Capítulo 8

Algoritmos e heurísticas para comparações exata e aproximada de seqüências

Guilherme P. Telles , Nalvo F. Almeida*, Fábio H. Viduani Martinez

Resumo

A comparação de seqüências é uma tarefa importante em ciência da computação, em particular nas áreas de bioinformática e mineração de dados em textos digitais. A grande massa de dados gerada por aplicações dessas áreas tem gerado desafios, traduzidos em questões a serem respondidas através da comparação de seqüências. Neste texto são apresentados alguns dos principais algoritmos e heurísticas para comparações exatas e aproximadas de seqüências. O objetivo deste texto é proporcionar ao leitor uma introdução ao assunto, sem esgotá-lo.

Abstract

Comparing digital sequences is an important task in computer science. Particularly, it is a central task in bioinformatics and text data mining. Such fields of application also offer huge datasets, what represents another degree of difficulty for pattern matching in sequences. In this text we review the main algorithms and heuristics for exact and inexact sequence comparison. The main goal of this text is to provide an introduction to the subject.

*Financiado pelo CNPq e pela Fundect-MS.

8.1. Introdução

Técnicas computacionais para a manipulação de cadeias de caracteres sempre fizeram parte daquilo que se considera importante em ciência da computação. Em particular, a comparação de seqüências vem se tornando, cada vez mais, um problema computacional de grande interesse.

O interesse pelo problema de comparação de seqüências tem aumentado principalmente por causa de duas vertentes distintas de aplicações. A primeira delas está relacionada à quantidade de informação disponível na *web* e à forma como essa informação pode ser recuperada eficientemente. A segunda vertente está relacionada aos inúmeros avanços da biologia molecular, em particular no desenvolvimento de técnicas de seqüenciamento de DNA.

No que diz respeito à informação disponível na *web*, os desafios impostos por essa fonte de informação e dados se traduzem em duas grandes categorias de problemas. A primeira delas diz respeito aos próprios dados armazenados. Os dados crescem exponencialmente, são voláteis e estão naturalmente armazenados de forma distribuída e redundante, em computadores ligados em uma topologia não-fixa, sem controle de qualidade, incluindo dados falsos, erros gramaticais e erros de digitação. Além disso, esses dados são heterogêneos, com formatos diferentes, escritos em linguagens diferentes e usando alfabetos variados.

A segunda categoria de problemas relacionados à informação disponível na *web* diz respeito à maneira como o usuário interage com essa massa de dados, ou seja, como ele pode conseguir, de forma eficiente e confiável, uma resposta satisfatória. Duas perguntas surgem naturalmente: qual a melhor maneira de se especificar uma consulta e como interpretar a resposta? Essas e outras perguntas ainda se relacionam a um outro problema: sintaticamente a informação devolvida pode estar correta, mas semanticamente pode não estar. Essas e outras questões relacionadas à recuperação de informação publicada na *web* estão fatalmente ligadas ao problema computacional de recuperar informação textual que, por sua vez, está diretamente ligado ao problema de se comparar seqüências.

Paralelamente, a biologia molecular tem contribuído muito para o aumento da necessidade por ferramentas computacionais de comparação de seqüências. A comparação de seqüências biológicas, sejam de DNA ou de proteínas, é um dos principais problemas da biologia computacional, área de pesquisa destinada à solução computacional de problemas de biologia molecular.

A semelhança entre seqüências moleculares pode ser usada como indicação de vários relacionamentos entre elas, como ancestralidade comum ou função similar. Vários eventos causam modificações no genoma de organismos. Alguns ocorrem naturalmente, como a recombinação de cromossomos que precede a formação de células reprodutivas, em que partes de cromossomos homólogos são trocadas. Outros são influenciados por fatores externos, como radiação e incorporação de material genético de vírus.

Há interesse em comparar seqüências de DNA e proteínas de diversas maneiras, como por exemplo:

- Comparar o genoma completo de dois organismos aparentados para identificar as

diferenças;

- Comparar uma seqüência de DNA de centenas de bases com um genoma de milhões de bases para verificar se aquele trecho ocorre no genoma; e
- Comparar seqüências de DNA (ou proteína) de algumas centenas de letras entre si para identificar as semelhanças.

Certamente o campo de aplicações que necessitam de técnicas para a comparação de seqüências não se limita aos exemplos citados acima, mas esses exemplos já nos fornecem motivação suficiente para a tentativa de propor um texto sobre o assunto, com base em experiências didáticas dos autores em cursos de algoritmos e estruturas de dados, além do convívio constante com problemas relacionados à comparação de seqüências biológicas. Este texto, portanto, trata de algoritmos e heurísticas para a o problema de comparação de seqüências, também conhecido como *string-matching* ou *pattern-matching*.

Dentro do escopo deste trabalho, um algoritmo não é somente uma seqüência finita de operações para resolver um problema bem definido, mas também uma seqüência de operações que vem acompanhada de uma garantia sobre a qualidade da solução final apresentada. Já uma heurística é um algoritmo que resolve o problema sem qualquer garantia a respeito da qualidade da solução encontrada ou em relação ao seu tempo de execução.

Quando falamos da qualidade da solução de um problema, estamos nos referindo a um problema de otimização, onde se quer maximizar ou minimizar uma função-objetivo previamente definida. No caso particular da comparação de seqüências, quando nos referimos a problemas que podem ser resolvidos por heurísticas, estamos nos referindo a problemas que podem ser modelados como um problema de otimização. Algumas vezes isso é possível. Por exemplo, quando queremos encontrar o melhor alinhamento entre duas seqüências biológicas, estamos procurando o alinhamento com maior pontuação, segundo critérios pré-definidos. Neste caso particular, um algoritmo garante a determinação de uma solução ótima para o problema, enquanto que uma heurística não garante.

Os problemas de comparação de seqüências que tratamos neste trabalho são separados em duas grandes categorias: comparação exata e comparação aproximada. A comparação exata consiste em determinar trechos de duas seqüências que são exatamente iguais, símbolo a símbolo, enquanto que a comparação aproximada permite diferenças.

Outro tipo de classificação que pode ser encontrada na literatura, e também neste texto, está relacionada aos vários tipos de problemas envolvendo comparação de seqüências e, conseqüentemente, às várias possíveis modelagens a serem propostas para os problemas, independentemente do tipo de comparação (exata ou aproximada). O problema de se comparar duas seqüências biológicas, por exemplo, tenta encontrar um alinhamento das duas seqüências que melhor expressa as similaridades entre elas. Já o problema de se comparar uma seqüência contra um conjunto contendo várias outras seqüências tenta encontrar aquela que lhe é mais similar. Há inúmeros outros problemas correlatos, porém sempre baseados na comparação de seqüências.

O objetivo deste texto é proporcionar ao leitor uma descrição dos principais pro-

blemas envolvendo comparação de seqüências, assim como as principais técnicas para resolvê-los, abordando, sempre que possível, aspectos relacionados às estruturas de dados apropriadas a cada algoritmo, bem como a complexidade de cada um deles.

Organização do texto

O texto está organizado da seguinte forma. Na seção 8.2 fazemos uma descrição dos principais conceitos e notações utilizados ao longo do texto, incluindo conceitos relacionados à complexidade de algoritmos. A seguir, na seção 8.3, descrevemos os principais algoritmos e heurísticas para a comparação de seqüências. Na seção 8.4 descrevemos técnicas que são mais adequadas para a comparação de textos longos, que são a representação do texto como um vetor de termos e a complexidade de Kolmogorov, uma indicação da quantidade de informação em um texto. Na seção 8.5 fazemos alguns comentários, incluindo apontadores para técnicas e aplicações não tratadas neste texto e finalmente apresentamos as referências bibliográficas.

8.2. Definições básicas e notação

Nesta seção fazemos uma descrição dos principais conceitos e notações utilizados ao longo do texto, incluindo conceitos relacionados à complexidade de algoritmos.

Uma **seqüência** ou **cadeia** é uma sucessão ordenada de símbolos de um alfabeto. O **tamanho** ou **comprimento** de uma seqüência s, denotado por $|s|$, é o número de símbolos em s. A cadeia vazia tem tamanho zero e é denotada ε. Os símbolos em uma cadeia s são indexados de 1 até $|s|$, sendo que o símbolo que ocupa a i-ésima posição em s é representado por $s[i]$. Uma **subcadeia** de uma cadeia s é formada por símbolos consecutivos de s, na mesma ordem. A subcadeia de s que contém os símbolos entre i e j inclusive é denotada $s[i..j]$. Se $j < i$ então $s[i..j] = \varepsilon$ e $s[0..0] = \varepsilon$. Uma cadeia s é uma **supercadeia** de uma cadeia t se t é uma subcadeia de s. Dizemos que t é uma **subseqüência** de s se t pode ser obtida a partir de s pela remoção de zero ou mais de seus símbolos. Note que, apesar de usarmos indistintamente os termos *seqüência* e *cadeia*, não fazemos o mesmo com os termos *subseqüência* e *subcadeia*. Note, também, que toda subcadeia é uma subseqüência, mas o contrário não acontece. Um **prefixo** de uma cadeia s é uma subcadeia de s da forma $s[0..i]$ para $0 \leq i \leq |s|$ (ε é prefixo de s). Um **sufixo** de uma cadeia s é uma subcadeia de s da forma $s[i..|s|]$ para $1 \leq i \leq |s|+1$ (ε é sufixo de s).

Neste texto, freqüentemente nos referiremos a seqüências biológicas. Tais seqüências podem ser de DNA, onde o alfabeto é $\{\texttt{A},\texttt{C},\texttt{G},\texttt{T}\}$, ou seqüências de proteínas, que são formadas pelos 20 aminoácidos conhecidos [33].

Quando medimos a eficiência de um algoritmo estamos interessados em saber quanto tempo ele necessita para ser executado. Note, no entanto, que medir empiricamente o tempo de execução de um algoritmo é uma tarefa complicada, pois depende de vários aspectos, como computadores de diferentes desempenho (velocidade de processamento e quantidade de memória), computadores com diferentes plataformas, compartilhamento de máquinas, etc.

Uma maneira mais geral, mais confiável e justa, de se medir a eficiência de um algoritmo é através da contagem do número de operações básicas que ele executa. Uma operação básica depende, obviamente, do algoritmo em questão. Em um algoritmo de ordenação por comparação, por exemplo, a principal operação consiste na comparação entre dois elementos do conjunto, enquanto que em um algoritmo de busca de uma chave em um conjunto a operação básica consiste em uma operação de comparação do elemento a ser procurado com cada elemento do conjunto. No nosso caso, como estamos interessados principalmente em algoritmos para comparar duas seqüências, a operação básica que deve ser levada em conta para a determinação da eficiência dos algoritmos é a comparação entre pares de símbolos das duas seqüências.

Essa medida de eficiência é, em geral, expressa através de uma função do tamanho da entrada do problema. No caso da comparação de duas seqüências, por exemplo, supõe-se em geral que o tamanho do problema seja o tamanho da maior das duas seqüências, digamos n. O interesse maior é pelo comportamento assintótico dessa função, digamos $f(n)$, quando n cresce. A idéia é medir o comportamento assintótico de $f(n)$ para a chamada *instância de pior caso* do problema, que consiste em uma entrada que proporcione o pior comportamento do algoritmo. Como exemplo, um algoritmo pode gastar tempo $f(n) = 3n^2 - 2n + 5$ no pior caso. Para simplificar essas funções, eliminamos todas as constantes e os termos de menor ordem, e o fazemos porque estamos preocupados apenas com o comportamento assintótico da função. Assim, dizemos que $f(n) = O(g(n))$, onde $g(n)$ é uma função que expressa assintoticamente o crescimento de $f(n)$, quando n cresce. Formalmente, $f(n) = O(g(n))$ se existem constantes c e n_0 tais que $f(n) \leq cg(n), \forall n \geq n_0$. Em outras palavras, $f(n)$ não cresce mais que $g(n)$, ou ainda, $g(n)$ é um limite superior para $f(n)$.

No exemplo citado no parágrafo acima, podemos dizer que $f(n) = O(n^2)$. Dizemos então que "o algoritmo é $O(n^2)$" ou que o algoritmo "leva tempo n^2". Um algoritmo $O(n)$ é dito ser "de tempo linear", ou simplesmente linear. Note que um algoritmo $O(n^2)$ pode ser mais rápido do que um algoritmo $O(n)$ para valores pequenos de n, devido às constantes, mas não para valores de n maiores ou iguais a n_0. Um algoritmo cujo tempo de execução é expresso por meio de uma função polinomial é dito ser um **algoritmo polinomial**. Um algoritmo é considerado **eficiente** para um problema de tamanho n se for polinomial em n.

8.3. Algoritmos, complexidades e aplicações

Nesta seção descrevemos várias técnicas para a comparação de seqüências. O objetivo aqui é o de explorar detalhadamente alguns dos principais métodos presentes na literatura, enfatizando a complexidade e, em alguns casos, suas aplicações.

Para o problema da comparação exata, primeiro apresentamos o algoritmo de força-bruta, que é o mais simples para a comparação de duas cadeias. Em seguida, descrevemos dois dos mais importantes algoritmos da literatura, o de Knuth, Morris e Pratt (KMP), que resolve o problema da comparação exata em tempo linear no tamanho das seqüências, e o algoritmo de Boyer e Moore (BM), cuja complexidade se assemelha assintoticamente à complexidade do algoritmo de força-bruta, mas que na prática tem um

comportamento excelente.

Ainda enfocando o problema da comparação exata, descrevemos duas variações de uma estrutura de dados conhecida como *árvore digital*, que é capaz de armazenar em suas arestas trechos de uma seqüência de símbolos, de tal forma que possa ser usada eficientemente na comparação.

A comparação aproximada começa a ser tratada quando descrevemos algoritmos baseados em uma conhecida técnica computacional denominada *programação dinâmica*. Neste ponto descrevemos um conceito muito utilizado em comparação de seqüências, em especial de seqüências biológicas, o *alinhamento* de seqüências. Basicamente, um alinhamento consiste em acrescentar espaços nas duas seqüências de tal forma que ambas fiquem do mesmo tamanho e as partes mais similares fiquem alinhadas nas colunas, quando colocamos uma seqüência sobre a outra. O problema de encontrar uma subseqüência comum de comprimento máximo entre duas seqüências também é descrito neste ponto.

Tratando ainda de comparações aproximadas e de seqüências biológicas, duas importantes heurísticas muito utilizadas hoje são descritas: FASTA e BLAST. São heurísticas usadas para o problema de se encontrar, dentre várias seqüências, aquelas mais similares a uma outra seqüência dada. Esse problema, também conhecido em biologia computacional como *busca em banco de seqüências*, é um dos mais importantes nessa área e tem atraído a atenção de cientistas da computação.

8.3.1. Algoritmos para comparação exata

Sejam s e p duas seqüências de n e m símbolos, respectivamente. A seqüência s é chamada **texto** e a seqüência p é chamada **padrão**. Dizemos que p **ocorre em** s **na posição** $d+1$ se $s[d+1..d+m] = p[1..m]$ para algum d, com $0 \leq d \leq n-m$. O índice d é chamado um **deslocamento** de p sobre s. Nesta seção queremos resolver o problema da comparação exata de duas seqüências (*string matching problem – SM*):

> **Problema** SM(s, p): *dada uma seqüência s de n símbolos e uma seqüência p de m símbolos, com $m \leq n$, encontrar todas as ocorrências de p em s.*

Algoritmo força-bruta

Um algoritmo ingênuo de força-bruta para comparação de duas seqüências s e p é uma seqüência de passos onde em cada um deles queremos saber se $s[d+1..d+m] = p[1..m]$, para todo deslocamento $d = 0, \ldots, n-m$. Ou seja, dado um deslocamento d, comparamos símbolo a símbolo uma subcadeia de s contendo m símbolos que inicia-se na posição $d+1$ e termina na posição $d+m$, para algum d válido, com a seqüência p. Se $s[d+1..d+m] = p[1..m]$, então p ocorre em s na posição $d+1$. Um exemplo de funcionamento deste algoritmo é mostrado na figura 8.1.

Observe que, no pior caso, em que o padrão não ocorre no texto, o algoritmo força-bruta para comparação de seqüências baseado na idéia acima tem tempo de execução $O(mn)$.

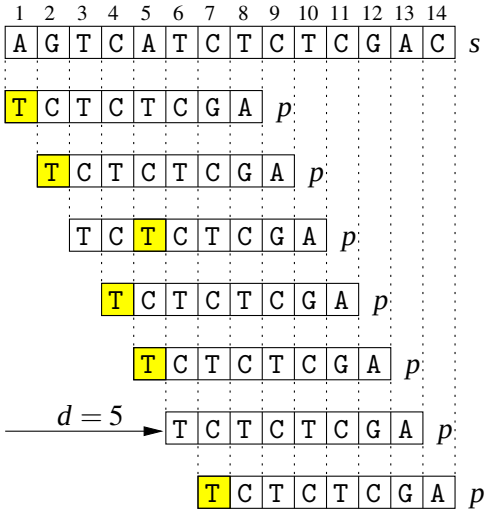

Figura 8.1: Exemplo de comparação de seqüências s = AGTCATCTCTCGAC e p =TCTCTCGA através do algoritmo força-bruta. Dizemos que ocorre um casamento entre as seqüências s e p na posição d = 5.

Algoritmo de Knuth, Morris e Pratt

O algoritmo de Knuth, Morris e Pratt [18] é um algoritmo de comparação exata de seqüências que soluciona o problema SM em tempo linear, isto é, tempo de execução $O(m+n)$ para um texto s de n símbolos e um padrão p de m símbolos, com $m \leq n$. Os símbolos de s e p são obtidos de um alfabeto finito Σ. Através do pré-processamento do padrão p, o algoritmo obtém informação suficiente sobre como o padrão se comporta quando confrontado com deslocamentos dele próprio. Esta informação é então utilizada para evitar verificações impróprias que um algoritmo ingênuo para comparação de seqüências faria [36, 12].

Como já mencionado, o algoritmo de Knuth, Morris e Pratt confronta o padrão com deslocamentos dele próprio e extrai informações suficientes desse processo para evitar comparações desnecessárias entre o texto e o padrão. Por exemplo, suponha que o padrão $p =$ AGAGAGGC é alinhado com o texto s da figura 8.2.

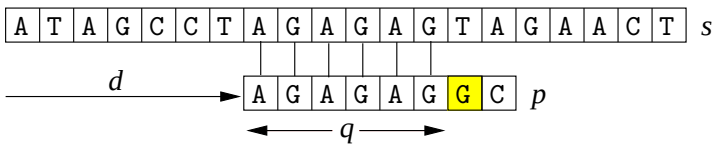

Figura 8.2: Uma comparação entre o padrão p e o texto s onde os q = 6 primeiros símbolos de p casam com os símbolos correspondentes em s.

Observe na figura 8.2 que uma comparação entre o texto s e o padrão p para o deslocamento $d+1$ não é necessária: do deslocamento d sabemos que o primeiro símbolo A do padrão p não irá coincidir com o símbolo da posição $d+1$ do texto s, um G. No entanto, note que o deslocamento $d' = d+2$ não pode ser descartado, como mostra a figura 8.3. Do que sabemos da comparação anterior, com deslocamento d, os quatro primeiros símbolos do padrão coincidem com os quatro símbolos do texto.

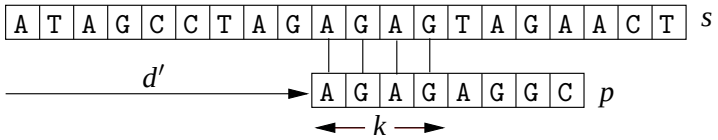

Figura 8.3: Uma comparação entre o padrão p e o texto s para um próximo deslocamento potencial d′ = d + 2.

A idéia central desse algoritmo é encontrar deslocamentos que podem ser seguramente descartados, isto é, cuja comparação entre as seqüências resulta em uma resposta certamente negativa. O algoritmo responde então à seguinte questão: dado que os símbolos $p[1..q]$ do padrão coincidem com os símbolos $s[d+1..d+q]$ do texto, qual o menor deslocamento $d' > d$ tal que

$$p[1..k] = s[d'+1..d'+k], \tag{8.1}$$

onde $d'+k = d+q$?

O melhor que pode acontecer é $d' = d+q$, ou seja, os deslocamentos entre $d'+1$ e $d'+q-1$ serem todos descartados. Além disso, observe que em qualquer deslocamento d' não há a necessidade de comparar os primeiros k símbolos de p com os símbolos correspondentes em s.

Observe também que $s[d'+1..d'+k]$ é um sufixo de $p[1..q]$ e essa informação é conhecida de $p[1..k]$. Como mostra a figura 8.4, obtemos essa informação comparando o padrão p consigo próprio. Isto é, queremos encontrar o maior $k < q$ tal que $p[1..k]$ é um sufixo de $p[1..q]$, uma outra forma de enxergar a equação (8.1). Assim, o próximo deslocamento a ser verificado é $d' = d+(q-k)$.

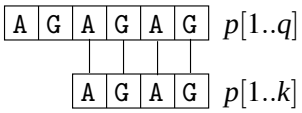

Figura 8.4: Comparando o padrão p consigo próprio, com q = 6 e k = 4.

Dado um padrão p de comprimento m, a **função prefixo** π é tal que:

$$\pi\colon \{1,\ldots,m\} \to \{0,\ldots,m-1\} \text{ e}$$
$$\pi(q) = \max\{k\colon k < q \text{ e } p[1..k] \text{ é sufixo de } p[1..q]\}.$$

O algoritmo FUNÇÃO-PREFIXO computa a função prefixo π relativa a um padrão p dado como entrada.

O tempo de execução do algoritmo FUNÇÃO-PREFIXO depende dos índices q e k. O índice q possui valor inicial 2, é incrementado a cada execução do laço da linha 3 e atinge o valor máximo m. O índice k possui valor inicial 0, é incrementado na linha 7, mas pode ser decrementado na linha 5. O tempo de execução do algoritmo é dado pelo número de vezes que o índice q é incrementado mais o número de vezes que o índice k

FUNÇÃO-PREFIXO(p): recebe uma seqüência p de m símbolos e devolve a função prefixo π para p.

```
1: π[1] ← 0
2: k ← 0
3: para q ← 2 até m faça
4:    enquanto k > 0 e p[k+1] ≠ p[q] faça
5:       k ← π[k]
6:    se p[k+1] = p[q] então
7:       k ← k+1
8:    π[q] ← k
9: devolva π
```

é decrementado, já que quando k é incrementado q também é. O número de vezes que o índice q é incrementado é no máximo m. Como k é decrementado no máximo m vezes, o tempo de execução do algoritmo FUNÇÃO-PREFIXO é $O(m)$, onde m é o comprimento do padrão p. Um exemplo de execução desse algoritmo para o padrão $p =$ AGAGAGGC é mostrado na figura 8.5.

	1	2	3	4	5	6	7	8
p	A	G	A	G	A	G	G	C
π	0	0	1	2	3	4	0	0

Figura 8.5: Computação da função prefixo para p =AGAGAGGC.

O algoritmo para comparação de seqüências de Knuth, Morris e Pratt recorre ao procedimento FUNÇÃO-PREFIXO para pré-processar o vetor π e, em seguida, realiza a busca do padrão p no texto s fornecidos como entrada.

KMP(s, p): recebe um texto s de n símbolos e um padrão p de m símbolos e realiza a comparação de s e p, devolvendo os índices em s onde p ocorre.

```
1: π ← FUNÇÃO-PREFIXO(p)
2: q ← 0
3: para i ← 1 até n faça
4:    enquanto q > 0 e p[q+1] ≠ s[i] faça
5:       q ← π[q]
6:    se p[q+1] = s[i] então
7:       q ← q+1
8:    se q = m então
9:       escreva "Padrão ocorre no texto com deslocamento" i − m
10:      q ← π[q]
```

O tempo de execução do algoritmo KMP depende do tempo de execução do algoritmo FUNÇÃO-PREFIXO mais o tempo gasto nas linhas de 3 a 10. Como já foi discutido, o tempo de execução do algoritmo FUNÇÃO-PREFIXO é $O(m)$. Usando para i e q no

algoritmo KMP o mesmo argumento que para q e k no algoritmo FUNÇÃO-PREFIXO, concluímos que o tempo de execução do laço das linhas 3 até 10 é $O(n)$, onde n é o comprimento do texto s. Dessa forma, o tempo de execução do algoritmo KMP é $O(m+n)$. Um exemplo de execução desse algoritmo é mostrado na figura 8.6.

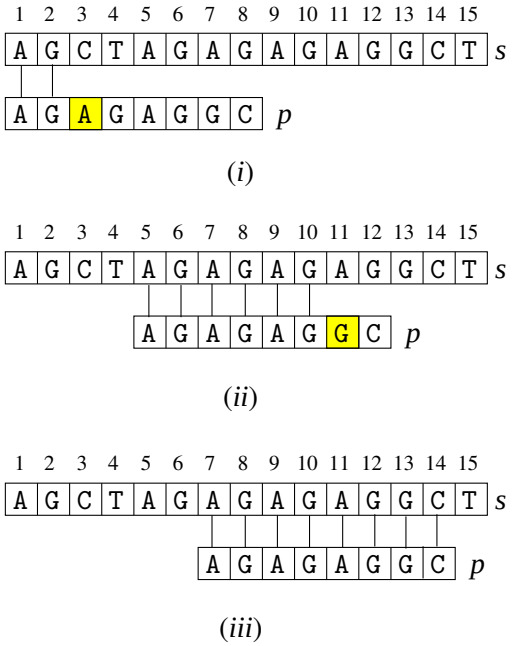

Figura 8.6: Um exemplo de execução do algoritmo KMP, tendo como entrada o texto $s =$ AGCTAGAGAGAGGCT e o padrão $p =$ AGAGAGGC. A função prefixo π para o padrão p é mostrada na Figura 8.5. (i) Comparação entre p e s com deslocamento 0 e a primeira diferença de símbolos ocorrendo na posição $i = 3$ de s. Neste ponto, $q = 2$, mas como $q > 0$ e $p[q+1] \neq s[i]$, o algoritmo faz $q \leftarrow \pi[q] = 0$. (ii) Comparação entre p e s com deslocamento 4 e a primeira diferença de símbolos ocorrendo na posição $i = 9$ de s. Observe que $q = 6$, mas como $q > 0$ e $p[q+1] \neq s[i]$, o algoritmo faz $q \leftarrow \pi[q] = 4$. (iii) Casamento das seqüências com deslocamento $6 = i - m = 14 - 8$.

Algoritmo de Boyer e Moore

O algoritmo de Boyer e Moore [8] é um algoritmo bastante utilizado na prática para solucionar o problema SM, apesar de sua complexidade assintótica ser equivalente à complexidade do algoritmo ingênuo. À primeira vista, esse algoritmo se assemelha ao algoritmo ingênuo para comparação exata de seqüências. Apesar dessa semelhança, o algoritmo de Boyer e Moore torna-se eficiente devido à inclusão de duas heurísticas que permitem descartar seguramente muitas comparações entre os símbolos das seqüências.

O algoritmo de Boyer e Moore inclui duas heurísticas para acelerar a comparação das seqüências denominadas *heurística do símbolo ruim* e *heurística do sufixo bom.*

Na figura 8.7 apresentamos um exemplo dessas heurísticas. Observe que o deslocamento d não é válido, isto é, as seqüências não se casam para o deslocamento $d = 2$. O algoritmo do símbolo ruim tenta movimentar o padrão sobre o texto de forma que o

símbolo ruim no texto seja casado com o mesmo símbolo mais à direita do padrão. Na figura 8.8 o padrão é movido 4 posições à direita e assim o símbolo ruim $s[7] = \texttt{A}$ do texto se casa com o símbolo $p[2] = \texttt{A}$ do padrão. Se o símbolo ruim não ocorre no padrão, então o padrão pode ser movimentado seguramente para a primeira posição após o símbolo ruim no texto.

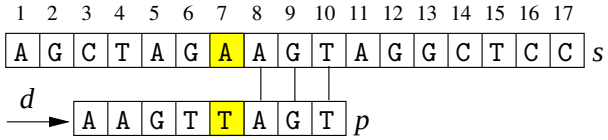

Figura 8.7: Comparação de p =AAGTTAGT e s =AGCTAGAAGTAGGCTCC com deslocamento d = 2 feita pelo algoritmo de Boyer e Moore. Essa comparação é realizada da direita para a esquerda e o primeiro símbolo onde ocorre uma diferença é destacado (p[5] = T ≠ A = s[7]).

Dado um deslocamento $d \geq 0$, se $p[j] \neq s[d+j]$ para algum j, com $1 \leq j \leq m$, então o algoritmo do símbolo ruim encontra o maior índice k, com $1 \leq k \leq m$, tal que $s[d+j] = p[k]$ se um tal k existe. Caso contrário, $k = 0$. Então, o algoritmo toma o próximo deslocamento como sendo $d + j - k$.

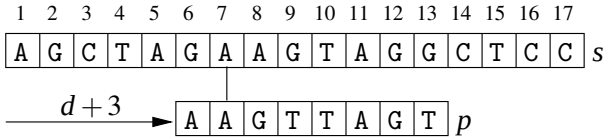

Figura 8.8: Deslocamento de $d+3$ posições do padrão p sobre o texto s da figura 8.7, devido ao símbolo ruim A em s.

A **função última ocorrência** λ implementa a idéia de deslocamentos baseados em um símbolo ruim e é definida da seguinte forma: $\lambda[a]$ é o índice da posição mais à direita no padrão em que o símbolo a ocorre, para cada símbolo $a \in \Sigma$. O algoritmo a seguir implementa essa função.

FUNÇÃO-ÚLTIMA-OCORRÊNCIA(p, Σ): recebe um padrão p de m símbolos e um alfabeto Σ e devolve a função última ocorrência λ para todo símbolo em Σ.

1: **para** cada símbolo $a \in \Sigma$ **faça**
2: $\quad \lambda[a] \leftarrow 0$
3: **para** $j \leftarrow 1$ **até** m **faça**
4: $\quad \lambda[p[j]] \leftarrow j$
5: **devolva** λ

O tempo de execução do algoritmo FUNÇÃO-ÚLTIMA-OCORRÊNCIA é $O(\Sigma + m)$. Um exemplo de execução do algoritmo FUNÇÃO-ÚLTIMA-OCORRÊNCIA sobre o padrão p da figura 8.8 é mostrado na figura 8.9.

A outra heurística utilizada no algoritmo de Boyer e Moore é o algoritmo do sufixo bom. Nesse caso, o padrão é movido para direita com um deslocamento que garante

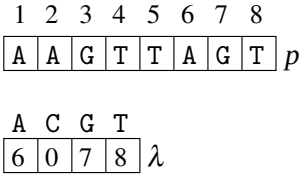

Figura 8.9: Execução do algoritmo Função-Última-Ocorrência sobre o padrão p da figura 8.8. Note, por exemplo, que 6 é a posição da última ocorrência do símbolo `A` em p.

que símbolos do padrão casam com o sufixo bom encontrado no texto. No exemplo da figura 8.7 o deslocamento obtido pelo algoritmo do sufixo bom é ilustrado na figura 8.10.

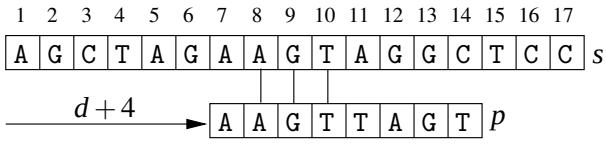

Figura 8.10: Deslocamento de d+4 posições do padrão p sobre o texto s da figura 8.7, devido ao sufixo bom `AGT`.

Sejam x e y duas seqüências sobre um alfabeto Σ. Dizemos que x é **similar** a y, e denotamos por $x \sim y$, se x é sufixo de y ou y é sufixo de x. Note que a relação $\sim$ tem a seguintes propriedades:

- se x e y são seqüências sobre um alfabeto Σ, então $x \sim y$ se e somente se $y \sim x$ (simétrica);
- se x, y e z são seqüências sobre um alfabeto Σ tais que x é sufixo de y e z é sufixo de y então $x \sim z$.

Na heurística do sufixo bom, se $p[j] \neq s[d+j]$ para $j < m$ na comparação do texto com o padrão, então o deslocamento d pode ser seguramente avançado de $\gamma[j]$ posições onde

$$\gamma[j] = m - \max\{k\colon 0 \leq k < m \text{ e } p[j+1..m] \sim p[1..k]\}.$$

A função γ é chamada **função sufixo bom** para o padrão p. Note que $\gamma[j]$ é o menor avanço possível para o deslocamento d para o qual os símbolos no sufixo bom $s[d+j+1..d+m]$ do texto se casam com os símbolos do sufixo correspondente do padrão.

Para efeito de computação, a função sufixo bom γ pode ser reescrita da seguinte forma:

$$\gamma[j] = \min\big(\{m - \pi[m]\} \cup \{l - \pi'[l]\colon 1 \leq l \leq m \text{ e } j = m - \pi'[l]\}\big),$$

onde π é a função prefixo para o padrão p e π' é a função prefixo para o reverso p' do padrão p. Um algoritmo FUNÇÃO-SUFIXO-BOM, que implementa essa fórmula, é apresentado a seguir.

Observe que o algoritmo FUNÇÃO-SUFIXO-BOM utiliza o algoritmo FUNÇÃO-PREFIXO descrita na seção 8.3.1 para computar a função prefixo π do padrão p e a função

FUNÇÃO-SUFIXO-BOM(p): recebe o padrão p de m símbolos e devolve a função sufixo bom γ para p.

```
1: π ← FUNÇÃO-PREFIXO(p)
2: p' ← reverso(p)
3: π' ← FUNÇÃO-PREFIXO(p')
4: para j ← 0 até m faça
5:     γ[j] ← m − π[m]
6: para l ← 1 até m faça
7:     j ← m − π'[l]
8:     se γ[j] > l − π'[l] então
9:         γ[j] ← l − π'[l]
10: devolva γ
```

prefixo π' do reverso π' do padrão π. Um exemplo de execução do algoritmo FUNÇÃO-SUFIXO-BOM sobre o padrão p da figura 8.7 é apresentado na figura 8.11. Por simples inspeção no pseudocódigo do algoritmo FUNÇÃO-SUFIXO-BOM notamos que o tempo de execução desse algoritmo é $O(m)$.

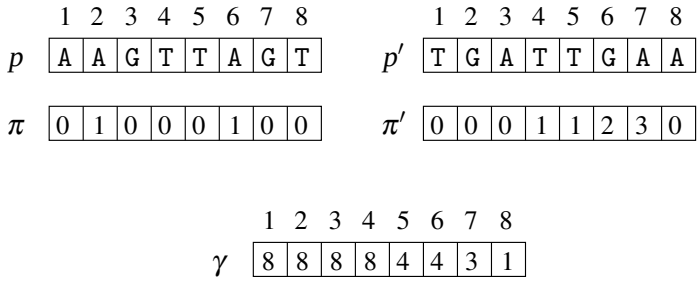

Figura 8.11: Execução do algoritmo Função-Sufixo-Bom sobre o padrão p da figura 8.7.

Finalmente, o algoritmo de Boyer e Moore para comparação de seqüências é apresentado a seguir. Esse algoritmo, à primeira inspeção, é parecido com o algoritmo força-bruta, a menos da comparação do padrão contra o texto realizada da direita para a esquerda e do uso das heurísticas do símbolo ruim e do sufixo bom, que trabalham independentemente. O maior deslocamento devolvido por ambas é então utilizado como próximo deslocamento confiável. São justamente essas heurísticas que garantem deslocamentos seguros de vários símbolos do texto.

Um exemplo de execução do algoritmo BM é mostrado na figura 8.12.

O tempo de execução do algoritmo BM é $O((n-m+1)m+|\Sigma|)$, que supera o tempo de execução do algoritmo KMP e é equivalente ao tempo de execução do algoritmo força-bruta. Entretanto, o algoritmo BM tem comportamento eficiente na prática e é, por isso, bastante utilizado.

8.3.2. Algoritmos para comparação exata baseados em árvores digitais

Nesta seção veremos estruturas de dados cuja principal aplicação é a busca de uma ocorrência exata de uma dada seqüência p em um texto s ou em cada um dos textos de

BM(s, p, Σ): recebe um texto s de n símbolos, um padrão p de m símbolos de um alfabeto Σ e devolve os índices em s onde p ocorre.

```
 1: λ ← FUNÇÃO-ÚLTIMA-OCORRÊNCIA(p, Σ)
 2: γ ← FUNÇÃO-SUFIXO-BOM(p)
 3: d ← 0
 4: enquanto d ≤ n − m faça
 5:     j ← m
 6:     enquanto j > 0 e p[j] = s[d + j] faça
 7:         j ← j − 1
 8:     se j = 0 então
 9:         escreva "Padrão ocorre no texto com deslocamento" d
10:         d ← d + γ[0]
11:     senão
12:         d ← d + max(γ[j], j − λ[s[d + j]])
```

uma coleção R de textos. A estrutura de dados é uma árvore que armazenará informações sobre cada seqüência de R ou sobre s. Essas informações são armazenadas de tal forma que cada aresta da árvore represente subcadeias das seqüências de R ou de s.

A principal característica desse tipo de estrutura é o fato de a busca não considerar p como um único elemento indivisível, mas sim como uma seqüência de símbolos pertencentes a um alfabeto Σ. A busca, denominada *busca digital*, consiste em comparar símbolos de p com símbolos da estrutura, individualmente, até que toda a seqüência seja encontrada ou a busca termine sem que p seja encontrado. A estrutura neste caso é chamada de *árvore digital*.

A idéia é que as arestas da árvore de alguma forma representem subcadeias das seqüências do conjunto. Veremos duas principais estruturas: na primeira consideraremos uma árvore digital geral, a qual chamaremos simplesmente de *árvore digital*, onde cada aresta representa exatamente um símbolo do alfabeto. Na segunda estrutura, veremos um tipo específico de árvore digital, denominada *árvore de sufixos*, onde cada aresta representa uma cadeia de símbolos do alfabeto.

Árvore digital

Supondo um alfabeto Σ, tal que seus símbolos possuam uma ordenação, uma **árvore digital** T para o conjunto de k seqüências não-vazias $R = \{r_1, r_2, \ldots, r_k\}$ é uma árvore não vazia onde cada nó interno tem $|\Sigma|$ filhos, tal que:

1. o j-ésimo filho de um nó interno v corresponde ao j-ésimo símbolo de Σ, para todo $1 \leq j \leq |\Sigma|$;
2. Para cada nó v, interno ou folha, a seqüência de símbolos definida pelo único caminho da raiz de T até v corresponde a um prefixo de alguma seqüência de R e
3. Toda seqüência de R corresponde a um caminho da raiz de T a um único nó v.

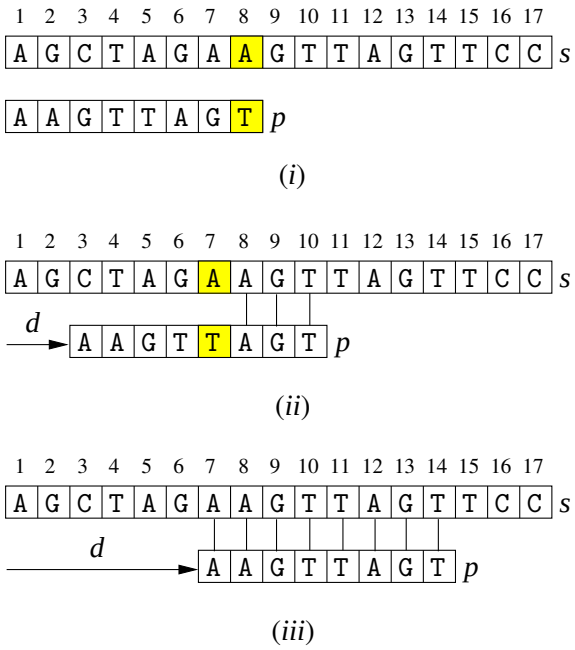

Figura 8.12: Um exemplo de execução do algoritmo BM para as entradas $s = \texttt{AGCTAGAAGTTAGTTCC}$ e $p = \texttt{AAGTTAGT}$. A função última ocorrência λ e sufixo bom γ para o padrão p são apresentadas nas figuras 8.9 e 8.11. (i) Comparação entre p e s com deslocamento 0 e a primeira diferença de símbolos ocorrendo na posição $j = 8$ ($T = p[8] \neq s[0+8] = A$). Como $\gamma[8] = 1$ e $j - \lambda[A] = 2$, o valor do próximo deslocamento é $d = 0+2$. (ii) Comparação entre p e s com deslocamento 2 e a primeira diferença de símbolos ocorrendo na posição $j = 5$ ($T = p[5] \neq s[2+5] = A$). Como $\gamma[5] = 4$ e $j - \lambda[A] = -1$, o valor do próximo deslocamento é $d = 2+4 = 6$. (iii) Casamento das seqüências com deslocamento 6. Neste ponto, como $j = 0$ e $\gamma[0] = 8$, temos que $d = d + \gamma[8] = 6+8 = 14 > 9 = n - m$ e o algoritmo termina.

A figura 8.13 mostra um exemplo de árvore digital. Note que existe na árvore um nó para cada seqüência de R. Esses são os nós representados por círculos cheios na figura e poderiam, por exemplo, apontar para o local onde a chave e demais informações agregadas a ela estariam localizados. A esses nós denominamos **terminais**. Vale lembrar que um nó terminal não necessariamente é um nó folha. Por outro lado, todo nó folha deve ser terminal. Note também que, como a ordenação dos símbolos de Σ pré-determina a ordenação dos filhos de cada nó da árvore, implicitamente a sua formação nos diz qual ou quais são os filhos que existem para cada nó.

A busca de uma cadeia p, de tamanho m, em uma árvore digital T_R já construída para o conjunto R é muito simples. Como a inclusão de uma nova seqüência em T_R depende da busca, descreveremos da busca e da inclusão ao mesmo tempo. A busca inicia-se pela raiz de T_R. A cada símbolo de p lido, tenta-se caminhar pelas arestas de T_R, comparando cada símbolo de p com cada símbolo encontrado em T_R, até que não seja mais possível. As possibilidades para o término da busca são as seguintes:

- toda a cadeia p é lida e a busca termina em um nó v. Neste caso, $p \in R$ se e somente se v é terminal. Mais ainda, se v é não-terminal, então a inclusão da cadeia p em T_R

é feita simplesmente pela transformação de v em terminal. Note que, neste caso, p é prefixo de alguma cadeia já representada em T_R;

- a segunda possibilidade acontece quando a busca termina em algum nó v sem que p tenha sido completamente lido. Seja $p[1..i], i < m$, o prefixo de p lido até esse momento. Neste caso $p \notin R$ e a inclusão de p em T_R consiste na inclusão do sufixo $p[i+1..m]$ na árvore, a partir de v, símbolo a símbolo, incluindo, portanto, $m-i$ novas arestas em T_R.

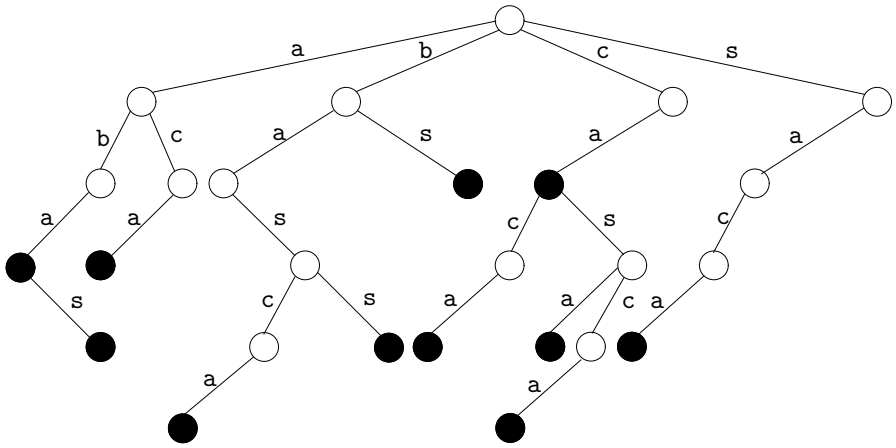

Figura 8.13: Árvore digital para o alfabeto $\blacksquare = \{\mathbf{a},\mathbf{b},\mathbf{c},\mathbf{s}\}$ e o conjunto $\mathbf{R} = \{$aba, abas, aca, basca, bass, bs, ca, caca, casa, casca, saca$\}$.

O algoritmo recursivo BUSCA-DIGITAL computa a busca de uma cadeia de entrada $p[1..m]$, em uma árvore digital apontada por *ptr*. Cada nó v da árvore é um registro composto de um vetor de apontadores *filho*$[1..|\Sigma|]$, tal que *filho*$[p[j]]$ não aponta para o endereço nulo se e somente se existe uma aresta rotulada com o símbolo $p[j]$ saindo de v, $1 \leq j \leq |\Sigma|$; e também de uma variável lógica *terminal*, que indica se v é ou não um nó terminal. Note que, assim, pode-se determinar em tempo constante se tal aresta existe, caso o alfabeto seja de tamanho constante.

O algoritmo tem como entrada quatro parâmetros:

- p, a cadeia a ser procurada;
- *ptr*, um apontador para a raiz da subárvore corrente;
- i, o último símbolo de p encontrado na busca; e
- *resultado*, que indica se p foi ou não encontrado.

Os parâmetros i e *resultado* devem ser passadas por referência e indicam, ao final, o resultado da busca, da seguinte forma: se *resultado* é igual a 1, então a busca terminou com sucesso no nó apontado por *ptr*. Neste caso i estará valendo m. Se ao final da busca *resultado* é igual a zero, então p não foi encontrado, e *ptr* deve ser usado para a inclusão do sufixo $p[i+1..m]$, $1 \leq i \leq m-1$, em T_R. A chamada externa deve ser feita como BUSCA-DIGITAL$(p, raiz, 0, 0)$, onde *raiz* é um apontador para a raiz de T_R.

BUSCA-DIGITAL(p, *raiz*, i, *resultado*): recebe p e *raiz* e devolve, através de i e *resultado*, o resultado da busca de p em uma árvore digital com raiz apontada por *raiz*.

```
1: se i < m então
2:    se ptr.filho[p[i+1]] ≠ nulo então
3:       i ←i+1
4:       BUSCA-DIGITAL(p,ptr.filho[p[i+1]],i,resultado)
5:    senão
6:       se ptr.terminal = 1 então
7:          resultado ←1
```

A inclusão de uma nova cadeia p em T_R deve ser feita, como vimos, após a sua busca na árvore. O algoritmo INCLUSÃO-DIGITAL[1], faz a inclusão de p em T_R. A inclusão consiste em acrescentar $m-i$ arestas em T_R, a partir do nó *ptr*, onde a busca foi encerrada, cada uma contendo um símbolo do sufixo $p[i+1..m]$ de p, $i \geq 0$. Se $i = m$, então é porque a busca terminou após a leitura de toda a cadeia p e, neste caso, o laço controlado pela variável j do algoritmo (linhas 8 a 16) sequer é executado e a única operação a ser feita é a transformação do nó *ptr* em um nó terminal.

INCLUSÃO-DIGITAL(p,*raiz*): recebe p, a cadeia a ser incluída na árvore digital, cuja raiz é apontada por *raiz*.

```
1: ptr ←raiz
2: i ←0
3: resultado ←0
4: Busca-Digital(p,ptr,i,resultado)
5: se resultado≠ 0 então
6:    escreva "p pertence a R"
7: senão
8:    para j←i+1 até m faça
9:       pts ←novo nó
10:      pts.terminal ←0
11:      para todo símbolo α ∈ Σ faça
12:         pts.filho[α] ←nulo
13:      ptr.filho[p[j]] ←pts
14:      ptr ←pts
15: ptr.terminal ←1
```

A construção de uma árvore digital T_R pode ser feita pela execução do algoritmo de inclusão, executado para cada seqüência de R, a partir de uma árvore contendo apenas o nó raiz. Supondo que R contém k cadeias com tamanho médio m, o espaço gasto por T_R é, no pior caso, $O(|\Sigma|mk)$. O pior caso acontece quando todos os prefixos são distintos.

A complexidade do algoritmo de busca depende diretamente do tamanho m da cadeia p e também do tamanho do alfabeto Σ. Cada comparação de um símbolo de p em um

[1]Cujo nome mais parece um programa social do governo.

nó v consiste em descobrir se a partir de v sai uma aresta rotulada com aquele símbolo. Como Σ tem os símbolos ordenados, isso pode ser determinado em tempo $O(\log|\Sigma|)$, usando uma busca binária no vetor de apontadores para os filhos de v. Assim, a busca gasta tempo $O(m\log|\Sigma|)$. Entretanto, para a maioria das aplicações, o alfabeto é de tamanho muito menor que o tamanho das seqüências. Assim, podemos considerar que o tempo da busca é $O(m)$, portanto proporcional ao tamanho da cadeia p. Usando o mesmo raciocínio, podemos concluir que a complexidade do algoritmo de inclusão também é $O(m)$.

Um tipo específico de árvore digital bastante considerado na literatura, mas não tratado neste texto, é a *árvore digital binária*, onde $\Sigma = \{0,1\}$. Os principais problemas com complexidade também acontecem quando se tem muitas cadeias armazenadas na árvore com prefixos distintos. No entanto, no caso binário, é possível uma implementação mais sofisticada, que elimina esse tipo de problema. Essa implementação de árvore digital binária recebe o nome de árvore *PATRICIA*, de *Practical Algorithm to Retrieve Information Coded in Alphanumeric* [36].

A seguir, descrevemos um tipo de árvore digital onde cada aresta, ao invés de representar um único símbolo, pode representar uma cadeia de símbolos do alfabeto.

Árvore de sufixos

Uma *árvore de sufixos* é uma árvore digital que armazena os sufixos de uma seqüência de símbolos. Essa estrutura foi inicialmente proposta para a solução de problemas de casamento exato de padrões, mas suas funcionalidades vão além disso, incluindo compressão de texto.

Neste trabalho, veremos que uma árvore de sufixos pode ser utilizada eficientemente na comparação exata de seqüências. Entretanto, ela pode ser utilizada em muitas aplicações mais complexas envolvendo seqüências. Uma descrição bastante detalhada, incluindo diversas aplicações de árvores de sufixos, pode ser encontrada em [16].

Nesta seção seguiremos o seguinte roteiro: primeiro definimos uma árvore de sufixos formalmente; em seguida descrevemos um algoritmo para localizar as ocorrências exatas de uma seqüência p de tamanho m em um texto s de tamanho n, representado por uma árvore de sufixos T_s já construída; depois descrevemos um algoritmo para a construção da árvore T_s; ao final, alguns detalhes são descritos e outras aplicações são citadas.

Seja $s[1..n]$ uma seqüência de n símbolos de um alfabeto Σ, onde $s[n] = \$$ é um símbolo especial que não ocorre em qualquer outra posição de s. A **árvore de sufixos** T_s para s é uma árvore enraizada, com n folhas e no máximo $n-1$ nós internos, tal que:

1. as arestas de T_s são orientadas no sentido raiz→folhas e são rotuladas com subcadeias de s;

2. cada nó interno tem pelo menos dois filhos;

3. quaisquer arestas distintas que saem de um mesmo nó estão rotuladas com subcadeias de prefixos diferentes;

4. cada folha está rotulada com um inteiro i, $1 \leq i \leq n$, que representa uma posição de s; e

5. a concatenação dos rótulos das arestas de um caminho da raiz até uma folha i corresponde ao sufixo de s que começa na posição i.

A figura 8.14 mostra a árvore de sufixos para a seqüência $s = \texttt{xabxa\$}$. Note que a árvore tem $n = 6$ folhas, numeradas de 1 a 6, de tal modo que o caminho da raiz até a folha i representa o sufixo de s que começa em s_i, $i = 1, \ldots, 6$. Denotaremos esse sufixo como **sufixo i de s.**

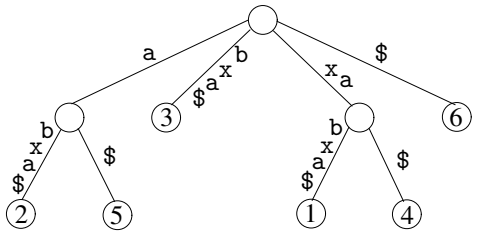

Figura 8.14: Árvore de sufixos para a seqüência xabxa$.

A presença do símbolo especial \$ se faz necessária pelo fato de ser impossível representar todos os sufixos de uma seqüência que tenha dois ou mais sufixos que compartilham um mesmo prefixo. A figura 8.15 mostra a árvore de sufixos para a seqüência $s = \texttt{xabxa}$. Note que os sufixos 4 e 5 não estão explicitamente representados na árvore, já que o sufixo 5 é prefixo do sufixo 4. Assim, para garantir que todos os sufixos de s sejam explicitamente representados, fazemos uso do símbolo especial \$. Suporemos, a partir de agora, que s sempre possui o símbolo especial. Além disso, suporemos que o tamanho de Σ é constante, o que na prática faz sentido para muitas aplicações.

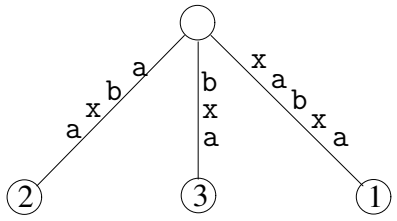

Figura 8.15: Árvore de sufixos para a seqüência xabxa (sem o símbolo especial).

Busca de uma seqüência em um texto usando árvore de sufixos

A aplicação clássica de árvores de sufixos é a busca de todas as ocorrências de uma seqüência p de tamanho m em um texto s de tamanho n, dada a árvore de sufixos T_s.

A idéia parte do princípio de que qualquer sufixo $s[i..m]$ de s deve estar representado na árvore, através de um caminho único da raiz de T_s até a folha rotulada com i. Por outro lado, qualquer subcadeia $s[i..j]$ de s, $i \leq j \leq m$, é um prefixo do sufixo $s[i..m]$ e, portanto, deve rotular a parte inicial desse caminho único. Assim, para descobrir se há ocorrências de p em s, deve-se percorrer esse caminho, comparando os símbolos de p com os símbolos que rotulam as arestas do caminho, até que p seja integralmente encontrado ou nenhum casamento seja mais possível. No primeiro caso, todas as folhas na subárvore abaixo do nó onde a busca terminou rotulam posições onde p ocorre em s. No segundo caso, p não ocorre em s.

O caminho a ser percorrido na busca é único porque quaisquer duas arestas que saem de um vértice interno de T_s são rotuladas com subcadeias de s que obrigatoriamente começam com símbolos diferentes, pela própria definição da árvore de sufixos.

Na figura 8.14 podemos ver que a busca por $p = xa$ terminará em um vértice interno, cuja subárvore tem as folhas rotuladas com 1 e 4, significando, portanto, que p ocorre nessas posições. Note que essa busca, mesmo que com resultado positivo, não necessariamente termina em um vértice interno. Ela pode terminar em uma folha, ou mesmo no interior de uma aresta. No caso da busca terminar em uma folha, tem-se apenas uma ocorrência de p em s, enquanto que se a busca terminar antes de alcançar algum vértice qualquer (interno ou folha), esse vértice determina a posição ou as posições de ocorrência.

No caso em que a busca termina sem que p seja integralmente avaliado, tem-se a situação onde p não ocorre em s. Essa busca também pode terminar em um vértice ou no interior de uma aresta.

Uma vez determinado que p ocorre em s, nas posições determinadas pelos rótulos das folhas abaixo do (ou exatamente no) vértice onde a busca terminou, basta executarmos um algoritmo qualquer de percurso na subárvore enraizada por esse vértice, coletando os rótulos das folhas. Assim, pode-se determinar as ocorrências de p em s em tempo $O(m+k)$, onde k é o número de ocorrências.

A construção da árvore T_s para uma seqüência s de tamanho n, como veremos adiante, pode ser feita em tempo $O(n)$. Dessa forma, temos um algoritmo que, em tempo $O(n+m)$, encontra todas as ocorrências exatas de uma cadeia de tamanho m em um texto de tamanho n. Esse custo é o mesmo de algoritmos como o KMP, descrito na seção 8.3.1. Entretanto, a distribuição de trabalho aqui é bem mais vantajosa, uma vez que gasta-se $O(n)$ para o pré-processamento (construção da árvore) e depois pode-se fazer várias buscas, cada uma com tempo $O(m+k)$. Como m e k, em geral, são muito menores que n, esse tipo de processamento é mais vantajoso com o uso da árvore de sufixos.

Construção da árvore de sufixos

Weiner [39] propôs, em 1973, o primeiro algoritmo linear no tamanho de s para a construção da árvore de sufixos T_s. Alguns anos mais tarde McCreight [24] também propôs um algoritmo linear, mais econômico em termos de espaço. Em 1995, Ukko-

nen [38] apresentou um algoritmo também linear, com as vantagens do algoritmo de McCreight, só que mais facilmente explicável.

O algoritmo de Ukkonen, apesar de mais simples do que os anteriores, ainda requer uma extensa e detalhada explicação, que será omitida neste texto. Uma excelente descrição do algoritmo linear de Ukkonen pode ser vista em [16]. Objetivando, no entanto, um melhor entendimento da estrutura, um algoritmo muito simples, porém de tempo $O(n^2)$, é descrito a seguir. A idéia consiste em incluir em uma árvore inicialmente contendo apenas o sufixo $s[1..n]$, todos os outros sufixos $s[i..n]$ de s, com $i = 2, \ldots, n$.

Vamos descrever agora esse algoritmo mais detalhadamente. O passo inicial consiste na construção de uma árvore T_s^1 contendo apenas a raiz, uma aresta rotulada com o sufixo $s[1..n]$ e uma folha rotulada com 1. Seja então T_s^i a árvore intermediária contendo todos os sufixos $s[j..n]$ de s, com $1 \leq j \leq i$.

A cada passo o algoritmo constrói a árvore T_s^{i+1}, acrescentando o sufixo $s[i+1..n]$ em T_s^i, da seguinte forma. Faz-se uma busca de $s[i+1..n]$ em T_s^i. Seja C o caminho percorrido em T_s^i na busca por $s[i+1..n]$. Note que C sempre termina antes que $s[i+1..n]$ seja encontrado integralmente, uma vez que não há sufixos que sejam prefixos de outros sufixos maiores. Seja então $s[r]$ o último símbolo encontrado no caminho C, $i+1 \leq r < n$, ou seja, o sufixo $s[i+1..n]$ tem duas partes: a primeira parte, $s[i+1..r]$, foi encontrada em C e a segunda parte, $s[r+1..n]$, não foi encontrada. Essa segunda parte deve ser inserida na árvore imediatamente após o símbolo de $s[r]$ em C. Duas situações podem ocorrer:

- a primeira situação ocorre quando, logo após $s[r]$ em C, temos um nó interno v. Neste caso o algoritmo insere uma nova aresta (v,w) em T_s^i, rotulada com a seqüência $s[r+1..n]$, e w é uma nova folha rotulada com $i+1$;
- a segunda situação ocorre quando $s[r]$ e $s[r+1]$ estão na mesma aresta de T_s^i. Neste caso um novo nó interno v é inserido exatamente entre $s[r]$ e $s[r+1]$, uma nova folha w também é criada e rotulada com $i+1$, e uma nova aresta (v,w) é criada com rótulo $s[r+1..n]$.

O algoritmo descrito acima é $O(n^2)$, uma vez que são feitas n inserções de novos sufixos, esses de tamanhos $n, n-1, \ldots, 1$; e cada inserção tem o custo do tamanho do sufixo, já que consiste no caminho que esse sufixo percorreria na árvore. Novamente, assim como no caso da árvore digital geral, estamos supondo que Σ tem tamanho fixo.

Alguns detalhes e outras aplicações

Algumas observações devem ser feitas com relação à implementação da árvore de sufixos. Com relação ao espaço utilizado, o número de folhas da árvore é fixo e igual a n, o número de sufixos de s. Como cada nó interno tem pelo menos dois filhos, o número de nós internos da árvore não pode ultrapassar $n-1$. Assim, o número total de nós é linear no tamanho de s.

Uma vez que sabemos antecipadamente que cada aresta (v,w) armazena uma subcadeia de s, e sabemos onde começa e onde termina essa subcadeia, essa informação

não precisa ser armazenada efetivamente na aresta da árvore, mas sim no nó w. Mais ainda, não precisamos armazenar a subcadeia, e sim apenas o início e o término dela, ou seja, usando apenas dois números inteiros. Mesmo considerando esses aspectos, e considerando que em geral $|\Sigma|$ pequeno comparado a n, deve-se tomar cuidado com a implementação. Um bom exemplo disso é a utilização de árvore de sufixos para armazenar uma seqüência genômica. O alfabeto Σ é muito pequeno, mas a árvore terá muitos nós, comprometendo a eficiência, em termos de espaço.

Uma opção para a economia de espaço é o uso do chamado *arranjo de sufixos*. Um **arranjo de sufixos** V para uma seqüência s é um vetor de inteiros que armazena, em ordem lexicográfica, a posição de início de todos os sufixos de s. Assim, o menor sufixo (lexicograficamente) de s, que começa na posição, digamos, j, é tal que $V[1] = j$, e assim por diante. A busca de uma cadeia p em V se dá pela busca binária de p entre os sufixos representados pelos índices armazenados em V. Dessa forma, pode-se encontrar todas as ocorrências exatas de p em s em tempo $O(m \log n)$.

Uma generalização natural do uso de árvore de sufixos é o armazenamento, na mesma árvore, dos sufixos de várias seqüências. Dessa forma a árvore pode ser usada para buscar p em um conjunto R de outras seqüências. Neste caso, cada folha armazena a informação do sufixo e também da seqüência de R que possui aquele sufixo. Uma mesma folha pode armazenar vários sufixos de distintas seqüências.

Outra importante aplicação de árvores de sufixos consiste na procura de repetições em uma mesma seqüência. A utilização da árvore de sufixos na procura de repetições reside em um fato muito importante e óbvio: o caminho da raiz até um nó interno v da árvore representa uma repetição de s. Essa repetição acontece nas posições $i_1, \ldots, i_k$, que são exatamente os rótulos das folhas da subárvore que têm v como raiz. Mais ainda, se uma cadeia α se repete em s, nas posições $i_1, \ldots, i_k$, certamente teremos k sufixos, cujas folhas correspondentes são rotuladas com esses k valores, descendentes de um nó interno v comum. Ou seja, todas as repetições de s estarão representadas em sua árvore de sufixos. A partir dessas repetições exatas encontradas em uma mesma seqüência s, é possível determinar a ocorrência de repetições aproximadas [1]. Muitas outras aplicações são possíveis e podem ser vistas em [16].

8.3.3. Algoritmos para comparação exata e aproximada baseados em programação dinâmica

O problema da comparação exata de duas seqüências é resolvido eficientemente pelo algoritmo de Knuth, Morris e Pratt (KMP), que é $O(m+n)$ para comparar seqüências de tamanhos n e m. Em aplicações como a comparação de seqüências de moléculas em biologia molecular estamos interessados no caso em que diferenças entre as seqüências devem ser admitidas, isto é, o caso em que as seqüências não precisam ser exatamente iguais para que sejam consideradas semelhantes.

Alinhamento de seqüências

O problema da comparação inexata de seqüências pode ser formalizado em termos de alinhamentos e similaridade. Dadas duas seqüências s e t sobre o mesmo alfabeto, um **ali-**

nhamento entre elas consiste na inserção de zero ou mais espaços em quaisquer posições de s e t de tal forma que s e t tenham o mesmo tamanho e tal que se $s[i]$ é um espaço então $t[i]$ não é um espaço e vice-versa. Depois do acréscimo dos espaços, já que as seqüências têm o mesmo tamanho, elas podem ser colocadas uma sobre a outra, mostrando a correspondência entre os caracteres e espaços na mesma posição. Por exemplo, sejam as seqüências

```
TAGCA
GCATCAT
```

Podemos construir os alinhamentos abaixo, representando espaços por hífens:

```
-TAGCA-        T-A-GCA        --TAG-CA-
  | ||           |               |  ||
GCATCAT        GCATCAT        GC-A-TCAT
```

Outros alinhamentos também são possíveis para essas seqüências. Como estamos interessados em avaliar o quanto as seqüências são parecidas, podemos definir uma pontuação (*score*) para o alinhamento entre duas seqüências de mesmo tamanho s e t:

$$score(s,t) = \sum_{i=1}^{|s|} c_i$$

onde

$$c_i = \begin{cases} 1 & \text{se} \quad s[i] = t[i] \\ -1 & \text{se} \quad s[i] \neq t[i] \\ -2 & \text{se} \quad s[i] = \text{`-'} \text{ ou } t[i] = \text{`-'} \end{cases}$$

Os alinhamentos no exemplo anterior têm pontuação $-3, -7$ e -9, respectivamente. Para um mesmo valor de pontuação pode ser possível obter mais de um alinhamento entre duas seqüências.

A **similaridade** entre duas seqüências é o valor de pontuação máxima obtido em um alinhamento entre elas. É possível definir outros esquemas de pontuação, envolvendo funções mais complexas inclusive. O esquema que ilustramos é bastante usado na prática para comparar seqüências de DNA, porque valoriza as colunas do alinhamento onde os símbolos coincidem e penaliza as colunas onde há discordância e inserção de espaços.

Nosso problema então é calcular a similaridade entre duas seqüências e construir um alinhamento que tenha a similaridade calculada. Uma solução possível é gerar todos os alinhamentos entre elas, calcular a pontuação e escolher o melhor. Entretanto, o número de possibilidades é exponencial, o que torna esta solução muito ineficiente.

Existe um algoritmo mais eficiente para calcular a similaridade dos alinhamentos ótimos entre duas seqüências. A idéia do algoritmo é calcular a similaridade estendendo alinhamentos entre prefixos das seqüências. Sejam s e t duas seqüências. Vamos supor que sabemos construir um alinhamento de pontuação máxima e tamanho k entre um prefixo $s[0..i]$ e um prefixo de $t[0..j]$. Seja A um desses alinhamentos como ilustrado abaixo.

colunas	1	2	3	...	k
seqüência s	□	□	□	...	□
seqüência t	□	□	□	...	□

Para obter um alinhamento de tamanho $k+1$ temos três possibilidades:

- Acrescentar $s[i+1]$ e $t[j+1]$ ao alinhamento, em uma nova coluna.

colunas	1	2	3	...	k	$k+1$
seqüência s	□	□	□	...	□	$s[i+1]$
seqüência t	□	□	□	...	□	$t[j+1]$

- Acrescentar $s[i+1]$ e um espaço ao alinhamento, em uma nova coluna.

colunas	1	2	3	...	k	$k+1$
seqüência s	□	□	□	...	□	$s[i+1]$
seqüência t	□	□	□	...	□	-

- Acrescentar um espaço e $t[j+1]$ ao alinhamento em uma nova coluna.

colunas	1	2	3	...	k	$k+1$
seqüência s	□	□	□	...	□	-
seqüência t	□	□	□	...	□	$t[j+1]$

Começando com os prefixos vazios, o algoritmo calcula os valores de similaridade entre todos os prefixos de s e todos os prefixos de t. Essa técnica chama-se programação dinâmica [12].

Seja $M[i,j]$ a célula na interseção da linha i e da coluna j de uma matriz M. Dadas as seqüências s de tamanho n e t de tamanho m, existem nm combinações de prefixos de s e t. O algoritmo constrói uma matriz $M[0..n,0..m]$, de tal forma que a célula $M[i,j]$

contenha a similaridade entre $s[1..i]$ e $t[1..j]$. Depois de preenchida, essa matriz contém a similaridade entre todos os prefixos de s e todos os prefixos de t, incluindo o prefixo vazio. A célula $M[n,m]$ contém a similaridade entre s e t. Cada célula (i, j) da matriz é preenchida com base nas células:

$M[i, j-1]$
$M[i-1, j-1]$
$M[i-1, j]$

		$\times$	$\times$	
		$\times$	i, j	

Estas células correspondem às três formas possíveis de estender um alinhamento, que descrevemos anteriormente. As primeiras linha e coluna são preenchidas com múltiplos do valor do espaço, porque correspondem a alinhar prefixos de s ou t a espaços na outra seqüência. Mais formalmente,

$$
\begin{aligned}
&M[0,0] = 0,\\
&M[i,0] = M[i-1,0] - 2,\\
&M[0,j] = M[0,j-1] - 2,
\end{aligned}
$$

$$
M[i,j] = \max \begin{cases} M[i,j-1] - 2, \\ M[i-1,j-1] + p(i,j), \\ M[i-1,j] - 2, \end{cases}
$$

onde

$$
p(i,j) = \begin{cases} 1 & \text{se} \quad s[i] = t[j], \\ -1 & \text{se} \quad s[i] \neq t[j]. \end{cases}
$$

Para exemplificar sejam $s = \texttt{ACGT}$ e $t = \texttt{AAT}$. A matriz de similaridade para s e t aparece na figura 8.16. Na figura, as setas indicam qual foi a célula a partir da qual o valor da similaridade foi obtido. O algoritmo FUNÇÃO-SIMILARIDADE preenche a matriz M.

A matriz M pode ser usada também para obter um ou mais alinhamentos ótimos entre s e t, já que o preenchimento de uma célula corresponde a uma extensão do alinhamento. Cada escolha possível, representada por uma seta na matriz, indica uma maneira de alinhar um caractere em s e outro em t:

- A seta $\leftarrow$ corresponde a um caractere de s alinhado a um espaço em t.
- A seta $\nwarrow$ corresponde a um caractere de t alinhado a um caractere de s.
- A seta $\uparrow$ corresponde a um caractere de t alinhado a um espaço em s.

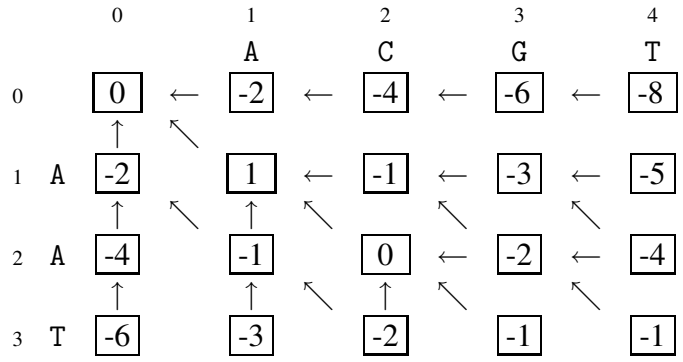

Figura 8.16: Matriz construída pelo algoritmo de programação dinâmica para a comparação inexata de seqüências. As células de fato aparecem delimitadas por quadrados. As setas indicam qual foi a célula a partir da qual o valor da similaridade foi obtido.

FUNÇÃO-SIMILARIDADE(s,t): recebe duas seqüências s e t, de comprimentos n e m, respectivamente, e devolve a similaridade máxima $M[n,m]$ entre elas.

```
 1: M[0,0]←0
 2: n←|s|
 3: m←|t|
 4: para j←1 até m faça
 5:    M[0,j]←M[0,j−1]−2
 6: para i←1 até n faça
 7:    M[i,0]←M[i−1,0]−2
 8:    para j←1 até m faça
 9:       se s[i] = t[j] então
10:          M[i,j]←M[i−1,j−1]+1
11:       senão
12:          M[i,j]←M[i−1,j−1]−1
13:       se M[i,j] < M[i−1,j]−2 então
14:          M[i,j]←M[i−1,j]−2
15:       se M[i,j] < M[i,j−1]−2 então
16:          M[i,j]←M[i,j−1]−2
17: devolva M[n,m]
```

Percorrendo a matriz de $M[n,m]$ até $M[0,0]$ podemos construir um alinhamento de pontuação máxima entre s e t. As células que estão na origem de duas setas permitem escolher caminhos diferentes e conseqüentemente construir alinhamentos diferentes. As setas não precisam necessariamente ser implementadas de forma explícita.

Este algoritmo constrói **alinhamentos globais**, isto é, as seqüências são comparadas como um todo. Este tipo de comparação é útil quando temos seqüências que diferem pouco entre si. Uma característica deste tipo de comparação é que se existem regiões das seqüências s e t de grande similaridade local, como subcadeias de s e t ou pontas de s e t, essas regiões podem ser diluídas pelo alinhamento global, como no exemplo abaixo.

```
ACTGGCG-CTAAT          ------ACTGGCGCTAAT
 ||   || ||                   ||||||
TCT-ACGACTGGC          TCTACGACTGGC------
```

Pequenas variações do algoritmo permitem outros dois tipos de comparações, a local e a semi-global.

Um **alinhamento local** entre as seqüências s e t é um alinhamento entre uma subcadeia de s e uma subcadeia de t. O algoritmo para encontrar a similaridade de um alinhamento global entre s e t pode ser modificado para encontrar um alinhamento local de similaridade máxima. A matriz será preenchida de tal forma que o alinhamento é finalizado quando o alinhamento local não pode ser estendido, caso em que o cálculo da similaridade é reiniciado.

Suponha que temos um alinhamento de tamanho k entre as seqüências s e t. Para estender esse alinhamento em busca de subcadeias parecidas, devemos poder finalizar o alinhamento entre certas subcadeias de s e t e reiniciar o cálculo da similaridade. Dadas seqüências s de tamanho n e t de tamanho m, o algoritmo constrói a matriz $M[0..n,0..m]$, onde a célula $M[i,j]$ tem a similaridade máxima entre um sufixo de $s[1..i]$ e um sufixo de $t[1...j]$, incluindo os sufixos vazios de s e t. O alinhamento com a cadeia vazia recebe pontuação igual a 0; desta forma todas as células terão valor maior ou igual a zero, porque sempre obteremos um valor de similaridade maior alinhando sufixos vazios do que se inserirmos espaços ou alinharmos letras diferentes. A matriz é preenchida assim:

$$
\begin{aligned}
&M[i,0] = 0,\\
&M[0,j] = 0,\\
&M[i,j] = \max \begin{cases} M[i,j-1] - 2, \\ M[i-1,j-1] + p(i,j), \\ M[i-1,j] - 2, \\ 0, \end{cases}
\end{aligned}
$$

onde

$$p(i,j) = \begin{cases} 1 & \text{se} \quad s[i] = t[j], \\ -1 & \text{se} \quad s[i] \neq t[j]. \end{cases}$$

Depois de preencher a matriz, o valor de similaridade máxima entre duas subseqüências de s e t estará armazenado em alguma célula $M[k,l]$. Usando a mesma técnica para encontrar alinhamentos, basta percorrer as setas a partir de $M[k,l]$ até encontrar a primeira célula com valor igual a 0 para produzir um alinhamento local de similaridade máxima.

Um **alinhamento semi-global** não penaliza espaços nas extremidades das seqüências, isto é, a similaridade pode ser calculada por uma variação do algoritmo para comparações globais em que espaços antes do primeiro e depois do último símbolo de s ou de t são ignorados. Este tipo de comparação é usado para encontrar alinhamentos em que uma seqüência está contida na outra ou entre um sufixo de uma e um prefixo da outra.

Em relação ao algoritmo original, a adaptação para alinhamentos semi-globais modifica apenas a maneira de preencher as primeiras linha e coluna e onde procurar o valor da similaridade do alinhamento. A maneira de preencher o resto da matriz permanece inalterada. Para ignorar espaços no início da primeira seqüência, a primeira linha deve ser inicializada com zeros; no início da segunda seqüência, a primeira coluna deve ser inicializada com zeros. Para ignorar espaços no fim da primeira seqüência o valor máximo de similaridade deve ser procurado na última linha; no fim da segunda seqüência, o valor deve ser procurado na última coluna. Os alinhamentos podem ser construídos da mesma maneira, tomando-se os devidos cuidados com o final das seqüências.

Os algoritmos que apresentamos nesta seção têm complexidade de tempo e espaço igual a $O(nm)$. Não se sabe exatamente qual o origem desses algoritmos; acredita-se que eles tenham sido redescobertos mais de uma vez em contextos diferentes. Existem ainda outras variações do algoritmo que têm como objetivo melhorar a complexidade de espaço e tempo, permitir outros esquemas de pontuação ou encontrar alinhamentos locais subótimos.

Subseqüência comum de comprimento máximo

Dadas duas seqüências s e t, dizemos que uma seqüência w é uma **subseqüência comum** de s e t se w é subseqüência de s e w é subseqüência de t. No problema da **subseqüência comum de comprimento máximo**, ou simplesmente problema da **maior subseqüência comum** (*longest common subsequence – LCS*), são dadas duas seqüências s e t, de comprimentos n e m respectivamente, e queremos encontrar a subseqüência comum de comprimento máximo de s e t:

> **Problema** LCS(s,t): *dadas a seqüência s de n símbolos e a seqüência t de m símbolos, encontrar uma subseqüência comum de comprimento máximo de s e t.*

O problema LCS tem a propriedade da subestrutura ótima de programação dinâmica. Para esse problema, essa propriedade é identificada da seguinte maneira. Sejam s e t seqüências de n e m símbolos, respectivamente, e seja w uma seqüência de k símbolos que é uma maior subseqüência comum de s e t. Então,

1. se $s[n] = t[m]$ então $w[k] = s[n] = t[m]$ e $w[1..k-1]$ e uma maior subseqüência comum de $s[1..n-1]$ e $t[1..m-1]$;
2. se $s[n] \neq t[m]$ então $w[k] \neq s[n]$ implica que $w[1..k]$ é uma maior subseqüência comum de $s[1..n-1]$ e $t[1..m]$;
3. se $s[n] \neq t[m]$ então $w[k] \neq t[m]$ implica que $w[1..k]$ é uma maior subseqüência comum de $s[1..n]$ e $t[1..m-1]$.

Além da propriedade da subestrutura ótima, o problema LCS também tem a propriedade da sobreposição de subproblemas. Essa propriedade permite que uma solução recursiva para o problema LCS seja obtida. Defina $c[i,j]$ o comprimento de um LCS das seqüências $s[1..i]$ e $t[1..j]$. Se $i = 0$ ou $j = 0$ então uma das seqüências tem comprimento 0 e assim o LCS tem comprimento 0. Note também que se $s[i] = t[j]$ para $i, j > 0$ então uma solução pode ser obtida recursivamente através da computação do LCS para as seqüências $s[1..i-1]$ e $t[1..j-1]$; como $s[i] = t[j]$, a solução é o comprimento do LCS para as $s[1..i-1]$ e $t[1..j-1]$, mais uma unidade relativa à igualdade entre os símbolos $s[i]$ e $t[j]$. Por fim, se $s[i] \neq t[j]$ para $i, j > 0$ então uma solução pode ser obtida através da computação do LCS para as seqüências $s[1..i]$ e $t[1..j-1]$ ou $s[1..i-1]$ e $t[1..j]$; o LCS entre $s[1..i]$ e $t[1..j]$ será o maior LCS entre esses dois. Essa fórmula pode então ser escrita da seguinte forma:

$$c[i,j] = \begin{cases} 0, & \text{se } i = 0 \text{ ou } j = 0, \\ c[i-1, j-1] + 1, & \text{se } i, j > 0 \text{ e } s[i] = t[j], \\ \max(c[i, j-1], c[i-1, j]), & \text{se } i, j > 0 \text{ e } s[i] \neq t[j]. \end{cases}$$

A computação da matriz de programação dinâmica c é descrita pelo algoritmo LCS. Essa matriz armazena em cada posição i, j, com $1 \leq i \leq n$ e $1 \leq j \leq m$, o valor de um LCS para as seqüências $s[1..i]$ e $t[1..j]$. Uma matriz auxiliar b é inserida nessa computação para que a maior subseqüência comum possa ser facilmente recuperada.

A figura 8.17 mostra um exemplo de execução do algoritmo LCS.

Observe que o tempo de execução do algoritmo LCS é $O(mn)$ e o espaço utilizado é $O(mn)$. Masek e Paterson [23] propuseram um algoritmo de complexidade assintótica $O(mn/\log n)$ para o problema LCS, onde as seqüências são obtidas de um alfabeto de tamanho limitado. Quando um símbolo do alfabeto não pode aparecer mais que uma vez nas seqüências de entrada, o problema pode ser solucionado por um algoritmo devido à Szymanski [37], com tempo de execução $O((m+n)\log(m+n))$. De qualquer forma, o espaço utilizado também pode ser reduzido para $O(m+n)$.

Para construir um LCS para duas seqüências de entrada s e t a partir da matriz b devolvida pelo algoritmo LCS basta tomar $b[n,m]$ e percorrê-la através de suas flechas.

LCS(s,t): recebe as seqüências s e t com n e m símbolos, respectivamente, e devolve a matriz de programação dinâmica c, contendo os valores dos LCS para subseqüências de s e t, e b, contendo indicadores para a recuperação do LCS.

```
1:  para i ← 1 até n faça
2:      c[i,0] ← 0
3:  para j ← 1 até m faça
4:      c[0,j] ← 0
5:  para i ← 1 até n faça
6:      para j ← 1 até m faça
7:          se s[i] = t[j] então
8:              c[i,j] ← c[i−1,j−1] + 1
9:              b[i,j] ← " ↖ "
10:         senão
11:             se c[i−1,j] ≥ c[i,j−1] então
12:                 c[i,j] ← c[i−1,j]
13:                 b[i,j] ← " ↑ "
14:             senão
15:                 c[i,j] ← c[i,j−1]
16:                 b[i,j] ← " ← "
17: devolva c e b
```

Note que se $b[i,j] =$ " $\nwarrow$ " para algum i,j, temos que $s[i] = t[j]$ é um símbolo de um LCS. Então, o algoritmo IMPRIME-LCS utiliza essa informação para construir a seqüência apropriada.

IMPRIME-LCS(b,s,i,j): recebe a matriz b devolvida pelo algoritmo LCS, a seqüência s com n símbolos e índices i e j para linha e coluna da matriz b e devolve um LCS de s e t.

```
1: se i > 0 e j > 0 então
2:     se b[i,j] = " ↖ " então
3:         IMPRIME-LCS(b,s,i−1,j−1)
4:         escreva s[i]
5:     senão
6:         se b[i,j] = " ↑ " então
7:             IMPRIME-LCS(b,s,i−1,j)
8:         senão
9:             IMPRIME-LCS(b,s,i,j−1)
```

A execução do algoritmo IMPRIME-LCS sobre as entradas s e b da figura 8.17 fornece como saída a seqüência ATGA, que é a maior subseqüência comum de $s =$ ACTTGA e $t =$ ATCGA. O tempo de execução desse algoritmo é $O(m+n)$, já que a cada chamada recursiva ou i ou j é decrementado.

Uma extensão natural do problema é considerarmos o alinhamento de mais de duas seqüências. Este tipo de alinhamento, o alinhamento múltiplo de seqüências, é bastante usado para comparar proteínas. Para resolver esse problema, podemos pensar em

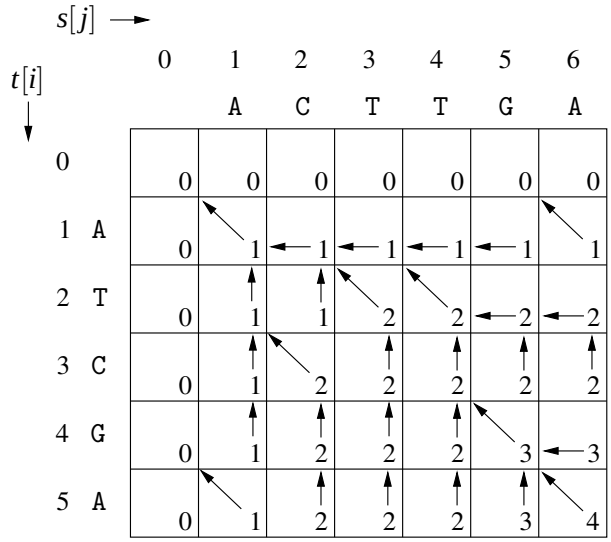

Figura 8.17: Um exemplo de execução do algoritmo LCS para as seqüências s =ACTTGA e t =ATCGA. As tabelas c e b são mostradas em conjunto. O comprimento da maior subseqüência comum de s e t é $\mathbf{c}[5,6] = 4$.

usar uma variação do algoritmo de programação dinâmica para alinhar k seqüências. Para tanto, é necessário definir um esquema de pontuação para a similaridade e preencher uma matriz com k dimensões. O problema desta abordagem é a complexidade: supondo que as k seqüências têm tamanho n, preencher a matriz requer visitar n^k células, sendo que cada uma depende de $2^k - 1$ outras células, além do custo de calcular a similaridade. Este algoritmo é exponencial e dificilmente poderá ser usado na prática. Além disso, o problema do alinhamento múltiplo de seqüências é NP-completo. Na prática este problema é resolvido abrindo-se mão da garantia de uma solução ótima em troca de aproximações aceitáveis produzidas mais rapidamente.

8.3.4. Heurísticas para comparação aproximada

Nos últimos anos, as seqüências de bases do DNA e as seqüências de aminoácidos de vários organismos vêm sendo determinadas e armazenadas em bancos de seqüências biológicas. Um banco de seqüências deve ser entendido neste contexto como um conjunto de seqüências armazenadas de alguma forma, como por exemplo um arquivo texto. Bancos de seqüências como o GenBank [15] e o Swissprot [35] contêm milhões de seqüências que representam moléculas de DNA e proteínas. Além de grandes, estes bancos de dados crescem rapidamente.

A semelhança entre seqüências biológicas, como já mencionamos, é uma indicação de funcionalidade comum. Uma questão que deve ser respondida então é, dada uma seqüência s de interesse e um banco de seqüências B, quais são as seqüências de B com as quais s se parece. As comparações locais são as mais interessantes para buscas em bancos de seqüências porque partes de moléculas que são semelhantes podem ser encontradas.

Se quisermos comparar uma seqüência s que nos interessa com as seqüências em um banco de seqüências grande, o algoritmo de programação dinâmica para comparação de seqüências que mostramos na seção anterior não é suficientemente eficiente. Esta

questão do desempenho motivou o desenvolvimento de heurísticas para realizar buscas em bancos de seqüências. Estas heurísticas abrem mão da exatidão (isto é, elas podem não encontrar os melhores alinhamentos entre s e as seqüências no banco ou podem não reportar a existência de similaridade entre s e alguma seqüência no banco) em favor do desempenho.

Nesta seção apresentamos duas heurísticas para a busca em bancos de seqüências que têm bom desempenho e sensibilidade: FASTA e BLAST. Estas duas heurísticas são membros de famílias de heurísticas que vêm sendo aprimoradas há alguns anos e são usadas intensivamente por biológos moleculares e bioinformatas.

Estas heurísticas podem ser consideradas otimizações do algoritmo exato baseado em programação dinâmica onde a matriz é preenchida apenas parcialmente, em torno de alinhamentos pequenos encontrados rapidamente.

Para calcular a similaridade entre seqüências de aminoácidos, o FASTA e o BLAST usam uma matriz S chamada **matriz de substituições**, que especifica a pontuação $S[i, j]$ do alinhamento entre todo par de aminoácidos i e j. Essas matrizes são tais que identidades e substituições prováveis têm pontuação positiva e substituições improváveis têm pontuação negativa.

FASTA

A primeira versão desta heurística para fazer buscas em bancos de seqüências de aminoácidos foi criada por Lipman e Pearson e se chama FASTP [22]. Ela foi aprimorada pelos mesmos autores e chamada de FASTA [26, 27]. O algoritmo FASTA compara uma seqüência s contra cada seqüência do banco.

O FASTA realiza os seguintes passos basicamente:

1. Identificação de alinhamentos exatos e regiões;
2. Pontuação das regiões;
3. Pontuação de união de regiões;
4. Produção de um alinhamento.

Identificação de alinhamentos exatos e regiões O primeiro passo do FASTA é identificar alinhamentos exatos e sem inserção de espaços entre subseqüências de tamanho k, **k-tuplas**, comuns à seqüência consultada s e à seqüência t no banco. Por exemplo, supondo que s e t sejam seqüências de proteínas, e que $k = 3$, o FASTA constrói um vetor A de tamanho $20^3 = 8000$, isto é, uma posição para cada cadeia que pode ser formada por 3 letras dentre as 20 que representam cada um dos aminoácidos existentes. O algoritmo armazena em A as posições de todas as triplas de s. Varrendo a seqüência t, o algoritmo identifica as posições em s e t onde há coincidência entre triplas.

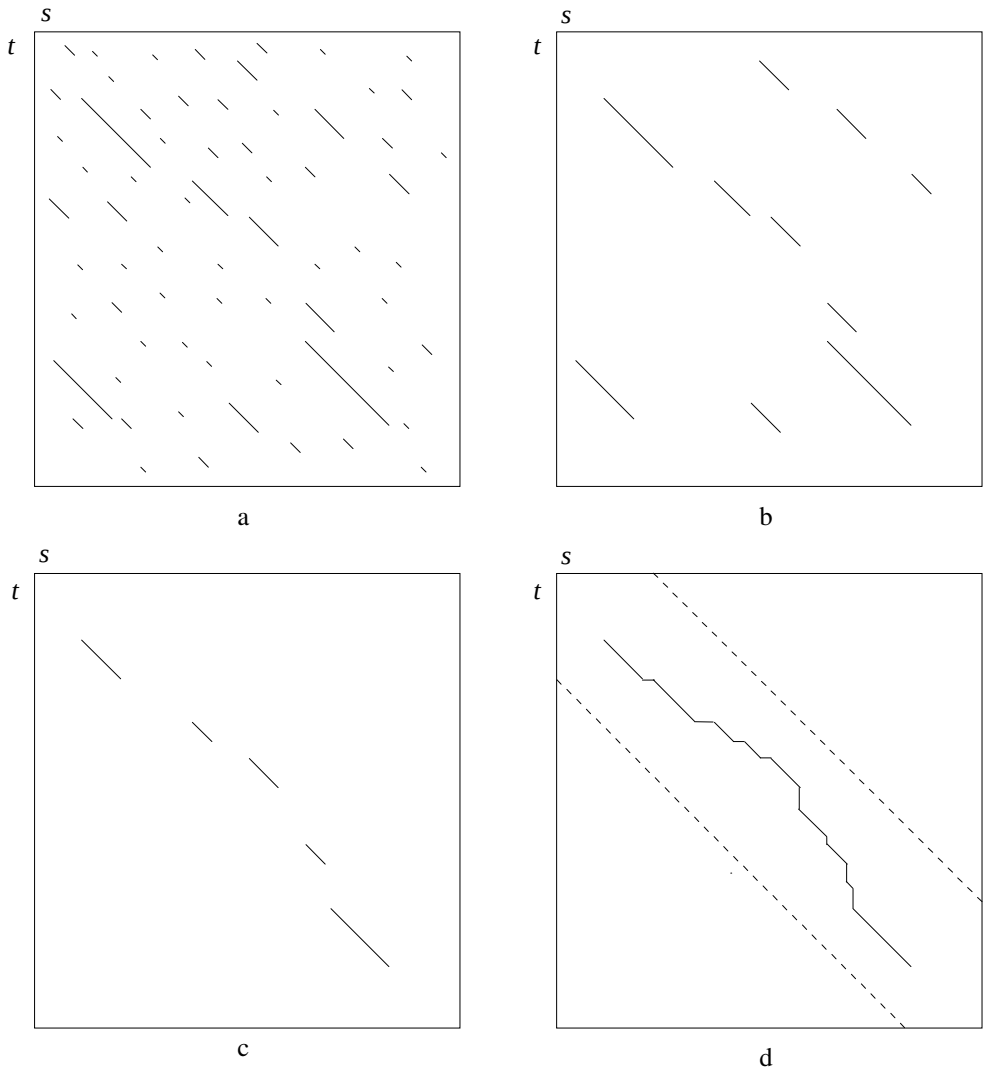

Figura 8.18: Passos do algoritmo FASTA. (a) Identificação de alinhamentos exatos de tamanho k e (b) seleção das dez melhores regiões. (c) Cálculo da pontuação e seleção de sub-regiões (d) Produção de um alinhamento restrito entre s e t.

Suponha que duas k-tuplas em s, s' e s'', são iguais a duas k-tuplas em t, t' e t''. Se a diferença entre a posição inicial de s' e t' é igual à diferença entre a posição inicial de s'' e t'', então os alinhamentos $s't'$ e $s''t''$ estão na mesma diagonal da matriz.

Depois de identificar k-tuplas comuns, o programa constrói **regiões**. Uma região é formada por uma ou mais k-tuplas na mesma diagonal que não estejam muito distantes, gerando uma única região alinhada sem espaços. Embora o FASTA não preencha uma matriz de programação dinâmica, podemos ilustrar um resultado possível da aplicação deste passo como na Figura 8.18a. Em seguida, as dez regiões de maior pontuação são escolhidas (Figura 8.18b).

Pontuação das regiões No segundo passo do FASTA, a pontuação das dez melhores regiões é recalculada usando um matriz de substituições. Cada região é podada, conservando apenas sua sub-região de pontuação maximal. Apenas regiões com pontuação acima de um certo limiar são consideradas nos próximos passos do algoritmo (Figura 8.18c). Este passo também considera apenas alinhamentos sem espaços. O programa registra a região de melhor pontuação entre s e cada seqüência t_i no banco.

Pontuação de união de regiões No terceiro passo, as sub-regiões para produzir um alinhamento com espaços, levando em conta a posição das sub-regiões, sua pontuação e uma função de penalidade para uní-las. Esta função de penalidade corresponde à inserção de espaços no alinhamento (Figura 8.18c). O programa registra a pontuação para este novo alinhamento entre s e cada seqüência t_i no banco.

Produção de um alinhamento Depois de computar a pontuação no terceiro passo para todas as seqüências no banco, as de melhor pontuação são consideradas no quarto passo do algoritmo. Neste passo é produzido um alinhamento entre s e uma seqüência t_i usando uma variação do algoritmo de programação dinâmica restrito a uma região em torno do alinhamento obtido no terceiro passo (Figura 8.18d).

O FASTA foi projetado para buscas em bancos de seqüências e constrói apenas um alinhamento entre a seqüência que está sendo pesquisada e as seqüências do banco. Outros programas da família FAST [14] encontram alinhamentos entre seqüências de DNA e alinhamentos locais entre pares de seqüências.

Embora tenha sido projetado para comparações entre seqüências de DNA e proteínas, o método de comparação do FASTA pode ser aplicado a qualquer alfabeto com esquemas de pontuação arbitrários.

BLAST

Nesta seção apresentamos a heurística para comparações locais chamada **BLAST** (*Basic Local Aligment Search Tool*). A primeira versão [2], publicada em 1990, não produz alinhamentos com espaços. A segunda versão [3], de 1997, produz alinhamentos com espaços é a que descrevemos a seguir.

A idéia central do algoritmo BLAST é que um alinhamento entre duas seqüências que seja estatisticamente significativo, isto é, não tenha alta probabilidade de ser aleatório, provavelmente contém pequenas cadeias alinhadas nas duas seqüências com alta similaridade. Estas cadeias pequenas são chamadas de **palavras**; no caso de aminoácidos normalmente são de tamanho 3. Dadas duas seqüências para comparar, o algoritmo faz o seguinte:

1. Seleciona, em cada seqüência, um conjunto de palavras que quando alinhadas com alguma outra podem pontuar acima de um certo limiar T;
2. Busca alinhamentos entre palavras selecionadas no Passo 1 que satisfaçam certas condições;
3. Estende os alinhamentos encontrados no Passo 2.

A seguir descrevemos estes passos em mais detalhes. Ao descrever o algoritmo vamos considerar que estamos comparando seqüências de aminoácidos s de tamanho n e t de tamanho m. As mesmas idéias são aplicadas a seqüências de DNA, com otimizações diferentes.

Compilação da lista de palavras A primeira tarefa é percorrer as seqüências e encontrar todas as palavras de tamanho w que podem produzir pelo menos um alinhamento com pontuação acima de um certo limiar T.

Busca de *hits* Duas palavras, uma de s e uma de t que se alinham com pontuação acima do limiar T são chamadas de ***hit***. O algoritmo procura pares de *hits* que:

- Não se sobrepõem;
- Estão na mesma diagonal. A diagonal de um *hit* envolvendo palavras que começam nas posições i e j respectivamente é definida como $i - j$;
- Estão no máximo à distância A entre si. A distância entre dois *hits* na mesma diagonal é a diferença entre suas coordenadas iniciais.

Extensão dos alinhamentos A extensão do alinhamento entre um par de *hits* selecionado previamente é feita em duas fases. Primeiramente, o segundo *hit* é estendido nas duas direções sem adicionar espaços. O *hit* deixa de ser estendido em uma direção quando a pontuação sofre uma variação negativa grande em relação à máxima pontuação alcançada por extensões menores. Estender o alinhamento sem adicionar espaços é mais rápido que estendê-lo adicionando espaços. Se o alinhamento α obtido com esta extensão tiver pontuação acima de um certo limiar S_g, então o *hit* passa para a segunda fase, em que é alinhado com espaços.

Para a extensão com espaços, inicialmente o algoritmo escolhe a **semente** do alinhamento. Do alinhamento α produzido na fase anterior é selecionada a seqüência de onze colunas consecutivas com maior pontuação (ou todo o alinhamento se ele tem menos que onze colunas) e sua coluna central é usada como semente.

A seguir o BLAST usa uma variação do algoritmo de programação dinâmica para comparação de seqüências em que a matriz é preenchida a partir da semente, estendendo o alinhamento em direção às duas extremidades das seqüências. Além disso, o BLAST preenche apenas as células em que a pontuação do alinhamento não diminui mais que X_g em relação à melhor pontuação obtida. Em outras palavras, o alinhamento deixa de ser estendido se forem inseridos muitos espaços ou caracteres discordantes. Na maioria dos casos essa variação faz com que apenas uma parte da matriz seja preenchida.

Os alinhamentos com espaços que tenham significância estatística aceitável são reportados pelo algoritmo, com a pontuação normalizada, que permite comparar alinhamentos produzidos com esquemas de pontuação diferentes, e a significância do alinhamento (*E-value*), que indica a probabilidade de que o alinhamento ocorra aleatoriamente entre seqüências daquele tamanho. O BLAST reporta não apenas o alinhamento local máximo, mas também outros alinhamentos com boa pontuação que não pode ser aumentada pela extensão ou redução do alinhamento. Tais alinhamentos locais são chamados de HSPs (*high scoring segment pairs*).

A probabilidade de que o BLAST perca alinhamentos existe e diminui rapidamente à medida que a pontuação do alinhamento aumenta. Para $T = 11$, a probabilidade de perder um alinhamento de pontuação normalizada 40 é menor que 4%. Para pontuação 45, é de aproximadamente 2%. Na maioria dos casos, no entanto, o desempenho da heurística compensa a pequena probabilidade de que algum alinhamento seja perdido.

A complexidade de pior caso do algoritmo é $O(nm)$ para comparar duas seqüências de tamanhos n e m, a mesma do algoritmo de programação dinâmica. Na prática, porém, o desempenho do BLAST fica bastante aquém deste limite.

As implementações do BLAST estão entre os programas mais usados pelos cientistas envolvidos com genômica. Exemplos de implementações disponíveis gratuitamente são `wublast` [40] e `blastall` [7]. Vários sites na Internet oferecem serviços de busca em bancos de dados de seqüências usando o BLAST. Há até a comercialização de hardware e software preparados para executar o BLAST massivamente [25].

Para dar uma idéia do desempenho desta heurística, usamos uma implementação do algoritmo de programação dinâmica e uma implementação do BLAST para comparar uma seqüência de aminoácidos contra 2.831 outras do mesmo tipo. Não nos preocupamos em analisar os alinhamentos produzidos pelos dois programas. O algoritmo exato levou 4,19 segundos. O BLAST levou 0,17 segundo, cerca de 20 vezes menos. No caso de DNA, por conta de certas otimizações, a diferença pode ser ainda maior. Como o BLAST é um algoritmo projetado principalmente para buscas em bancos de seqüências, normalmente o banco é pré-processado e todas as palavras que podem pontuar acima de T em todas as seqüências do banco são indexadas. Subseqüentemente, quando uma seqüência s vai ser comparada com o banco, as palavras de s que podem atingir pontuação pelo

menos T e que existem nas seqüências do banco são encontradas mais rapidamente. A indexação é necessária apenas uma vez. O pré-processamento das 2.831 seqüências levou 0,27 segundo[2].

8.4. Métodos para comparação de textos

Até este ponto tratamos da comparação entre duas seqüências e da comparação entre uma seqüência e todas as seqüências em um conjunto. Nesta seção vamos considerar o problema da comparação entre dois textos.

Vamos supor que temos um conjunto de textos $T = \{t_1, t_2, \ldots, t_n\}$. As seguintes questões podem ser feitas em relação a T:

- Dado um texto x, há algum texto em T com o qual x tem similaridade, isto é, trata do mesmo assunto ou pertence ao mesmo domínio de conhecimento?
- Dado um texto x, qual a similaridade de x com cada um dos textos de T?
- Como os textos em T se relacionam entre si, isto é, quais são similares ou dissimilares?

Apesar de podermos considerar um texto como uma seqüência, se usarmos um algoritmo de comparação entre dois textos, o resultado provavelmente não será muito satisfatório. Considere por exemplo, o algoritmo de programação dinâmica da seção 8.3.3. Aquele algoritmo será capaz de construir um alinhamento entre os dois textos, mas o valor de similaridade entre eles pode ser muito baixo mesmo que os textos sejam sobre o mesmo assunto, bastando que a ordem relativa das palavras seja moderadamente diferente. Por exemplo, o alinhamento global produzido pelo algoritmo de programação dinâmica entre as frases "enquanto eu trabalho, outros descansam" e "outros descansam enquanto eu trabalho" é

```
enquanto -e-u trabalho, outros descansam
   |     | |      |              |     |
o--utros descansam enquanto eu trabalho-
```

O melhor alinhamento local entre essas mesmas seqüências será

```
enquanto eu trabalho
||||||||||||||||||||
enquanto eu trabalho
```

[2]Estas comparações foram realizadas usando um Pentium IV a 1,7 GHZ com 512 MB de RAM rodando Linux 2.4.18-27.7. A implementação do algoritmo exato é o SWAT versão 0.990319 (`www.phrap.org`) e a do BLAST é o `blastall` versão Aug-28-2002 (`www.ncbi.nlm.nih.gov`). As seqüências selecionadas são as traduções dos 2.831 genes da *Xylella fastidiosa* e o gene XF0001 dessa mesma bactéria (`www.lbi.ic.unicamp.br`). A matriz de substituições BLOSUM62 foi usada pelos dois programas.

mas as seqüências têm ainda mais semelhança. Não é difícil perceber que se os textos são grandes, fica ainda mais difícil contornar essas limitações.

Duas maneiras de tratar os problemas listados acima são representando cada texto do conjunto como um vetor de termos e através da complexidade de Kolmogorov, uma medida da quantidade de informação em cada texto. Nas seções a seguir consideramos brevemente essas duas abordagens. Os vetores de termos são discutidos em [10], [32] e [34]. A complexidade de Kolmogorov é abordada intensivamente em [21].

Vetor de termos

Usando esta técnica, cada texto do conjunto T é representado como um vetor multidimensional. Cada dimensão dos vetores representa um termo. Um **termo** é uma palavra do texto. A escolha dos termos para a construção dos vetores pode ser feita a partir de um certo vocabulário curado ou podemos tomar como vocabulário o conjunto de termos que aparecem em algum texto em T. Normalmente artigos, preposições, conjunções e outras palavras pouco discriminativas não são consideradas e os termos são reduzidos a seus radicais para diminuir a influência de tempos verbais e flexões.

Seja $\mathscr{V}$ o vocabulário escolhido e V_{t_i} o vetor de termos para o texto t_i. Cada elemento $V_{t_i}[x]$ deste vetor contém o número de vezes que o termo x aparece no texto t_i. O valor de $V_{t_i}[x]$ pode ser normalizado em função do tamanho dos documentos.

Normalmente multiplica-se o valor de cada posição do vetor por um fator que representa a importância do termo no conjunto. Um indicador possível é o inverso da freqüência com que o termo aparece nos documentos em T, ou seja, se um termo aparece muitas vezes nos textos, sua importância para caracterizar cada texto t_i é menor.

Depois que os vetores foram construídos, a similaridade entre dois textos pode ser computada como um produto entre os vetores

$$sim(t_i, t_j) = \sum_{\forall x \in \mathscr{V}} V_{t_i}[x] \times V_{t_j}[x].$$

No exemplo que demos no início desta seção, os vetores de termos seriam iguais.

A representação vetorial dos textos é bastante utilizada [31] e [6], mas para grandes conjuntos de texto há alguns problemas. Em primeiro lugar, quando a dimensão dos vetores é muito grande em geral é necessário selecionar um conjunto de termos relevantes para que o método seja capaz de discriminar os textos, e este é um processo *ad-hoc*. Pode-se ainda aplicar alguma técnica para reduzir a dimensionalidade dos vetores. Em segundo lugar, o fato de que quando o conjunto de textos se altera, o conjunto de termos pode se modificar o suficiente para que seja necessário recomputar os vetores de todo o conjunto.

Exemplos de aplicações que usam esta representação aparecem em [31] e [6].

Quantidade de informação

Outra forma de comparar textos é com base na complexidade de Kolmogorov. Intuitivamente, a complexidade de Kolmogorov é a quantidade de informação que uma cadeia contém. Aqui apresentaremos apenas a definição e algumas relações importantes para a aplicação à comparação de textos.

Dados dois textos s e t quaisquer, a **complexidade condicional de Kolmogorov** de s dado o texto t, $K(s|t)$, é o tamanho do menor programa que, tendo t como entrada, gera s. Intuitivamente, a complexidade condicional de Kolmogorov é a quantidade de informação que a cadeia t não tem a respeito de s. Quando mais informação t possui a respeito de s, menos complicada é a geração de s a partir de t e, portanto, menor é a complexidade condicional de Kolmogorov, $K(s|t)$. Observe que s não é dado como entrada para o programa.

A complexidade de Kolmogorov de uma cadeia s é $K(s|\varepsilon) = K(s)$, isto é, o tamanho do menor programa que gera s tendo a cadeia vazia como entrada.

Dadas duas cadeias s e t, é possível calcular

$$d(s,t) = 1 - \frac{K(s) - K(s|t)}{K(st)},$$

que é uma métrica entre s e t a menos de um fator logarítmico aditivo, onde st é a concatenação de s e t. Note que $d(s,t) = 0$ quando $s = t$ e $d(s,t)$ tende a 1 quanto mais distintos são s e t [19].

Aplicando essa métrica, teoricamente poderíamos calcular o quanto dois textos estão relacionados. No entanto, a complexidade de Kolmogorov não é computável, isto é, não existe um algoritmo que calcule $K(s|t)$. Mas é possível aproximar seu valor comprimindo o texto. Usando um compactador que tem certas características, como `gzip` [17], `bzip2` [9] ou `PPMZ` [30], podemos calcular o que é conhecido como *normalized information distance*,

$$NCD(s,t) = \frac{C(st) - \min\{C(s), C(t)\}}{\max\{C(s), C(t)\}}$$

onde $C(\cdot)$ é o tamanho da versão compactada de um texto. Essa relação também é uma métrica entre s e t a menos de um fator logarítmico aditivo [20].

Portanto, é possível estabelecer uma métrica de relacionamento entre dois textos com base na informação contida neles. Exemplos de aplicação desta técnica aparecem em [11], [19] e [20].

8.5. Considerações finais

O problema de comparação de seqüências tem atraído uma maior atenção, em função do aumento de aplicações relacionadas a duas áreas distintas. A primeira delas relaciona-se

à grande quantidade de informação disponível na *web* e à forma como essa informação pode ser recuperada eficientemente. A segunda relaciona-se aos avanços da biologia molecular, em particular no desenvolvimento de técnicas de seqüenciamento de DNA. Essas áreas proporcionam inúmeras aplicações cujo principal problema remete ao tratamento de cadeias de símbolos, em especial à comparação de seqüências.

Neste texto apresentamos alguns algoritmos e heurísticas para os problemas de casamento exato e casamento aproximado de seqüências, abordando suas complexidades e algumas aplicações. O objetivo não foi o de ser uma coletânea de técnicas de comparação de cadeias, mas sim o de servir com um texto inicial, onde o leitor pode ter uma primeira visão detalhada dos principais métodos mais utilizados.

Alguns tópicos não foram tratados neste texto. A comparação de textos comprimidos é um tópico de interesse por conta do volume de dados existente. Baeza-Yates e Ribeiro-Neto [6] e Ziviani [41] abordam este problema. Algoritmos paralelos para problemas de comparação de seqüências têm sido desenvolvidos [5, 4]. Outros algoritmos para comparação exata de seqüências, ou mesmo extensões daqueles que apresentamos aqui, foram deixados de fora deste texto. Os textos de Pearson e Miller [28], Gusfield [16], Setubal e Meidanis [33], Brito [13] e Pevzner [29] contém discussões detalhadas sobre o problema da comparação de seqüências, incluindo outros algoritmos e abordagens para resolvê-lo.

Referências

[1] N.F. Almeida. *Tools for genome comparison*. PhD thesis, Institute of Computing, University of Campinas, May 2002. In Portuguese.

[2] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[3] S.F. Altschul, T.L. Madden, A.A. Schäffer, et al. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[4] C.E.R. Alves, E.N. Cáceres, and S.W. Song. Computing maximum sequence in parallel. In *Proceedings of the II Brazillian Workshop on Bioinformatics*, pages 80–87, Macaé-Brazil, 2003.

[5] C.E.R. Alves, E.N. Cáceres, and S.W. Song. An all-substrings common subsequence algorithm. In *Proceedings of the 2nd Brazillian Symposium on Graphs, Algorithms and Combinatorics (GRACO 2005)*, pages 110–116, Angra dos Reis-Brazil, 2005.

[6] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[7] Blastall. `www.ncbi.nlm.nih.gov/BLAST`.

[8] R.S. Boyer and J.S. Moore. A fast string-searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[9] Bzip2. www.bzip2.org.

[10] S. Chakrabarti. *Mining the web: discovering knowledge from hypertext data*. Morgan Kaufmann Publishers, 2003.

[11] R. Cilibrasi and P.M.B. Vitányi. Clustering by compression. *IEEE Trans. Information Theory*, 51(4):1546–1555, 2005.

[12] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[13] R.T. de Brito. Alinhamento de seqüências biológicas. Master's thesis, Instituto de Matemática e Estatística, Universidade de São Paulo, 2003.

[14] Fasta family of programs. fasta.bioch.virginia.edu/.

[15] Genbank. www.ncbi.nlm.nih.gov/GenBank.

[16] D. Gusfield. *Algorithms on Strings, Trees, and Sequences. Computer Science and Computational Biology*. Cambridge University Press, 1997.

[17] Gzip. www.gzip.org.

[18] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[19] M. Li, J. H. Badger, X. Chen, S. Kwong, P. Kearney, and H. Zhang. An information-based sequence distance and its application to whole mitochondrial genome phylogeny. *Bioinformatics*, 17(2):149–154, 2001.

[20] M. Li, X. Chen, X. Li, B. Ma, and P. Vitanyi. The similarity metric. *IEEE Transactions on Information Theory*, 50(12):3250–3264, 2004.

[21] M. Li and P. Vitányi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Verlag, 2nd edition, 1997.

[22] D.J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity search. *Science*, 227:1435–1441, 1985.

[23] W.J. Masek and M.S. Paterson. A faster algorithm computing string edit distances. *Journal of Computer and System Sciences*, 20(1):18–31, 1980.

[24] E.M. McCreight. A space-economical suffix tree construction algorithm. *J.ACM*, 23:262–272, 1976.

[25] Paracel. www.paracel.com.

[26] W. R. Pearson. Rapid and sensitive sequence comparisions with FASTP and FAST. *Methods in Enzymology*, 183:63–98, 1985.

[27] W.R. Pearson and D.J. Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85:2444–2448, 1988.

[28] W.R. Pearson and W. Miller. Dynamic programming algorithms for biological sequence comparison. In Ludwig Brand and Michael L. Johnson, editors, *Numerical Computer Methods*, volume 210, pages 575–601. Academic Press, 1992.

[29] P.A. Pevzner. *Computational Molecular Biology: An algorithmic approach*. MIT Press, 2000.

[30] Ppmz. `www.cbloom.com/src/ppmz.html`.

[31] M. Rasmussen and G. Karypis. gCLUTO - an interactive clustering, visualization, and analysis system. Technical Report CSE/UMN TR 04-021, Univ. of Minnesota, Dep. of Computer Science and Engineering, 2004.

[32] G. Salton. Developments in automatic text retrieval. *Science*, 253:974–980, 1991.

[33] J.C. Setubal and J. Meidanis. *Introduction to computational molecular biology*. PWS Publishing Co., 1997.

[34] A. Singhal and G. Salton. Automatic text browsing using vector space model. In *Proceedings of the Dual-Use Technologies and Applications Conference*, pages 318–324, 1995.

[35] Swissprot. `www.ebi.ac.uk/swissprot/`.

[36] J.L. Szwarcfiter and L. Markenzon. *Estruturas de Dados e seus Algoritmos*. LTC, 1994.

[37] T.G. Szymanski. A special case of the maximal common subsequence problem. Technical Report TR-170, Computer Science Laboratory, Princeton University, 1975.

[38] E. Ukkonen. On-line costruction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[39] P. Weiner. Linear pattern matching algorithms. In *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*, pages 1–11, 1973.

[40] Wublast. `blast.wustl.edu`.

[41] N. Ziviani. *Projeto de algoritmos*. Thomson, 2 edition, 2004.

Guilherme Pimentel Telles, 32, é doutor em Ciência da Computação pela Unicamp. Participou do primeiro projeto genoma brasileiro, o projeto de seqüenciamento genético da bactéria *Xylella fastidiosa*, financiado pela Fapesp, pelo que recebeu a distinção Honra ao Mérito Científico e Tecnológico do Governo do Estado de São Paulo. Participou também do projeto genoma da cana-de-açúcar. Atualmente é professor do Departamento de Ciência da Computação e Estatística do ICMC da USP. Suas áreas de interesse são complexidade de algoritmos e estruturas de dados, grafos e bioinformática.

Nalvo Franco de Almeida Junior, 40, é doutor em Ciência da Computação pela Unicamp e pesquisador nível 2 do CNPq. Sua tese de doutorado, que propõe ferramentas para a comparação de genomas, recebeu prêmio de 3o. lugar no CTD'2003. Participou dos seguintes projetos genoma: *Xylella fastidiosa* (onde recebeu a distinção Honra ao Mérito Científico e Tecnológico do Governo do Estado de São Paulo), *Xanthomonas citri*, *Xanthomonas campestris* e *Xylella fastidiosa Pierce's Disease*, financiados pela Fapesp; *Agrobacterium tumefaciens* (artigo de capa da revista Science, Vol. 294,#5550), financiado pela Dupont e pela University of Washington (Seattle, USA); *Paracoccidioides brasilienses* e *Anaplasma marginale*, financiados pelo CNPq. É professor do Departamento de Computação e Estatística da UFMS desde 1987. Suas áreas de interesse são Biologia Computacional, Complexidade de Algoritmos e Estruturas de Dados.

Fábio Henrique Viduani Martinez, 34, é graduado em Ciência da Computação pela Universidade Federal de Mato Grosso do Sul. É mestre em Matemática Aplicada pelo Instituto de Matemática e Estatística da Universidade de São Paulo, quando estudou algoritmos paralelos para construção da árvore dos sufixos. É doutor em Ciência da Computação pelo Instituto de Matemática e Estatística da Universidade de São Paulo, quando estudou algoritmos de aproximação para restrições do problema de Steiner em grafos. Desde de 1996 é professor do departamento de Computação e Estatística da Universidade Federal de Mato Grosso do Sul. Suas áreas de interesse são Algoritmos de Aproximação, Teoria dos Grafos e Otimização Combinatória