contém colunas consistindo somente de espaços em branco. Por este motivo, estaremos trabalhando apenas com alinhamentos *sem* colunas compostas unicamente por espaços.

Um possível método para resolver o problema poderia ser gerar exaustivamente os alinhamentos A entre s e t e, dentre tais alinhamentos, escolher um que minimize a quantidade $c(A)$. Esta estratégia, embora simples, não é prática a menos que as seqüências s e t sejam muito pequenas, uma vez que o número de alinhamentos entre duas seqüências cresce rapidamente conforme o tamanho das seqüências aumenta.

Na realidade, se considerarmos apenas os alinhamentos entre s e t sem colunas compostas apenas de espaços, então o número de tais alinhamentos é $\sum_{i=0}^n \binom{2n-i}{n-i, n-i, i}$, onde $\binom{2n-i}{n-i, n-i, i} = \frac{(2n-i)!}{(n-i)!(n-i)!i!}$. Mas apenas o termo correspondente a $i = 0$ neste somatório é $\binom{2n}{n} \sim \frac{2^{2n}}{\sqrt{\pi n}} = \frac{4^n}{\sqrt{\pi n}}$. Desta forma,

$$\sum_{i=0}^n \binom{2n-i}{n-i, n-i, i} = \Omega(4^n/\sqrt{n}),$$

o que nos mostra que o número de alinhamentos entre duas seqüências é uma grandeza exponencial no tamanho das seqüências de entrada e que, ainda que consigamos examinar cada alinhamento em tempo $O(1)$, um algoritmo baseado em enumeração de alinhamentos é intrinsecamente lento.

É interessante notar que a função custo c definida acima é aditiva no seguinte sentido: se A é decomposto em dois outros alinhamentos, digamos A_E e A_D , de forma que $A = (A_E : A_D)$, então temos que $c(A) = c(A_E) + c(A_D)$. Esta propriedade irá nos permitir escrever um algoritmo simples e de tempo polinomial para obter uma solução para o problema do alinhamento de duas seqüências.

4.1 Solução via Programação Dinâmica

Como a função c possui a propriedade de ser aditiva, em particular, sabemos que se A é um alinhamento entre s e t decomposto como $A = (A' : A'')$, onde A'' é a última coluna de A , então $c(A) = c(A') + c(A'')$. Com isto em mente, podemos concluir que existem apenas três possibilidades para a última coluna A'' de A :

- o último símbolo de s está alinhado a um espaço em A'' , isto é, $A'' = \binom{s[m]}{\square}$;
- o último símbolo de s está alinhado ao último símbolo de t em A'' , isto é, $A'' = \binom{s[m]}{t[n]}$;
- o último símbolo de t está alinhado a um espaço em A'' , isto é, $A'' = \binom{\square}{t[n]}$.

Note-se que como estamos considerando apenas alinhamentos sem colunas contendo apenas espaços em branco, estas são todas as possibilidades para a última coluna de um alinhamento.

As observações feitas acima nos sugerem uma maneira recursiva de solução do problema 2-ALIN, descrita no algoritmo ALINHA-REC.

Infelizmente, o algoritmo ALINHA-REC executa um número exponencial de operações no tamanho das seqüências de entrada para calcular alinhamentos.

Algoritmo 1 ALINHA-REC(s, t)

- 1: **se** $m = 0$ **então**
 - 2: Devolva t alinhada a espaços;
 - 3: **se** $n = 0$ **então**
 - 4: Devolva s alinhada a espaços;
 - 5: $A_1 \leftarrow$ ALINHA-REC($s[1..m-1], t[1..n]$);
 - 6: $A_2 \leftarrow$ ALINHA-REC($s[1..m-1], t[1..n-1]$);
 - 7: $A_3 \leftarrow$ ALINHA-REC($s[1..m], t[1..n-1]$);
 - 8: Seja A um alinhamento de menor pontuação dentre $(A_1 : \begin{smallmatrix} s[m] \\ \sqcup \end{smallmatrix})$, $(A_2 : \begin{smallmatrix} s[m] \\ t[n] \end{smallmatrix})$, $(A_3 : \begin{smallmatrix} \sqcup \\ t[n] \end{smallmatrix})$
 - 9: Devolva A
-

Uma pergunta que surge naturalmente é “É possível, dadas duas seqüência s e t , encontrar um alinhamento entre elas de maneira mais rápida?” Procuraremos responder a esta pergunta a partir da próxima seção.

4.1.1 Método de Programação Dinâmica para $k = 2$

Não é complicado observar que o algoritmo 1 realiza implicitamente uma enumeração exaustiva de todos os possíveis alinhamentos entre s e t sem colunas compostas apenas por espaços.

Apesar de este algoritmo não ser eficiente, ele torna claras algumas propriedades do problema 2-ALIN:

- Se A é um alinhamento ótimo entre s e t e $A = (A' : A'')$, onde A'' é a última coluna de A , então o alinhamento A' também é um alinhamento ótimo entre os segmentos de s e de t alinhados por A' . Caso contrário, se existisse um outro alinhamento, digamos, A''' entre os segmentos de s e t alinhados por A' de modo que $c(A''') < c(A')$, então teríamos que $c(A''') + c(A'') < c(A') + c(A'') = c(A)$ e o alinhamento A não seria ótimo, uma vez que o alinhamento $(A''' : A'')$ teria pontuação menor do que a de A .
- Para encontrar um alinhamento ótimo entre as seqüências $s[1..m]$ e $t[1..n]$, nossa solução recursiva descrita acima resolve, por exemplo, o subproblema de encontrar um alinhamento entre $s[1..m-1]$ e $t[1..n-1]$ em mais de uma ocasião: uma das vezes em que este subproblema é resolvido é durante uma chamada recursiva de ALINHA-REC($s[1..m-1], t[1..n]$), outra ocorre na chamada explícita a ALINHA-REC($s[1..m-1], t[1..n-1]$) e, outra, em uma chamada recursiva de ALINHA-REC($s[1..m], t[1..n-1]$). De um modo geral, o algoritmo ALINHA-REC computa, durante a solução do problema original, a solução para um dado subproblema mais de uma vez.

A primeira das propriedades acima é, em conjunto com o fato de existirem apenas três possibilidades para a última coluna de um alinhamento, uma demonstração de que o algoritmo 1 está correto, isto é, de que dadas duas seqüências s e t , o algoritmo ALINHA-REC(s, t) de fato computa um alinhamento A entre s e t de pontuação mínima.

Problemas que, como o 2-ALIN, podem ser resolvidos recursivamente e que possuem as duas características citadas acima podem ser resolvidos pelo paradigma de programação dinâmica [4]. Sucintamente, uma solução por programação dinâmica consiste em ordenar de maneira conveniente os passos da solução recursiva a um problema, armazenando em

uma tabela as soluções para subproblemas, de modo que cada subproblema seja resolvido apenas uma vez.

A solução para encontrar um alinhamento ótimo entre duas seqüências s e t em relação a uma função de pontuação p foi proposta por Needleman e Wunsch em 1970 e adaptada de diversas maneiras desde então. Uma versão bastante conhecida, que permite realizar alinhamentos locais (i.e., entre segmentos) das seqüências, é a de Smith-Waterman [19].

O algoritmo de programação dinâmica para encontrar um alinhamento ótimo entre s e t pode ser observado como o resultado de duas etapas: na primeira, computa-se a distância entre s e t e, na segunda, o alinhamento propriamente dito entre as seqüências é calculado.

A solução de programação dinâmica para o problema de encontrar um alinhamento ótimo entre s e t preenche uma tabela a de 2 dimensões, indexada por $\{0, \dots, m\}$ e por $\{0, \dots, n\}$, onde a posição (i, j) da matriz a contém a distância entre os prefixos de s e t de comprimento i e j respectivamente. Em outras palavras, a posição (i, j) de a é tal que $a[i, j] = d(s[1..i], t[1..j])$, para todo $i = 0, \dots, m$ e para todo $j = 0, \dots, n$. A distância entre s e t está portanto, computada em $a[m, n]$.

Naturalmente, para $j = 0, \dots, n$, se $i = 0$, pela definição de $a[i, j]$, temos que $a[0, j] = d(s[1..0], t[1..j]) = d(\varepsilon, t[1..j])$. No caso particular em que existe uma constante g tal que $d(\sqcup, \sigma) = g$ para todo $\sigma \in \Sigma$, $a[0, j] = gj$ (uma vez que o único alinhamento entre ε e uma seqüência qualquer é dado por um espaço alinhado com cada caractere da seqüência). De modo análogo, podemos concluir que $a[i, 0] = gi$, para todo $i = 0, \dots, m$.

Se $i > 0$ e $j > 0$, então, por nossa discussão anterior de como é a última coluna de um alinhamento, sabemos que

$$a[i, j] = \min \left\{ \begin{array}{l} a[i-1, j] + p(s[i], \sqcup) \\ a[i-1, j-1] + p(s[i], t[j]) \\ a[i, j-1] + p(\sqcup, t[j]) \end{array} \right\}.$$

Escolhendo-se uma ordem conveniente, podemos preencher a matriz a de modo iterativo usando a fórmula para $a[i, j]$ e dependendo apenas de entradas de a já computadas no momento de preencher a entrada $a[i, j]$. Uma possibilidade para computar os valores de a seria, digamos, computá-los linha por linha, a partir da linha correspondente a $i = 0$.

Um pseudo-código que implementa estas idéias é o algoritmo 2. Para este algoritmo, assumimos que existe uma constante g tal que $d(\sqcup, \sigma) = g$, para todo símbolo σ do alfabeto considerado. Caso contrário, são necessárias algumas pequenas modificações ao que que apresentamos aqui.

4.1.2 Calculando um Alinhamento via Matriz de Programação Dinâmica

Até o presente momento, apenas a primeira etapa para o cálculo de um alinhamento foi concluída: a que corresponde ao cálculo da pontuação de um alinhamento ótimo (distância entre seqüências). Terminado este pré-processamento, podemos usar a tabela a resultante da primeira etapa para construir os alinhamentos ótimos.

A construção de um alinhamento ótimo é feita observando-se qual (ou quais, se estivermos interessados em vários alinhamentos ótimos) das pontuações de $a[m-1, n]$, $a[m-1, n-1]$ e $a[m, n-1]$ produziu a pontuação $a[m, n]$ (correspondente à pontuação ótima de todos os m caracteres de s alinhados a todos os n caracteres de t) e decidindo, portanto, qual é a última coluna de um alinhamento ótimo.

Desta forma, supondo-se que $a[i, j]$ seja a posição dentre as três descritas a produzir a pontuação $a[m, n]$, podemos obter as demais colunas do alinhamento repetindo o processo,

Algoritmo 2 DIST(s, t)

```
1:  $m \leftarrow |s|$ ;  $n \leftarrow |t|$ ;  
2: para  $j = 0, \dots, n$  faça  
3:    $a[0, j] \leftarrow g \cdot j$ ;  
4: para  $i = 1, \dots, m$  faça  
5:    $a[i, 0] \leftarrow g \cdot i$ ;  
6:   para  $j = 1, \dots, n$  faça  
7:      $a[i, j] \leftarrow a[i - 1, j] + p(s[i], \sqcup)$ ;  
8:     se  $a[i, j] > a[i - 1, j - 1] + p(s[i], t[j])$  então  
9:        $a[i, j] \leftarrow a[i - 1, j - 1] + p(s[i], t[j])$ ;  
10:    se  $a[i, j] > a[i, j - 1] + p(\sqcup, t[j])$  então  
11:       $a[i, j] \leftarrow a[i, j - 1] + p(\sqcup, t[j])$ ;  
12: Devolva  $a[m, n]$ ;
```

usando (i, j) no lugar de (m, n) e prosseguir até que a posição $(0, 0)$ de a seja atingida, momento em que todas as colunas do alinhamento estarão determinadas.

Um algoritmo recursivo que resume a discussão acima é o algoritmo 3.

Algoritmo 3 ALINHA(a, s, t)

```
1:  $m \leftarrow |s|$ ;  $n \leftarrow |t|$ ;  
2: se  $m = 0$  então  
3:   Devolva os caracteres de  $t$  alinhados a espaços em  $s$ ;  
4: se  $n = 0$  então  
5:   Devolva os caracteres de  $s$  alinhados a espaços em  $t$ ;  
6: se  $a[m, n] = a[m - 1, n] + p(s[m], \sqcup)$  então  
7:   Devolva  $\left( \text{ALINHA}(a, s[m - 1], t) : \begin{smallmatrix} s[m] \\ \sqcup \end{smallmatrix} \right)$   
8: se  $a[m, n] = a[m - 1, n - 1] + p(s[m], t[n])$  então  
9:   Devolva  $\left( \text{ALINHA}(a, s[m - 1], t[n - 1]) : \begin{smallmatrix} s[m] \\ t[n] \end{smallmatrix} \right)$   
10: se  $a[m, n] = a[m, n - 1] + p(\sqcup, t[n])$  então  
11:   Devolva  $\left( \text{ALINHA}(a, s, t[n - 1]) : \begin{smallmatrix} \sqcup \\ t[n] \end{smallmatrix} \right)$ 
```

4.1.3 Complexidade do Método

Vamos analisar a complexidade de tempo e de espaço dos algoritmos vistos há pouco. Para evitar ambigüidade, convencionaremos que, nesta seção, a palavra espaço será utilizada para nos referirmos a quantidade de memória requerida para os algoritmos em vez de designar o caractere espaço (\sqcup) como nas demais seções.

O algoritmo 2 para computar $d(s, t)$ inicializa a primeira linha da matriz a em tempo $\Theta(n)$ (linhas 2 e 3). Após a inicialização, os laços encaixados em i e em j (linhas 4–11) são executados. Cada iteração destes laços (isto é, cada execução das linhas 7–11) leva tempo constante e são realizadas um total de $\Theta(mn)$ iterações. Ao longo do algoritmo, a linha 5 é executada um total de m vezes e, como cada execução toma tempo constante, a parcela de tempo total do algoritmo referente a sua execução é $\Theta(m)$. Disto podemos concluir que o algoritmo leva tempo $\Theta(n) + \Theta(m) + \Theta(mn) = \Theta(mn)$.

O espaço usado pelo algoritmo é também $\Theta(mn)$, uma vez que os recursos de memória empregados são, basicamente, a matriz a , de tamanho $(m + 1) \times (n + 1)$ e as variáveis de controle do algoritmo (i, j, m, n) , de onde segue a afirmação.

O algoritmo 3, que constrói um alinhamento ótimo entre s e t , opera em tempo e espaço linear (não contando o espaço requerido para a matriz a). Para nos convenceremos disto, lembremos que cada coluna do alinhamento construído como solução requer que, no máximo, 3 posições da matriz a sejam analisadas (vide linhas 6–11). Como cada um dos testes é feito em tempo constante, a determinação de uma coluna do alinhamento final toma tempo $\Theta(1)$. Ademais, todos os alinhamentos que estamos considerando (i.e., sem colunas tendo apenas brancos) possuem comprimento máximo de $m + n$ caracteres. Daí, concluímos que o algoritmo 3 leva tempo $\Theta(m + n)$.

Para o espaço usado pelo algoritmo 3, observemos que, excetuando-se a matriz a , tudo o que é necessário é armazenar as variáveis de controle do algoritmo e o alinhamento produzido como resposta (que, como mencionado acima, usa espaço $\Theta(m+n)$). Logo o algoritmo opera em espaço linear.

Como um comentário adicional, gostaríamos de ressaltar que, embora tenhamos descrito algoritmos para o problema de encontrar um alinhamento entre pares de seqüências usando espaço quadrático, é possível, por meio de uma técnica de divisão e conquista, realizar todo o procedimento em espaço $\Theta(m+n)$, *mantendo ainda* a complexidade de tempo $\Theta(mn)$ [12].

Capítulo 5

Alinhamentos Múltiplos Exatos

5.1 Programação dinâmica para k seqüências

O problema k -ALIN de encontrar alinhamentos para k seqüências é uma generalização natural do problema de alinhamentos de pares de seqüências e, mais uma vez, o paradigma de programação dinâmica pode ser usado para obter uma solução algorítmica.

Uma aplicação direta das idéias de programação dinâmica para o problema k -ALIN nos dá um algoritmo de complexidade de espaço $\Theta(n^k)$ e de complexidade de tempo $\Omega(2^k n^k)$, para k seqüências de comprimento n . Naturalmente, tal algoritmo não é polinomial nos dados k e n da entrada.

Embora o algoritmo obtido não seja eficiente do ponto de vista prático, ele é a base para interpretações do problema k -ALIN em outros contextos e algumas heurísticas e alguns algoritmos desenvolvem-se com base nestas idéias.

Pretendemos estudar as referências [12, 5, 8] e apresentar uma descrição detalhada do problema e de sua solução via programação dinâmica na versão final da dissertação.

5.2 O Método de Carrillo-Lipman

Uma tentativa de tornar o algoritmo de programação dinâmica mais prático foi proposta em 1988 por Carrillo e Lipman [3]. A idéia básica do método é interpretar o problema k -ALIN como o problema de encontrar um caminho em um reticulado e procurar reduzir o número de nós do reticulado que são visitados na construção do caminho, diminuindo, assim o tempo de execução do algoritmo de programação dinâmica.

Para este assunto, estudaremos o artigo original de Carrillo e Lipman [3], além das referências [5, 12].

Capítulo 6

O Problema k-ALIN é NP-difícil

O problema k-ALIN é NP-difícil e a primeira demonstração deste fato foi realizada por Wang e Jiang em 1994 [18]. A demonstração dada no artigo consiste em computar uma redução polinomial do problema da superseqüência de comprimento mínimo (SCM) ao problema de encontrar um alinhamento de custo mínimo.

Em linhas gerais, a redução dada por Wang e Jiang considera versões de decisão dos problemas de superseqüência de comprimento mínimo e de alinhamento de múltiplas seqüências e constrói um número polinomial de instâncias do problema k-ALIN a partir de uma instância do SCM. Entretanto, a matriz de pontuação utilizada nesta demonstração é bastante peculiar e, em particular, não satisfaz os axiomas de métrica: a pontuação de alinhar dois caracteres iguais de acordo com esta matriz não necessariamente é zero.

Em um recente artigo [2], Bonizzoni e Vedova generalizam o resultado de Wang e Jiang, mostrando que o problema de computar alinhamentos ótimos entre várias seqüências com pontuação SP é NP-difícil no caso de seqüências sobre um alfabeto binário alinhadas de acordo com uma métrica.

A demonstração de Bonizzoni e Vedova computa¹ uma redução do problema da cobertura mínima de arestas por vértices ao problema de alinhamento de múltiplas seqüências.

Um artigo de Just [9], posterior ao de Bonizzoni e Vedova, generaliza os resultados de [18] e de [2] para uma classe maior² de matrizes/funções de pontuação (inclusive no caso especial de lacunas serem permitidas apenas no início ou no fim de seqüências alinhadas) e mostra também que existe uma função de pontuação para qual o problema k-ALIN é MAXSNP-difícil, o que dá indícios de que possivelmente não existe um esquema de aproximação polinomial para o k-ALIN.

Pretendemos estudar as demonstrações de [18], de [2] e de [9] e apresentá-la nas versão final da dissertação.

¹Em versões de decisão.

²No sentido de inclusão.

Capítulo 7

Usando Menos Recursos para Calcular Alinhamentos

Uma vez que o problema k -ALIN é NP-difícil, nossas esperanças de encontrar um algoritmo eficiente (i.e., com tempo de execução polinomial em função do tamanho das entradas) ficam reduzidas. Uma alternativa para este impasse é relaxarmos a condição de procurar *sempre* soluções ótimas para o problema e passarmos a procurar soluções “boas” (mas não necessariamente ótimas), contanto que estas soluções sejam obtidas “rapidamente”.

Com os novos objetivos em mente, podemos estar interessados em heurísticas que produzam alinhamentos de maneira “rápida” ou em algoritmos de aproximação (caso em que, além de desejarmos que os alinhamentos sejam encontrados de forma mais rápida do que pelo algoritmo de programação dinâmica, também queremos que os alinhamentos produzidos tenham pontuações próximas à pontuação ótima para a instância).

7.1 Heurísticas

Uma alternativa à construção de alinhamentos ótimos (usando, por exemplo, o método de programação dinâmica apresentado anteriormente) é o uso de heurísticas. Esta estratégia gera bons resultados quando as seqüências da entrada são similares, embora não haja nenhuma garantia de que os alinhamentos produzidos tenham pontuação próxima à pontuação dos alinhamentos ótimos (como é o caso dos algoritmos de aproximação, vistos a seguir).

7.1.1 Divisão e Conquista

Uma heurística baseada no paradigma de divisão e conquista para resolver o problema k -ALIN foi proposta em 1996 [15] e aprimorada em 1998 [13], com objetivo de tornar mais prática a obtenção de alinhamentos de múltiplas seqüências.

Pretendemos estudar os artigos citados acima e apresentar uma descrição geral do método empregado na versão final da dissertação.

7.2 Algoritmos de Aproximação

Os algoritmos de aproximação para o problema k -ALIN possuem uma particularidade: eles assumem que a função objetivo é uma métrica para poderem fornecer a razão de aproximação, fato ignorado pelas heurísticas.

7.2.1 Algoritmo de Alinhamento Estrela

Um algoritmo de aproximação simples para o problema k-ALIN foi apresentado em 1993 por Gusfield [7].

O algoritmo de Gusfield, também chamado de algoritmo de alinhamentos estrela (por causa da forma como os alinhamentos são produzidos) é uma $(2 - 2/k)$ -aproximação para o k-ALIN, onde k , como anteriormente, é o número de seqüências da entrada. Isto significa que se A é um alinhamento produzido pelo algoritmo e A^* é um alinhamento ótimo, então o alinhamento A é tal que $c(A^*) \leq c(A) \leq (2 - 2/k)c(A^*)$.

Vamos estudar o algoritmo proposto em [7] e apresentá-lo na versão final da dissertação.

7.2.2 Algoritmo de l -estrelas

Em 1997, Bafna, Lawler e Pevzner [1] generalizaram o algoritmo de Gusfield de tal forma que, em vez de considerar estrelas (em que os vértices são seqüências), o algoritmo trabalhe com grafos chamados l -estrelas, para $l < k$, l inteiro. Com isto, obtiveram uma melhora na razão de aproximação, obtendo uma $(2 - l/k)$ -aproximação ao problema k-ALIN.

Pretendemos estudar o artigo [1] e apresentar o algoritmo na dissertação.

Capítulo 8

Alinhamentos de Múltiplas Seqüências na Prática

Conforme indicamos na introdução, o problema k-ALIN é de grande importância em Biologia Computacional. Uma de suas principais aplicações neste contexto é no estudo evolucionário de espécies onde, em geral, deseja-se construir uma árvore evolucionária (também chamada de árvore filogenética ou, simplesmente, de filogenia) a partir de seqüências (de DNA, de RNA ou de aminoácidos) das espécies em estudo.

A construção de filogenias é um ramo amplo e importante de Biologia Computacional e várias técnicas para construção de filogenias estão intimamente relacionadas à construção de alinhamentos de seqüências de espécies. Na realidade, alguns programas para construção de filogenias tomam como entrada um alinhamento de múltiplas seqüências. Exemplos deste tipo de programa são os programas do pacote PHYLIP¹.

8.1 ClustalW

O programa ClustalW é um dos programas mais populares e largamente utilizados para a construção de alinhamentos de múltiplas seqüências. Escrito em 1994 [14], ele possui muitas adaptações e, inclusive, está disponível para uso através de vários lugares da World Wide Web. Ele conta com versões especiais e um exemplo de tais modificações é a versão desenvolvida pela SGI para paralelizar o algoritmo do ClustalW².

Essencialmente, o ClustalW opera em três etapas:

1. Todas as distâncias/similaridades entre cada par de seqüências da entrada são computadas, isto é, computam-se todos os $\binom{k}{2}$ alinhamentos ótimos entre as k seqüências;
2. Constrói-se uma “árvore guia” que determina qual deve ser a ordem em que as seqüências devem ser alinhadas para obter o alinhamento resultado;
3. Cálcula-se o alinhamento propriamente dito determinado pela árvore guia.

Pretendemos estudar o artigo [14] sobre a implementação do ClustalW e apresentar uma descrição detalha do programa na versão final da dissertação.

¹<http://evolution.genetics.washington.edu/phylip.html>

²<http://www.sgi.com/solutions/sciences/chembio/resources/clustalw/>

8.2 MSA

Uma modificação do algoritmo básico de programação dinâmica com as idéias do método de Carrillo-Lipman foi apresentada por Lipman, Altschul e Kececioglu em 1989 e modificada posteriormente na implementação do programa MSA, detalhada no artigo de Gupta, Kececioglu e Schäffer [6].

Pretendemos estudar o artigo [6] e descrever na dissertação a operação do programa MSA em linhas gerais.

Capítulo 9

Apêndice

9.1 Algoritmo de Ukkonen para o Cálculo de Distância de Edição

Ainda que nosso interesse principal seja em alinhamentos de várias seqüências, o estudo de alinhamentos de pares de seqüências é de grande importância: através dele revelam-se idéias que, se generalizadas, podem ajudar a produzir bons algoritmos para várias seqüências e, além disso, algoritmos para alinhamentos de pares de seqüências são geralmente empregados como subrotinas para métodos e heurísticas de alinhamentos de múltiplas seqüências.

Assim, há interesse em desenvolver algoritmos eficientes para alinhamentos de pares de seqüências. Em 1983 e 1985, Ukkonen apresentou um algoritmo para calcular a distância de edição $d(s, t)$ de duas seqüências s e t com tamanhos m e n respectivamente em tempo $O(d(s, t) \min(m, n))$ e espaço $O(d^2(s, t))$.

Pretendemos estudar os artigos [16, 17] e apresentar uma visão geral do algoritmo na dissertação.

9.2 Alinhamentos Ótimos de 3 seqüências

Assim como para o caso de alinhamento de pares de seqüências é possível descrever algoritmos como o de Ukkonen que são mais eficientes do que o algoritmo básico de programação dinâmica, para o caso particular de 3 seqüências também é possível ter um algoritmo que utiliza menos recursos: o algoritmo de Powell, Allison e Dix [11] tem complexidade de tempo $O(d^3 + n)$ na média e $O(nd^2)$ no pior caso para seqüências de comprimento n e distância de edição d .

Pretendemos estudar o artigo citado acima e descrevê-lo na versão final da dissertação.

Capítulo 10

Cronograma de Trabalho

Começamos a estudar o tema de alinhamentos de múltiplas seqüências no início do primeiro semestre de 2001. Durante este período, estive cursando disciplinas para cumprir a meta de créditos do programa de Mestrado do IME/USP, sendo que agora todos os créditos referentes a disciplinas foram obtidos.

A partir da data de apresentação do trabalho para a Banca Examinadora no Exame de Qualificação, a versão final da Dissertação de Mestrado será escrita tendo-se como base as referências bibliográficas apresentadas neste texto e as recomendações da banca.

Estima-se que o estudo dos artigos e capítulos de livros citados como referências e a redação da dissertação tomem o restante do segundo semestre de 2001 e o primeiro semestre de 2002, com a apresentação final da dissertação sendo realizada aproximadamente em julho ou agosto de 2002.

Referências Bibliográficas

- [1] V. Bafna, E. L. Lawler, and P. A. Pevzner. Approximation algorithms for multiple sequence alignment. *Theoretical Computer Science*, 182:233–244, 1997.
- [2] P. Bonizzoni and G. D. Vedova. The complexity of multiple sequence alignment with sp-score that is a metric. *Theoretical Computer Science*, 259:63–79, 2001.
- [3] H. Carrillo and D. Lipman. The multiple sequence alignment problem in biology. *SIAM Journal of Applied Math*, 48(5):1073–1082, October 1988.
- [4] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, 1st edition, 1990.
- [5] G. Fuellen. A gentle guide to multiple alignment. World Wide Web, <http://www.techfak.uni-bielefeld.de/bcd/Curric/MulAli/mulali.html>, March 1997.
- [6] S. K. Gupta, J. D. Kececioglu, and A. A. Schäffer. Improving the practical space and time efficiency of the shortest-paths approach to sum-of-pairs multiple sequence alignment. *Journal of Computational Biology*, 2(3):459–472, 1995.
- [7] D. Gusfield. Efficient methods for multiple sequence alignment with guaranteed error bounds. *Bulletin of Mathematical Biology*, 55(1):141–154, 1993.
- [8] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge Press, 1997.
- [9] W. Just. Computational complexity of multiple sequence alignment with SP-score. World Wide Web, <http://www.math.ohiou.edu/~just/>, 2001.
- [10] P. A. Pevzner. Multiple alignment, communication cost, and graph matching. *SIAM Journal of Applied Math*, 52(6):1763–1779, December 1992.
- [11] D. R. Powell, L. Allison, and T. I. Dix. Fast, optimal alignment of three sequences using linear gap costs. *J. Theor. Biol.*, 207:325–336, 2000.
- [12] J. Setubal and J. Meidanis. *Introduction to Computational Molecular Biology*. PWS Publishing Company, 1997.
- [13] J. Stoye. Multiple sequence alignment with the divide-and-conquer method. *Gene*, 211:45–56, 1998.

- [14] J. D. Thompson, D. G. Higgins, and T. J. Gibson. Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Research*, 22:4673–4680, 1994.
- [15] U. Tönges, S. W. Perrey, J. Stoye, and A. W. M. Dress. A general method for fast multiple sequence alignment. *Gene*, 172:33–41, 1996.
- [16] E. Ukkonen. On approximate string matching. In *Proceedings of the 24th IEEE Annual Symposium on Foundations of Computer Science*, pages 487–495, 1983.
- [17] E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64:100–118, 1985.
- [18] L. Wang and T. Jiang. On the complexity of multiple sequence alignment. *Journal of Computational Biology*, 1:337–348, 1994.
- [19] M. S. Waterman. *Introduction to Computational Biology: Maps, sequences and genomes*. Chapman and Hall, 1995.