Filtros para a busca e extração de padrões aproximados em cadeias biológicas

Domingos Soares Neto

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
Obtenção do Título
de
Mestre em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. José Augusto Ramos Soares

— São Paulo, 10 de Setembro de 2008 —

– Durante a realização desse trabalho, o autor recebeu apoio financeiro do Conselho Nacional de Pesquisa (CNPq). –

Filtros para a busca e extração de padrões aproximados em cadeias biológicas

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por Domingos Soares Neto
e aprovada pela Comissão Julgadora.

Banca Examinadora

Membros titulares:

Prof. Dr. José Augusto Ramos Soares (orientador) - IME-USP;

Prof. Dr. José Coelho de Pina Junior - IME-USP;

Prof. Dr. Fábio Henrique Viduani Martinez - UFMS.

Membros suplentes:

Prof. Dr. João Meidanis - IC-UNICAMP;

Profa. Dra. Yoshiko Wakabayashi - IME-USP.

Agradecimentos

Uma dissertação de mestrado é um trabalho extenuante, não apenas para o pesquisador, mas também para todos aqueles com os quais ele convive durante a elaboração do trabalho e que colaboram, direta ou indiretamente, para o sucesso desta empreitada. Essas pessoas, no meu caso, formam um grupo volumoso. Logo, precisarei de muitas linhas para prestar os devidos agradecimentos, mas faço questão, na medida que a minha péssima memória permitir, de deixar cada um deles aqui registrado. Exceção feita aos agradecimentos prestados à minha filha — sem você não encontraria forças sequer para começar a trilhar esse caminho —, a ordem dos mesmos não representa sua importância: se a arquitetura caneta-papel me permitisse, agradeceria a todos em paralelo.

À Maria Fernanda, minha filha: palavras são insuficientes para lhe agradecer de acordo. Espero ser capaz de demonstrar o meu infinito amor e gratidão a cada instante do tempo em que eu estiver aqui presente. Sua chegada me fez enxergar o que era realmente importante nesse mundo e me deu coragem para seguir em frente quando parecia não haver esperanças.

À Maria Claudia, minha esposa: há muitos anos trilhamos o mesmo caminho e durante todo esse percurso tive a sorte e honra de poder contar com o seu amor, amizade, apoio, bom-humor e infinita paciência, especialmente nos momentos mais difíceis, mas também nos de grande alegria. O único de modo de lhe fazer juz seria com o seu nome na capa deste trabalho: perdoe-me por não poder fazê-lo.

Aos meus pais, Admilson e Terezinha, e meu irmão, Sandro: família é um porto-seguro; o lugar para onde corremos nos momentos de maior necessidade, ou até de quase desespero. Pude contar com minha família em todos os momentos difíceis, que foram muitos durante esse tempo. Se eu consegui chegar até aqui, relativamente inteiro, foi graças a vocês. Muito, muito obrigado. Isso vale, também, para os meus cunhados, Michelle, Paula, Mara, Quinho, Almir e Gisele; meus sogros, Roque e Lourdes, e meus sobrinhos, Júlia, Mateus, Luíza, e Bruno. Também não posso esquecer o apoio dos meus primos e tios.

Ao meu orientador, Zé Augusto: afortunado é o aluno que pode usar de palavras como admiração, sabedoria, compreensão, paciência, inteligência, amizade e

genialidade para descrever seu orientador; é com muito carinho que faço uso de todas estas e mais algumas. Deste mestrado, levo não apenas o título, mas tudo aquilo que aprendi em nossas inúmeras conversas, nas quais falávamos sobre ciência, política e vida. Em especial, aprendi o valor da simplicidade, tanto na ciência quanto na vida. Muito obrigado por tudo.

Ao Coelho: devo muito a você. Obrigado pelo imenso apoio demonstrado desde o início do curso e pela valiosa amizade. Obrigado também pela oportunidade de assistir áquilo que é sem dúvida uma aula perfeita: ministrada com absoluto domínio do assunto, mas acima de tudo tendo o aprendizado do aluno como objetivo principal. Suas aulas são uma demonstração de talento que marca seus alunos para sempre.

Ao Alair e a Yoshiko: muito obrigado pelas conversas, pela amizade e pelo excelente trabalho na banca de qualificação, marcada por imenso capricho na correção, extremamente criteriosa, e por sugestões indispensáveis.

Ao Carlinhos e a Cris: obrigado pelas muitas conversas, e também pela amizade e carinho sempre demonstrados.

Aos membros da banca de defesa, Coelho e Fábio: obrigado pelo tempo e atenção dedicados à correção, assim como pelas inúmeras e valiosas sugestões.

Ao Feoffilof: muito obrigado por me ensinar o que é uma demonstração matemática "de gente grande". O seu rigor matemático, a beleza de suas desmonstrações e a dedicação demonstrada em suas aulas o fazem um dos grandes pilares do Instituto. Como costumo dizer: "quem estudou no IME e não foi aluno do Feofiloff, não estudou no IME.".

Um muito obrigado a todos os professores da IME com os quais tive o prazer de conviver ou ter aulas durante esse mestrado: Yoshiharu, Leliane, Siang, Fabio Kon, Alfredo, Flávio Soares e Marcelo Queiroz.

Ao Cassio (Polpo de Campos): muito obrigado pelo apoio, amizade, inspiração, pelos treinos para a maratona e por proporcionar os primeiros contatos com o IME.

Ao Renato (Lucindo): obrigado pela amizade, daquelas que se constrói poucas vezes durante uma vida inteira e que deve ser preservada com muito carinho. Um especial obrigado pelas inúmeras discussões, as vezes um pouco acaloradas, mas que sempre nos proporcionaram muito aprendizado.

Obrigado aos professores Paulino Ng, Robert Liang Koo, Fernando Giorno e Ítalo Vega, que me apoiaram desde os tempos de graduação.

Um imenso abraço de agradecimento aos meus grandes amigos: Marcelo (Hashimoto), Marcel, Cardonha, Ellen, Domingos Dellamonica, Fabricio Benevides, Pil (Aritanan Gruber), Cris Sato, Goca (Gordana), Mario (Leston), Rafael (Luna), Rafael (Lobato), Marina, Chicão, Antonio, Jú (Barby), Bug (Ricardo Herrmann), David, Igor Sucupira, Igor Stockler, Cristiano Breuel, Marcelo (Miranda), Rodrigo Soboleski, Rodrigo Pedrosa, Said Sadique, Dubs (Rodrigo Dambrowsky), Arthur Gregório, Casca (Rodrigo Cascarrolho), Thadeu Russo, Ricardo Abrantes, Paulo Azevedo Silveira, Paulo Silveira, Marcio Oshiro, Paulo Salvador, Cabelo (Bruno Maimoni), Mituru Yamamoto e Fabio Pavan.

Resumo

Esta dissertação de mestrado aborda formulações computacionais e algoritmos para a busca e extração de padrões em cadeias biológicas. Em particular, o presente texto concentra-se nos dois problemas a seguir, considerando-os sob as distâncias de Hamming e Levenshtein:

a) como determinar os locais nos quais um dado padrão ocorre de modo aproximado em uma cadeia fornecida;

b) como extrair padrões que ocorram de modo aproximado em um número significativo de cadeias de um conjunto fornecido.

O primeiro problema, para o qual já existem diversos algoritmos polinomiais, tem recebido muita atenção desde a década de 60, e ganhou novos ares com o advento da biologia computacional, nos idos dos anos 80, e com a popularização da Internet e seus mecanismos de busca: ambos os fenômenos trouxeram novos obstáculos a serem superados, em razão do grande volume de dados e das bastante justas restrições de tempo inerentes a essas aplicações.

O segundo problema, de surgimento um pouco mais recente, é intrinsicamente desafiador, em razão de sua complexidade computacional, do tamanho das entradas tratadas nas aplicações mais comuns e de sua dificuldade de aproximação. Também é de chamar a atenção o seu grande potencial de aplicação.

Neste trabalho são apresentadas formulações adequadas dos problemas abordados, assim como algoritmos e estruturas de dados essenciais ao seu estudo. Em especial, estudamos a extremamente versátil árvore dos sufixos, assim como uma de suas generalizações e sua estrutura irmã: o vetor dos sufixos.

Grande parte do texto é dedicada aos filtros baseados em q-gramas para a busca aproximada de padrões e algumas de suas mais recentes variações.

Estão cobertos os algoritmos bit-paralelos de Myers e Baeza-Yates-Gonnet para a busca de padrões; os algoritmos de Sagot para a extração de padrões; os algoritmos de filtragem de Ukkonen, Jokinen-Ukkonen, Burkhardt-Kärkkäinen, entre outros.

Abstract

This thesis deals with computational formulations and algorithms for the extraction and search of patterns from biological strings. In particular, the present text focuses on the following problems, both considered under Hamming and Levenshtein distances:

1. How to find the positions where a given pattern approximatelly occurs in a given string;
2. How to extract patterns which approximatelly occurs in a certain number of strings from a given set.

The first problem, for which there are many polinomial time algorithms, has been receiving a lot of attention since the 60's and entered a new era of discoveries with the advent of computational biology, in the 80's, and the widespread of the Internet and its search engines: both events brought new challenges to be faced by virtue of the large volume of data usually held by such applications and its time constraints.

The second problem, much younger, is very challenging due to its computational complexity, approximation hardness and the size of the input data usually held by the most common applications. This problem is also very interesting due to its potential of application.

In this work we show computational formulations, algorithms and data structures for those problems. We cover the bit-parallel algorithms of Myers, Baeza-Yates-Gonnet and the Sagot's algorithms for patterns extraction. We also cover here the oustanding versatile suffix tree, its generalised version, and a similar data structure: the suffix array.

A significant part of the present work focuses on q-gram based filters designed to solve the approximate pattern search problem. More precisely, we cover the filter algorithms of Ukkonen, Jokinen-Ukkonen and Burkhardt-Kärkkäinen, among others.

Sumário

0.1 A Evolução de Darwin 1

0.2 Padrões em Cadeias Biológicas 2

0.3 Trabalhos Anteriores 4

0.4 Conteúdo do Presente Trabalho 5

0.5 Relevância Prática e Teórica 8

1 Notação e Definições Básicas **11**

1.1 Cadeias de Caracteres 11

1.2 Operadores de Edição 13

1.3 Distância entre Cadeias 14

1.3.1 (d, r)-vizinhança de Uma Cadeia 15

2 Estruturas de Dados **17**

2.1 Introdução 17

2.2 A Árvore dos Sufixos 18

2.2.1 Árvore dos Sufixos e a Busca Exata de Padrões 21

2.3 O Vetor dos Sufixos 24

2.3.1 Algoritmos para a Construção do Vetor dos Sufixos 28

2.4 Comentários Bibliográficos 31

3 Busca Aproximada **33**

3.1 Introdução 33

3.2 O Bit-Paralelismo 35

3.3 O Modelo Computacional 36

3.4 O Algoritmo Baeza-Yates – Gonnet 37

3.5 O Algoritmo de Myers 39

3.5.1 As Matrizes Δv e Δh 41

3.5.2 Representando Δv e Δh Através de Vetores Binários 43

3.5.3 O Algoritmo 46

3.6 Comentários Bibliográficos 52

4 Extração de Padrões **55**

4.1 Introdução 55

4.2 Algoritmo de Sagot para a Extração de Motifs Repetidos 57

4.3 Algoritmo de Sagot para a Extração de Motifs Comuns 61

4.4 Extração de Motifs Estruturados 66

4.5 Comentários Bibliográficos 71

5 Algoritmos de Filtragem **75**

5.1 Filtros baseados em q-gramas 78

5.1.1 O Teorema das q-gramas 79

5.1.2 Cálculo da Distância de q-gramas 80

5.2 O Algoritmo de Ukkonen 85

5.3 O Algoritmo de Jokinen e Ukkonen 88

5.3.1 O Pré-Processamento 88

5.3.2 A Etapa de Filtragem 89

5.3.3 A Etapa de Verificação 91

5.3.4 Resultados Experimentais 91
5.4 O Algoritmo Burkhardt-Kärkkäinen 94
5.4.1 Q-gramas 94
5.4.2 O Teorema das Q-gramas 96
5.4.3 Determinação do Limiar de Filtragem 97
5.4.4 Determinação do Formato Ideal 100
5.4.5 Construção das Estruturas de Índices 101
5.5 Extensões do Algoritmo de Burkhardt e Kärkkäinen 101

6 Conclusão **105**

A Demonstração do Filtro de Ukkonen **107**

Prefácio

0.1 A Evolução de Darwin

Imaginemos, por um minuto, que uma nave alienígena houvesse chegado à Terra e pretendêssemos desvendar, para os seus curiosos ocupantes, o pouco que sabemos sobre nós mesmos, o mundo no qual vivemos e os demais seres que o compartilham conosco. Poderíamos nos afundar nas reflexões de Freud, debater as teorias de Newton ou, talvez, reverenciar a beleza de Shakeaspeare, Mozart e Michelangelo. Mas, seria apenas à força dos argumentos de Darwin, suplantável apenas pela simplicidade com a qual são apresentados, que veríamos nossos verdinhos visitantes exprimirem um "Ahh!", de súbita compreensão.

A ciência nos permite guardar reservas quanto ao surgimento do universo, ou à origem da vida na Terra, mas sempre teremos o conforto de saber — talvez sem todos os pormenores — como e por que, partindo de uma pequena e desprezível forma primitiva de vida, nos tornamos, se não os mais capazes, sem dúvida os mais arrogantes representantes de uma classe que Dawkins [Daw85] batizou de **objetos complexos**: arranjos de matéria cuja constituição, complexa e organizada, não pode ser atribuída ao acaso, e cuja origem só pode ser compreendida à luz da evolução[1].

Todo objeto complexo possui a capacidade de reproduzir-se; isto é, pode propagar, para os seus descendentes, o conjunto de instruções existentes no material genético, DNA ou RNA, contido em cada uma de suas células, juntamente com os possíveis erros inerentes ao processo. Ademais, todos os seus descendentes também farão parte dessa mesma classe, portanto estarão irremediavelmente sujeitos à inclemência da Evo-

[1] Os objetos complexos equivalem, portanto, à união dos conjuntos dos seres vivos e dos vírus.

lução.[2] Porém, a reprodução não é um direito universal inalienável, mas um benefício conquistado. A seleção darwiniana concede o direito de transmitir o seu código genético, à geração seguinte, apenas aos indivíduos cujas características, ditadas por esse mesmo código genético, tornam-os capazes de resistir à pressão exercida pelo meio-ambiente e atingir a idade adulta. Logo, instruções cujas características sejam benéficas aos indivíduos que as possuem tendem a ser conservadas com o passar das gerações, talvez com algumas mutações, enquanto instruções danosas tendem ao descarte.[3]

A forma geral desse intrincado conjunto de fenômenos é clara e bem conhecida, porém inúmeros detalhes permanecem obscuros ou apenas recentemente foram elucidados, em virtude dos recentes avanços nos métodos de seqüenciamento genético e das descobertas obtidas pela genômica funcional; área de estudos que ganhou notoriedade nos últimos anos, firmando-se como um campo de pesquisa próspero e que propõe uma infinidade de problemas interessantes e desafiadores, nas mais diversas áreas de conhecimento. Dentre tais problemas, ocupa uma posição de destaque, em razão de seu inestimável valor e complexidade, o estudo de técnicas computacionais para a localização, determinação e representação de padrões contidos em cadeias de DNA, RNA, proteínas ou de seus constituintes aminoácidos.

0.2 Padrões em Cadeias Biológicas

Lembrando que trechos do material genético que exibam algum tipo de função primordial ao sucesso evolutivo de um determinado organismo tendem a ser conservados pela evolução, mesmo que não de modo exato [SNH04, SBM03], é fácil aceitarmos que tais trechos, chamados de regiões funcionais, podem ser representados por padrões recorrentes em um conjunto de genomas relacionados. Reciprocamente, padrões

[2] Se nos permitirmos abusar um pouco da terminologia, isto equivale a dizer que a classe dos objetos complexos é fechada por reprodução.

[3] Se analisada com superficialidade, a argumentação pode ser tomada por controversa: instruções cuja expressão fenotípica não implica em nenhum benefício direto para o organismo em questão, por vezes podem ser preservadas pela evolução. É o princípio do *gene egoísta*, perfeitamente ilustrado pelo macho do Louva-Deus: seus genes o impelirão à cópula, mesmo que isso signifique a sua morte ao ser devorado pela fêmea. Não há contradição. Note que apesar de danosos ao espécime macho devorado, os genes acabam por favorecer seu próprio sucesso evolutivo: a fecundação foi realizada com sucesso, os genes serão propagados para a geração seguinte.

que ocorram em todos ou em grande parte dos genomas, de um mesmo conjunto, com grande chance representam regiões funcionais [CS01, SW03]. O que nos induz, diretamente, a dois dos problemas apresentados neste trabalho: (i) dado um padrão, encontrar os locais em que ele ocorre, possivelmente com erros, em uma dada cadeia; (ii) extrair, a partir de um conjunto de cadeias biológicas, padrões que ocorram, possivelmente com erros, como segmentos de todas ou de grande parte das cadeias fornecidas. Ambos os problemas podem ser formulados de diversas formas, dependendo da definição de "padrão" adotada. Aqui, seguindo a tendência da área, vamos adotar a definição baseada em funções de distância entre cadeias.

Por vezes, regiões funcionais não se apresentam como um único segmento de nucleotídeos, mas como um conjunto de segmentos não-contíguos distribuídos pelo DNA de forma não necessariamente uniforme. Esse detalhe torna sua determinação e localização tarefas sobremaneira mais complexas, como iremos ver no presente texto.

A manipulação de cadeias biológicas é, intrinsecamente, uma atividade desafiadora. O genoma da maior parte dos organismos, dentre os seqüenciados até o momento, ultrapassa um bilhão de pares de bases de comprimento[4]. Tal volume de dados exige a adoção de algoritmos eficientes para o seu processamento. É importante ressaltarmos que o consumo de tempo de algoritmos polinomialmente eficientes é, muitas vezes, proibitivo, sendo necessária a adoção de algoritmos cujas necessidades, de espaço e tempo, cresçam apenas linearmente, ou mesmo sublinearmente, em relação ao tamanho da entrada. Infelizmente, para a a maioria dos problemas que aqui veremos, algoritmos com essa eficiência no pior caso são inexistentes ou, até o momento, desconhecidos pela ciência. Uma solução natural para esse impasse é a busca por algoritmos heurísticos, de aproximação, ou com consumo de tempo de caso médio satisfatório; é nessa última alternativa que depositamos nossos esforços ao abordar algoritmos de filtragem.

[4]No momento da redação dessa dissertação o projeto *Animal Genome Size* [Gre05] contava com 4056 genomas em sua base de dados. Destes, 96% apresentavam comprimento superior a 10^9 pares de bases, 10% superior a 10^{10} pares de bases e $1{,}9\%$ superior a 10^{11} pares de bases.

0.3 Trabalhos Anteriores

A busca de padrões em cadeias é um dos problemas fundamentais da Ciência da Computação. Logo, foi ampla e profundamente estudado e é, evidentemente, longevo: segundo Knuth [Knu98][5], o primeiro método algorítmico para um problema de busca em cadeias foi proposto em 1946, por John Mauchly [Mau46]. O caso particular da busca aproximada de padrões também não é novo: trabalhos dedicados ao tema já existiam no final da década de 60 [Nav98], porém seu nascimento pode ter se dado antes disso. Sankoff e Kruskal [SK83] fornecem um pequeno apanhado histórico da área, enquanto uma visão abrangente e detalhada da busca aproximada de padrões pode ser encontrada na excelente tese de doutorado de Navarro [Nav98] e em outro trabalho do mesmo autor [Nav01]. É evidente que o amplamente estudado problema da **busca exata de padrões**, para o qual existem diversos algoritmos eficientes, em relação ao consumo de tempo e espaço [KMP77, Knu98, CLRS01], não é, senão, um caso particular da busca aproximada de padrões, notadamente aquele em que a entrada não permite a existência de erros. Exceto em um ou outro ponto de nosso trabalho, e sempre com o intuito de introduzir algum problema mais complexo, não vamos abordar a busca exata de padrões.

A extração de padrões é tratada por Crochemore e Sagot [CS01], que apresentam uma resenha de grande parte dos problemas e algoritmos que aqui veremos; uma discussão semelhante, porém mais atual, é fornecida por Sagot e Wakabayashi [SW03].

Fatia considerável do conteúdo da primeira parte desse texto, em especial as estru-

[5] A bem da verdade, Knuth situa a origem do problema em um patamar bem anterior: em seu livro, a referência mais antiga ao problema data de 300 A.C. e remete ao Império Babilônico. Entre suas inúmeras e interessantes citações, uma em especial vale comentar, por seu caráter hilário. Segue o excerto, traduzido do original em inglês:

> O primeiro dicionário de inglês, *Table Alphabetical*, de Robert Cawdrey (Londres, 1604), contém as seguintes instruções: "Se a palavra que se deseja encontrar inicie com (a), procure-a no início dessa tabela, porém caso inicie com (v) procure-a próximo ao final. Agora, se a palavra iniciar com (ca), procure-a no início da letra (c). Se for (cu), então procure-a no final da letra e assim por diante". É interessante notar que Cawdrey estava ensinando o alfabeto **a si próprio** enquanto preparava seu dicionário. Inúmeras palavras aparecem incorretamente posicionadas nas primeiras páginas do livro, enquanto as últimas encontram-se quase que em perfeita ordem alfabética!

turas de dados tratadas no capítulo 2, e pequenos tópicos da segunda parte, podem ser encontrados no livro de Gusfield [Gus97], porém de modo bastante disperso. Crochemore [CR94] trata de parte da temática de Gusfield, porém insere tópicos que este omite, como os autômatos dos sufixos, e aprofunda-se em outros nos quais Gusfield apenas resvala, como os algoritmos paralelos. Um excelente e rigoroso debate sobre árvores dos sufixos e algumas de suas aplicações é fornecido por Lago e Simon [dLS03].

Além desses trabalhos, na última década diversas teses de doutorado e dissertações de mestrado exploraram o tema:

- Carvalho [Car04] estuda métodos para a extração de motifs estruturados e propõe um algoritmo baseado no uso de **ligações de blocos**;
- Fonseca [Fon03] discute **estruturas de índices** para parte dos problema discutidos por Sagot [CS01];
- Burkhardt [Bur02] aborda o uso de filtros para a localização de motifs simples, introduzindo uma ferramenta baseada no algoritmo JOKINEN-UKKONEN, chamada QUASAR, e um algoritmo de filtragem baseado no processamento de Q-gramas [BK01, BK03a], que discutiremos, em detalhes, no Capítulo 5;
- Pisanti [Pis02] propõe conjuntos de **motifs fundamentais** para a extração de motifs curingas, assim como modificações no algoritmo de Marsan e Sagot [MS00a, MS00b] para a extração de motifs estruturados sob a distância de Hamming;
- Karpischeck [Kar93] trata do autômato dos sufixos e, em menor profundidade, da árvore dos sufixos e do vetor dos sufixos; estruturas de dados fundamentais em nosso trabalho;
- Peterlongo [Pet06] discute algoritmos de filtragem para a extração de repetições aproximadas.

0.4 Conteúdo do Presente Trabalho

À primeira vista, padrões em cadeias parece um assunto muito explorado pela ciência, talvez até excessivamente. De fato, a oferta de textos que abordam os problemas aqui

discutidos é vasta, porém poucos desses trabalhos englobam o uso de algoritmos de filtragem, e um número ainda menor é rigoroso o suficiente para permitir uma análise detalhada da área. Pretendemos preencher essas e outras lacunas com o presente trabalho, oferecendo uma discussão rigorosa e, em boa medida, abrangente sobre a busca e extração de padrões em cadeias. Vamos enfatizar a adoção de algoritmos de filtragem e a aplicação das técnicas aqui discutidas para o processamento de cadeias biológicas, cobrindo os avanços mais recentes na área, que não foram poucos.

O texto está logicamente dividido em duas partes. A primeira, que compreende os capítulos 1 a 4, apresenta boa parte das definições e terminologia adotadas, formaliza os problemas centrais e discute algoritmos exatos e sem filtragem para os mesmos, assim como as estruturas de dados nos quais esses algoritmos se baseiam. A segunda parte, que compreende o capítulo 5, trata de algoritmos de filtragem para alguns dos problemas expostos na primeira parte. Cada capítulo se inicia com uma seção introdutória e, exceção feita ao capítulo 1, é concluído com uma seção de comentários bibliográficos. Em detalhes, o texto está organizado como segue:

- o capítulo 1 apresenta a notação e algumas das definições básicas que serão adotadas no decorrer do texto. Ao apresentá-las, procuramos ser tão econômicos e precisos quanto foi possível. Acreditamos que uma leitura superficial do capítulo é essencial, porém o leitor pode, sem prejuízo de compreensão, evitar aprofundar-se nos detalhes, bastando retornar ao capítulo sempre que julgar necessário;

- o capítulo 2 expõe as principais estruturas de dados utilizadas pelos algoritmos aqui discutidos e suas propriedades, assim como enuncia e demonstra alguns dos teoremas que fundamentam tais estruturas. Uma breve discussão sobre os algoritmos disponíveis para a discussão de tais estruturas também se faz presente;

- o capítulo 3 discute a busca aproximada de padrões. Após a apresentação de alguns conceitos iniciais, o algoritmo de Myers [Mye99] é discutido em profundidade;

- o capítulo 4 aborda a extração de padrões a partir de conjuntos de cadeias. O capítulo está estruturado de modo semelhante ao anterior: inicialmente os conceitos básicos são apresentados, juntamente com a formalização do problema e o capítulo é concluído com a discussão do algoritmo de Sagot [Sag98]. Para tornar

o texto mais interessante, assim como facilitar a compreensão do tópico, também apresentaremos a extração de padrões repetidos em uma única cadeia;

- o capítulo 5 trata de filtros para a busca aproximada de padrões em cadeias. Conceitos básicos são apresentados e algumas soluções, propostas e analisadas.

Como não pretendemos esgotar o assunto, diversos tópicos interessantes foram omitidos. Em maiores detalhes:

- trataremos exclusivamente dos filtros sem perda. Logo, alguns dos algoritmos heurísticos presentes nas ferramentas mais utilizadas não constam neste texto. Para maiores detalhes sugerimos Gusfield [Gus97], Myers *et al.* [AGM+90] e Pearson *et al.* [PL88]. É válido lembrar que parte significativa das técnicas adotadas por esses algoritmos encontra-se coberta pelo conteúdo da segunda parte de nosso trabalho: algoritmos de filtragem;

- não abordaremos, diretamente, soluções para a busca exata de padrões [KMP77, Knu98, CLRS01]. Porém, como algoritmos para esse problema servem de base para a solução de outros mais complexos, entre os quais há muitos de nosso interesse, além de possuírem um enorme valor paradigmático e histórico, algumas técnicas originalmente aplicadas à busca exata serão aqui apresentadas;

- algoritmos de aproximação, demonstrações da complexidade computacional dos problemas abordados e algoritmos probabilísticos também não constam deste texto, apesar de constituírem áreas fascinantes de estudo. Recomendamos [LMW99], [LMW02] e [FGN06];

- não levaremos em conta formulações dos problemas sob outras distâncias entre cadeias que não Hamming e Levenshtein. Para maiores informações, *vide* Ukkonen [Ukk92], Damerau [Dam64], Fredrikkson *et al.* [FMN06] e Makinen *et al.* [MNU05].

Sempre que possível, as devidas referências aos tópicos omitidos serão incluídas no texto, para que um leitor interessado possa aprofundar-se em algum desses temas, se assim o desejar.

0.5 Relevância Prática e Teórica

A principal motivação para a realização desse trabalho se deu em razão do grande interesse surgido, nos últimos anos, em torno do uso de algoritmos de filtragem para aplicações destinadas a recuperação de informações, em especial no que tange o processamento de cadeias biológicas e a busca de páginas na web. Ambas as aplicações possuem, entre outras, uma característica em comum de vital importância: o volume de dados que deverá ser processado é monstruoso. Esse detalhe torna essas duas aplicações cenários ideais para o uso das técnicas aqui descritas que, por sua vez, também servem a diversas outras aplicações. Em maiores detalhes, podemos justificar a relevância de nosso trabalho trazendo ao leitor alguns problemas interessantes:

- Recuperação de Informações na Internet [BYRN99]: poucas áreas têm experimentado o aparecimento de novas técnicas em um ritmo tão intenso quanto a recuperação de informações armazenadas em páginas da Internet. No início, os mecanismos de busca proporcionavam uma experiência de usuário extenuante e frustrante: os resultados de buscas eram pouco relevantes e a atividade improdutiva. Novas técnicas de processamento distribuído, aprendizado de máquina, dentre outras, contribuíram para essa recente onda de avanços, mas só o uso intensivo de boas estruturas de dados para a indexação de cadeias, como as que estudamos nesse trabalho, e a descoberta de novos algoritmos de busca, tornaram tal avanço possível. Alguns mecanismos de busca, como o Google, disponibilizaram recentemente serviços de correção automática e sugestão de palavras chaves. Tais serviços fazem proveito do uso de estruturas de indexação de q-gramas e, na grande maioria dos casos, adota a distância de Levenshtein para compor o espaço métrico adotado. Além disso, recentemente o Google passou a incorporar a busca aproximada de padrões em seu mecanismo de busca que, até então, buscava apenas ocorrências exatas das palavras chaves inseridas.

- Segurança de Dados [PCGS03a, RKI06, CMS05]: Em 1992, Udi Manber, em coautoria com Sun Wu, [MW94] propôs uma solução para o seguinte problema: determinar se uma dada palavra assemelha-se à alguma palavra de um conjunto fornecido. A aplicação que os autores tinham como foco consistia em identificar a ocorrência de ataques por força bruta para a descoberta de senhas de usuário. Assim como essa aplicação, diversas outras relacionadas à segurança de dados

podem ser modeladas através dos problemas aqui abordados e o volume de trabalhos publicados nesse segmento é enorme.

- Computação Musical [CIR98, IKMV00, CCI+02]: uma peça musical pode ser representada por meio de cadeias de caracteres. Tomando um exemplo bastante rudimentar, é representação válida de uma dada peça uma cadeia sob um alfabeto que relaciona o conjunto de notas na escala diatônica. Visto isso, diversos problemas da computação musical como: classificação e reconhecimento de melodias, identificação de acordes e rastreamento de evoluções, apenas para citar alguns, podem ser modelados como problemas de processamento de cadeias, o que abre um grande leque de opções de algoritmos e estruturas de dados aplicáveis a tais problemas [CIR98].

- Compressão de Dados [AP03]: O uso das estruturas de dados exploradas nesse trabalho se estende por diversas áreas, mas é nesses dois tópicos fundamentais, recuperação de informações e compressão de dados, que toda a sua funcionalidade se faz mostrar. Tanto a árvore dos sufixos quanto o vetor dos sufixos se prestam a essa função: de algoritmos da família Ziv-Lempel [ZL77] aos mais recentes e promissores algoritmos Borrows-Wheller [BW94], diversos exploraram essa funcionalidade.

Capítulo 1

Notação e Definições Básicas

1.1 Cadeias de Caracteres

Definição 1 (Alfabeto)**.** Um **alfabeto** Σ é um conjunto finito de caracteres.

Definição 2 (Cadeia de Caracteres)**.** Toda seqüência formada sobre elementos de um alfabeto Σ é uma **cadeia sobre** Σ, ou simplesmente **cadeia** quando o alfabeto for óbvio ou irrelevante.

`Alguns Exemplos:`

1. As palavras não acentuadas da língua portuguesa são cadeias sobre o alfabeto de 26 letras;
2. `1234` e `01001` são cadeias sobre o alfabeto $\{\texttt{0}, \texttt{1}, \ldots, \texttt{9}\}$.

Com freqüência, adotaremos como alfabeto o conjunto de nucleotídeos $\{\texttt{a}, \texttt{c}, \texttt{t}, \texttt{g}\}$, no caso das cadeias de DNA, ou o alfabeto de 20 letras que representam o conjunto dos aminoácidos, para cadeias de proteína. Denotamos o comprimento de uma cadeia $X = x_1 \ldots x_m$ por $|X| = m$. Uma cadeia de comprimento zero é dita vazia e denotada por ϵ. O conjunto de todas as cadeias formadas sobre um alfabeto Σ é denotado por Σ^*. De modo análogo, o conjunto de todas as cadeias sobre Σ, e de comprimento k, é

denotado por Σ^k. Σ^+ é o conjunto $\Sigma^* \backslash \{\epsilon\}$. Para o que segue, considere $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$ duas cadeias sobre um alfabeto Σ.

Definição 3. A **concatenação** de X e Y é a cadeia $x_1 \dots x_m y_1 \dots y_n$, denotada por $X.Y$ ou simplesmente XY. Claramente, $|X.Y| = |X| + |Y|$. Note, ainda, que a operação de concatenação é não-simétrica, associativa e tem ϵ como o elemento neutro.

Para dados $I \subseteq \mathbb{Z}$ e $j \in \mathbb{Z}$, vamos definir os operadores **deslocamento à esquerda** ($\ominus$) e **deslocamento à direita** ($\oplus$):

$$\begin{aligned} I \ominus j &= \{i - j \colon i \in I\}, \\ I \oplus j &= \{i + j \colon i \in I\}. \end{aligned}$$

Definição 4. Suponha uma parte I do conjunto $\{1, \dots, m\}$ de índices de caracteres de X, e seja $\langle i_1, \dots, i_{m'} \rangle$ a seqüência em ordem estritamente crescente dos elementos de I. A cadeia $x_{i_1} x_{i_2} \dots x_{i_{m'}}$ é uma **subseqüência** de X, no caso, a subseqüência de X **induzida** por I, denotada por $X[I]$.

Informalmente, subseqüência de X é toda cadeia obtida a partir da remoção de zero, ou mais, caracteres de X.

Definição 5. Fator ou **segmento** de X é toda cadeia Y que pode ser escrita na forma $x_i x_{i+1} \dots x_j$, para algum par $i, j \in \mathbb{Z}$ tal que $1 \leq i \leq j + 1 \leq m + 1$; nesse caso dizemos que Y **ocorre** em X na posição i. Também, podemos escrever $X_{i..j}$, para representar Y. Se $i > j$, então por convenção Y é vazia. Um k**-fator** de X é todo fator de X de comprimento k.

O número de ocorrências de Y como fator de X é dado por $\kappa(Y, X)$; ou seja,

$$\kappa(Y, X) = \sum_{1 \leq i \leq m-n+1} \begin{cases} 1, & \text{se } Y = X_{i..i+n-1} \\ 0, & \text{se } Y \neq X_{i..i+n-1}. \end{cases}$$

O conjunto de fatores de X é denotado por $\mathsf{Ft}(X)$, enquanto o conjunto de suas subseqüências é denotado por $\mathsf{Ss}(X)$.

Definição 6. Prefixo de X é toda cadeia $X_{1..j}$, para algum $0 \leq j \leq m$. De modo semelhante, **sufixo** de X é toda cadeia $X_{i...m}$, para algum $1 \leq i \leq m + 1$.

Notemos que as cadeias vazias $X_{m+1..m}$ e $X_{1..0}$ são, respectivamente, sufixo e prefixo de X. O conjunto de prefixos de X é denotado por $\mathtt{Pf}(X)$, enquanto o conjunto de seus sufixos é denotado por $\mathtt{Sf}(X)$.

> `Exemplo:` a cadeia é `abc`. Suas subseqüências são: ϵ, a, b, c, `ab`, `bc`, `ac` e `abc`; seus fatores são: ϵ, a, b, c, `ab`, `bc` e `abc`; seus sufixos são: ϵ, abc, bc e c; seus prefixos são: ϵ, a, ab e abc.

Vamos denotar o sufixo de X obtido a partir da remoção do prefixo Y da cadeia por $Y^{-1}X$, isto é,

$$Z = Y^{-1}X \iff X = YZ.$$

Analogamente,

$$Z = XY^{-1} \iff X = ZY.$$

1.2 Operadores de Edição

Para o que segue, considere Σ um alfabeto.

Definição 7. Um **operador de edição** é uma função que assume uma das formas abaixo

$$\begin{aligned} \Sigma^* \times \Sigma \times \mathbb{Z} &\rightarrow \Sigma^* \\ \text{ou} & \\ \Sigma^* \times \mathbb{Z} &\rightarrow \Sigma^*. \end{aligned}$$

No presente texto, vamos considerar apenas os operadores **inserção**, **remoção** ou **substituição**.

Definição 8. O **operador de substituição** é a função $sub : \Sigma^* \times \Sigma \times \mathbb{Z} \rightarrow \Sigma^*$ tal que, para toda cadeia $X = x_1 \ldots x_m$, caractere y e inteiro i,

$$sub(X, y, i) \mapsto X_{1\ldots i-1} y X_{i+1\ldots m}.$$

Definição 9. O **operador de inserção** é uma função $ins : \Sigma^* \times \Sigma \times \mathbb{Z} \rightarrow \Sigma^*$ tal que, para toda cadeia $X = x_1 \ldots x_m$, caractere y e inteiro i,

$$ins(X, y, i) \mapsto X_{1\ldots i} y X_{i+1\ldots m}.$$

Definição 10. O **operador de remoção** é a função $del : \Sigma^* \times \mathbb{Z} \to \Sigma^*$ tal que, para toda cadeia $X = x_1 \dots x_m$ e inteiro i,

$$del(X, i) \mapsto X_{1\dots i-1} X_{i+1\dots m}.$$

`Exemplo:` a cadeia X é `abca`:

- $sub(X, \mathtt{d}, 3) \mapsto$ `abda`;
- $ins(X, \mathtt{d}, 3) \mapsto$ `abdca`;
- $del(X, 3) \mapsto$ `aba`.

1.3 Distância entre Cadeias

Para o que segue, seja Σ um alfabeto.

Definição 11. Uma **função distância**, ou **métrica**, em Σ^* é uma função $d \colon \Sigma^* \times \Sigma^* \to \mathbb{R}$ para a qual valem as propriedades a seguir, para quaisquer $X, Y, Z \in \Sigma^*$:

$$d(X, Y) = d(Y, X); \tag{1.1a}$$
$$d(X, X) = 0; \tag{1.1b}$$
$$X \neq Y \Leftrightarrow d(X, Y) > 0; \tag{1.1c}$$
$$d(X, Y) + d(Y, Z) \geq d(X, Z). \tag{1.1d}$$

Logo, (Σ^*, d) é um **espaço métrico**. É evidente, em virtude da propriedade $1.1b$, que $d(\epsilon, \epsilon) = 0$. Apresentamos, a seguir, as funções de distância que consideramos de maior importância para o presente trabalho. Vamos considerar

$$\alpha(a, b) = \begin{cases} 0 & \text{se} \quad a = b \\ 1 & \text{se} \quad a \neq b \end{cases}, \forall a, b \in \Sigma.$$

Distância de Hamming É denotada por d_H e definida pela recorrência abaixo:

$$d_H(X_{1..n}, Y_{1..m}) = \begin{cases} 0 & \text{se } n = 0 \text{ e } m = 0 \\ d_H(X_{1..n-1}, Y_{1..m-1}) + \alpha(x_n, y_m) & \text{se } n = m \text{ e } n, m > 0 \\ \infty & \text{se } n \neq m. \end{cases}$$

Informalmente, $d_H(X, Y)$ corresponde ao número mínimo de operações de substituição que transformam X em Y.

Distância de Levenshtein Também chamada de **distância de edição**, é denotada por d_L e definida pela recorrência abaixo:

$$d_L(X_{1..n}, Y_{1..m}) = \begin{cases} \max\{n, m\} & \text{se } n = 0 \text{ ou } m = 0 \\ \min \left\{ \begin{array}{l} d_L(X_{1..n-1}, Y_{1..m-1}) + \alpha(x_n, y_m) \\ d_L(X_{1..n}, Y_{1..m-1}) + 1 \\ d_L(X_{1..n-1}, Y_{1..m}) + 1 \end{array} \right\} & \text{se } n, m > 0. \end{cases}$$

Informalmente, $d_L(X, Y)$ corresponde ao número mínimo de operações de inserção, remoção ou substituição que transformam X em Y. A distância entre duas cadeias pode ser calculada, para as funções apresentadas, através de programação dinâmica, com consumo de tempo $O(n^2/\lg^2 n)$ [MP80].

1.3.1 (d, r)-vizinhança de Uma Cadeia

Definição 12. Dados duas cadeias X e Y, uma métrica d e um inteiro r, dizemos que X e Y são cadeias (d, r)**-vizinhas** se $d(X, Y) \leq r$.

Definição 13. Dados uma cadeia X, uma métrica d e um inteiro r, a (d, r)**-vizinhança** de X é o conjunto $\{Y \in \Sigma^* : d(X, Y) \leq r\}$.

Vamos denotar a (d_H, r)-vizinhança de X por $NH_r(X)$ e sua (d_L, r)-vizinhança por $NL_r(X)$.

Capítulo 2

Estruturas de Dados

2.1 Introdução

Apresentaremos, nesse capítulo, algumas das estruturas de dados adotadas pelos algoritmos estudados no presente trabalho. Dada uma cadeia X, estamos particularmente interessados em estruturas que nos permitam representar, de modo eficiente, o conjunto de fatores de X. Por eficiente, entenda-se uma representação com ambos, consumo de espaço e tempo de construção, lineares no comprimento da cadeia, ou algo próximo disso. Ademais, é desejável que possamos verificar a pertinência de uma cadeia Y no conjunto de fatores de X em tempo linear no comprimento de Y.

Observe, que o tamanho de $\mathsf{Ft}(X)$ é, no pior caso, dado por

$$1 + \sum_{k=1}^{n} k \quad = \quad \Theta(n^2).$$

Ademais, novamente no pior caso, o número de caracteres necessários para representar a concatenação de todos os fatores de X é dado por

$$\sum_{k=1}^{n} k(n-k+1) \quad = \quad \Theta(n^3).$$

Em virtude do exposto, é surpreendente podermos construir representações eficientes do conjunto de fatores de uma cadeia. De fato, em 1970, Knuth [KMP77, Apo85, dLS03] conjecturou que o maior fator repetido de uma cadeia não poderia ser obtido em tempo

$O(n)$; um problema de fácil solução quando se tem em mãos uma estrutura de dados com as propriedades acima descritas. Felizmente, em 1973, Weiner [Wei73] mostrou que ele estava errado, com a publicação do primeiro algoritmo linear para a construção de uma **árvore dos sufixos**, então eleito por Knuth o algoritmo do ano [Apo85, dLS03]. A árvore dos sufixos é uma estrutura de dados extremamente poderosa, que nos permite representar e obter de modo eficiente todas as cadeias que são prefixo de algum sufixo de uma dada cadeia. Logo, permite a obtenção de todos os fatores da cadeia. É recorrente na literatura afirmar que a árvore dos sufixos resolve uma miríade de problemas sobre cadeias, em virtude dessa estrutura de dados possuir aplicações das mais diversas, com certo destaque para recuperação de informações e compressão de dados [AKO04, AOK02, Apo85, GK97].

2.2 A Árvore dos Sufixos

Definição 14 (Árvore dos Sufixos)**.** Seja $T = (N, A, \lambda)$ uma árvore orientada com raiz r em N, tal que para todo nó u existe um único caminho orientado de r a u. Ademais, seja $\lambda\colon A \to (\Sigma \cup \{\$\})^+$ uma rotulação nos arcos de T, onde $\$ \notin \Sigma$. O rótulo de um nó $u \in N$ é denotado por $\phi(u)$ e definido como o resultado da concatenação dos rótulos dos arcos no caminho de r a u. Dada uma cadeia X, dizemos que T é uma **árvore dos sufixos** de X quando:

(p1) todo nó interno (não folha) em T possui dois ou mais filhos;

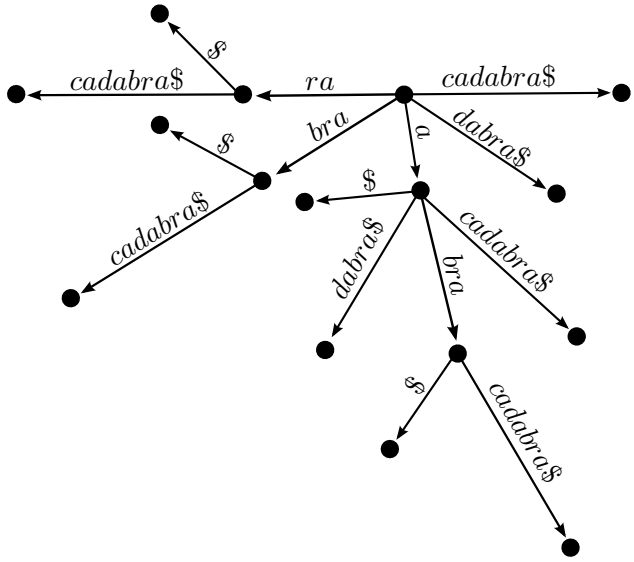
(p2) para todo par de arcos $e, f \in A$, com origem em um mesmo nó, os rótulos de e e f iniciam-se com caracteres distintos entre si;

(p3) se um nó u é folha de T, então o rótulo de u é sufixo da cadeia $X\$$;

(p4) para toda cadeia Y, sufixo de $X\$$, existe uma folha $u \in T$ com rótulo Y.

Para um dado nó u, seja $\phi'(u)$ a cadeia obtida após a remoção do último caractere de $\phi(u)$. Note que se u é folha de T, então o último caractere de $\phi(u)$ será sempre \$. Da propriedade (p3), segue imediatamente que para toda folha u de T, $\phi'(u)$ é sufixo de X.

Figura 2.1 Árvore dos sufixos da cadeia `abracadabra`



Os teoremas que apresentamos a seguir asseguram a versatilidade da estrutura.

Teorema 15. *Para todo par de nós $u, v \in N$, u é ancestral de v se, e somente se, $\phi(u)$ é prefixo de $\phi(v)$.*

Demonstração. Provaremos inicialmente a necessidade. Seja $u, v \in N$ um par de nós tal que u é ancestral de v. Vamos definir o rótulo de um caminho em T como o resultado da concatenação dos arcos no caminho; isto é, o rótulo do caminho $P = \langle p_1, \ldots, p_m \rangle$ é $\lambda(P) = \lambda(p_1p_2) \cdot \lambda(p_2p_3) \ldots \lambda(p_{m-1}p_m)$. Por definição, o rótulo de um nó é o rótulo do caminho da raiz até o nó. Também por definição, o caminho de r até v contém u . Logo,

$$\phi(v) = \lambda(r \rightsquigarrow u).\lambda(u \rightsquigarrow v) = \phi(u).\lambda(u \rightsquigarrow v).$$

Concluímos que $\phi(u)$ é prefixo de $\phi(v)$.

Para provar a suficiência, tomemos novamente dois nós $u, v \in N$ e suponhamos que $\phi(u)$ seja prefixo de $\phi(v)$. A prova é por indução em $|\phi(u)|$. Se $|\phi(u)| = 0$, então u é raiz da árvore e, portanto, ancestral de v. Agora, vamos supor $|\lambda(u)| \geq 1$ e a validade da hipótese de indução. Seja w o pai de u em T. Pela prova da necessidade, $\phi(w)$ é prefixo de $\phi(u)$ e, por conseguinte, prefixo de $\phi(v)$. Como $|\phi(w)| < |\phi(u)|$, pela hipótese de indução w é ancestral de v. Seja a a primeira letra de $\lambda(wu)$. Como $\phi(w)$ é prefixo

de ambos, $\phi(u)$ e $\phi(v)$, e $\phi(u)$ é prefixo de $\phi(v)$, temos que $\lambda(wu)$ é prefixo de $\lambda(w \rightsquigarrow v)$. Portanto, a também é primeira letra do primeiro arco no caminho $w \rightsquigarrow v$, digamos wx. Porém, como wx e wu partem de um mesmo nó, nessa caso w, a propriedade $(p2)$ assegura-nos que $wx = wu$, o que implica que $x = u$. Logo, o único caminho de $r \rightsquigarrow v$ contém u, seguindo imediatamente que u é ancestral de v. □

Desse teorema, seguem os seguintes corolários.

Corolário 16. *Para todo par de nós u e v, $u = v$ se, e somente se, $\phi(u) = \phi(v)$;*

Corolário 17. *para todo par de nós distintos u e v, u é ancestral de v se, e somente se, $\phi(u)$ é prefixo próprio de $\phi(v)$.*

Definição 18. Definimos um **lugar** em uma árvore dos sufixos T como um par ordenado (u, Y), tal que u é nó de T e Y é prefixo próprio do rótulo de um arco que sai de u.

Vamos estender um pouco a definição da função ϕ para definir o rótulo de um lugar, como segue:

$$\phi(u, Y) \quad = \quad \phi(u) \cdot Y.$$

Dizemos que (u, Y) é lugar da cadeia X em T se u é o nó mais profundo em T cujo rótulo é prefixo de X e $\phi(u, Y) = X$.

Nesse trabalho, adotamos a representação compacta da árvore dos sufixos, seguindo uma certa tendência da literatura [dLS03, Gus97, CR94], em vez da representação não-compacta. Na representação não-compacta, o rótulo de cada arco possui comprimento unitário e, portanto, as definições de lugar e lugar da cadeia tornam-se perfeitamente dispensáveis. A escolha deu-se em razão da representação compacta adequar-se melhor à implementações eficiente da estrutura, facilitando a discussão dos detalhes de mais baixo nível. Por um outro lado, a representação compacta requer que apresentemos, além das citadas, as definições de transição e adjacência entre lugares, como segue.

Definição 19. Dados dois lugares $g = (u, Y)$ e $h = (v, Z)$, dizemos que existe **transição** t de g para h, denotada $t = g \rightarrow h$, se existe caractere $a \in (\Sigma \cup \{\$\})$ tal que $\phi(g).a = \phi(h)$. Nesse caso, o rótulo de t é $\lambda(t) = a$.

Definição 20. Dois lugares $g = (u, Y)$ e $h = (v, Z)$ são **adjacentes** se existe transição $g \to h$ ou $h \to g$.

Definição 21. Dado um lugar h na árvore dos sufixos, vamos definir $Pai(h)$ como sendo o lugar g tal que existe transição $g \to h$.

Teorema 22. *Seja T árvore dos sufixos de uma cadeia X e u um nó interno de T. Para todo lugar (u, Y) em T, e toda folha v descendente de (u, Y), é verdade que $|X| - |\phi'(v)| + 1$ é posição de uma ocorrência de $\phi(u, Y)$ em X.*

Demonstração. Pela propriedade $(p3)$, $\phi(v)$ é sufixo de $X\$$, seguindo imediatamente que $\phi'(v)$ é sufixo de X. Logo, $\phi'(v)$ ocorre em X na posição $|X| - |\phi'(v)| + 1$. Como u é ancestral de v, temos, pelo teorema 15, que $\phi(u)$ é prefixo próprio de $\phi(v)$, do que segue imediatamente que $\phi(u)$ e prefixo de $\phi'(v)$. Como Y é prefixo próprio de um arco que sai de u, e v é descendente de (u, Y), logo esse mesmo arco pertence ao caminho $u \rightsquigarrow v$, vale afirmar que $Y \neq \$$ e $\phi(u, Y)$ é prefixo de $\phi'(v)$. Portanto, $\phi(u, Y)$ também ocorre em X na posição $|X| - |\phi'(u)| + 1$. □

A construção de uma árvore dos sufixos não é tarefa trivial, tão pouco o é a análise dos diversos algoritmos existentes para a tarefa, que pode ser realizada em tempo e espaço $O(n)$, onde n é o número de símbolos em X. Desde a invenção do algoritmo pioneiro de Weiner [Wei73], diversos algoritmos foram propostos para sua construção, destacando-se os algoritmos propostos por Ukkonen [Ukk95] e McCreight [McC76]. Não entraremos em maiores detalhes sobre algoritmos de construção da árvore dos sufixos. Para o leitor interessado, recomendamos as referências apresentadas na seção "Comentários Bibliográficos" do presente capítulo.

2.2.1 Árvore dos Sufixos e a Busca Exata de Padrões

Dado $T = (N, A, \lambda)$, árvore dos sufixos de uma cadeia $X = x_1 \dots x_n$, verificar se $P = p_1 \dots p_m$ é fator de X é tarefa fácil: basta localizar o lugar em T cujo rótulo é o maior prefixo comum entre P e um sufixo de X; se tal prefixo tiver comprimento máximo, isto é, for igual a P, então P é fator de X. Em outras palavras, estamos verificando a pertinência do lugar de P na árvore e, caso tal lugar exista, determinando a sua localização na estrutura. Chamamos esse procedimento de **soletrar** P em T. Em detalhes,

partindo-se da raiz de T e varrendo-se P da esquerda para a direita, deve-se percorrer na árvore o caminho determinado por P, como segue: o próximo nó a ser visitado, a partir do nó atual u, é v, adjacente a u e tal que o rótulo do arco (u, v) é prefixo do restante de P que ainda não foi lido até o momento. Se não existir tal arco, então o lugar a ser devolvido é (u, Y), onde Y é o maior prefixo do rótulo de um arco que sai de u e que seja prefixo do restante de P, podendo ser a cadeia vazia. Se existe transição com origem em (u, Y) e rótulo \$, então P, além de fator, é sufixo de X (veja algoritmo 2.1) De modo bastante natural, o algoritmo SOLETRA pode ser adotado como sub-rotina de

Algoritmo 2.1 Algoritmo SOLETRA, que recebe uma cadeia P e uma árvore dos sufixos T, de uma cadeia X, e devolve o lugar (u, Y) em T cujo rótulo é o maior prefixo comum a P e um sufixo de X. SOLETRA também devolve o nó v, ponta do arco associada a (u, Y).

SOLETRA$(P_{1..m}, T)$

1 $u \leftarrow$ raiz de T
2 $P' \leftarrow P$
3 **enquanto** $|P'| > 0$ e $\exists e = (u, v) \in A | \lambda(e) \in \mathsf{Pf}(P')$ **faça**
4 $\quad P' \leftarrow \lambda(u, v)^{-1} P'$
5 $\quad u \leftarrow v$
6 **se** $|P'| > 0$ e $\exists e = (u, v) \in A | p'_1 \in \mathsf{Pf}(e)$
7 **então** $Y \leftarrow$ maior prefixo comum a $\lambda(e)$ e P'
8 **senão**
9 $\quad Y \leftarrow \epsilon$
10 $\quad v \leftarrow u$
11 **devolva** $((u, Y), v)$

um algoritmo que localiza as posições iniciais de ocorrências exatas de P em X. Basta, partindo do lugar (u, Y), realizar uma busca, em largura ou profundidade, na árvore e, para cada folha descendente desse lugar, calcular, com base no teorema 22, a posição da ocorrência do padrão determinada por essa folha, como podemos ver no algoritmo 2.2. Analisando esse mesmo algoritmo em maiores detalhes, podemos ver que o conjunto V é utilizado apenas para marcar os nós já visitados durante a busca na árvore, e pode ser implementado através de um simples vetor, consumindo espaço e tempo

de inicialização $O(n)$ e tempo de atualização $O(1)$ (linhas 8 e 15). Logo, o consumo de tempo do algoritmo é ditado pelo consumo de tempo da linha 1 e da chamada à função SOLETRA. A linha 1 consome tempo $O(n)$, visto que a árvore dos sufixos pode ser construída em tempo linear no comprimento da cadeia. Analisando SOLETRA com atenção, podemos ver que o consumo de tempo do algoritmo é dependente do tempo necessário para decidir a satisfazibilidade das condições das linhas 3 e 6. Se conseguirmos, dado um nó u, encontrar, em tempo constante, qual arco sai de u e tem rótulo iniciado com um determinado caractere a, então a linha 3 pode ser decidida em tempo linear no comprimento do rótulo do arco e e a linha 6 em tempo constante, nesse caso a chamada à SOLETRA, realizada por BUSCA-EXATA-AS, consome tempo $O(m)$, e o consumo de tempo total, por parte de BUSCA-EXATA-AS, resulta $O(m+n)$. Para que tudo isso seja possível, basta armazenar, para cada nó interno da árvore, um vetor de arcos indexado pelos elementos em Σ; o elemento indexado pelo caractere a, no vetor definido para um determinado nó u, irá apontar para algum nó caso exista arco saindo de u e com rótulo iniciado por a. Caso tal arco não exista, esse elemento do vetor é nulo. Além do ponteiro para o próximo nó, é conveniente que esse mesmo elemento do vetor também contenha os índices da cadeia X que representam o rótulo do arco (u, v). Essa estratégia faz com que toda a estrutura ocupe espaço $\Omega(n|\Sigma|)$. Uma outra alternativa consiste na substituição do vetor de arcos, anteriormente empregado, por uma árvore binária de busca balanceada, que iria consumir espaço $O(k)$ para cada nó u, onde k é o número de filhos de u. Cada operação de busca ou inserção de filhos de u consome espaço $O(\lg k)$. Note que $O(\lg k) = O(\lg |\Sigma|)$. Essa alternativa faz sentido quando k, e conseqüentemente Σ, são grandes o suficiente para justificar a manutenção da árvore de busca. Há outras alternativas, mas de modo geral, apesar de assintóticamente linear, o consumo de espaço das implementações existentes de árvores dos sufixos é elevado, não só em razão do espaço necessário para representar os arcos que saem de cada nó, mas também do espaço necessário para representar os rótulos dos arcos e dos nós da árvore. A implementação de Kurtz [Kur99], a mais econômica conhecida até o momento, ocupa $20n$ bytes de memória total[1], porém sua aplicação está restrita a cadeias com comprimento inferior a 135 milhões de caracteres, e sua adaptação para cadeias que ultrapassem esse limite resultaria em um algoritmo com consumo de tempo significativamente superior. Na verdade, as necessidades de espaço da árvore dos sufixos tornam a sua adoção impraticável em determinadas aplicações, especialmente naque-

[1]Supomos aqui uma máquina com palavras de 32 bits.

las onde o tamanho do alfabeto é grande. Além disso, a construção de uma árvore dos sufixos em tempo linear se dá a partir de ligações de sufixos, elementos estruturais que permitem que o algoritmo se movimente rapidamente entre pontos da árvore distantes entre si, o que complica bastante o trabalho de paginação de memória que deverá ser realizado pelo Sistema Operacional caso a árvore não possa ser mantida inteiramente em memória. Para contornar essas deficiências apresentadas pela árvore dos sufixos, Manber e Myers [MM93] propuseram o vetor dos sufixos, uma estrutura de dados mais espaço-econômica e que fornece, quando adotada isoladamente, parte das funcionalidades da árvore dos sufixos, podendo substituí-la por completo se adotado em conjunto com uma tabela auxiliar [AKO04]. Ainda não se sabe se um vetor de sufixos pode substituir completamente uma árvore dos sufixos sem a necessidade de alguma estrutura auxiliar e sem acréscimo no consumo de tempo das operações realizadas sobre a estrutura [AKO04], apesar de improvável que isso seja possível.

2.3 O Vetor dos Sufixos

Um modo bastante intuitivo de representarmos $\mathsf{Sf}(X)$, para uma cadeia X, consiste na construção de uma estrutura de acesso aleatório contendo todos os sufixos de X. É claro que uma construção ingênua dessa estrutura, na forma de um matriz $n \times n$, consome espaço $\Omega(n^2)$, entretanto, após certa reflexão, fica evidente que podemos implementá-la na forma de um vetor, consumindo apenas espaço $O(n)$, como segue.

Definição 23. O **vetor dos sufixos** de uma cadeia $X = x_1 \dots x_n$ é um vetor de inteiros $V[1..n{+}1]$, tal que $X_{V[j]\dots n} \preceq X_{V[j+1]..n}$, para $1 \leq j \leq n$, onde $\preceq$ determina precedência lexicográfica ou igualdade entre cadeias de caracteres.

O vetor dos sufixos de X codifica todos os sufixos da cadeia: $V[i]$ representa o sufixo $X_{V[i]..n}$ de X, para $1 \leq i \leq n+1$. Logo, os sufixos encontram-se lexicograficamente ordenados no vetor, o que permite-nos verificar se uma cadeia Y é fator de X em tempo $O(|Y| \log |X|)$: basta realizarmos uma busca binária no vetor. Em maiores detalhes, seja $\mathsf{Pf}_q(X)$ o prefixo de X de comprimento q ou a própria cadeia X caso $|X| \leq q$; sejam $\prec_q$ e $\preceq_q$ operadores que determinam relação de ordem lexicográfica entre os prefixos de comprimento q dos operandos, isto é, $X \prec_q Y$ se e somente se $\mathsf{Pf}_q(X) \prec \mathsf{Pf}_q(Y)$, e de modo análogo estão definidos os operadores $\preceq_q$ e $=_q$. Supo-

Algoritmo 2.2 Recebe duas cadeias, P e X, e imprime as posições de ocorrências exatas de P em X através do processamento da árvore dos sufixos de X .

BUSCA-EXATA-AS$(P_{1..m}, X_{1..n})$

```
 1  T ← ÁRVORE-DOS-SUFIXOS(X)
 2  ⟨(u, Y), v⟩ ← SOLETRA(P, T)
 3  se φ(u, Y) ≠ P
 4    então
 5      imprima "P não ocorre em X."
 6    senão
 7      S ← {v}
 8      V ← {v}
 9      enquanto S ≠ ∅ faça
10            escolha um nó s de S
11            se s é folha de T
12              então imprima "P ocorre em X na posição |X| − |φ'(s)| + 1"
13            se ∃(s, w) ∈ A tal que w ∉ V
14              então
15                V ← V ∪ {w}
16                S ← S ∪ {w}
17              senão
18                S ← S\{s}
```

nha que desejemos encontrar as ocorrências da cadeia $P = p_1 \dots p_m$ em $X = x_1 \dots x_n$ e seja $V[1..n+1]$ o vetor dos sufixos de X. P ocorre em X se P é prefixo de algum sufixo de X; isto é, se $P = \mathsf{Pf}_m(x_i \dots x_n)$ para algum $1 \leq i \leq n - m + 1$. A ordem lexicográfica do vetor nos garante que se $P \prec_m x_{V[i]} \dots x_n$ então P não é prefixo de nenhum dos sufixos em $V[i+1\,..\,n]$, só podendo ocorrer como prefixo de algum sufixo em $V[1\,..\,i]$, visto que $P \prec x_{V[i]} \dots x_n \iff P \prec_m x_{V[i]} \dots x_n$; informação análoga é inferida se $P \succ_m x_{V[i]} \dots x_n$. Além disso, se $P =_m X_{V[i]..n}$ e $P =_m X_{V[j]..m}$, para $i \leq j$, então toda posição $V[i\,..\,j]$ de X é ocorrência de P, e se i e j são, respectivamente, menor e maior valor para os quais tais igualdades valem, então toda e qualquer ocorrências de P em X está em $V[i\,..\,j]$. Logo, o problema se resume a encontrar, através de busca binária, os valores l_P e r_P tais que $l_P = \min\{i : P =_m X_{V[i]..n}\}$ e $r_p = \max\{j : P =_m X_{V[j]..m}\}$. O algoritmo 2.3, apresentado na página 27, devolve o valor de l_p de acordo com os parâmetros fornecidos; processo semelhante devolve o valor de r_p. O algoritmo final (algoritmo 2.4) é apresentado na página 30. O consumo de tempo do algoritmo é determinado pelo tempo de construção do vetor dos sufixos e o consumo das chamadas aos métodos LIMITE-ESQUERDO e LIMITE-DIREITO, e resulta $O(n + m \lg n)$. Como cada elemento do vetor armazena apenas a posição de início do sufixo, é possível implementar a estrutura ocupando, por instância, apenas $4n$ bytes de memória para cadeias de até $4GB$ caracteres de comprimento[2]. Em comparação com a árvore dos sufixos, o vetor dos sufixos apresenta-se como uma solução muito mais econômica, principalmente em se tratando de espaço, mas também quando levamos em consideração o tempo de construção: apesar de assintoticamente equivalentes, na prática o tempo de construção do vetor dos sufixos é significativamente inferior ao tempo de construção da árvore dos sufixos [BK03b]. Porém, ainda não se sabe como extrair do vetor dos sufixos, com o mesmo consumo de tempo, a funcionalidade fornecida pela árvore dos sufixos, sem que para isso seja necessário a adoção de alguma estrutura de dados auxiliar; conjectura-se fortemente que isso não seja possível [AKO04]. Abouelhoda e colegas [AKO04] demonstraram que qualquer problema solucionável através de uma árvore dos sufixos também é solucionável, com o mesmo consumo de tempo, através de um vetor dos sufixos quando adotado em conjunto com duas tabelas auxiliares, as tabelas do maior prefixo comum e de ancestrais, que juntas ocupam $3n$ bytes de espaço adicional.

[2]Novamente, assumimos uma máquina de 32 bits.

Algoritmo 2.3 Recebe duas cadeias, P e X, e V, vetor dos sufixos de X. O algoritmo realiza busca binária em V e devolve $l_P = \min\{i : P =_m X_{V[i]..n}\}$, que é adotado pelo algoritmo 2.4 para encontrar as ocorrências do padrão no texto através do vetor dos sufixos.

LIMITE-ESQUERDO(P, X, V)

```
1  se P >_m X_{V[n]..n}
2    então devolva n + 1
3  l ← 1
4  r ← n
5  enquanto l < r faça
6     q ← ⌊(l + r)/2⌋
7     se P ⪯_m X_{V[q]..n}
8       então r ← q
9       senão l ← q + 1
10 l_p ← r
11 devolva l_p
```

2.3.1 Algoritmos para a Construção do Vetor dos Sufixos

A obtenção de um vetor dos sufixos, a partir de uma árvore de sufixos e em tempo linear no comprimento da cadeia, é tarefa trivial [Gus97]. Infelizmente, essa estratégia requer memória disponível para abrigar toda a árvore, reduzindo as vantagens da adoção do vetor dos sufixos em substituição à árvore dos sufixos. Porém, diversos algoritmos são capazes de construir, diretamente, um vetor dos sufixos para uma dada cadeia. Puglisi [PST07] classifica tais algoritmo em três categorias principais, sobre as quais faremos algumas breves considerações a seguir:

- **Algoritmos de Duplicação de Prefixos:** algoritmos que se enquadram nessa categoria baseiam-se fortemente em uma idéia apresentada em 1972 por Karp, Miller e Rosenberg [KMR72], e ordenam os sufixos da cadeia a partir de uma seqüência de etapas de refinamentos. Cada etapa consome tempo linear no número de sufixos e consiste em, a partir dos sufixos ordenados lexicograficamente em relação aos seus primeiros k caracteres, ordená-los em relação aos primeiros $2k$ caracteres. No total, há $O(\lg n)$ etapas de refinamento, logo o consumo de tempo dos algoritmos de duplicação de prefixos é $O(n \lg n)$. O algoritmo Manber-Myers, proposto no artigo original, em 1993, pertence a esse grupo e a cada etapa de refinamento ordena os sufixos através de um algoritmo de ordenação misto dos algoritmos radix-sort e bucket-sort. Outro algoritmo nessa categoria é o algoritmo Larsson-Sadakane [LS99], de 1999, que utiliza a variação do algoritmo quicksort apresentada por Bentley e McIlroy [BM93]. Apesar do consumo assintótico de tempo elevado, o algoritmo Larsson-Sadakane é, na prática, mais eficiente do que os algoritmos lineares de Kärkkäinen e Sanders, Ko e Aluru, e Kim e colegas [PST05] e, geralmente, dez vezes mais eficiente do que o algoritmo Manber-Myers. Devido a sua robustez, Seward [Sew00], assim como Puglisi e colegas [PST05], afirmam que ferramentas de propósito geral devem adotar o algoritmo Larsson-Sadakane como algoritmo auxiliar sempre que algum caso patológico do algoritmo principal ocorrer, como o faz o poderoso pacote de compactação *bzip2* [Sew07].

- **Algoritmos Recursivos:** algoritmos recursivos para a construção do vetor dos sufixos são assim chamados por dividirem os sufixos da cadeia em dois grupos, ordenar recursivamente um desses grupos e utilizar a ordenação obtida para ordenar os sufixos da parte restante, intercalando ambas em seguida. Essa classe

de algoritmos nasceu no ano de 2003, e toma como base o interessante trabalho de Farach [Far97], onde foi proposto um algoritmo para a construção de árvores dos sufixos para alfabetos grandes. Todos os algoritmos com consumo de tempo linear, conhecidos até o momento, enquadram-se nessa categoria; dentre estes, os três primeiros foram propostos simultânea e independentemente no ano de 2003, por Kärkkäinen e Sanders [KS03], Ko e Aluru [KA03] e Kim e colegas [KSPP03]. Infelizmente, o menor consumo assintótico de tempo dos algoritmos lineares para a construção do vetor dos sufixos ainda não se refletiu em um melhor desempenho em testes experimentais; na prática o desempenho de tais algoritmos deixa bastante a desejar quando comparados com alguns outros algoritmos de maior consumo assintótico [PST05, PST07].

- **Algoritmos de Cópia Induzida:** Essa categoria foi inaugurada com a criação, em 2004, do algoritmo de Seward [Sew00] e, assim como os algoritmos recursivos, os algoritmos dessa classe se baseiam na premissa de que uma ordenação completa de um subconjunto dos sufixos pode ser adotada para acelerar a ordenação dos sufixos restantes. A diferença, bastante sutil por sinal, está no fato desses algoritmos executarem de modo iterativo em vez de recursivamente, além de apresentarem desempenho bastante superior, fato ilustrado pelo algoritmo de Maniscalco e Puglisi [MP07], tido como o método mais eficiente para a construção do vetor dos sufixos dentre os conhecidos até o momento. Grande parte dos algoritmos de cópia induzida operam de modo heurístico e essa é uma das razões pela qual seu desempenho na prática não reflete o consumo assintótico de tempo declarado pelos seus autores. De modo geral, em testes experimentais com entradas de interesse prático, os algoritmos de cópia induzida se saem melhor do que os demais algoritmo [PST07].

Em um trabalho publicado enquanto essa dissertação encontrava-se nos estágios finais de redação, Puglisi e colegas [PST07] realizaram um estudo comparativo dentre diversos algoritmos disponíveis para a construção do vetor dos sufixos, cujos resultados principais encontram-se, parcialmente, expostos no quadro a seguir:

Algoritmo 2.4 Recebe duas cadeias P e X e imprime as ocorrências exatas de P em X, através da construção e processamento do vetor dos sufixos de X. O algoritmo consome tempo $O(n + m \lg n)$

BUSCA-EXATA-VS(P, X)
1 $V \leftarrow$ VETOR-DOS-SUFIXOS(X)
2 $l_p \leftarrow$ LIMITE-ESQUERDO(P, X, V)
3 $r_p \leftarrow$ LIMITE-DIREITO(P, X, V)
4 **para** $i \leftarrow l_p$ **até** r_p **faça**
5 imprima "P ocorre em X na posição $V[i]$".

	Algoritmo	Tempo (Assintótico)	Tempo de Execução	Espaço
Duplicação de Prefixo				
	Manber e Myers	$O(n \lg n)$	30	$8n$
	Larsson e Sadakane	$O(n \lg n)$	3	$8n$
Recursivos				
	Ko e Aluru	$O(n)$	2,5	$10n$
	Kim e colegas	$O(n)$	–	–
	Kärkkäinen e Sanders	$O(n)$	4,7	$13n$
Cópia Induzida				
	Maniscalco e Puglisi	$O(n^2 \lg n)$	1	$6n$
	Burkhardt e Kärkkäinen	$O(n \lg n)$	3,5	$6n$
	Seward	$O(n^2 \lg n)$	3, 5	$5n$
	Manzini e Ferragina	$O(n^2 \lg n)$	1,7	$5n$

O tempo de execução é apresentado como um fator multiplicativo sobre o tempo de execução do algoritmo de Maniscalco e Puglisi [MP07], o mais rápido algoritmo dentre todos os algoritmos testados. É importante notar que Maniscalco e Puglisi não demonstraram a validade do limite assintótico apresentado: o mesmo é apenas conjecturado.

2.4 Comentários Bibliográficos

A primeira referência à árvore dos sufixos na forma como a vemos atualmente, assim como a autoria do primeiro algoritmo linear para a sua construção, é atribuído a Weiner [Wei73], porém as raízes históricas da árvore dos sufixos remontam a duas estruturas de dados propostas anos antes: o **Trie**, proposta por Edward Fredkin [Fre60], em 1960, e a **Patricia Trie**, proposta por Donald Morrison [Mor68] em 1968. Uma Patricia Trie, quando aplicada ao conjunto dos sufixos de uma cadeia X, é o que aqui chamamos de árvore dos sufixos de X em sua forma compacta, enquanto a Trie, quando aplicada ao mesmo conjunto, é a árvore dos sufixos de X, porém apresentada na forma não compacta. Três anos depois do artigo de Weiner, McCreight [McC76] propôs uma solução com menor consumo de espaço, e em 1993 Ukkonen [Ukk93, Ukk95] publicou um algoritmo *on-line* com o mesmo consumo de espaço do algoritmo de McCreight, discretamente menos eficiente em termos práticos, porém de demonstração e análise mais simples. Uma comparação, em termos práticos e teóricos, dos três algoritmos citados pode ser encontrada em Giegerich e Kurtz,[GK97], enquanto Kurtz [Kur99] discute diferentes estratégias espaço-econômicas para a implementação de árvores dos sufixos. A propósito, Burkhardt e Kärkkäinen afirmam que a implementação de Kurtz ocupa entre $8n$ e $14n$ bytes de memória, em uma maquina de 32 bits [BK03b]. Essa afirmação, até onde nossa compreensão nos permite afirmar, está em desacordo com o exposto por Kurtz.

O ano de 2003 foi o divisor de águas na pesquisa envolvendo vetores dos sufixos: foi nesse ano que surgiram os primeiros algoritmos lineares para a construção dessa estrutura de dados e, desde então, viu-se uma verdadeira explosão de artigos relacionados ao tema. De quando foi proposto, em 1993, por Manber e Myers [MM93], até 2003, todos os algoritmos que construíam um vetor dos sufixos, sem a construção prévia de uma árvore de sufixos, o faziam com consumo de tempo $\Theta(n \log n)$. Porém, em 2003, Kärkkäinen e Sanders [KS03], Ko e Aluru [KA03] e Kim e colegas [KSPP03], independentemente, propuseram um algoritmo que constrói, diretamente, um vetor de sufixos com consumo de tempo e espaço adicional proporcional a $O(n)$. No mesmo ano, Burkhardt e Kärkkäinen [BK03b] apresentaram um modo de construir um vetor dos sufixos consumindo tempo $O(n \log n)$, porém exigindo memória adicional apenas sublinear. Em 2007, Maniscalco e Puglisi [MP07] publicaram o que é hoje tido como o melhor algoritmo, em termos práticos para a construção dessa estrutura de dados. A

literatura sobre o vetor dos sufixos confunde-se com a literatura que aborda algoritmos de compressão de dados através da transformada de Burrows-Wheeler (BWT) [BW94]. Isso ocorre em virtude da ordenação dos sufixos do texto a ser comprimido ser a etapa mais custosa dessa transformada, portanto qualquer redução no tempo de construção do vetor dos sufixos representa uma redução no tempo de execução de tais algoritmos. Como demonstra a intensidade com que novos resultados têm aparecido, a pesquisa dessa estrutura de dados encontra-se em franca atividade e é muito provável que significativos avanços sejam alcançados nos próximos anos.

Capítulo 3

Busca Aproximada

3.1 Introdução

Nesse capítulo, vamos nos ater ao problema da **busca aproximada de padrões em cadeias**. Sob uma visão geral, e informal, o objetivo é apontar ocorrências de uma palavra em um texto que, de algum modo, tenha sido corrompido. É claro que o problema pode ser formulado de diversas formas. Na que estamos interessados no momento, o objetivo é encontrar as partes do texto cuja distância do padrão, sob uma métrica fornecida, esteja abaixo de um certo limiar, também fornecido. A dificuldade do problema está intimamente relacionada à métrica adotada e varia de problemas solucionáveis em tempo linear a problemas NP-DIFÍCEIS. Claramente, há tendência na literatura em se discutir o problema sobre as distâncias de Hamming e Levenshtein; isto se dá em razão dessas distâncias modelarem satisfatoriamente uma grande gama de problemas e aplicações, sem que isso incorra em uma dificuldade computacional intolerável [Nav01].

Definição 24 (Ocorrência aproximada de um padrão)**.** Dadas duas cadeias $P = p_1 \dots p_m$ (o padrão) e $X = x_1 \dots x_n$ (o texto), uma métrica d e um inteiro r, dizemos que $X_{i..j}$ é (d, r)**-ocorrência** de P em X se $d(P, X_{i..j}) \leq r$. Também, dizemos que P (d, r)**-ocorre** em X e i e j são, respectivamente, posição inicial e final de uma (d, r)**-ocorrência** de P em X.

`Exemplo:` As cadeias são $P =$ `acat` e $X =$ `acgtacacatg`, a métrica é Levenshtein e $r = 2$. Nesse caso, as posições 1, 5 e 7 são início de (d_L, r)-

ocorrências de P em X, e as posições 4, 8 e 10, final.

Agora, estamos prontos para enunciar o problema abaixo:

Problema 25 (Busca Aproximada de Padrões em Cadeias)**.** *Dadas as cadeias X (o texto) e P (o padrão), uma métrica d e um inteiro r, encontrar todas as* ***posições finais*** *de (d, r)-ocorrências de P em X.*

Vamos considerar duas variações para a busca aproximada de padrões. Basicamente, elas diferem entre si apenas nos critérios adotados para a análise de uma eventual solução para o problema.

Busca *on-line*: Também conhecida como **busca aproximada de padrões em textos dinâmicos**, essa variante do problema contabiliza, como parte do tempo gasto para a realização da busca, o tempo consumido por todo e qualquer processamento do texto. Todo programa-aplicativo de dimensões suficientemente grandes é uma aplicação em potencial da busca em textos dinâmicos. Bons exemplos são os editores de textos, ambientes integrados de desenvolvimento, navegadores de páginas na Internet e reprodutores de formatos de arquivos de áudio.

Busca *off-line*: Essa variante, também conhecida como **busca aproximada de padrões em textos estáticos**, permite o pré-processamento do texto fornecido; isto é, o tempo consumido para extrair previamente informações do texto, e construir estruturas de dados que armazenem tais informações (ditas **estruturas de índices**), não é computado como parte do tempo gasto para a realização da busca. A busca em textos estáticos é de importância fundamental em se tratando de aplicações da biologia computacional e busca de páginas na Internet. Intuitivamente, podemos imaginar o texto como uma cadeia armazenada em uma base de dados, sobre a qual diversas operações de busca, de padrões possivelmente dois-a-dois distintos, serão realizadas. Mesmo que, eventualmente, o tempo gasto durante o pré-processamento esteja longe de ser desprezível, ele será amortizado pelo número total de operações de busca.

Esse capítulo aborda a busca em textos dinâmicos e discute, em detalhes, o algoritmo de Myers [Mye99], enquanto a busca em textos estáticos será discutida no capítulo 5, quando então adotaremos algoritmos para a busca em textos dinâmicos como sub-rotinas para os algoritmos que veremos.

Figura 3.1 Alguns dos mais populares algoritmos para a busca aproximada de padrões. O consumo de tempo do algoritmo de Ukkonen refere-se ao consumo de tempo de pior caso e de caso médio. Ademais, leia-se: PD: Programação Dinâmica; BP: Bit Paralelismo; H: Heurística; 4R: Método dos Quatro Russos; NFA: Autômato não-determinístico; SA: Autômato dos Sufixos; ST: Árvore dos Sufixos; w: Comprimento da palavra de máquina.

Trabalho	Abordagem	Consumo de Tempo	Consumo de Espaço
Sellers, 1980	PD	$\Theta(nm)$	$O(nm)$
Masek e Paterson, 1980	PD + 4R	$O(nm/\log n)$	$O(nm)$
Ukkonen, 1985	PD + H	$\Theta(nm)$ — $O(rn)$	$O(nm)$
Wu e Manber, 1992	NFA + BP	$O(rnm/w)$	$O(mr)$
Ukkonen e Wood, 1993	PD + SA	$O(m^2)$ — $O(rn)$	$O(m)$
Chang e Lawler, 1994	PD + ST	$O(rn)$	$O(m)$
Myers, 1999	PD + BP	$O(nm/w)$	$O(m)$

3.2 O Bit-Paralelismo

Em 1999, Eugene W. Myers [Mye99] desenvolveu um algoritmo para a busca aproximada de padrões com base no preenchimento bit-paralelo de uma tabela de programação dinâmica. O bit-paralelismo[1] é uma técnica de programação cuja autoria é atribuida a Baeza-Yates em sua tese de doutorado [BY89] defendida em 1989, apesar de Van Emde Boas ter empregado a técnica quase vinte e cinco anos antes [vEBKZ76]. Basicamente, o bit-paralelismo consiste na exploração da capacidade dos computadores modernos de executar, em tempo constante, operações básicas em cadeias binárias de comprimento igual ou inferior ao de uma palavra da máquina, que aqui denotaremos por w. Com certa dose de engenhosidade, essa técnica permite reduzir o consumo de tempo de certos algoritmos por um fator w, que geralmente é igual a 32 ou 64, podendo chegar a 128 em alguns processadores modernos e para um conjunto restrito de operações, e ainda tende a aumentar com o avanço tecnológico. A paralelização de tais operações pode ser aplicada para promover a aceleração da execução de uma

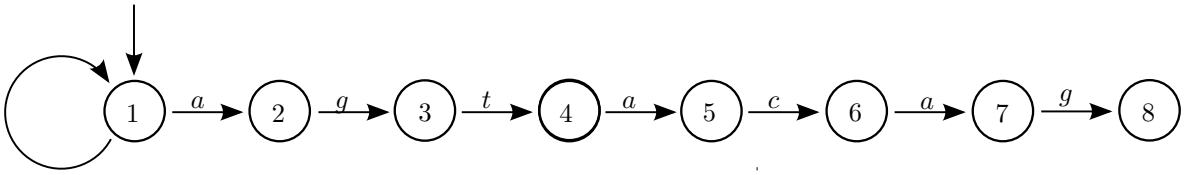
[1] Também é possível encontrar na literatura os termos paralelismo em nível de palavra ⟨ *word level parallelism* ⟩, paralelismo em nível de bits ⟨ *bit-level parallelism* ⟩ e paralelismo interno a palavra ⟨ *within-word parallelism* ⟩.

grande gama de algoritmos, inclusive algoritmos de busca e ordenação, mas é aplicada com maior freqüência em algoritmos que simulam autômatos, determinísticos ou não, ou que adotam programação dinâmica. Apesar de, algumas vezes, a adoção de bit-paralelismo não representar uma melhora no consumo assintótico de tempo do algoritmo, quando em comparação com a versão sequencial (não bit-paralelizada), algoritmos bit-paralelos costumam estar entre os algoritmos mais eficientes na prática em se tratando de problemas relacionados a busca de padrões [Nav01].

3.3 O Modelo Computacional

Modelo computacional é a definição formal de um computador abstrato. Um modelo computacional deve especificar quais instruções o computador é capaz de executar e o custo computacional, em tempo e espaço, inerente a cada uma dessas operações. É desejável que um modelo computacional seja abstrato o suficiente para simplificar o processo de contagem do número de operações executadas pelos algoritmos escritos para o modelo. Porém, por vezes, é preciso que o modelo reflita com certo grau de fidelidade a um computador real. Logo, a escolha de um modelo representa um equilíbrio entre realidade e tratabilidade matemática. Até o momento, os algoritmos apresentados neste trabalho assumiam o modelo RAM (Máquina de Acesso Aleatório). As instruções possíveis de serem executadas em uma máquina RAM, consumindo tempo e espaço constante, são: matemáticas (adição, subtração, multiplicação, divisão e resto de divisão inteira), movimentação de dados (atribuição e cópia) e de controle (comparação, repetição, chamada de sub-rotina e devolução de valor). Algoritmos bit-paralelos assumem um modelo um pouco mais próximo dos computadores reais modernos, o modelo w-RAM [Hag98, vEBKZ76]. No modelo w-RAM cada palavra da memória do computador é composta por w bits, portanto capaz de manter um inteiro no intervalo $[-2^{w-1} \ldots 2^{w-1} - 1]$. As mesmas instruções do modelo RAM também podem ser executadas em uma máquina w-RAM, porém elas consomem tempo constante mesmo quando executadas sobre cadeias binárias de comprimento w. Até onde o nosso conhecimento nos permite afirmar, todos os computadores modernos adotados em escala razoável são compatíveis com o modelo w-RAM para algum comprimento de palavra não unitário. Tem havido uma onda crescente de interesse no desenvolvimento de algoritmos para esse modelo e que exploram em profundidade o conjunto de operações

Figura 3.2 Autômato-Não-Determinístico (AND) que verifica ocorrências (exatas) do padrão `agtacag` na cadeia varrida pelo autômato. Para maior clareza, foram omitidas arestas de retorno ao estado inicial.



e o comprimento de palavra; esses algoritmos são ditos bit-paralelos. É interessante observar que o interesse por algoritmos bit-paralelos se estende a problemas clássicos da computação, como os problemas da busca e ordenação: uma resenha publicada em 1998 [Hag98] cita 20 artigos publicados no período de três anos compreendido entre 1995 e 1998 e que abordam esses dois problemas sob o modelo w-RAM. No que tange o problema que estamos vendo, Grabowski e Fredriksson [GF08] afirmam que a técnica se tornou uma das mais populares.

3.4 O Algoritmo Baeza-Yates – Gonnet

Antes de introduzir o algoritmo de Myers, na próxima seção, vamos fazer algumas considerações sobre bit-paralelismo e ilustrar a técnica com o algoritmo **Shift-Or**, de Baeza-Yates e Gonnet [BYG89], que determina as ocorrências exatas de uma cadeia $P = p_1 \dots p_m$ (o padrão), em outra cadeia $X = x_1 \dots x_n$ (o texto), ambas sob um alfabeto Σ. Conforme Navarro [Nav01] observou, o algoritmo pode ser visto como uma simulação bit-paralela de um autômato não-determinístico construído para o padrão em questão e que recebe o texto como entrada (Figura 3.2), porém essa analogia não está explicitada no artigo original. O algoritmo constrói uma tabela B contendo $|\Sigma|$ cadeias binárias, todas de comprimento m, tal que o bit b_i da cadeia $B[x] = b_m \dots b_1$ é 0 se e somente se o caractere x ocorre na posição i do padrão[2]. No decorrer da execução do algoritmo, o estado atual da simulação (estado ativo do autômato) é mantido em um vetor $D = d_m \dots d_1$, tal que d_i é igual a 0 se o estado i no autômato é ativo; isto é,

[2]Note que por uma questão de praticidade, visto que estamos mais acostumados com máquinas que adotam a orientação dos bits da direita para a esquerda ⟨ *big endian* ⟩, seguiremos a mesma abordagem, isto é, o bit mais significativo encontra-se à esquerda da palavra.

Algoritmo 3.1 Algoritmo Shift-Or que determina as ocorrências de uma cadeia em outra através do bit-paralelismo. O algoritmo recebe como entrada duas cadeias P e X e imprime as posições finais de ocorrências exatas de P em X.

```
SHIFT-OR(P_{1..m}, X_{1..n})
 1  para i ← 1 até |Σ| faça
 2     para j ← 1 até m faça
 3           B[i][j] ← 1
 4  para j ← 1 até m faça
 5     B[p_j][j] ← 0
 6  para k ← 1 até m faça
 7     D[k] ← 1
 8  para j ← 1 até n faça
 9     D ← D << 1 ∨ B[x_j]
10     se d_m = 0
11       então imprima "P ocorre em X na posição j".
```

se a cadeia $P_{1..i}$ ocorre ao final do prefixo do texto que foi lido até o momento. Logo, temos uma ocorrência sempre que d_m é igual a zero. Vamos denotar os operadores de deslocamento binário à esquerda e direita respectivamente por $<<$ e $>>$, e os operadores lógicos binários por $\wedge$ (e) e $\vee$ (ou). Inicialmente, todas as posições de D são iguais a um (estado inicial) e, para cada caractere x_j na entrada, D é atualizado como segue:

$$D \leftarrow (D << 1) \vee B[x_j] \tag{3.1}$$

d_i torna-se zero se e somente se d_{i-1} era zero (ativo) para o caractere de entrada anterior e o caractere atual ocorre na posição i do padrão, como demonstraremos a seguir. Em outras palavras, a cadeia $P_{1..i}$ ocorre ao final da cadeia $X_{1..j}$ somente se $P_{1..i-1}$ ocorre ao final de $X_{1..j-1}$ e $p_i = x_j$. Em suma, a cada iteração do laço da linha 8, vale o invariante a seguir:

(I1) Para todo $1 \leq i \leq m$, $d_{i-1} = 0$ se e somente se $P_{1..i-1} = X_{j-i+1..j-1}$.

[Prova de (I1)] No início da primeira iteração, temos que $d_i = 1$, para todo $1 \leq i \leq m$, $j = 0$ e $X_{j-i+1..j-1} = \epsilon$. Logo, o invariante vale trivialmente. Tomemos uma iteração

qualquer do algoritmo e vamos supor que o invariante vale para essa iteração. Seja $d_{i-1} = 0$ e, portanto, $P_{1..i-1} = X_{j-i+1..j-1}$. Se $p_i = x_j$, então $b_i = 0$. Logo,

$$\begin{aligned} d_i &= d_{i-1} \vee b_i \\ &= 0 \vee 0 \\ &= 0 \end{aligned}$$

e $P_{1..i-1}p_i = X_{j-i+1..j-1}x_j$. Agora, vamos supor $d_{i-1} = 1$, que implica $P_{1..i-1} \neq X_{j-i+1..j-1}$. Temos que

$$\begin{aligned} d_i &= d_{i-1} \vee b_i \\ &= 1 \vee b_i \\ &= 1 \end{aligned}$$

e, evidentemente, $P_{1..i-1}p_i \neq X_{j-i+1..j-1}p_j$. Do invariante, segue imediatamente a correção do algoritmo.

Se o comprimento do padrão é menor ou igual ao de uma palavra da máquina, o vetor D pode ser atualizado em tempo constante. Nesse caso, a simulação do autômato consome tempo $O(n)$. Como a construção da tabela B consome tempo $O(|\Sigma|m)$, toda a execução do algoritmo, nesse caso, consome tempo $O(n)$. Agora, caso o padrão seja maior do que uma palavra da máquina, a cada atualização de D é preciso efetuar um número constante de operações binárias sobre m/w blocos de comprimento w. Logo, todo o algoritmo consome tempo $O(mn/w)$.

Muitos algoritmos de processamento de cadeias são apenas implementações de autômatos finitos, geralmente em sua forma determinística. O bit-paralelismo apresenta-se como uma opção em diversos casos; é possível simular alguns autômatos não-determinísticos sem precisar convertê-los para a sua forma determinística. Infelizmente, algoritmos bit-paralelos são particularmente eficientes quando o comprimento do padrão não supera o de uma palavra de máquina, caso contrário muitas vezes a adaptação produz um algoritmo cuja eficiência deixa a desejar [Nav98].

3.5 O Algoritmo de Myers

A idéia central do algoritmo de Myers é acelerar o processamento de uma matriz de programação dinâmica através do cálculo simultâneo de blocos de células da matriz

usando o bit-paralelismo. Essa idéia foi testada, pela primeira vez, por Wright [Wri94], em 1994, porém seu algoritmo era competitivo apenas para alfabetos muito pequenos [Nav98]. Apesar de muitos dos conceitos nos quais o algoritmo de Wright se fundamenta também servirem de base para o algoritmo de Myers, este, por sua vez, é bastante eficiente [Nav98, SM98].

A abordagem clássica de programação dinâmica para a busca aproximada de padrões é a dada pela recorrência a seguir, proposta por Sellers [Sel74] e baseada no algoritmo para o cálculo da distância de edição. Considere $\alpha(a, b) = 0$, se $a = b$, ou $\alpha(a, b) = 1$, se $a \neq b$.

$$C[i,j] = \begin{cases} i & \text{se } i = 0 \text{ ou } j = 0 \\ \min \left\{ \begin{array}{l} C[i-1, j-1] + \alpha(p_i, x_j) \\ C[i-1, j] + 1 \\ C[i, j-1] + 1 \end{array} \right\} & \text{se } i > 0 \text{ e } j > 0. \end{cases} \tag{3.2}$$

A matriz de programação dinâmica que essa recorrência dá conta pode ser preenchida de diversas formas. A trivial é iniciar o preenchimento na posição $C[0, 0]$ e avançar, em colunas, até a última posição, $C[m, n]$. Ao término do processamento, teremos, em cada posição $C[i, j]$ da matriz, a menor distância de edição entre $P_{1..i}$ e um sufixo de $X_{1..j}$. Logo, todo j tal que $C[m, j] \leq r$ é posição final de uma (d, r)-ocorrência de P em X.

Assim como ocorre com o cálculo da distância de edição entre duas cadeias, é possível preencher a matriz de programação dinâmica consumindo apenas espaço linear no comprimento do padrão, visto que para calcular $C[i, j]$ é preciso olhar apenas para valores em $\bigcup_{i=0}^{m}\{C[i, j]\}$; isto é, para determinar todos os valores de uma determinada coluna $C_j = \langle C[1, j] \ldots C[m, j]\rangle$, depende-se apenas de valores da coluna anterior C_{j-1}. Uma intuição muito interessante pode ser extraída dessa simples observação: o processamento da matriz pode ser abstraído como a simulação de um autômato não-determinístico, onde cada coluna da matriz representa um estado do autômato e, ao ler o caractere de entrada x_j, estamos transitando do estado C_{j-1} para o estado C_j. A simulação inicia-se no estado C_0 e qualquer estado C_j, tal que existe $C[m, j] \leq r$, é final. Essa analogia foi adotada por Myers no artigo original.

3.5.1 As Matrizes Δv e Δh

Masek e Paterson [MP80] demonstraram, em 1980, que, a diferença entre duas posições, vertical ou horizontalmente adjacentes, da matriz dada pela recorrência de Sellers é -1, 0 ou $+1$; isto é, elas diferem por, no máximo, 1.[3] Dada uma matriz $C[1\,..\,m, 1\,..\,n]$ definida pela recorrência e um par ordenado $(i, j) \in \{1, m\} \times \{1, n\}$, vamos definir as matrizes **delta horizontal**, $\Delta h[i, j]$, e **delta vertical**, $\Delta v[i, j]$, como segue:

$$\Delta h[i, j] = C[i, j] - C[i, j-1] \tag{3.3}$$

$$\Delta v[i, j] = C[i, j] - C[i-1, j] \tag{3.4}$$

Note que para determinar um estado C_j é suficiente saber a **coluna de diferenças** $\Delta v_j = \langle \Delta_v[1, j] \ldots \Delta_v[m, j] \rangle$, uma vez que todos os valores da linha inicial são sempre zero. Como veremos em maiores detalhes mais à frente, é possível resolver o problema calculando-se apenas a matriz Δv, composta por todas as colunas de diferenças de C.

Agora, vamos ver como calcular os valores de uma coluna Δv_j com base nos valores da coluna Δv_{j-1}. Para tanto, considere o trecho da matriz de programação dinâmica composto pelas posições (i, j), $(i-1, j-1)$, $(i-1, j)$ e $(i, j-1)$. Há quatro deltas associados a esse conjunto de posições: dois deltas verticais, $\Delta v[i, j]$ e $\Delta v[i, j-1]$, e dois deltas horizontais $\Delta h[i, j]$ e $\Delta h[i-1, j]$. Vamos definir a função α como de costume, isto é $\alpha(a, b) = 0$, se $a = b$, e $\alpha(a, b) = 1$, se $a \neq b$. Também, vamos definir a função α' como $\alpha'(a, b) = 0$, se $a = b$, e $\alpha'(a, b) = 1$, se $a \neq b$; isto é, $\alpha(a, b) = 1 - \alpha'(a, b)$. Pelas definições de delta horizontal e delta vertical e pela recorrência de Sellers (3.2), temos

[3] Masek e Paterson provaram a afirmação para a recorrência do cálculo da distância de edição e adotaram essa propriedade na aplicação do método dos 4-russos ao problema. Ambas as recorrências são muito semelhantes, e a mesma propriedade vale para a recorrência fornecida por Sellers.

que

$$
\begin{aligned}
\Delta v[i,j] &= C[i,j] - C[i-1,j] \\
&= \min \left\{ \begin{array}{l} C[i-1,j-1] + \alpha(p_i, x_j), \\ C[i-1,j] + 1, \\ C[i,j-1] + 1. \end{array} \right\} - C[i-1,j] \\
&= \min \left\{ \begin{array}{l} C[i-1,j-1] + (1 - \alpha'(p_i, x_j)), \\ C[i-1,j] + 1, \\ C[i,j-1] + 1. \end{array} \right\} - (C[i-1,j-1] + \Delta h[i-1,j]) \\
&= \min \left\{ \begin{array}{l} C[i-1,j-1] - \alpha'(p_i, x_j) + 1, \\ C[i-1,j-1] + \Delta v[i,j-1] + 1, \\ C[i-1,j-1] + \Delta h[i-1,j] + 1. \end{array} \right\} - (C[i-1,j-1] + \Delta h[i-1,j]) \\
&= \min \left\{ \begin{array}{l} -\alpha'(p_i, x_j) + 1, \\ \Delta v[i,j-1] + 1, \\ \Delta h[i-1,j] + 1. \end{array} \right\} + C[i-1,j-1] - C[i-1,j-1] - \Delta h[i-1,j] \\
&= \min \left\{ \begin{array}{l} -\alpha'(p_i, x_j), \\ \Delta v[i,j-1], \\ \Delta h[i-1,j]. \end{array} \right\} + 1 - \Delta h[i-1,j].
\end{aligned}
\tag{3.5}
$$

Desenvolvendo a equação 3.3 de modo muito similar, também temos que:

$$
\Delta h[i,j] = \min \left\{ \begin{array}{l} -\alpha'(p_i, x_j), \\ \Delta v[i,j-1], \\ \Delta h[i-1,j]. \end{array} \right\} + 1 - \Delta v[i,j-1]. \tag{3.6}
$$

Como as recorrências acima deixam claro, é possível preencher as matrizes Δv e Δh sem recorrer a valores da matriz C. Mas, para resolver o problema tendo em mãos apenas essas duas matrizes, um pequeno detalhe deve ser levado em conta: é preciso saber se uma dada coluna Δv_j é final; isto é, saber se $\sum_{i=1}^{m} \Delta v[i,j] = C[m,j] \leq r$. Isso pode ser feito ao final da construção da matriz, em tempo $O(m)$, ou incrementalmente, de modo que ao término do preenchimento da matriz a desigualdade possa ser avaliada em tempo constante. Para isso, basta manter um vetor $S[1\,..\,n]$, atualizado a cada iteração do algoritmo, juntamente com a matriz, e tal que $S[j] = C[m,j]$, tendo como valores iniciais $S[0] = m$ e $S[j] = S[j-1] + \Delta h[m,j]$. Em qualquer uma das abordagens o consumo de tempo do algoritmo tradicional permanece o mesmo. Porém, em se tratando do nosso algoritmo bit-paralelo, é fundamental que adotemos a segunda opção, baseada no processamento do vetor. Essa escolha ficará clara a seguir.

3.5.2 Representando Δv e Δh Através de Vetores Binários

A idéia central do algoritmo de Myers é adotar o bit-paralelismo para calcular simultaneamente blocos de células das matrizes Δv e Δh. Notadamente, uma coluna por vez. Para isso, é preciso codificá-las em vetores binários, que podem ser manipulados através das operações básicas permitidas pelo modelo w-RAM. Tomemos uma célula da matriz, digamos a célula $C[i, j]$. Essa célula tem seu valor determinado através das células $C[i-1, j-1]$, $C[i-1, j]$ e $C[i, j-1]$, e também da validade da comparação $p_i = x_j$. Por sua vez, $C[i, j]$ ajudará a compor o valor das células $C[i+1, j+1]$, $C[i+1, j]$ e $C[i, j+1]$. Codificando valores da matriz C através das matrizes Δv e Δh, Myers, intuitivamente, considerou cada célula da matriz como um processador, que recebe dados de entrada e devolve dados de saída: $C[i, j]$ recebe como entradas os valores $\Delta v[i, j-1]$, $\Delta h[i-1, j]$ e $\alpha'(p_i, x_j)$, e devolve como saída os valores $\Delta v[i, j]$ e $\Delta h[i, j]$. Para deixar essa abordagem ainda mais explícita e tornar a discussão mais fácil, vamos tomar $C[i, j]$ isoladamente das demais células, eliminando os índices por alguns instantes e definindo os valores Δv_{in}, $\Delta_{h_{in}}$ e α' como a sua entrada e $\Delta_{v_{out}}$ e $\Delta_{h_{out}}$ como sua saída, isto é

$$\begin{aligned} \alpha' &= \alpha'(p_i, x_j), \\ \Delta v_{in} &= \Delta v[i, j-1], \\ \Delta_{h_{in}} &= \Delta h[i-1, j], \\ \Delta_{v_{out}} &= \Delta v[i, j], \\ \Delta_{h_{out}} &= \Delta h[i, j]. \end{aligned}$$

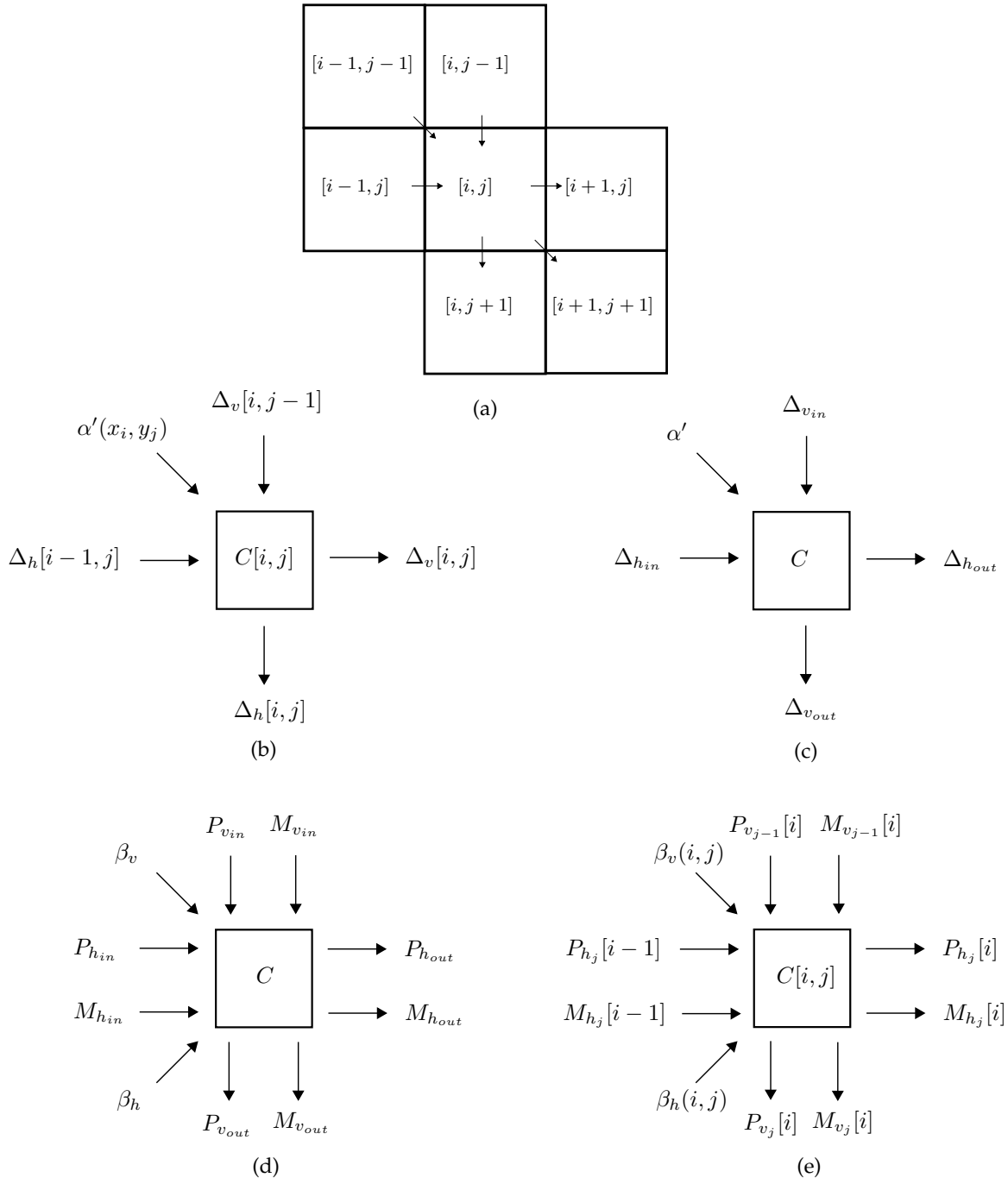
Para calcular $\Delta_{v_{out}}$ e $\Delta_{v_{out}}$ bit-paralelamente e, desse modo, processar diversas células das matrizes Δ_v e Δ_h em tempo $O(1)$, vamos representar a entrada e saída de cada célula através de dois conjuntos de variáveis binárias P_η e M_η, onde $\eta \in \{v_{in}, v_{out}, h_{in}, h_{out}\}$ e tais que

$$P_\eta = \begin{cases} 1 & \text{se } \Delta_\eta = +1 \\ 0 & \text{caso contrário,} \end{cases}$$

$$M_\eta = \begin{cases} 1 & \text{se } \Delta_\eta = -1 \\ 0 & \text{caso contrário.} \end{cases}$$

É evidente que P_η e M_η não podem ser ambos simultaneamente iguais a um. Note também que $\Delta_\eta = 0$ sempre que $P_\eta = 0$ e $M_\eta = 0$. Logo, todos os possíveis valores Δ_η

Figura 3.3 Uma célula $[i, j]$ da matriz de programação dinâmica C, recebe valores de células adjacentes e ajuda a compor valores de outras células (a). Essa mesma célula, pode ser vista como um processador, que recebe como entrada os valores $\Delta_v[i, j-1]$, $\Delta_h[i-1, j]$ e $\alpha'(p_i, x_j)$ e devolve como saída os valores $\Delta_v[i, j]$ e $\Delta_h[i, j]$. Para facilitar a visualização é conveniente remover os índices, explicitando ainda mais essa analogia(c). Os valores de entrada e saída são codificados através de vetores binários, permitindo assim o processamento bit-paralelo da matriz (d). A mesma representação após o retorno dos índices (e).



estão perfeitamente codificados em P_η e M_η. Sejam β_v e β_h variáveis binárias definidas como segue:

$$\beta_v = \alpha'(p_i, x_j) \vee M_{v_{in}} \tag{3.7}$$

e

$$\beta_h = \alpha'(p_i, x_j) \vee M_{h_{in}}. \tag{3.8}$$

Logo, β_v codifica ambos $\Delta_{v_{in}}$ e α' enquanto β_h codifica $\Delta_{h_{in}}$ e α'. A tabela a seguir apresenta os valores de $\Delta_{v_{out}}$ em função de β_v e $\Delta_{h_{in}}$.

$\Delta_{v_{out}}$		βv	
		0	1
$\Delta_{h_{in}}$	-1	$+1$	$+1$
	0	$+1$	0
	$+1$	0	-1

(3.9)

De modo análogo, a mesma tabela vale para $\Delta_{h_{out}}$ em função de β_h e $\Delta_{v_{in}}$. Agora, com base nessa tabela e na tabela verdade 3.1 é possível verificar que as igualdades a seguir são válidas:

$$P_{v_{out}} = M_{h_{in}} \vee \neg(\beta_v \vee P_{h_{in}}) \tag{3.10}$$

e

$$M_{v_{out}} = P_{h_{in}} \wedge \beta_v. \tag{3.11}$$

E, de modo análogo,

$$P_{h_{out}} = M_{v_{in}} \vee \neg(\beta_h \vee P_{v_{in}}) \tag{3.12}$$

e

$$M_{h_{out}} = P_{v_{in}} \wedge \beta_h. \tag{3.13}$$

Tabela 3.1 Tabela verdade referente aos valores Δ e suas representações binárias P e M.

$\Delta_{h_{in}}$	$\Delta_{v_{in}}$	$-\alpha'$	$(1-\Delta_{h_{in}})$	$(1-\Delta v_{in})$	$\Delta_{v_{out}}$	$\Delta_{h_{out}}$	$P_{h_{in}}$	$M_{h_{in}}$	$M_{v_{in}}$	β_v	$P_{v_{out}}$	$M_{v_{out}}$
0	0	0	1	1	+1	+1	0	0	0	0	1	0
0	0	−1	1	1	0	0	0	0	0	1	0	0
0	+1	0	1	0	+1	0	0	0	0	0	1	0
0	+1	−1	1	0	0	−1	0	0	0	1	0	0
+1	0	0	0	1	0	+1	1	0	0	0	0	0
+1	0	−1	0	1	−1	0	1	0	0	1	0	1
0	−1	0	1	2	0	+1	0	0	1	1	0	0
0	−1	−1	1	2	0	+1	0	0	1	1	0	0
−1	0	0	2	1	+1	0	0	1	0	0	1	0
−1	0	−1	2	1	+1	0	0	1	0	1	1	0
+1	+1	0	0	0	0	0	1	0	0	0	0	0
+1	+1	−1	0	0	−1	−1	1	0	0	1	0	1
−1	−1	0	2	2	+1	+1	0	1	1	1	1	0
−1	−1	−1	2	2	+1	+1	0	1	1	1	1	0
−1	+1	0	2	0	+1	−1	0	1	0	0	1	0
−1	+1	−1	2	0	+1	−1	0	1	0	1	1	0
+1	−1	0	0	2	−1	+1	1	0	1	0	0	1
+1	−1	−1	0	2	−1	+1	1	0	1	1	0	1

3.5.3 O Algoritmo

Agora que já apresentamos a representação binária de Δ_v e Δ_h, podemos voltar a olhar para as demais células da matriz e nos livrar, de uma vez por todas, das variáveis $\Delta_{v_{in}}$, $\Delta_{h_{in}}$, $\Delta_{v_{out}}$, $\Delta_{h_{out}}$, adentrando nos detalhes do algoritmo. Para isso, vamos estender a definição das variáveis P_η e M_η para definir os conjuntos de vetores a seguir:

$$P_{v_j}[i] = \begin{cases} 1 & \text{se } \Delta_v[i,j] = +1 \\ 0 & \text{caso contrário} \end{cases}$$

e

$$M_{v_j}[i] = \begin{cases} 1 & \text{se } \Delta_v[i,j] = -1 \\ 0 & \text{caso contrário.} \end{cases}$$

O ponto principal do algoritmo está em, para cada caractere x_j lido, atualizar o vetor $S[j]$ e os vetores binários P_{v_j} e M_{v_j} a partir dos vetores $S[j-1]$, $P_{v_{j-1}}$ e $M_{v_{j-1}}$; esse é o processamento realizado a cada iteração do algoritmo, enquanto a inicialização se dá

como segue:

$$\begin{aligned} P_{v_0}[i] &= 1, \\ M_{v_0}[i] &= 0, \\ S[0] &= m. \end{aligned}$$

A atualização é realizada em dois estágios. No primeiro, todos os valores da coluna Δ_{h_j}, codificada nos vetores P_{v_j} e M_{v_j}, são atualizados simultaneamente com base em valores de $\Delta_{v_{j-1}}$, codificada nos vetores $P_{v_{j-1}}$ e $M_{v_{j-1}}$, como segue:

$$P_{h_j}[i] = M_{v_{j-1}}[i] \vee \neg(\beta_{h_j}[i] \vee P_{v_{j-1}}[i]) \tag{3.14}$$

e

$$M_{h_j}[i] = P_{v_{j-1}}[i] \wedge \beta_{h_j}[i]. \tag{3.15}$$

No segundo estágio, os valores da coluna Δ_{v_j}, são atualizados com base nos valores obtidos no primeiro estágio, como segue:

$$\begin{aligned} P_{h_j}[0] &= M_{h_j}[0] = 0, \\ P_{v_j}i] &= M_{h_j}[i-1] \vee \neg(\beta_{v_j}[i] \vee P_{h_j}[i-1]) \\ M_{v_j}[i] &= P_{h_j}[i-1] \wedge \beta_{v_j}[i]. \end{aligned}$$

Também é necessário atualizar o vetor $S[1\,..\,n]$, como segue

$$S[j] = S[j-1] + [P_{h_j}[m] = 1] - [M_{h_j}[m] = 1]. \tag{3.16}$$

Tal atualização pode ser feita entre o primeiro e o segundo estágio, uma vez que atualizar S requer apenas valores obtidos após o primeiro estágio.

A execução do algoritmo propriamente dito é antecedida por uma etapa de processamento que devolve um conjunto de $|\Sigma|$ vetores $W = W_a, W_b, \ldots, W_{|\Sigma|}$ tal que $W_a[i] = 1$, se $a = p_i$, ou $W[a,i] = 0$, caso contrário. Tal tabela pode ser facilmente construída em tempo e espaço $O(|\Sigma| + m)$. O papel dessa tabela ficará claro a seguir. As equações fornecidas não deixam claro como é realizada, bit-paralelamente, a atualização dos valores β_v e β_h. Da definição 3.7, desenvolvemos o que segue:

$$\beta_{v_j}[i] = W_{y_j}[i] \vee M_{v_{j-1}}[i] \tag{3.17}$$

$$\beta_{h_j}[i] = W_{y_j}[i] \vee M_{h_j}[i-1] \tag{3.18}$$

Aqui o leitor irá notar um pequeno impasse: calcular β_{h_j} requer valores de M_{h_j} que, por sua vez, requer valores de β_{h_j}. A saída para o impasse é fornecida pelo lema a seguir, que nos diz como calcular β_{h_j} sem depender de M_{h_j}.

Lema 26. $\beta_{h_j}[i] = \exists k \leq i, W_j[k] \wedge (\forall \gamma \in [k, i-1], P_{v_{j-1}}[\gamma])$

Demonstração. Em virtude de 3.15, temos que:

$$M_{h_j}[i] = P_{v_{j-1}}[i] \wedge \beta_{h_j}[i]$$

de onde, aplicando 3.18, temos

$$\begin{aligned} M_{h_j}[i] &= P_{v_{j-1}}[i] \wedge (W_{y_j}[i] \vee M_{h_j}[i-1]) \\ &= (P_{v_{j-1}}[i] \wedge W_{y_j}[i]) \vee (P_{v_{j-1}}[i] \wedge M_{h_j}[i-1]), \end{aligned}$$

que, por sua vez, se aplicada repetidas vezes sobre 3.18, resulta

$$\begin{aligned} \beta_{h_j}[i] &= W_{y_j}[i] \vee M_{h_j}[i-1] \\ &= W_{y_j}[i] \vee (P_{v_{j-1}}[i-1] \wedge W_{y_j}[i-1]) \\ &\quad \vee (P_{v_{j-1}}[i-1] \wedge M_{h_j}[i-2]) \\ &= W_{y_j}[i] \vee (P_{v_{j-1}}[i-1] \wedge W[i-1, j]) \\ &\quad \vee (P_{v_{j-1}}[i-1] \wedge (P_{v_{j-1}}[i-2] \wedge W_{y_j}[i-2]) \\ &\quad \vee (P_{v_{j-1}}[i-1] \wedge M_{h_j}[i-3]) \\ &= W_{y_j}[i] \vee (P_{v_{j-1}}[i-1] \wedge W_{y_j}[i-1]) \\ &\quad \vee (P_{v_{j-1}}[i-1] \wedge (P_{v_{j-1}}[i-2] \wedge W_{y_j}[i-2]) \\ &\quad \vee \ldots \\ &\quad \vee (P_{v_{j-1}}[i-1] \wedge P_{v_{j-1}}[i-2] \wedge \ldots \wedge P_{v_{j-1}}[1] \wedge W_{y_j}[1]) \\ &\quad \vee (P_{v_{j-1}}[i-1] \wedge P_{v_{j-1}}[i-2] \wedge \ldots \wedge P_{v_{j-1}}[1] \wedge M_{h_j}[0]). \end{aligned}$$

Daí, lembrando que $M_{h_j}[0] = 0$, deriva imediatamente a validade da afirmação. □

Ainda nos resta esclarecer como calcular β_h em tempo constante. Basicamente, o lema anterior afirma que o i-ésimo bit de β_{h_j} estará ligado sempre que existir algum

k-ésimo bit ligado em W_{y_j} e uma seqüência $P_{v_{j-1}}[k\,..\,i-i]$ de bits ligados. Observe o exemplo a seguir:

$$\begin{array}{ll} P_{v_{j-1}} & 00011111111000 \\ W[j] & 00100100100010 \\ \hline \beta_{h_j} & 00111111100010 \end{array}$$

Aqui, é conveniente fazer uma pequena observação: há analogia entre a operação realizada entre os vetores $P_{v_{j-1}}$ e W_{y_j}, do qual resulta β_{h_j} e a adição binária de inteiros. Um bit ligado em $W[j]$, implica no mesmo bit ligado em β_{h_j}, e é propagado (*carry*) por β_{h_j} enquanto houver bits correspondentes ligados em $P_{v_{j-1}}$. A adição de inteiros conserva um comportamento análogo, porém preservando certas diferenças. Observe o exemplo a seguir, no qual realizamos a adição dos mesmos vetores $P_{v_{j-1}}$ e W_{y_j}:

$$\begin{array}{lll} P_{v_{j-1}} & 00011111111000 & + \\ W_{y_j} & 00100100100010 & \\ \hline C & 01000100011010 & \end{array}$$

Vamos considerar inicialmente apenas as posições de W_{y_j} que emparelham bits ligados em $P_{v_{j-1}}$. Cada bit ligado em W_{y_j} inicia a propagação de uma seqüência de bits desligados no vetor resultante C, com exceção das posições que correspondem a bits ligados em W_{y_j}: estas posições estarão ligadas em C. Os bits desligados de W_{y_j} que não pertencem a uma seqüência de propagação como a descrita resultam ligados em C. Agora, seja i tal $Pv_{j-1}[i] = 0$. Nesse caso, valem as regras a seguir:

1. se $W_{y_j}[i] = 1$:

   se $W_{y_j}[i]$ é parte de uma seqüência de bits ligados em W_{y_j} que precede imediatamente uma seqüência de propagação por bits ligados de $P_{v_{j-1}}$, então $C[i] = 0$, caso contrário $C[i] = 1$;

2. se $W_{y_j} = 0$:

   se $W_{y_j}[i]$ precede imediatamente uma seqüência de bits ligados em W_{y_j} que, por sua vez, precede imediatamente uma seqüência de propagação por bits ligados de $P_{v_{j-1}}$, então $C[i] = 1$, caso contrário $C[i] = 0$.

Para realizar a operação que desejamos, e obter β_{h_j} tomando como base a adição de inteiros, precisamos inicialmente zerar os bits de W_{y_j} que não emparelham bits ligados de $P_{v_{j-1}}$: isso será feito calculando $W_{y_j} \wedge P_{v_{j-1}}$ e somando o resultado a $P_{v_{j-1}}$. Feito isso, é necessário capturar todos os bits de W_{y_j} cobertos por alguma seqüência de propagação; faremos isso através da operação *ou-exclusivo* entre o resultado que temos até o momento e $P_{v_{j-1}}$. Ainda precisamos capturar os bits de W_{y_j} que não emparelham bits ligados de $P_{v_{j-1}}$ e tampouco iniciam uma seqüência de propagação; para isso realizaremos a operação $\vee$ entre o resultado e W_{y_j}. Sumarizando, a intuição nos leva a conjecturar que o vetor β_{h_j} pode ser calculado de acordo com a equação abaixo:

$$\beta_{h_j} = ((W_{y_j} \wedge P_{v_{j-1}}) + P_{v_{j-1}}) \oplus P_{v_{j-1}} \vee W_{y_j}. \tag{3.19}$$

Porém, ainda é preciso amparar formalmente essa observação, o que é conseguido com o lema a seguir.

Lema 27. *Se* $\beta_{h_j} = ((W_{y_j} \wedge P_{v_{j-1}}) + P_{v_{j-1}}) \oplus P_{v_{j-1}} \vee W_{y_j}$ *então* $\beta_{h_j}[i] = \exists k < i, W_j[k] \wedge \forall x \in [k, i-1] P_{v_{j-1}}[x]$

Demonstração. Considere o transdutor determinístico da figura 3.4. Esse transdutor recebe como entrada dois inteiros binários, digamos A e B, e devolve sua soma. O rótulo $(a, b)/c$ de cada transição representa um par (a, b) de dígitos dos operandos de entrada, e a saída c fornecida pelo transdutor. O transdutor devolve 1 quando no estado *sem carry* caso os operandos sejam iguais e 0 caso contrário. No estado *com carry* o comportamento do transdutor é o inverso: devolve 0 caso os operandos sejam iguais e 1 caso contrário. Para que o transdutor encontre-se no estado *com carry* quando estiver lendo dígitos na posição i dos operandos é necessário que exista $k < i$ tal que $A[k] = B[k] = 1$ e que para todo j, $k < j < i$, $A[j] = 0$ ou $B[j] = 0$. Logo,

$$(A + B)[i] = ((\exists k < i, A[k] \wedge B[k] \wedge (\forall x \in [k, i-1], A[x] \vee B[x])) \equiv (A[i] \equiv B[i]))$$

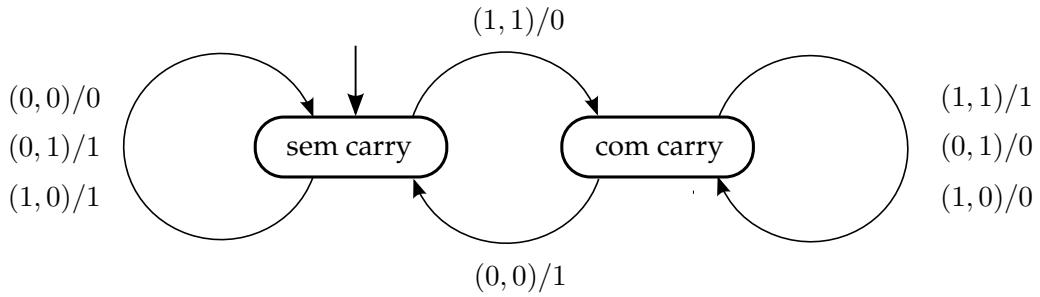
Considerando $B = P_{v_{j-1}}$:

$$(A + P_{v_{j-1}})[i] = ((\exists k < i, A[k] \wedge P_{v_{j-1}}[i] \wedge \forall x, k < x < i-1, (A[x] \vee P_{v_{j-1}}[i])) = (A[i] = P_{v_{j-1}}[i]))$$

Agora, seja $A = W_{y_j} \wedge P_{v_{j-1}}$. Em virtude da associatividade dos operadores lógicos envolvidos e do princípio da absorção, temos:

$$((W_{y_j} \wedge P_{v_{j-1}}) + P_{v_{j-1}})[i] = ((\exists k < i, W_{y_j} \wedge (\forall x \in [k, i-1], P_{v_{j-1}}[i])) \equiv (W_{y_j} \vee \neg P_{v_{j-1}}[i]))$$

Figura 3.4 Transdutor que define o resultado da soma de dois inteiros codificados em binário. O rótulo $(a, b)/c$ de cada transição representa um par (a, b) de dígitos dos operandos de entrada, e a saída c fornecida pelo transdutor. O transdutor possui apenas dois estados: o estado "com carry", que indica que uma seqüência de propagação está em andamento, e o estado inicial "sem carry", que indica o oposto.



Seja $C = (W_{y_j} \wedge P_{v_{j-1}}) + P_{v_{j-1}}) \oplus P_{v_{j-1}}$. Da equação acima, temos que:

$$C[i] = ((\exists k < i, W_{y_j} \wedge (\forall x \in [k, i-1], P_{v_{j-1}}[i])) \equiv (W_{y_j} \vee \neg P_{v_{j-1}}[i])) \oplus P_{v_{j-1}}[i]$$

De onde segue facilmente a equação:

$$C[i] = ((\exists k < i, W_{y_j} \wedge (\forall x \in [k, i-1], P_{v_{j-1}}[i])) \equiv \neg(W_{y_j} \wedge P_{v_{j-1}}[i]))$$

O que conclui a nossa demonstração. □

Do que foi discutido até o momento, obtemos o algoritmo 3.2. Como comentado anteriormente, a execução do algoritmo propriamente dito á antecedida por uma etapa de pré-processamento na qual é construída a matriz W (linha 1). Note que para atualizar os vetores binários em uma dada iteração do algoritmo do algoritmo é preciso apenas dos valores da iteração imediatamente anterior. Logo, sobrescrevendo os valores da iteração anteriores com os da iteração atual é possível executar o algoritmo com apenas sete vetores binários que serão atualizados inteiramente; não será necessário trabalharmos com índices. Em virtude disso, o consumo de espaço do algoritmo é dominado pelo consumo da matriz W: $O(|\Sigma|m)$. Após o pré-processamento, os vetores binários P_v e M_v são inicializados de acordo. Durante a varredura do texto, da posição $j = 1$ até a posição final, os vetores β_h e β_v são atualizados de acordo com a fórmula , para que contenham, respectivamente, os valores β_{h_j} e β_{v_j}. P_h e M_h são então calculados, abrigando os delta horizontal e vertical referentes a j-ésima coluna. Em seguida,

S é atualizado, assim como P_v e M_v, que passam a codificar os deltas horizontal e vertical da j-ésima coluna. Ao final de cada iteração, o valor de S é conferido (linhas 19 e 20 do algoritmo) para determinar se uma (d, r)-ocorrência foi encontrada. O consumo de tempo do algoritmo -e dominado pelo consumo do laço *para* da linha 5: $O(m + n)$, caso $m \leq w$, ou $O(mn/w)$ no caso geral. Lembrando que no caso geral ($m > w$) cada uma das operações de atualização dos vetores binários manipulados é, na verdade, um conjunto de $\lceil m/w \rceil$ operações sobre vetores binários de comprimento não superior a w. Maiores detalhes sobre a divisão dos vetores em blocos de comprimento w, e a manipulação dos mesmos, podem ser encontrados no trabalho original, juntamente com o esboço de uma biblioteca adequada a tarefa.

3.6 Comentários Bibliográficos

Segundo Navarro [Nav01], a busca aproximada de padrões passou a receber considerável atenção após o final da década de 60, quando já se discutia a sua aplicabilidade em problemas de biologia computacional, processamento de sinais e edição de textos, tendo essa última, provavelmente, sido a primeira motivação para o problema. Porém, foi com os avanços das técnicas de seqüênciamento genético, nos idos dos anos 90, e o surgimento da World Wide Web, aproximadamente no mesmo período, que o problema ganhou notoriedade e passou a ser exaustivamente pesquisado. Depois do surgimento do algoritmo de Sellers [Sel74] – primeira solução a adotar programação dinâmica – diversas abordagens foram propostas, sob diferentes paradigmas. Em 1994, Chang e Marr [CM94] provaram a cota inferior, para o caso médio, e forneceram um algoritmo, adotando filtragem, que atinge esse limiar ótimo, porém de importância apenas teórica: seu desempenho em termos práticos deixa muito a desejar. Em sua excelente tese de doutorado, Navarro [Nav98] explorou a fundo o problema, com ênfase na adoção de algoritmos bit-paralelos para sua solução[4]. Posteriormente, o mesmo autor apresentou um resumo das técnicas até então existentes, porém restringindo o debate a busca *on-line*[Nav01] e, em 2004 [HFN04], dessa vez com colegas, apresentou uma versão do algoritmo bit-paralelo de Myers [Mye99] que, segundo os autores, é ex-

[4]O bit-paralelismo nasceu a partir da tese de doutorado de Baeza-Yates [BY89] e consiste na exploração da capacidade do processador de realizar, em tempo constante, operações em cadeias binárias de comprimento menor ou igual ao comprimento de uma palavra da máquina.

Algoritmo 3.2 Algoritmo de Myers para a Busca Aproximada de Padrões. O algoritmo recebe duas cadeias, P e X, e imprime as posições finais de (d, r)-ocorrências de P em X.

MYERS$(P_{1..m}, X_{1..n})$

1 $W_{\{\sigma_1,\sigma_2,\ldots,\sigma_p\}}[1\,..\,m] \leftarrow$ PRÉ-PROCESSA(P)
2 $P_v \leftarrow 1\ldots1$
3 $M_v \leftarrow 0$
4 $S \leftarrow m$
5 **para** $j \leftarrow 1$ **até** n **faça**
6 $\quad\alpha' \leftarrow W_{x_j}$
7 $\quad\beta_v \leftarrow \alpha' \vee M_v$
8 $\quad\beta_h \leftarrow (((\alpha' \wedge P_v) + P_v) \oplus P_v) \vee \alpha'$
9 $\quad P_h \leftarrow M_v \vee \neg(\beta_v \vee P_v)$
10 $\quad M_h \leftarrow P_v \wedge \beta_h$
11 $\quad$**se** $P_h \wedge 100\ldots00$
12 $\quad\quad$**então** $S \leftarrow S + 1$
13 $\quad$**senão se** $M_h \wedge 100\ldots00$
14 $\quad\quad$**então** $S \leftarrow S - 1$
15 $\quad P_h \leftarrow P_h << 1$
16 $\quad M_h \leftarrow M_h << 1$
17 $\quad P_v \leftarrow M_h \vee \neg(\beta_v \vee P_h)$
18 $\quad M_v \leftarrow P_h \wedge \beta_v$
19 $\quad$**se** $S \leq r$
20 $\quad\quad$**então** imprima "P ocorre em X na posição j"

perimentalmente a solução mais eficiente para a versão *on-line* do problema conhecida até o momento. Em 2004, Navarro e Fredriksson [FN04] propuseram modificações no algoritmo de Chang e Marr, e afirmaram que o novo algoritmo era atraente não apenas sob o ponto de vista teórico, visto que preserva a otimalidade do algoritmo de Chang e Marr, mas também na prática, uma vez que se equipara com diversos algoritmo conhecidos e tidos como eficientes. A busca aproximada de padrões também é abordada por diversos outros trabalhos [Gus97, CR94, BYRN99, Hyy03, KM97].

Capítulo 4

Extração de Padrões

4.1 Introdução

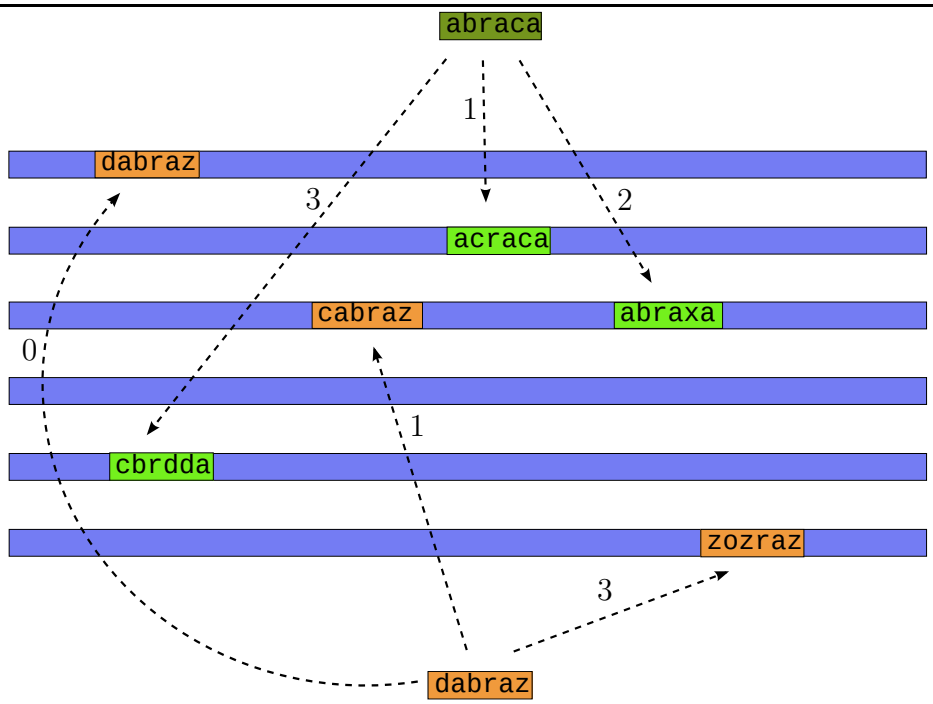
Neste capítulo, vamos estudar modelos e técnicas computacionais para a extração de padrões a partir de conjuntos de cadeias. É comum na literatura pertinente o uso do termo motif para denotar uma abstração de um padrão recorrente em um conjunto finito. Termo esse que, recentemente, passou a ser adotado no estudo dos mais diversos problemas e sob uma enorme gama de modelos matemáticos. Em sintonia com o restante do texto, e apesar da extensão do assunto, restringiremos nossa atenção aos motifs em cadeias.

O problema no qual estamos interessados consiste em encontrar todos os padrões que ocorrem, possivelmente com erros, em um número significativo de cadeias de um conjunto fornecido; isto é, encontrar os motifs recorrentes no conjunto. Chamamos esse problema de **extração de motifs comuns** e o formalizamos como segue:

Problema 28 (Extração de motifs comuns)**.** *Dados um conjunto de cadeias S, uma métrica d e inteiros r, q e l, encontrar todas as cadeias de comprimento l e que (d, r)-ocorrem em pelo menos q cadeias de S.*

A extração de motifs é um problema duplamente atraente, visto que, além de encontrar aplicação em diversos problemas computacionais (e.g. construção de árvores filogenéticas, detecção de plágio em textos, determinação de padrões de comportamento e preferências de usuário, entre outros), é computacionalmente desafiador em

Figura 4.1 A figura mostra as cadeias `abraca` e `dabraz`: ambas são motifs comuns no conjunto de cadeias apresentado, para dados $l = 6$, $q = 3$ e $r = 3$.



qualquer formulação razoavelmente interessante, dentre as que foram propostas até o momento, que são inúmeras. Em um trabalho divulgado em 2002 e publicado em 2006, Fellows e colegas [FGN06] demonstraram que a modelagem do problema que vamos estudar neste trabalho é NP-Difícil e também demonstraram que diversos de seus casos particulares também se encontram nessa mesma classe de complexidade, como o caso no qual apenas a distância de Hamming é considerada, ou o caso em que o alfabeto é binário. Além disso, o problema é de difícil aproximação [LMW99, LLM+99, FGN06, LMW02] e até o momento tem resistido à busca por heurísticas eficazes. Nesse capítulo veremos o algoritmo exato e não-polinomial proposto por Sagot [Sag98], em 1998, para a extração de motifs comuns sob a distância de Hamming. No mesmo trabalho, Sagot propôs também um algoritmo para resolver o problema da extração de motifs em uma cadeia, também sob a distância de Hamming. Vamos apresentar, inicialmente, esse último problema e o respectivo algoritmo, para então introduzir o algoritmo de Sagot para a extração de motifs comuns.

Algoritmo 4.1 Pré-processamento de árvore dos sufixos generalizada cuja execução antecede o algoritmo de Sagot para a extração de motifs repetidos.

PRÉ-PROCESSA-T(v)

1 **se** v é uma folha da árvore
2 **então**
3 $F[v] \leftarrow 1$
4 **senão**
5 $F[v] \leftarrow 0$
6 **para** cada $(v, u) \in A$ **faça**
7 $F[v] \leftarrow F[v] +$ PRÉ-PROCESSA-T(u)
8 **devolva** $F[v]$;

4.2 Algoritmo de Sagot para a Extração de Motifs Repetidos

O problema da extração de motifs repetidos é formalizado como segue:

Problema 29 (Extração de motifs de uma cadeia)**.** *Dado uma cadeia $X = x_1 \dots x_n$, uma métrica d e inteiros r, q e l, encontrar todas as cadeias de comprimento l e que (d, r)-ocorrem em pelo q posições distintas X.*

O algoritmo de Sagot dedica-se ao problema apenas sob a distância de Hamming, portanto para o restante da seção vamos considerar $d = d_H$. O algoritmo baseia-se no processamento da árvore dos sufixos de X, previamente modificada para que possamos dispor, para cada nó v da árvore, do número de folhas descendentes de v; informação que será armazenado no vetor F, indexado pelo conjunto N dos nós da árvore. Esse pré-processamento da árvore é trivial: basta percorrê-la em profundidade, como o faz recursivamente o algoritmo 4.1. O algoritmo deve ser executado recebendo como parâmetro a raiz de T e, por simplicidade, F é manipulada como de escopo global. Assim como definimos anteriormente o lugar de uma cadeia M em uma árvore dos sufixos T, vamos definir abaixo o lugar aproximado, ou $(d, r)-$lugar, de M em T.

Definição 30. Seja T a árvore dos sufixos de uma cadeia X. Dados uma cadeia M,

uma métrica d e um inteiro r, dizemos que um lugar (v, Y) é (d, r)-**lugar** de M em T se $d(\phi(v, Y), M) \leq r$.

Note que ao contrário do que ocorre com o lugar de uma cadeia na árvore, não existe uma bijeção entre $(d, r)-$lugares e cadeias: uma mesma cadeia pode ter diversos $(d, r)-$lugares e um lugar (u, Y) pode ser $(d, r)-$lugar de diversas cadeias.

Seja M uma cadeia de comprimento l e seja (u, Y) um (d, r)-lugar de M em T. Do teorema 22, segue imediatamente que cada folha descendente de (u, Y) implica em uma ocorrência de $\phi(u, Y)$ em X, portanto implica, também, em uma (d, r)-ocorrência de M em X.

Logo, para resolver o problema basta determinar as cadeias de comprimento l e cujo total de folhas descendentes de seus (d, r)-lugares seja igual ou superior a q; tais cadeias são motifs repetidos na cadeia dada. O seguinte teorema dá base para o desenvolvimento do algoritmo.

Lema 31. *$h = (u, Y)$ é (d, r)-lugar de Ma na árvore dos sufixos T, para $a \in \Sigma$, se e somente se uma das duas afirmações a seguir é verdadeira:*

1. *$Pai(h)$ é (d, r)-lugar de M e $\lambda(Pai(h) \rightarrow h) = a$;*
2. *$Pai(h)$ é $(d, r-1)$-lugar de M e $\lambda(Pai(h) \rightarrow h) \neq a$.*

A abordagem do algoritmo é recursiva. Para tanto, vamos utilizar um algoritmo adaptativo (*wrapper*) para receber os dados de entrada do problema e executar o algoritmo recursivo como sub-rotina. O algoritmo recursivo recebe uma cadeia M tal que M é um motif repetido em X, respeitando-se o quórum mínimo q, e de comprimento menor ou igual a l, e recebe também um conjunto L de pares ordenados (g, j) tais que g é (d, j)-lugar de M na árvore.

O algoritmo estende M a taxa de um caractere por chamada recursiva, usando todos os caracteres do alfabeto e gerando, desse modo, todas as Σ^l possíveis cadeias sob Σ e de comprimento l. Para cada cadeia gerada, digamos Ma, um conjunto L' é criado para indicar os seus (d, r)-lugares. Com o auxílio do vetor F, obtido na etapa de pré-processamento da árvore, o algoritmo irá calcular o total de folhas descendentes dos (d, r)-lugares obtidos e caso o total indique Ma como um motif válido, uma nova chamada recursiva é efetuada passando-se Ma e L' como parâmetros. Na primeira

Algoritmo 4.2 Algoritmo para a extração de motifs repetidos na cadeia X. O algoritmo adota como sub-rotina o algoritmo recursivo EXTRAI-MOTIFS-REPETIDOS-REC

EXTRAI-MOTIFS-REPETIDOS(X, r, q, l)
1 $T \leftarrow$ ÁRVORE-DOS-SUFIXOS(X)
2 $u \leftarrow$ raiz de T
3 PRÉ-PROCESSA-T(u)
4 $M = \epsilon$
5 $L = \{((u, \epsilon), 0)\}$
6 EXTRAI-MOTIFS-REPETIDOS-REC(M, L, T, l, q, r)

execução (primeiro nível da recursão) temos $M = \epsilon$ e L possui apenas a raiz da árvore sem nenhum erro relacionado. É importante observar que F codifica o número de folhas descendente de cada um dos nós da árvore dos sufixos, e não de seus lugares, logo obter o número de folhas descendentes de um determinado lugar (u, Y), através de F, requer um pequeno processamento adicional que consome apenas tempo constante: basta obter o número de folhas descendentes de u, caso $Y = \epsilon$, ou o número de folhas descendentes do nó ponta do arco no qual (u, Y) se encontra, isto é, o nó descendente imediato de (u, Y). Para preservar a simplicidade do algoritmo, vamos apenas abstrair esse processamento e assumir que $F[h]$, onde h é um lugar na árvore, fornece o número de folhas descendentes de h. Para extrair os motifs de comprimento l, é preciso descer na árvore dos sufixos a uma profundidade de, no máximo, l transições. Seja p o número de lugares com profundidade l. O número de cadeias na (d, r)-vizinhança do rótulo de cada um desses p lugares é dado por:

$$\sum_{i=0}^{r} \binom{l}{i} (|\Sigma| - 1)^i = O(l^r |\Sigma|^r).$$

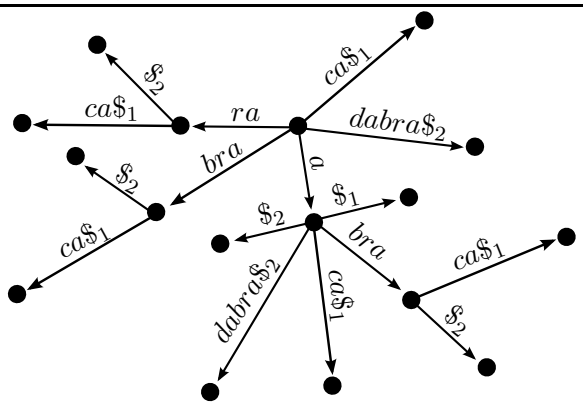
Observe que esse número é um delimitante superior do número de vezes que cada um dos p lugares é visitado durante a execução do algoritmo. Como cada uma dessas visitas consome tempo constante (linhas 10-17 do algoritmo), e como $p \leq n$, visto que o número de lugares em uma determinada profundidade da árvore é menor que o número total de folhas da mesma, temos que o consumo de tempo total do algoritmo é $O(nl^r|\Sigma|^r)$.

Algoritmo 4.3 Algoritmo recursivo para a extração de motifs repetidos a partir da árvore dos sufixos T. O algoritmo é adotado como sub-rotina pelo algoritmo EXTRAI-MOTIFS-REPETIDOS

EXTRAI-MOTIFS-REPETIDOS-REC$(M_{1..m}, L, T, l, q, r)$

```
1  se l = m
2   então
3    imprima "M é um motif do conjunto"
4   senão
5    para todo a ∈ Σ faça
6          total ← 0
7          L' ← ∅
8          para todo ((u, y), k) ∈ L faça
9            para cada transição t = (u, y) → (v, z) faça
10                 se a = λ(t)
11                  então
12                   L' ← L' ∪ {((v, z), k)}
13                   total ← total + F((v, z))
14                 senão se k < r
15                  então
16                   L' ← L' ∪ {((v, z), k + 1)}
17                   total ← total + F((v, z))
18         se total ≥ q
19          então
20           EXTRAI-MOTIFS-REPETIDOS-REC(M_{1..m}.a, L', T, l, q, r)
```

Figura 4.2 Árvore dos sufixos generalizada do conjunto de cadeias {`abraca`, `adabra`}



4.3 Algoritmo de Sagot para a Extração de Motifs Comuns

O algoritmo de Sagot para a extração de motifs comuns assemelha-se bastante ao algoritmo que acabamos de ver, porém tem como base o processamento de uma árvore dos sufixos generalizada, em vez da árvore dos sufixos tradicional. A **árvore dos sufixos generalizada** é uma estrutura de dados que abriga todos os sufixos das cadeias de um determinado conjunto e é construída em tempo e espaço linear na soma dos comprimentos das cadeias dos conjuntos, bastando para isso pequenas modificações nos algoritmos tradicionais de construção de uma árvore dos sufixos. O algoritmo de Sagot baseia-se na construção da árvore dos sufixos generalizada do conjunto de cadeias fornecido para, em seguida, extrair da estrutura de dados as cadeias motifs.

Tanto a definição formal da estrutura como os algoritmos para sua construção assemeham-se bastante aos da árvore dos sufixos.

Definição 32 (Árvore dos Sufixos Generalizada)**.** Seja $S = \{S_1, S_2, \ldots, S_n\}$ um conjunto de cadeias sobre Σ e $GT = (N, A, \lambda)$ uma árvore orientada tal que para todo nó $u \in N$ existe um único caminho orientado da raiz de GT a u. Ademais, seja $\lambda\colon A \to (\Sigma \cup \{\$_1, \$_2, \ldots, \$_n\})^+$ uma rotulação nos arcos de GT, para $\$_i \notin \Sigma$. O rótulo de um nó $u \in N$ é denotado por $\phi(u)$ e definido como o resultado da concatenação dos rótulos dos arcos no caminho de r a u. Dizemos que GT é **árvore dos sufixos generalizada** do conjunto S quando:

(p1) todo nó interno (não folha) em GT possui dois ou mais filhos;

(p2) para todo par de arcos $e, f \in A$, com origem em um mesmo nó, os rótulos de e e f iniciam-se com caracteres distintos;

(p3) toda folha de T tem como rótulo um sufixo de uma cadeia $S_i\$_i$;

(p4) para toda cadeia Y, sufixo de alguma cadeia $S_i\$_i$, existe uma folha em T com rótulo Y.

A construção da árvore dos sufixos generalizadas pode ser feita de modo similar à construção da árvore dos sufixos tradicional pelo algoritmo de McCreight, bastando para isso concatenar ao final de cada cadeia S_i de S um símbolo $\$_i$ tal que $\$_i \notin \Sigma$ e tal que para qualquer par de cadeias S_i e S_j tenhamos $\$_i \neq \$_j$. Feito isso, basta, iniciando-se com a árvore dos sufixos da cadeia $S_1\$_1$, construirmos a árvore dos sufixos da cadeia $S_i\$_i$ sobre a árvore dos sufixos generalizada construída sobre as cadeias anteriores.

Seja M uma cadeia e seja (u, Y) um (d, r)-lugar de M em GT. Do teorema 22, segue imediatamente que cada folha descendente de (u, Y) implica em uma ocorrência de $\phi(u, Y)$ em alguma cadeia de S, portanto implica, também, em uma (d, r)-ocorrência de M nessa mesma cadeia. Seja $S_M \subseteq S$ o conjunto de cadeias S_i tais que $\$_i$ é final do rótulo de uma folha descendente de um (d, r)-lugar de M. Se $|S_M| \geq q$, então M é um motif comum em S. Logo, para resolver o problema basta determinar toda cadeia M, de comprimento l, tal que $|S_M| \geq q$.

Após a construção da árvore generalizada e antes da extração dos motifs, propriamente dita, é conveniente pré-processar a estrutura de dados de modo que, para cada lugar $h = (u, Y)$, tenhamos um vetor binário $C_h[1\,..\,n]$ tal que

$$C_h[i] = \begin{cases} 1 & \text{se } \phi(h) \text{ é fator de } S_i, \\ 0 & \text{caso contrário.} \end{cases}$$

Note que $C_h[i] = 1$ equivale a afirmar que ao menos uma folha descendente de h tem como rótulo algum sufixo da cadeia $S_i\$_i$. O pré-processamento é realizado com uma busca em profundidade na árvore e, para cada lugar h da árvore, o vetor C_h é atualizado de modo bit-paralelo em tempo $O(n/w)$, onde, novamente, w é o comprimento de uma palavra da máquina[1]. Esse conjunto de vetores binários é gerado com o propósito de permitir, para cada lugar h da árvore, o cálculo rápido do número de cadeias de S

[1]O leitor notará que aqui assumimos, mais uma vez, o modelo w-RAM

das quais o rótulo de h é fator. Em outras palavras, queremos calcular rapidamente $\sum_{i=0}^{|S|} C_h[i]$. O algoritmo estende M a taxa de um caractere por chamada recursiva, usando todos os caracteres do alfabeto e gerando, desse modo, todas as Σ^l possíveis cadeias sob Σ e de comprimento l. Para cada cadeia gerada, digamos Ma, um conjunto L' é criado para indicar os seus (d, r)-lugares. Com o auxílio do vetor binário C, obtido na etapa de pré-processamento da árvore, o algoritmo irá calcular o total de cadeias das quais Ma é fator e caso o total indique Ma como um motif válido, uma nova chamada recursiva é efetuada passando-se Ma e L' como parâmetros. Na primeira execução (primeiro nível da recursão) temos $M = \epsilon$ e L possui apenas a raiz da árvore sem nenhum erro relacionado.

A análise do consumo de tempo e espaço do algoritmo é similar a análise do algoritmo 4.2, exceto pelos custos inerentes à construção e manutenção dos vetores C_h. Como dito anteriormente, a árvore dos sufixos generalizada é construída em tempo e espaço linear na soma dos comprimentos das cadeias, portanto, onde N é o maior comprimento de uma cadeia em S, temos $O(nN)$ como o tempo gasto para a construção da estrutura e o mesmo vale para o espaço ocupado por esta. Os vetores C_h, por se tratarem de vetores binários atualizados bit-paralelamente, consomem espaço $O(N/w)$ por lugar da árvore, onde w é, novamente, o comprimento da palavra da máquina. O número de lugares na árvore é $O(nN)$, logo o consumo de espaço total é $O(nN^2/w)$.

Para extrair os motifs de comprimento l é preciso descer na árvore dos sufixos generalizada a uma profundidade de, no máximo, l transições. Seja p o número de lugares com profundidade l. O número de cadeias na (d, r)-vizinhança do rótulo de cada um desses p lugares é dado por:

$$\sum_{i=0}^{r} \left(\begin{array}{c} l \\ i \end{array} \right) (|\Sigma| - 1)^i = O(l^r |\sigma|^r).$$

Observe que esse número é um delimitante superior do número de vezes que cada um dos p lugares é visitado durante a execução do algoritmo. Como cada uma dessas visitas consome tempo $O(N/w)$, e como $p \leq nN$, visto que o número de lugares em uma determinada profundidade da árvore é menor que o número total de folhas da mesma, temos que o consumo de tempo total do algoritmo é $O(nN^2 l^r |\Sigma|^r / w)$.

Algoritmo 4.4 Pré-processamento de árvore dos sufixos generalizada cuja execução antecede o algoritmo de Sagot para a extração de motifs comuns.

PRÉ-PROCESSA-GT$(h = (u, Y))$

1 $C_h \leftarrow \langle 0, \ldots, 0 \rangle$
2 **se** $Y = \$_i$
3 **então**
4 $C_h[i] \leftarrow 1$
5 **senão**
6 **para cada transição** $(u, Y) \rightarrow (v, Z)$ **faça**
7 PRÉ-PROCESSA-GT$(g = (v, Z))$
8 $C_h \leftarrow C_h \vee C_g$

Algoritmo 4.5 Algoritmo para a extração de motifs comuns na cadeia X. O algoritmo adota como sub-rotina o algoritmo recursivo EXTRAI-MOTIFS-COMUNS-REC

EXTRAI-MOTIFS-COMUNS(S, r, q, l)

1 $GT \leftarrow$ ÁRVORE-DOS-SUFIXOS-GENERALIZADA(S)
2 $u \leftarrow$ raiz de GT
3 PRÉ-PROCESSA-GT(u)
4 $M = \epsilon$
5 $L = \{((u, \epsilon), 0)\}$
6 EXTRAI-MOTIFS-COMUN-REC(M, L, GT, l, q, r)

Algoritmo 4.6 Algoritmo recursivo para a extração de motifs comuns a partir da árvore dos sufixos generalizada GT. O algoritmo é adotado como sub-rotina pelo algoritmo EXTRAI-MOTIFS-COMUNS

EXTRAI-MOTIFS-COMUNS-REC$(M_{1..m}, L, GT, l, q, r)$

```
1  se l = m
2   então
3    imprima "M é um motif do conjunto"
4   senão
5    para todo a ∈ Σ faça
6          L' ← ∅
7          C' ← ⟨0 .. 0⟩
8          para todo ((u, Y), k) ∈ L faça
9            para cada transição t = (u, Y) → (v, Z) faça
10                 se a = λ(t)
11                  então
12                   L' ← L' ∪ {((v, Z), k)}
13                   C' ← C' ∨ C(v, Z)
14                 senão se k < r
15                  então
16                   L' ← L' ∪ {((v, Z), k + 1)}
17                   C' ← C' ∨ C(v, Z)
18         se ∑{i ∈ C'} ≥ q
19          então
20           EXTRAI-MOTIFS-COMUNS-REC(M_{1..m}.a, L', GT, l, q, r)
```

4.4 Extração de Motifs Estruturados

O problema que apresentaremos a seguir tem importância fundamental na identificação de partes do genoma diretamente envolvidas na síntese protéica, visto que, por vezes, regiões funcionais não se apresentam como um único segmento de nucleotídeos comuns a diversas cadeias biológicas distintas, mas como um conjunto de segmentos não-contíguos comuns à diversas cadeias e cuja distribuição em cada uma das cadeias segue uma mesma estrutura. Esse detalhe torna sua determinação e localização tarefas sobremaneira mais complexas. Chamaremos uma abstração de padrões desse tipo de **motif estruturado**. Como ocorre com os motifs comuns, vistos anteriormente, esse problema também permite diversas definições. Neste trabalho, vamos adotar a definição que segue.

Definição 33 (Motif estruturado)**.** Um **motif estruturado** é uma tripla (M, e_{min}, e_{max}), onde:

- M é uma p-tupla de motifs simples $(M_1, .., M_p)$;
- e_{min} e e_{max} são inteiros não negativos.

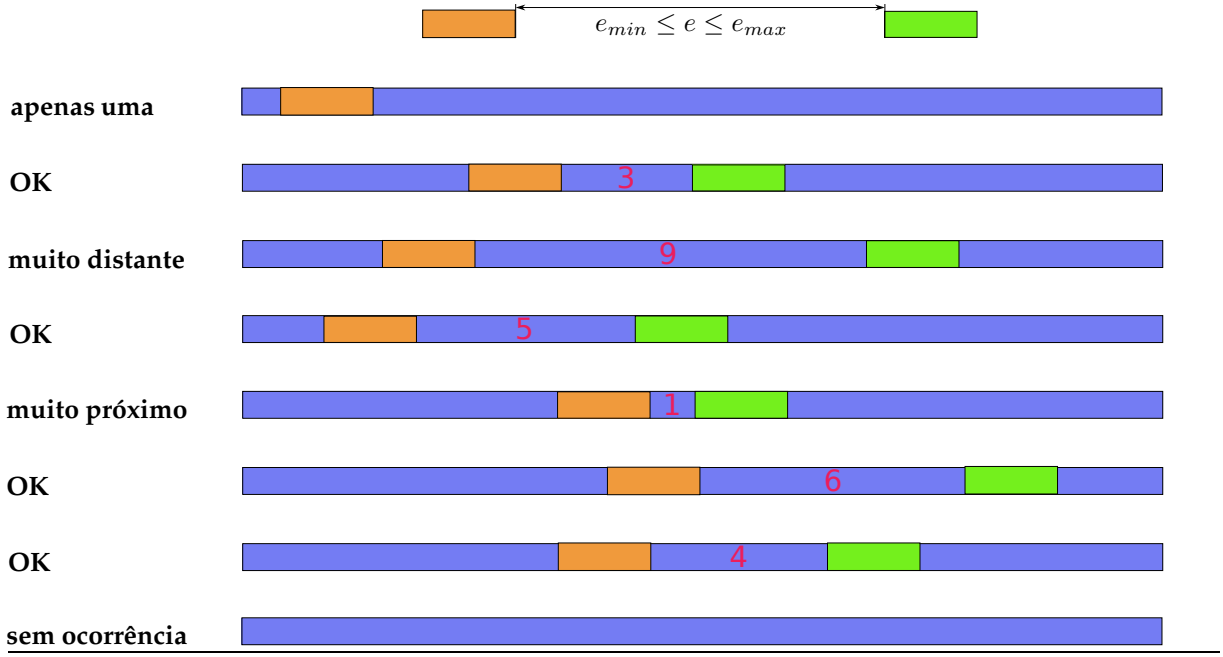
Definição 34 ((d, r)-ocorrência de um motif estruturado)**.** Dada uma cadeia $X = x_1 \dots x_n$, uma métrica d e um inteiro r, dizemos que um motif estruturado (M, e_{min}, e_{max}) (d, r)-**ocorre** em X se existem $O_1, \dots, O_p$ (d, r)-ocorrências (disjuntas) de $M_1, \dots, M_p$ em X, tais que

$$O_j = X_{k..l} \text{ e } O_{j+1} = X_{k'..l'} \Rightarrow e_{min} \leq k' - l - 1 \leq e_{max}, \forall j \in [1, p-1].$$

Problema 35 (Extração de motifs estruturados comuns)**.** *Dado um conjunto de cadeias S, uma métrica d e inteiros r, q, l e p, encontrar todos os motifs estruturados (M, e_{min}, e_{max}) tais que toda cadeia em M possui comprimento l e (M, e_{min}, e_{max}) (d, r)-ocorre em pelo menos q cadeias de S.*

A extração de motifs comuns é um caso particular da extração de motifs estruturados comuns: mais especificamente o caso em que $p = 1$. Logo, assim como o anterior, esse problema também é NP-Difícil. O algoritmo exato — evidentemente, de complexidade exponencial — que veremos a seguir, foi proposto, em 2000, por Marsan e Sagot [MS00a, MS00b] e apresentando na forma duas variações distintas. Cobriremos

Figura 4.3 A figura mostra um motif estruturado composto por dois blocos e comum ao conjunto de cadeias dado, considerando-se os valores $q = 4$, $e_{min} = 3$ e $e_{max} = 6$.



ambas as variações neste texto. Assim como o algoritmo para a extração de motifs comuns, o algoritmo de Marsan e Sagot baseia-se no processamento de uma árvore dos sufixos generalizada, construída sobre as cadeias do conjunto S, porém a diferença entre ambos está no fato de uma das variações do algoritmo Marsan-Sagot explorar as ligações de sufixo da árvore para movimentar-se através da mesma, em vez de apenas através dos arcos da estrutura. Logo, essa variação do algoritmo exige que as ligações de sufixos sejam preservadas após a construção da árvore dos sufixos generalizada.

Definição 36. É **caminho de uma cadeia** X em uma árvore dos sufixos T um caminho dirigido com extremos (u, Y) e (v, Z) tal que $\lambda(u, Y)^{-1}.\lambda(v, Z) = X$. Analogamente, é (d, r)-caminho de X em T o caminho dirigido com extremos (u, Y) e (v, Z) tal que $d(\lambda(u, Y)^{-1}.\lambda(v, Z), X) \leq r$.

Note que todo (d, r)-lugar de X é final de um (d, r)-caminho da mesma cadeia. Intuitivamente, a (d, r)-ocorrência de um motif estruturado $(\langle M_1, .., M_p\rangle, e_{min}, e_{mzx})$ em uma cadeia X é uma (d, r)-ocorrência de M_1, seguida de um segmento cujo comprimento está entre e_{min} e e_{max} caracteres, seguido de uma (d, r)-ocorrência de M_2, seguindo-se novamente um segmento de comprimento entre e_{min} e e_{max} caracteres, e

assim por diante. Já na árvore dos sufixos de X esse mesmo motif estruturado aparece como um caminho da raiz até um (d, r)-lugar aproximado de M_1, seguido de um caminho cujo comprimento está entre e_{min} e e_{max} lugares, seguido de um (d, r)-caminho de M_2 e assim por diante, incluindo $(d, r)-$caminhos de todas as cadeias em M, intercalados por caminhos de comprimento entre e_{min} e e_{max} lugares, e terminando no último lugar de um (d, r)-caminho de M_p. Vamos chamar o lugar final desse caminho que acabamos de traçar (i.e. lugar final do (d, r)-caminho de M_p) de (d, r)-lugar do motif estruturado $(\langle M_1, \ldots, M_p\rangle, e_{min}, e_{max})$ em GT. Assim como foi feito nos algoritmos estudados anteriormente, o objetivo do algoritmo Marsan-Sagot é encontrar todo motif estruturado que obedeça as restrições definidas pelos parâmetros l, e_{min}, e_{max} e p, e cujo conjunto $S_M \subseteq S$, de cadeias S_i tais que $\$_i$ é folha descendente de algum (d, r)-lugar de $(\langle M_1, \ldots, M_p\rangle, e_{min}, e_{max})$ em T, possua cardinalidade igual ou superior a q. O algoritmo se baseia no seguinte lema.

Lema 37. *Se $g = (u, Y)$ é (d, r)-lugar de um motif estruturado $(\langle M_1, \ldots, M_p\rangle, e_{min}, e_{max})$ então existe um lugar $h = (v, Z)$, ascendente de g tal que $l + e_{min} \leq |\phi(h)| - |\phi(g)| \leq l + e_{max}$ e h é (d, r)-lugar de $(\langle M_1, \ldots, M_{p-1}\rangle, e_{min}, e_{max})$*

Também é fundamental, para a compreensão do algoritmo, observar que se (M, e_{min}, e_{max}) é um motif estruturado comum em S, com quórum igual ou superior a q, então cada uma das cadeias em M é um motif comum em S. Essa observação permite que o algoritmo 4.5 seja adotado como sub-rotina do algoritmo que veremos em instantes. O algoritmo, cujo modo de trabalho é bastante intuitivo, extrai cada motif estruturado de S de modo incremental, buscando, a cada iteração, um motif comum válido de acordo com os parâmetros de entrada e que possa compor, juntamente com o motif estruturado obtido até o momento, um motif estruturado e de maior comprimento. O processo é repetido até que seja obtido um motif estruturado composto por p cadeias. Os autores propuseram duas versões do algoritmo para o problema. A primeira move-se na árvore apenas através dos arcos da estrutura, enquanto a segunda percorre os arcos e também as ligações de sufixos, além de modificar a árvore e restaurá-la antes da iteração seguinte. Para maior clareza, vamos omitir a alteração da árvore recebida; trata-se de um processo bastante simples. Em vez disso, o algoritmo irá gerar uma nova estrutura. Note que o processo de modificação e restauração é necessário para que o consumo de espaço da segunda versão do algoritmo não seja proibitivo. No texto que segue vamos discutir cada um dos algoritmos.

Primeiro Algoritmo: percorrendo os arcos da estrutura Vamos supor que, em um dado instante da execução do algoritmo, temos em mãos um motif estruturado composto por $i-1$ blocos, digamos $(\langle M_1 \,..\, M_{i-1}\rangle, e_{min}, e_{max})$. Seja L o conjunto de (d,r)-lugares desse motif estruturado na árvore GT. Seja L' o conjunto de pares ordenados $(h,0)$ tais que h é lugar de GT descendente de algum lugar g de L e cuja profundidade supera g no mínimo e_{min} lugares e no máximo e_{max} lugares, isto é, L' é o conjunto de lugares que descendem de lugares em L e de profundidade entre $(i-1)l + ie_{min}$ e $(i-1)l + ie_{max}$. O algoritmo precisa determinar uma cadeia M_i de comprimento l e tal que existam ao menos q (d,r)-caminhos de M_i com origem em lugares de L'. Isso é feito com a chamada a uma versão modificada do algoritmo para a extração de motifs comuns, que aqui chamamos de EXTRAI-MOTIFS-COMUNS', que em vez de apenas imprimir os motifs encontrados retorna um conjunto L'' de pares ordenados (M_i, L_i) tais que M_i é um motif válido no conjunto de cadeias, de acordo com os parâmetros fornecidos ao algoritmo, e L_i é o conjunto de pares ordenados contendo o conjunto de (d,r)-lugares de M_i juntamente com o respectivo erro associado. Para cada um dos motifs comuns encontrados o algoritmo irá compor um novo motif estruturado, passando-o para uma nova chamada recursiva, caso o comprimento desse motif estruturado ainda seja insuficiente, ou imprimindo-o, caso o mesmo componha p blocos.

O lugar mais profundo que o algoritmo pode visitar em GT tem profundidade $lp + e_{max}(p-1)$. A análise consiste na delimitação do número de vezes que algum lugar entre a raiz e lugares de profundidade $lp+e_{max}(p-1)$ pode ser visitado pelo algoritmo. O (d,r)-caminho de um motif estruturado composto por p blocos termina em um lugar de profundidade entre $pl+(p-1)e_{min}$ e $pl+(p-1)e_{max}$. O número de motifs estruturados que tal lugar pode ser (d,r)-lugar é

$$\left(\sum_{i=0}^{r} \binom{l}{i} (|\Sigma|-1)^i \; c \right)^p = O(l^{pr}|\sigma|^{pr}).$$

O número de lugares na árvore cuja profundidade está entre $pl + (p-1)e_{min}$ e $pl + (p-1)e_{max}$ é, no máximo, $(e_{max} - e_{min} + 1)nN = O(nN)$. Cada visita a um lugar da árvore consome tempo $O(N)$. Logo, o consumo de tempo do algoritmo é $O(nNl^{pr}|\sigma|^{pr})$.

Segundo Algoritmo: percorrendo as ligações de sufixos Assim como fizemos anteriormente, vamos supor que, em um dado instante da execução do algoritmo, temos em

Algoritmo 4.7 Algoritmo para a extração de motifs estruturados no conjunto de cadeias S. O algoritmo adota como sub-rotina uma das duas versões do algoritmo-EXTRAI-MOTIFS-ESTRUTURADOS-REC

EXTRAI-MOTIFS-ESTRUTURADOS$(S, r, q, l, e_{min}, e_{max})$

1 $GT \leftarrow$ ÁRVORE-DOS-SUFIXOS-GENERALIZADA(S)
2 $u \leftarrow$ raiz de GT
3 PRÉ-PROCESSA-GT(u)
4 $M = \epsilon$
5 $L = \{((u, \epsilon), 0)\}$
6 EXTRAI-MOTIFS-ESTRUTURADOS-REC$(M, L, GT, r, q, l, e_{min}, e_{max})$

Algoritmo 4.8 Primeira versão do algoritmo recursivo para a extração de motifs estruturados a partir da árvore dos sufixos generalizada. O algoritmo é adotado como sub-rotina pelo algoritmo EXTRAI-MOTIFS-ESTRUTURADOS

EXTRAI-MOTIFS-ESTRUTURADOS-REC-V1$((\langle M_1, \ldots, M_{i-1}\rangle, e_{min}, e_{max}), L, GT, p)$

1 **se** $i > 1$
2 **então**
3 **para todo** $g \in L$ **faça**
4 $L' \leftarrow L' \cup \{(h, 0)$: h é descendente de g
e $(i-1)l + (i-1)e_{min} \leq$prof$(h) \leq (i-1)l + (i-1)e_{max}\}$
5 **senão**
6 $L' \leftarrow \{(($raiz$[GT], \epsilon), 0)\}$
7 $L'' \leftarrow$ EXTRAI-MOTIFS-COMUNS$'(\epsilon, L', GT)$
8 **para todo** $(M_i, L_i) \in L''$ **faça**
9 **se** $i < p$
10 **então**
11 EXTRAI-MOTIFS-ESTRUTURADOS-REC-V1$((\langle M_1, \ldots, M_i\rangle, e_{min}, e_{max}), L_i, GT, p)$
12 **senão**
13 imprima "$(\langle M_1, \ldots, M_i\rangle)$ é um motif do conjunto S"

mãos um motif estruturado composto por $i-1$ blocos, digamos $(\langle M_1 \,..\, M_{i-1}\rangle, e_{min}, e_{max})$. Seja L o conjunto de (d, r)-lugares desse motif estruturado na árvore GT. Seja L' o conjunto de pares ordenados $(h, 0)$ tais que h é lugar de GT descendente de algum lugar g de L e cuja profundidade supera a profundidade deste em no mínimo e_{min} e no máximo e_{max} lugares, isto é, L' é o conjunto de lugares que descendem de lugares em L e de profundidade entre $il+(i-1)e_{min}$ e $il+(i-1)e_{max}$. A partir desse ponto, o algoritmo irá, para cada lugar de L', caminhar apenas pelas ligações de sufixos da árvore até encontrar um lugar de profundidade l; cada um dos lugares encontrados é inserido no conjunto E. Em seguida, uma nova árvore GT' é gerada; notadamente a árvore obtida a partir da união de todos os caminhos existentes em GT da raiz até um lugar do conjunto E. Também é preciso calcular os vetores C para cada lugar da árvore GT' recém gerada; isso é feito a partir dos valores gerados para os lugares em E, que serão folhas na nova árvore. A partir desse ponto o algoritmo é semelhante ao algoritmo anterior, exceto que a árvore GT' é processado em vez de GT.

4.5 Comentários Bibliográficos

Fellows e colegas [FGN06] demonstraram que o problema 28 é NP-Difícil, mesmo para o caso em que $d = d_H$. Em 1998, Sagot [Sag98] propôs os algoritmos, de complexidade não polinomial, para a extração de motifs repetidos e para a extração de motifs comuns, ambos sob a distância de Hamming, que discutimos aqui. Segundo a autora, ambos os algoritmos podem, facilmente, ser estendidos para versões mais gerais dos problemas, entre elas as versões nas quais a distância de Levenshtein é considerada. De fato, quatro anos depois, Adebiyi e Kaufmann [AK02] abordaram a extração de motifs comuns, sob a distância de Levenshtein. Em 2005, Iliopoulos e colegas [IMP+05], Sagot entre eles, propuseram três algoritmos para diferentes formulações da extração de motifs estruturados, todas polinomialmente tratáveis, entre as quais alguns casos particulares das formulações apresentadas por Marsan e Sagot [MS00a, MS00b].

Com o intuito de reduzir o espaço de busca, e, talvez, tornar o problema mais tratável, diversas versões restritas dos problemas acima surgiram nos últimos anos. Provavelmente, a mais discutida seja a versão proposta por Pevzner [PS00], chamada de **extração de motifs plantados**, e que enunciamos a seguir. Um capítulo inteiro de seu livro [Pev00] é dedicado ao problema.

Algoritmo 4.9 Segunda versão do algoritmo recursivo para a extração de motifs estruturados a partir da árvore dos sufixos generalizada. O algoritmo é adotado como sub-rotina pelo algoritmo EXTRAI-MOTIFS-ESTRUTURADOS

EXTRAI-MOTIFS-ESTRUTURADOS-REC-V2$((\langle M_1, \ldots, M_{i-1}\rangle, e_{min}, e_{max}), L, GT, p)$

1 **se** $i > 1$
2 **então**
3 **para todo** $g \in L$ **faça**
4 $L' \leftarrow L' \cup \{(h, 0)$: h é descendente de g
e $il + (i-1)e_{min} \leq$prof$(h) \leq il + (i-1)e_{max}\}$
5 **para todo** $h \in L$ **faça**
6 $h' \leftarrow$ lugar de profundidade l alcançavel a partir de
h via caminho de ligações de sufixo
7 **se** h' não está marcado como visitado
8 **então**
9 marque h' como visitado
10 $C_{h'} \leftarrow \langle 0 \ldots 0\rangle$
11 $E \leftarrow E \cup \{h'\}$
12 $C_{h'} \leftarrow C_{h'} \vee C_h$
13 $GT' \leftarrow$ árvore dos sufixos de profundidade máxima l
obtida a partir da união dos caminhos da raiz de GT até lugares em E
14 **senão**
15 $GT' \leftarrow GT$
16 $L' \leftarrow \{((\text{raiz}[GT'], \epsilon), 0)\}$
17 $L'' \leftarrow$ EXTRAI-MOTIFS-COMUNS$'(\epsilon, L', GT')$
18 **para todo** $(M_i, L_i) \in L''$ **faça**
19 **se** $i < p$
20 **então**
21 EXTRAI-MOTIFS-ESTRUTURADOS-REC-V2$((\langle M_1, \ldots, M_i\rangle, e_{min}, e_{max}), L_i, GT', p)$
22 **senão**
23 imprima "$(\langle M_1, \ldots, M_i\rangle)$ é um motif do conjunto S"

Problema 38 (Extração de motifs plantados)**.** *Dado um conjunto de cadeias S e inteiros r, q e l, encontrar todos as cadeias de comprimento l que ocorrem, com distância de Hamming exatamente r, em todas as cadeias de S.*

Durante nossa pesquisa não obtivemos conhecimento da existência de um algoritmo polinomial para a extração de motifs plantados. Rajasekaran e colegas [RBH05] propuseram um algoritmo exponencial, enquanto heurísticas são tratadas por Pevzner [PS00, KP02] e Buhler [BT02].

Uma abordagem muito interessante desses problemas, porém de interesse fundamentalmente teórico, consiste na determinação e extração de um conjunto de **motifs fundamentais**; um subconjunto do conjunto de todos os motifs comuns de acordo com os parâmetros de entrada, tal que todo motif não pertencente ao subconjunto possa ser gerado, em tempo constante, a partir de algum motif do subconjunto. Motifs fundamentais são discutidos em diversos trabalhos de Pisanti e colegas [Pis02, PCGS03a, PCGS03b].

Ao contrário do que ocorre com a busca aproximada de padrões, não é possível apontar um modelo para a extração de padrões aproximados como hegêmonico; pelo contrário: há uma profusão de formas de modelar computacionalmente o problema e ainda não é possível vislumbrar qual dentre estas prevalecerá na literatura sobre as demais. Pierre Peterlongo [PPBS05], em co-autoria com Pisanti e Sagot, apresentou a formulação a seguir, batisada de extração de repetições aproximadas, e propôs um algoritmo para resolvê-la, baseado em filtragem e que adota uma generalização do vetor dos sufixos batizada pelos autores de vetor de k-fatores.

Definição 39 (Repetições Aproximadas)**.** Dados um conjunto de cadeias $S = \{S_1, ..., S_n\}$ e inteiros r, l e q, chamamos de uma (r, q, l)-repetição em S^* a um conjunto de inteiros $\{\delta_1, ..., \delta_q\}$ tal que, para todo par de inteiros $i, j \in [1, q]$, vale que

$$d_H(S_{i\delta_i...\delta_i+l-1}, S_{j\delta_j...\delta_j+l-1}) \leq r.$$

Problema 40 (Extração de repetições aproximadas)**.** *Dado um conjunto de cadeias S e inteiros r, q e l, encontrar todos as (r, q, l)-repetições em S^*.*

Capítulo 5

Algoritmos de Filtragem

Um **filtro** é um algoritmo que descarta partes de uma entrada arbitrária para um problema específico; a rigor, apenas as partes que não satisfazem uma condição previamente estabelecida, chamada **critério de filtragem**. Desse modo, o tempo necessário para o processamento da entrada fornecida pode ser significativamente reduzido. Nessa segunda parte do texto, vamos discutir alguns filtros para o pré-processamento de instâncias da busca aproximada de padrões.

O critério de filtragem é uma condição necessária para que uma determinada parte da entrada seja preservada pelo filtro, porém não é condição suficiente para que a mesma seja relevante na busca por uma solução. Logo, um filtro não é capaz de resolver uma dada instância de um problema; é preciso usá-lo em conjunto com um **algoritmo secundário**, capaz de receber as partes da entrada original, preservadas pelo filtro, e devolver uma solução correspondente.

Um algoritmo de filtragem é **sem perda** se, para toda instância do problema, apenas partes irrelevantes da entrada são removidas; isto é, apenas aquelas que não são imprescindíveis na busca por uma solução. Do contrário, dizemos que o filtro é **com perda** (e.g. no caso da busca aproximada de padrões, um filtro é sem perda se qualquer algoritmo exato para o problema for capaz de encontrar todas as ocorrências do padrão no texto original, recebendo como entrada apenas a saída do filtro). A exemplo do BLAST [AGM$^+$90], filtros com perda têm sido amplamente adotados hoje em dia, em razão da velocidade com que são processados e do fato dos possíveis erros provenientes de seu uso não comprometerem a viabilidade da solução, para a grande

maioria das aplicações práticas. Porém, nosso trabalho será fortemente focado em algoritmos sem perda e essa escolha é facilmente justificável: o uso de tais algoritmos aumentou sensivelmente nos últimos anos, principalmente em virtude do tempo de processamento médio ter sido reduzido por várias ordens de magnitude, além das técnicas envolvidas em seu estudo se estenderem por diversos ramos da computação e sua aplicabilidade ser bastante diversa, como vimos na introdução deste trabalho.

Um filtro é composto por duas etapas bem definidas. Na primeira (**etapa de filtragem**), o texto é inspecionado e, com base no critério de filtragem, as partes irrelevantes são descartadas. Na segunda etapa (**etapa de verificação**), as partes preservadas na etapa anterior são encaminhadas ao algoritmo secundário, possivelmente uma por uma, e uma solução para a instância original é construída a partir dos resultados fornecidos pelo algoritmo secundário.

A eficiência de um algoritmo de filtragem sem-perda é avaliada, teórica e experimentalmente, com base em suas necessidades de espaço e tempo e em sua **eficiência de filtragem**; isto é, a competência em reduzir o tamanho da entrada. A eficiência de filtragem é, em grande parte, ditada pelo critério de filtragem adotado. Um critério de filtragem muito "agressivo" pode levar à remoção de partes relevantes do texto, por outro lado um critério de filtragem muito "conservador" pode tender à preservação da maior parte do texto, sobrecarregando, assim, a etapa de verificação. Seja X uma cadeia e Λ uma função de Σ^* no conjunto dos reais, que quantifica uma determinada propriedade da cadeia dada; o modo como tal função opera é irrelevante no momento. Vamos supor que $\Lambda(X) \succcurlyeq l$ seja o critério de filtragem adotado por um determinado algoritmo de filtragem, $\succcurlyeq$ um operador de comparação qualquer e l um valor de referência previamente determinado; se a condição $\Lambda(X) \succcurlyeq l$ é satisfeita, então a cadeia X deve ser preservada pela etapa de filtragem, caso contrário X deve ser descartada. Ambos, eficiência de filtragem e tempo de execução do algoritmo, são intimamente dependentes de l. Se alterarmos levemente o valor de l, é possível fazer uma "sintonia fina" na condição dada, aumentando ou reduzindo a eficiência de filtragem, assim como a velocidade de execução. Chamamos de **limiar ótimo de filtragem** ao valor de l para o qual o algoritmo e critério de filtragem adotados promovem a remoção da maior fração possível do texto, sem comprometer o caráter sem perda do algoritmo.

É importante notar que um filtro não promove, no pior caso, uma redução no tempo necessário para processar a entrada, mas apenas no caso médio; sempre há a possibi-

lidade do filtro fornecer ao algoritmo secundário exatamente aquilo que recebeu, sem que nenhuma parte da entrada tenha sido removida. Logo, é de se esperar que uma análise representativa da eficiência de algoritmos de filtragem tome como base o caso médio, precisando, para isso, assumir certas propriedades quanto à distribuição dos dados da entrada, o que nem sempre é conveniente. Com isso, torna-se muito importante a análise experimental de tais algoritmos. Evidentemente, é desejável que o filtro reduza tanto quanto o possível o tamanho da entrada, porém é fundamental que a etapa de filtragem seja concluída rapidamente, caso contrário a adoção do filtro pode ser pouco benéfica ou, em casos extremos, prejudicial. Logo, o projeto de um algoritmo que adote filtragem representa, em grande parte, a busca por um ponto de equilíbrio entre tempo de execução e eficiência de filtragem.

Algoritmos de filtragem para a busca aproximada de padrões procuram determinar e, de algum modo, quantificar um conjunto de características estruturais comuns a segmentos do texto e o padrão. Os segmentos que apresentem características do padrão, em quantidade suficiente, serão preservados, caso contrário serão descartados. Em outras palavras, o filtro procura identificar as partes do texto que se assemelhem ao padrão e, portanto, podem conter alguma ocorrência do mesmo.

De modo geral, filtros para a busca aproximada de padrões constroem alguma estrutura de índices a partir do texto, do padrão ou de ambos. Naturalmente, essa particularidade torna o uso de filtros especialmente interessante para aplicações da busca aproximada de padrões em textos estáticos: o tempo consumido pela construção da estrutura de índices é desconsiderado durante a análise. Por outro lado, o tempo gasto pelos filtros conhecidos para a busca em textos dinâmicos é diretamente proporcional ao erro permitido, tornando-os pouco interessantes para a busca com muitos erros [Nav01]. Logo, o foco de nosso interesse neste capítulo será a busca em textos estáticos, porém a busca em textos dinâmicos não será completamente deixado de lado.

A eficiência de filtragem para a busca aproximada de padrões é determinada pelo grau de ocorrências de **falsos positivos** e **falsos negativos**. Falsos positivos são partes da entrada preservadas pelo filtro e que não abrigam, de fato, alguma ocorrência do padrão. Falsos negativos são partes da entrada descartadas pelo filtro e que contêm alguma ocorrência do padrão. Como filtros sem perda não geram falsos negativos, estamos interessados apenas na ocorrência de falsos positivos, portanto podemos ousar afirmar que quanto menor a ocorrência de falsos positivos mais eficiente é o filtro. Em

um filtro ideal, toda posição preservada faz parte de alguma ocorrência do padrão, isto é, não há falsos positivos.

5.1 Filtros baseados em q-gramas

Como já foi dito, filtros para a busca aproximada de padrões procuram identificar os segmentos do texto que se assemelhem ao padrão, e portanto devem ser preservados pela etapa de filtragem. Um modo, adotado com freqüência, de quantificar essa semelhança consiste na contagem do número de cadeias que são fatores do texto e também do padrão. Entretanto, em vez de considerarmos todos os segmentos, o que seria pouco informativo, é interessante nos restringirmos a cadeias de um comprimento específico, definido de antemão. É tendo isso em mente que vamos introduzir, nessa seção, o conceito de q-grama. O conceito é simples ao ponto de dispensar um formalismo e denotação específica. Porém, para facilitar a introdução às suas extensões, que veremos nas próxima seções, vamos seguir a tendência da literatura e adotá-lo formalmente.

Definição 41. **q-grama** de uma cadeia X é todo fator de X de comprimento q.

`Exemplo:` a cadeia é `abcabcd`. Para $q = 2$, suas q-gramas são: `ab`, `bc`,`ca`, `ab`, `bc` e `cd`.

A quantificação da semelhança entre duas cadeias, com base em suas q-gramas, se dá através da definição e adoção de uma função de distância entre cadeias, baseada no número de q-gramas "comuns" as cadeias. Observe a definição a seguir:

Definição 42. A **distância de q-gramas** entre duas cadeias $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$ é

$$d_q(X, Y) = \sum_{Z \in \Sigma^q} |\kappa(Z, X) - \kappa(Z, Y)|.$$

Lembrando que $\kappa(Z, X)$ é o número de ocorrências de Z como fator de X.

`Exemplo:` as cadeias são `abcabcd` e `abcd`. Para $q = 2$ a distância de q-gramas entre ambas é $d_q = 3$.

Neste momento, vamos nos permitir um pequeno abuso de notação: a função apresentada não é, de fato, uma métrica. A propriedade 1.1c da definição 11 não é satisfeita, uma vez que podemos ter $d_q(X, Y) = 0$ e $X \neq Y$. Por exemplo, a propriedade não é satisfeita para $q = 2$, $X = \texttt{aabda}$ e $Y = \texttt{abdaa}$. Como d_q satisfaz as demais propriedades da definição 11, ela é uma pseudo-métrica.

Teorema 43. *A distância de q-gramas é uma pseudo-métrica.*

Demonstração. Claramente, as propriedades 1.1a e 1.1b, da definição 11, valem. Agora, para provar a desigualdade triangular, vamos considerar três cadeias X, Y e Z. Note que é suficiente provar que, para qualquer cadeia $\alpha \in \Sigma^q$, temos

$$|\kappa(\alpha, X) - \kappa(\alpha, Y)| \leq |\kappa(\alpha, X) - \kappa(\alpha, Z)| + |\kappa(\alpha, Y) - \kappa(\alpha, Z)|.$$

Mas, isso é evidente. Basta considerar todas as 6 possíveis relações de ordem entre $\kappa(\alpha, X)$, $\kappa(\alpha, Y)$ e $\kappa(\alpha, Z)$, e verificar que a desigualdade vale para cada uma delas. □

5.1.1 O Teorema das q-gramas

Agora, estamos prontos para enunciar o seguinte teorema, que sustenta os filtros que veremos nesta seção.

Teorema 44. *Considere duas cadeias $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$, uma métrica $d \in \{d_H, d_L\}$ e inteiros j e r, tais que j seja a posição final de uma (d, r)-ocorrência de X em Y. Pode-se afirmar que $d_q(X, Y_{j-m+1\dots j}) \leq 2qr$, para todo inteiro positivo q.*

Demonstração. É suficiente provar que o teorema vale para $d = d_L$. Seja X' uma (d, r)-ocorrência de X em Y que termine em y_j e chamemos a cadeia $Y_{j-m+1\dots j}$ de Y'. Note que X' pode ser obtida a partir de X após, no máximo, r operações de edição, o que nos permite construir uma seqüência de cadeias $\mathcal{C} = \langle C_1 = X, C_2, C_3, \dots, C_{r'+1} = X' \rangle$ tal que $r' \leq r$ e $d(C_i, C_{i+1}) = 1$. Seja $P = \langle p_1, p_2, \dots, p_{r'} \rangle$ a seqüência de operações de edição tal que p_i transforma C_i em C_{i+1}. Vamos escolher um inteiro positivo qualquer q e tomar uma operação p_i de P. Se p_i é de substituição, digamos, do caractere na posição k de C_i, então, no máximo, q das q-gramas de C_i não são q-gramas de C_{i+1} (notadamente, aquelas que contêm o caractere na posição k de C_i) e q das q-gramas

de C_{i+1} não são q-gramas de C_i (aquelas que contêm o caractere na posição k de C_{i+1}). Logo, $d_q(C_i, C_{i+1}) \leq q + q = 2q$. Por outro lado, se p_i é de inserção, digamos, de um caractere α entre posições k_1 e k_2 de C_i, então, no máximo, $q - 1$ das q-gramas de C_i não são q-gramas de C_{i+1} (aquelas que contêm ambos os caracteres nas posições k_1 e k_2) e q das q-gramas de C_{i+1} não são q-gramas de C_i (aquelas que contêm α). Logo, $d_q(C_i, C_{i+1}) \leq q + q - 1 = 2q - 1$. Observe que a desigualdade também vale quando p_i é uma operação de remoção.

Seja r'_1 o número de operações de inserção, ou remoção, em P, e r'_2 o número de operações de substituição. Pela desigualdade triangular,

$$\begin{aligned} d_q(X, X') &\leq \sum_{1 \leq i \leq r'} d_q(C_i, C_{i+1}) \\ &\leq r'_1(2q-1) + r'_2(2q) \\ &= 2qr'_1 - r'_1 + 2qr'_2 \\ &= 2q(r'_1 + r'_2) - r'_1 \\ &= 2qr' - r'_1. \end{aligned}$$

Agora, observe que X' é um sufixo de Y' ou Y' é um sufixo de X'. Note, também, que $|X'| \leq |Y'| + r'_1$. Daí, segue que $d_q(X', Y') \leq r'_1$. Sumarizando, e novamente pela desigualdade triangular, temos que

$$\begin{aligned} d_q(X, Y') &\leq d_q(X, X') + d_q(X', Y') \\ &\leq 2qr' - r'_1 + r'_1 \\ &= 2qr'. \end{aligned}$$

□

5.1.2 Cálculo da Distância de q-gramas

Com certa dose de criatividade, podemos adotar a distância de q-gramas para o desenvolvimento de um filtro para a busca aproximada de padrões. De modo geral, basta determinar, para cada fator do texto, de comprimento m, a distância de q-gramas até o padrão. Os fatores cuja distância for superior a $2qr$ terão sua última posição considerada para descarte, caso contrário a posição será preservada. Nesse primeiro algoritmo

que veremos, a condição $d_q(X, Y) \leq l$ é o critério de filtragem adotado. Aqui, l é o **limiar de filtragem**, isto é, o referencial adotado para decidir quais posições do texto serão preservadas. A seguir fornecemos um exemplo que, com base no teorema anterior, mostra que $l = 2qr$ é o limiar ótimo (mínimo) para o critério de filtragem em questão, a saber, o limiar de filtragem que garante que a execução do algoritmo seja sem perda e com eficiência de filtragem máxima entre todas as execuções possíveis: neste caso, o menor valor que pode ser adotado como limiar de filtragem.

> `Exemplo:` As cadeias são $X =$`acagctta` e $Y =$`acacctta` e as distâncias de Hamming e Levenshtein entre ambas é 1. Para $q = 3$ e $r = 1$ temos $d_q(X, Y) = 2qr = 6$. Portanto, $2qr - 1$ não é um delimitante superior para a distância de q-gramas entre cadeias (d_L, r)-vizinhas. Como $2qr$ é um delimitante superior, concluímos que é mínimo.

A seguir, veremos dois métodos para calcular a distância de q-gramas entre duas dadas cadeias X e Y de modo eficiente.

Primeiro Método Extremamente simples, consiste na contagem do número de ocorrências de cadeias de comprimento q, em ambas as cadeias X e Y, através de uma tabela hash. Um modo, muito natural, de concebermos uma função hash para essa tarefa é, para cada $Z \in \mathsf{Ft}_q(X)$, codificar Z como um inteiro na base $|\Sigma|$. Sejamos mais específicos: seja $|\Sigma| = p$, $\Sigma = \{\sigma_0, \ldots, \sigma_{p-1}\}$ e $Z = x_1, \ldots, x_q$. O código inteiro de Z é

$$\tilde{Z} = \tilde{x}_1 p^{q-1} + \tilde{x}_2 p^{q-2} + \ldots + \tilde{x}_q p^0, \text{onde } \tilde{x}_i = j \text{ se } x_i = \sigma_j.$$

Agora, como é de costume, seja $X = x_1 \ldots x_m$ e, ainda, seja $Z_i = x_i \ldots x_{i+q-1}$. Temos que

$$\tilde{Z}_i = (\tilde{Z}_{i-1} - \tilde{x}_{i-1} p^{q-1})p + \tilde{x}_{i+q-1}.$$

Com isso, ficou evidente que podemos calcular a codificação inteira, na base p, para todas as q-gramas de X em tempo linear. Mais ainda: para cada uma dessas q-gramas, podemos calcular o número de posições nas quais a mesma ocorre em X. É o que faz o algoritmo a seguir, que devolve o vetor $V_X[0 \ldots p^q - 1]$, tal que $V_X[\tilde{Z}]$ contém o número de ocorrências de Z em X.

VETOR-DE-Q-OCORRÊNCIAS(X, q)

1 $V_X[0\,..\,p^q - 1] \leftarrow \langle 0, \ldots, 0 \rangle$
2 $\tilde{Z}_1 \leftarrow \tilde{x}_1 p^{q-1} + \tilde{x}_2 p^{q-2} + \ldots + \tilde{x}_q p^0$
3 $V_X[\tilde{Z}_1] \leftarrow 1$
4 **para** $i \leftarrow 2$ **até** $m - q + 1$ **faça**
5 $\quad \tilde{Z}_i \leftarrow (\tilde{Z}_{i-1} - \tilde{x}_{i-1} p^{q-1}) p + \tilde{x}_{i+q-1}$
6 $\quad V_X[\tilde{Z}_i] \leftarrow V_X[\tilde{Z}_i] + 1$
7 **devolva** V_X

O consumo de tempo total do algoritmo é dominado pelo consumo das linhas 1 e 4. Logo, o consumo de tempo do algoritmo é proporcional a $(p^q) + (m - q + 1 - 2 + 1) = O(p^q + m)$. Aqui, cabe uma pequena observação: no artigo original Ukkonen desconsidera o tempo necessário para a inicialização do vetor V_X (linha 1). Curiosamente, o tempo gasto nessa operação está longe de ser desprezível, chegando a dominar o consumo de tempo total do algoritmo. Por um outro lado, em termos práticos e se tratando de filtros para aplicações de biologia computacional, é razoável considerarmos $p^q = O(m)$; em geral os alfabetos são pequenos, as cadeias possuem comprimento enorme e os testes experimentais desencorajam o uso de valores de q superiores a 20 [BCF+99]. Após executar o algoritmo, para ambas as cadeias, o somatório abaixo nos dará a distância de q-gramas entre X e Y.

$$d_q(X, Y) = \sum_{i \in (\mathsf{Ft}(X) \cup \mathsf{Ft}(Y))} |V_X[i] - V_Y[i]|.$$

Segue o algoritmo completo para o cálculo da distância:

DISTÂNCIA-DE-Q-GRAMAS-VERSÃO-1(X, Y, q)

1 $V_X \leftarrow$ VETOR-DE-Q-OCORRÊNCIAS(X, q)
2 $V_Y \leftarrow$ VETOR-DE-Q-OCORRÊNCIAS(Y, q)
3 $d \leftarrow 0$
4 **para todo** $Z \in (\mathsf{Ft}(X) \cup \mathsf{Ft}(Y))$ **faça**
5 $\quad d \leftarrow d + |V_X[Z] - V_Y[Z]|$
6 **devolva** d

O consumo de tempo do algoritmo é dado pelo consumo das linhas 1, 2 e 4. Logo, o algoritmo consome tempo proporcional a $O(p^q+m)+O(p^q+n)+O(m+n) = O(p^q+m+$

$n)$. Por sua vez, o consumo de espaço é dado pelo espaço usado para abrigar o conjunto $\mathsf{Ft}(X) \cup \mathsf{Ft}(Y)$ e cada um dos vetores V_X e V_Y, ou seja $O(p^q + m + n)$. Sumarizando, todo o processo consome tempo $O(p^q + m + n)$ e espaço $O(p^q + m + n)$. Logo, é verdadeira a seguinte afirmação.

Teorema 45. *A distância $d_q(X, Y)$ entre duas cadeias $X = x_1 \ldots x_m$ e $Y = y_1 \ldots y_n$ pode ser calculada em tempo $O(p^q + m + n)$ e espaço $O(p^q + m + n)$, onde $p = |\Sigma|$.*

Note que ambos os vetores V_X e V_Y são pouco aproveitados e esparsos: no máximo, apenas $m + n - 2q + 2 = O(m + n)$ elementos de cada vetor são efetivamente utilizados e, no máximo, $m - q + 1$ elementos de V_X, e $n - q + 1$ de V_Y, são diferentes de zero. Em virtude disso, a adoção de uma função hash diferente da apresentada, como exposto por Karp e Rabin [KR87], pode reduzir o consumo de espaço do algoritmo proposto para $O(m + n)$.

Apesar de mais onerosa do que a tabela de busca de palavras, uma estrutura de dados mais flexível foi sugerida por Burkhardt [BCF+99] e se baseia no uso misto de um vetor dos sufixos e da TBP para armazenar as ocorrências das q-gramas no texto. Entraremos em maiores detalhes na próxima seção.

Segundo Método No artigo original, Ukkonen adotou o uso de um autômato dos sufixos levemente modificado e lembrou que árvores dos sufixos também poderiam ser adotadas, porém sem dar maiores detalhes de como isso poderia ser feito. Aqui, optamos por desenvolver, inteiramente, uma solução baseada no uso de árvores dos sufixos. Assim como o algoritmo para a extração de motifs estruturados, que vimos anteriormente, esse método faz uso das ligações de sufixos da árvore, que devem, portanto, ser preservadas após a construção da estrutura de dados. A rigor, vamos precisar de informações da árvore até uma profundidade máxima q, logo, para economizar espaço e simplificar a implementação do algoritmo, iremos construir a árvore podando-a de acordo, isto é, todo lugar de profundidade q na árvore original será um folha na árvore obtida após a poda[1].

Seja $G_q(X)$ o conjunto de q-gramas de X. Da definição da distância de q-gramas,

[1] Essa estrutura é conhecida como árvore de fatores k-profundos, e maiores detalhes podem ser obtidos em [AS04]. Note que, em nosso caso, mesmo com a poda o consumo de espaço da árvore permanece assintóticamente linear no comprimento da cadeia.

vale a equação a seguir:

$$d_q(X,Y) = \sum_{Z \in G_q(X)} |\kappa(Z,X) - \kappa(Z,Y)| + \sum_{W \in (\Sigma^q \setminus G_q(X))} \kappa(W,Y).$$

Logo, para calcular a distância entre X e Y precisamos apenas considerar o número de ocorrências, em ambas as cadeias X e Y, das cadeias que são q-gramas de X e o número de ocorrências em Y de q-gramas de Y que não ocorrem em X.

O algoritmo é logicamente dividido em duas etapas: na primeira é construída T, a árvore dos sufixos de X como descrita anteriormente, e a cadeia X é soletrada em T; na segunda etapa, Y é soletrada em T.

É também na primeira etapa que o algoritmo irá determinar:

- $\langle G_1, .., G_Q \rangle$, seqüência dos elementos do conjunto $G_q(X)$, ordenados pela posição da primeira ocorrência em X;
- o valor I_l tal que, para cada lugar l em T, dentre os que possuem profundidade q, $I_l = i$ se $\lambda(l) = G_i$;
- o vetor $P_X[1..Q]$ tal que, para cada lugar l em T, dentre os que possuem profundidade q, $P_X[I_l]$ contém o número de ocorrências de $\lambda(l)$ em X.

A primeira etapa, após a construção da árvore, consiste em soletrar X de modo semelhante ao tradicional, com apenas uma pequena modificação: sempre que não existir uma transição na árvore que corresponda ao último caractere lido, isto é, sempre que o lugar corrente possuir profundidade q, o algoritmo atualiza I_l e P_X, movendo-se em seguida através da ligação de sufixo, sem que um novo caractere seja lido, e o atual, descartado.

Na segunda etapa, o algoritmo irá determinar o vetor $P_Y[1..Q]$ tal que, para cada lugar l em T, dentre os que possuem profundidade q, $P_Y[I_l]$ contém o número de ocorrências de $\lambda(l)$ em Y. Assim como na primeira etapa, P_y é atualizado a cada vez em que um lugar de profundidade q é alcançado. Porém, note que, ao soletrar Y, apenas verificar a não-existência de transição que corresponda ao caractere corrente não é suficiente para determinar se a profundidade atual é igual a q: é preciso verificar se o lugar atual é, efetivamente, uma folha de T, ou anotar a profundidade corrente a cada iteração. Por simplicidade, esses detalhes foram omitidos do algoritmo apresentado.

Ao término da segunda etapa, a distância de q-gramas entre X e Y é dada por:

$$\sum_{i=1}^{Q} |P_X[i] - P_Y[i]| + (n - q + 1) - \sum_{j=1}^{Q} P_Y[j]. \tag{5.1}$$

Teorema 46. *A distância $d_q(X, Y)$ entre duas cadeias $X = x_1...x_m$ e $Y = y_1 \dots y_n$ pode ser calculada em tempo $O(m|\Sigma| + n)$ e espaço $O(m|\Sigma|)$.*

5.2 O Algoritmo de Ukkonen

Como visto anteriormente, o teorema das q-gramas nos fornece uma delimitação superior para a distância de q-gramas entre duas cadeias (d, r)-vizinhas. Com base nesse teorema, diversos autores propuseram filtros para a busca aproximada de padrões, com destaque para Ukkonen que, em dois trabalhos seminais, propôs um algoritmo de filtragem para a busca aproximada de padrões em textos estáticos [Ukk92] e um para a busca em textos dinâmicos, esse último tendo Jokinen como autor principal [JU91]. Nesta seção, trataremos do algoritmo de Ukkonen para textos dinâmicos. O algoritmo inicia com uma etapa de pré-processamento, durante a qual são calculadas as distâncias de X até todos os fatores de Y de comprimento m com base no método para o cálculo da distância de q-gramas através da árvore dos sufixos. Essa etapa de pré-processamento consome tempo $O(m|\Sigma| + n)$, porém não entraremos em maiores detalhes, os quais podem ser conseguidos no artigo original [JU91]. O resultado dessa etapa é armazenado no vetor $V[1\,..\,m]$ de modo que $V[j]$ é a distância de q-gramas entre X e $Y[j - m + 1\,..\,j]$. Após o pré-processamento, o algoritmo percorre o vetor V procurando posições em Y candidatas a posições finais de uma (d, r)-ocorrência de X, isto é, posições j tais que $V[j] \leq 2qr$. Para cada posição candidata j encontrada, a cadeia $x[j - (m + r) + 1\,..\,j]$ é considerada e, portanto, fornecida ao algoritmo auxiliar: isso garante com que todas as possíveis ocorrências do padrão sejam levadas em conta, sem perda. Um certo cuidado na manipulação dos índices se faz necessário para garantir que as partes enviadas ao algoritmo sejam disjuntas. A validade da abordagem é diretamente inferida a partir do teorema das q-gramas e do que foi discutido até o momento, contudo apresentamos uma demonstração detalhada do funcionamento do algoritmo, a partir de suas relações invariantes, nos apêndices finais desse trabalho. É

Algoritmo 5.1 Algoritmo SOLETRA, que recebe uma cadeia P e uma árvore dos sufixos T, construída sobre uma cadeia X, e devolve o lugar (u, Y) em T e cujo rótulo é o maior prefixo comum a P e um sufixo de X. SOLETRA também devolve o nó v, ponta do arco associado a (u, Y).

DISTÂNCIA-DE-Q-GRAMAS-VERSÃO-2(X, Y, q)

```
 1  T ← árvore dos sufixos de X, podada a profundidade q
 2  I_l ← 0, para todo lugar l de profundidade q
 3  g ← raiz de T
 4  para i ← 1 até m faça
 5     se existe transição t = g → h : x_i = λ(t)
 6      então g ← h
 7      senão
 8           se I_g = 0
 9            então
10              I_g ← i
11              P_X[I_g] = 1
12            senão P_X[I_g] ← P_X[I_g] + 1
13           g ← ligação-de-sufixo(g)
14           i ← i − 1
15  g ← raiz de T
16  P_Y[1..Q] ← ⟨0..0⟩
17  para i ← 1 até n faça
18     se existe transição t = g → h : y_i = λ(t)
19      então g ← h
20      senão
21           se |λ(g)| = q
22            então P_Y[I_g] ← P_Y[I_g] + 1
23           g ← ligação-de-sufixo(g)
24           i ← i − 1
25  devolva Σ_{i=1}^{Q} |P_X[i] − P_Y[i]| + (n − q + 1) − Σ_{j=1}^{Q} P_Y[j]
```

```
FILTRO-DE-UKKONEN(X, Y, r, q)
 1  V[1..n] ← CALCULA-DISTÂNCIAS(X, Y, q)
 2  S ← ∅
 3  i ← 1
 4  k ← 0
 5  para j ← m até n faça
 6     se V[j] ≤ 2qr
 7      então k ← j
 8      senão
 9           se j − i + 1 = m + r
10            então
11             se Y_{i..k} ≠ ϵ
12              então
13                   S' ← ALGORITMO-SECUNDÁRIO(X, Y_{i..k}, r)
14                   S ← S ∪ (S' ⊕ i)
15                   i ← k + 2
16           senão i ← i + 1
17     devolva S
```

evidente que, em adição ao tempo gasto no pré-processamento, o consumo de tempo do algoritmo é proporcional ao número de posições do texto que são fornecidas ao algoritmo secundário e ao consumo de tempo do mesmo; vamos chamar esse valor de ω. No pior caso nenhuma posição do texto é descartada, logo, se supormos como algoritmo secundário o algoritmo de programação dinâmica tradicional, cujo consumo de tempo é $O(mn)$, temos um consumo de tempo $O(n\omega + m\omega)$. [2]

5.3 O Algoritmo de Jokinen e Ukkonen

No problema da busca aproximada de padrões em textos estáticos, o tempo gasto para pré-processar o texto não é considerado durante a análise. A motivação se dá a partir de aplicações onde diversas buscas, de padrões distintos, serão realizadas em um texto imutável; um cenário recorrente na resolução de problemas da biologia molecular. Como o custo do pré-processamento do texto será amortizado por diversas operações de busca, sua relevância não é fundamental para a eficiência da solução final.

O algoritmo que veremos nesta seção foi proposto por Jokinen e Ukkonen [JU91] e inspirou o surgimento de toda uma classe de filtros baseados em q-gramas, como veremos. Após a etapa de pré-processamento, o teorema 44 é tomado como base para a determinação das partes do texto que serão descartadas.

5.3.1 O Pré-Processamento

Para pré-processar o texto, vamos adotar uma estrutura de dados chamada de **tabela de busca de palavras**, abreviadamente **TBP**. Apesar do nome sofisticado, a estrutura não passa de um simples vetor de listas ligadas de inteiros, indexado através de uma função hash. Podemos adotar a mesma representação de cadeias, em inteiros na base

[2] No artigo original, Ukkonen exemplifica o algoritmo secundário com a adoção do algoritmo Ukkonen-Wood, cujo consumo de tempo é $O(rn)$. O seu uso, em conjunto com o algoritmo aqui apresentado, requer uma certa dose de engenhosidade e foge do escopo de nosso trabalho; é suficiente dizer que, nesse caso, a etapa de processamento passa a consumir tempo $O(m|\Sigma| + m^2)$ e a filtragem $O(n + \omega r^2)$. No pior caso, o consumo permanece o que seria gasto pelo algoritmo secundário quando aplicado integralmente ao texto: $O(rn)$.

$|\Sigma|$, que aplicamos para calcular a distância de q-gramas, ou ainda o proposto por Karp e Rabin [KR87]. Por simplicidade, optamos pela primeira alternativa.

O texto Y é pré-processado como segue: para cada cadeia Z de Σ^q, vamos construir uma lista W_Z contendo todas as posições onde Z ocorre em Y. Modificando levemente o algoritmo VETOR-DE-Q-OCORRÊNCIAS, podemos construir o conjunto $\{W_Z : Z \in \Sigma^q\}$ em tempo $O(|\Sigma|^q + n)$. É o que faz o algoritmo a seguir, que devolve a tabela de busca de palavras para as q-gramas de uma cadeia Y:

TABELA-DE-Q-PALAVRAS(Y, q)
1 $W_Y[0, \ldots, |\Sigma|^q - 1] \leftarrow \langle \emptyset, \ldots, \emptyset \rangle$
2 $\tilde{Z}_1 \leftarrow \tilde{y}_1 p^{q-1} + \tilde{y}_2 p^{q-2} + \ldots + \tilde{y}_q p^0$
3 $W_Y[\tilde{Z}_1] \leftarrow \{1\}$
4 **para** $i \leftarrow 2$ **até** $m - q + 1$ **faça**
5 $\quad \tilde{Z}_i \leftarrow (\tilde{Z}_{i-1} - \tilde{y}_{i-1} p^{q-1})p + \tilde{y}_{i+q-1}$
6 $\quad W_Y[\tilde{Z}_i] \leftarrow W_Y[\tilde{Z}_i] \cup \{i\}$
7 **devolva** W_Y

Claramente, o consumo de tempo do algoritmo é $O(m + |\Sigma|^q)$. O consumo de espaço é dado pelo consumo das listas, $m - q + 1 = O(m)$, e do vetor, $O(|\Sigma|^q)$, resultando em um consumo de espaço total $O(m + |\Sigma|^q)$.

5.3.2 A Etapa de Filtragem

Agora, vamos supor que já temos em mãos a tabela de busca de palavras, construída para as q-gramas de Y, e pretendemos encontrar todas as (d_L, r)-ocorrências de X em Y.

A estratégia é a que segue: para acelerar a filtragem, vamos considerar o texto dividido em blocos de tamanho $2(m-1)$ e, através da TBP, contar o número de ocorrências de q-gramas de X em cada um dos blocos. Com base no teorema 44, os blocos cujo número de ocorrências for satisfatório serão preservados para a etapa de verificação. O tamanho definido para cada bloco implica em uma sobreposição de tamanho $m - 1$ entre cada par de blocos consecutivos. Essa sobreposição é necessária para garantir

Algoritmo 5.2 Etapa de filtragem do algoritmo Jokinen-Ukkonen.

```
1  B[0, . . , ⌈n/(m−1)⌉ + 1] ← ⟨0, . . , 0⟩
2  Z̃_1 ← x̃_1 p^{q−1} + x̃_2 p^{q−2} + . . + x̃_q p^0
3  i ← 1
4  V ← ∅
5  enquanto i ≤ m − q + 1 faça
6     se Z̃_i ∉ V
        então
7        para todo j ∈ W_Y[Z̃_i] faça
8           B_{⌊(j−1)/(m−1)⌋} ← B_{⌊(j−1)/(m−1)⌋} + 1
9           B_{⌊(j−1)/(m−1)⌋+1} ← B_{⌊(j−1)/(m−1)⌋+1} + 1
10          V ← V ∪ {Z̃_i}
11    i ← i + 1
12    Z̃_i ← (Z̃_{i−1} − x̃_{i−1} p^{q−1})p + x̃_{i+q−1}
   . . .
```

que o filtro não tenha perda, pois, desse modo, todo m-fator de Y estará inteiramente "coberto" por algum dos blocos. Logo, caso alguma posição de Y seja final de uma (d, r)-ocorrência de X em Y então, pelo teorema 44, o bloco que a cobre inteiramente conterá, pelo menos, $m + 1 - (r + 1)q$ ocorrências de q-gramas de X, atingindo o limiar de filtragem. Não é difícil ver que essa abordagem pode implicar em um grande número de falsos-positivos: o número de ocorrências de q-gramas de X na região coberta por algum bloco pode atingir o limiar de filtragem, no entanto sem que alguma das posições da região o faça. Desse modo a etapa de filtragem é concluída mais rapidamente do que se as posições do texto fossem consideradas em separado. Trata-se de uma simples troca: maior velocidade de processamento em detrimento da eficiência de filtragem. No algoritmo 5.2 apresentamos a etapa de filtragem do algoritmo. O consumo de tempo é dominado pelo consumo das linhas 7-9. Logo, o consumo de tempo da etapa de filtragem do algoritmo é $O(m + n)$.

Algoritmo 5.3 Etapa de verificação do algoritmo Jokinen-Ukkonen.

```
1  ...
2  S ← ∅
3  j ← 1
4  para i ← 0 até ⌈n/(m−1)⌉ + 1 faça
5     se B[i] ≥ m + 1 − (r + 1)q
       então
6           k ← (i + 1)(m − 1)
7      senão
8           se Y_{j..k} ≠ ϵ
             então
9             S′ ← ALGORITMO-SECUNDÁRIO(X, Y_{j..k}, r)
10            S ← S ∪ (S′ ⊕ j)
11          j ← i(m − 1) − m − k + 2
12 devolva S
```

5.3.3 A Etapa de Verificação

Cada um dos $\lceil \frac{n}{m-1} \rceil + 2$ blocos é verificado quanto ao número de ocorrências de q-gramas do padrão, devidamente contabilizadas na etapa anterior (filtragem). Se o limiar de filtragem é atingido por algum dos blocos, digamos B_i, então, pelo teorema 44, a posição final de algum m-fator de Y "coberto" por B_i pode ser final de uma (d, r)-ocorrência do padrão X. Logo, uma (d, r)-ocorrência de X pode existir na cadeia $Y_{j..k}$, onde $j = i(m-1) - m - r + 2$ e $k = (i+1)(m-1)$. Essa parte do texto será verificada pelo algoritmo secundário.

5.3.4 Resultados Experimentais

No artigo original, Jokinen e Ukkonen [JU91] não oferecem nenhuma consideração quanto à eficiência de filtragem de seu algoritmo, análise do consumo de tempo esperado ou dados experimentais sobre o desempenho do mesmo na prática. Feliz-

Algoritmo 5.4 Algoritmo Jokinen-Ukkonen. O algoritmo recebe duas cadeias X e Y, assim como um inteiro q e W_Y, a tabela de busca de palavras de Y. Adotando uma sub-rotina, o algoritmo irá encontrar todas as $(d, r)-$ocorrências de X em Y.

JOKINEN-UKKONEN(X, Y, W_Y, q)

1 $B[0, . . , \lceil \frac{n}{m-1} \rceil + 1] \leftarrow \langle 0, . . , 0 \rangle$
2 $\tilde{Z}_1 \leftarrow \tilde{x}_1 p^{q-1} + \tilde{x}_2 p^{q-2} + . . + \tilde{x}_q p^0$
3 $i \leftarrow 1$
4 $V \leftarrow \emptyset$
5 **enquanto** $i \leq m - q$ **faça**
6 **se** $\tilde{Z}_i \notin V$
então
7 **para todo** $j \in W_Y[\tilde{Z}_i]$ **faça**
8 $B_{\lfloor \frac{j-1}{m-1} \rfloor} \leftarrow B_{\lfloor \frac{j-1}{m-1} \rfloor} + 1$
9 $B_{\lfloor \frac{j-1}{m-1} \rfloor + 1} \leftarrow B_{\lfloor \frac{j-1}{m-1} \rfloor + 1} + 1$
10 $V \leftarrow V \cup \{\tilde{Z}_i\}$
11 $i \leftarrow i + 1$
12 $\tilde{Z}_i \leftarrow (\tilde{Z}_{i-1} - \tilde{x}_{i-1} p^{q-1}) p + \tilde{x}_{i+q}$
13 $S \leftarrow \emptyset$
14 $j \leftarrow 1$
15 **para** $i \leftarrow 0$ **até** $\lceil \frac{n}{m-1} \rceil + 1$ **faça**
16 **se** $B[i] \geq m + 1 - (r + 1)q$
então
17 $k \leftarrow (i + 1)(m - 1)$
18 **senão**
19 **se** $Y_{j..k} \neq \epsilon$
então
20 $S' \leftarrow$ ALGORITMO-SECUNDÁRIO$(X, Y_{j..k}, r)$
21 $S \leftarrow S \cup (S' \oplus j)$
22 $j \leftarrow i(m - 1) - m - k + 1$
23 **devolva** S

mente, em 1999, Burkhardt *et al.* [BCF+99] apresentaram uma ferramenta chamada QUASAR(*Q-gram Alignment based on Suffix ARrays*) fortemente baseada no algoritmo JOKINEN-UKKONEN e que foi extensivamente adotada no agrupamento de seqüências genéticas durante seqüênciamento do genoma do camundongo. Apesar de o QUASAR apresentar algumas modificações em relação ao algoritmo original, a idéia central é a mesma e os resultados experimentais apresentados permitem-nos analisar a eficiência da abordagem.

Em vez da TBP tradicional, os autores adotaram, para facilitar a realização dos experimentos com diversos valores de q, o uso misto de um vetor dos sufixos do texto e da tabela de busca de palavras. O vetor dos sufixos tinha o papel de permitir a construção rápida da tabela de busca de palavras a partir de uma tabela de busca de palavras pré-existente, construída para algum valor distinto de q, e servir como índice para as posições da TBP.

Também foram realizados testes com blocos de tamanho igual ou superior a $2(m-1)$, porém sempre mantendo a sobreposição de metade do comprimento de um bloco entre cada par de blocos consecutivos, para manter o caráter sem-perda do algoritmo. Como algoritmo secundário foi adotado o NCBI BLAST versão 2.0.3 [3] e os resultados obtidos foram comparados com os resultantes de execuções independentes da mesma versão do BLAST. Os testes foram realizados sobre as bases de dados do genoma humano (723675 seqüências e um total de 279,5 milhões de pares de bases) e genoma do camundongo (198323 seqüências e um total de 75,2 milhões de pares de bases) disponíveis no ano de 1999, data da realização dos experimentos.

O tempo gasto com a construção do vetor dos sufixos e do pré-processamento (construção da TBP) não foi computado como parte da execução do QUASAR (em média, 114 segundos para o genoma humano e 30 segundos para o genoma do camundongo), porém o tempo de pré-processamento do BLAST foi computado como parte da execução do mesmo. Segundo os autores, não foi possível deixar de fazê-lo por razões técnicas, porém os mesmos estimam que o tempo de pré-processamento do BLAST tenha sido significativamente inferior a 1% do tempo total. Os testes foram conduzidos

[3] O BLAST é um algoritmo heurístico que implementa uma etapa de filtragem, o que, em tese, poderia afetar os resultados dos testes. Não está claro o motivo da escolha do BLAST como algoritmo secundário, mas acreditamos que ela tenha se dado em função da aplicação. Talvez os autores estivessem apenas interessados em demonstrar a aplicabilidade do QUASAR quando adotado em conjunto com o BLAST ou outros algoritmos semelhantes.

Tabela 5.1 Comparação dos tempos de execução do QUASAR, adotando BLAST como algoritmo secundário, e do BLAST quando executado diretamente. Os valores representam a média de de 1000 execuções dos algoritmo.

Base de Dados (genoma)	Tamanho do Texto (Mpb)	Tamanho do Padrão (pb)	r_f	Tempo de CPU	
				QUASAR	BLAST
Camundongo	73,5	368	0,24	0,123	3,37
Humano	279,5	393	0,17	0,38	13,27

em uma máquina Sun Entreprise 10000 com um único processador SUN Ultra SPARCII de 333 Mhz e 4 GB de memória RAM. Como pode ser observado na tabela 5.1, os testes indicam um desempenho 34 vezes superior à apresentada pelo BLAST durante o processamento do genoma humano, e a 27 vezes a apresentada durante o processamento do genoma do camundongo.

5.4 O Algoritmo Burkhardt-Kärkkäinen

5.4.1 Q-gramas

Em 2001, Stefan Burkhardt e Juha Kärkkäinen [BK01] estenderam a definição de q-gramas que apresentamos na seção anterior e propuseram novas técnicas de filtragem em um trabalho que obteve enorme repercussão, culminando no despertar de uma linha de estudos que esteve inerte por anos, e que após esse artigo vislumbrou um período de atividade persistente até a data da escrita dessa dissertação de mestrado. Basicamente, em vez de levar em conta todos os fatores comuns de comprimento q, o trabalho de Burkhardt e Kärkkäinen considera as subseqüências de comprimento q comuns ao padrão e ao texto e que apresentam um determinado formato previamente definido. Inicialmente, os autores restringiram o algoritmo à distância de Hamming, modificando-o levemente em um trabalho posterior para tratar da distância de Levenshtein. A seguir, apresentaremos em detalhes o algoritmo Burkhardt-Kärkkäinen para a busca aproximada de padrões sobre a distância de Hamming e, ao final da seção, faremos algumas breve considerações sobre a versão modificada para a distância de Levenshtein, assim como sobre o trabalho de Kucherov, Noé e Roytberg [KNR04], que estende a idéia de Burkhardt-Kärkkäinen ao considerar a utilização de subseqüên-

cias que apresentam um dentre diversos formatos pré-definidos em vez de um único formato.

Definição 47. Um **formato** Q é um conjunto de inteiros que contém o elemento zero. A **cobertura** de Q é $\text{cob}(Q) = \max Q + 1$.

É através da definição prévia de um formato que é restringido o conjunto de subseqüências que o algoritmo irá adotar durante a etapa de filtragem. Burkhardt e Kärkkäinen concentraram o foco de seu trabalho em como, dado um formato, determinar o limiar ótimo de filtragem para processar instâncias do problema com um determinado comprimento do padrão e uma dada distância máxima permitida. Apesar de fazer diversas considerações a respeito, o trabalho não esclarece como deve-se determinar o formato que promoverá a maior redução possível na entrada. É apenas apresentado um algoritmo que, a cada iteração, modifica um conjunto de formatos existente, gerando um conjunto de formatos cuja eficiência de filtragem é superior a do conjunto anterior. Nenhuma demonstração ou consideração é fornecida sobre a correção ou exatidão do algoritmo.

A busca por formatos eficientes acabou por originar uma nova linha de pesquisa, voltada unicamente para o problema de se determinar o formato que irá promover a maior remoção possível: diversas formulações do problema podem ser encontradas na literatura. Algumas dessas formulações provaram ser NP-difíceis [NR05, NR07]. Curiosamente, nos últimos anos a literatura vislumbrou poucos avanços no sentido de se encontrar algoritmos eficientes para a determinação dos melhores formatos para a filtragem de instâncias da busca aproximada de padrões, ou sequer de formatos cuja eficiência possa ser garantida ou atestada de algum modo que não seja a simples realização de testes empíricos [4].

Definição 48. Sejam $X = x_1 \ldots x_m$ uma cadeia e Q um formato, tais que $\text{cob}(Q) \leq m$, e convencionemos $Q_i = Q \oplus i$. Dado um inteiro positivo i, a Q**-grama** de X na posição i é a cadeia $X[Q_i]$.

É claro que $X[Q_i]$ é uma subseqüência de X, de comprimento $|Q|$. Observe o exemplo a seguir:

[4] Note que aqui não usamos o termo eficiente para denotar um algoritmo cujo consumo de tempo é polinomial, mas apenas para denotar um algoritmo cujo consumo de tempo é tolerável dentro de uma aplicação ou problema específico.

`Exemplo:` a cadeia é `abcabcd`. Para $Q = \{0, 2\}$, suas Q-gramas são: `ac`, `ba`, `cb`, `ac` e `bd`.

Um formato também pode ser representado, de modo menos formal, porém de visualização mais fácil, como uma cadeia sobre o alfabeto binário $\{\#, -\}$: o caractere na posição i da cadeia que representa um formato Q é $\#$, se $i - 1$ é um elemento de Q, e $-$, caso contrário.

`Exemplo:` O formato $\{0, 1, 2, 4, 6, 9, 10, 11\}$ é representado pela cadeia $\#\#\# - \# - \# - -\#\#\#$.

Dadas duas cadeias $X = x_1 \dots x_m$ e $Y = y_1 \dots y_m$, considere o conjunto $M(X, Y)$ definido como segue:

$$M(X, Y) = \{i : x_i = y_i\}.$$

$M(X, Y)$ é o conjunto das posições nas quais os caracteres das cadeias são iguais. Para o restante da seção, tomemos um formato Q e seja $c = cob(Q)$. Vamos definir a função similaridade abaixo:

Definição 49. A **similaridade de Q-gramas**, entre duas cadeias $X = x_1 \dots x_m$ e $Y = y_1 \dots y_m$, é

$$s_Q(X, Y) = |\{i : X[Q_i] = Y[Q_i]\}|.$$

5.4.2 O Teorema das Q-gramas

Teorema 50 (Teorema das Q-gramas)**.** *Dados um formato Q, duas cadeias $X = x_1 \dots x_m$ e $Y = y_1 \dots y_n$ e inteiros j e r, tais que j seja a posição final de uma (d_H, r)-ocorrência de X em Y, temos que*

$$s_Q(X, Y_{j-m+1..j}) \geq \min_{M \subseteq \{1,..,m\}, |M| = m-r} |\{i : Q_i \subseteq M\}|.$$

Demonstração. A demonstração é trivial. Como X e $Y_{j-m+1..j}$ são (d_H, r)-vizinhas, temos

$$s_Q(X, Y_{j-m+1..j}) \geq \min\{s_Q(X, Z) : Z \in \mathsf{NH}_r(X)\}.$$

Trabalhando a inequação, temos que

$$\begin{aligned} s_Q(X, Y_{j-m+1..j}) &\geq \min|\{i\colon X[Q_i] = Z[Q_i] \text{ e } Z \in \mathsf{NH}_r(X)\}| \\ &= \min|\{i\colon Q_i \subseteq M(X,Z) \text{ e } Z \in \mathsf{NH}_r(X)\}| \\ &= \min_{M\subseteq\{1,\ldots,m\},|M|=m-r} |\{i\colon Q_i \subseteq M\}|. \end{aligned}$$

□

5.4.3 Determinação do Limiar de Filtragem

O teorema anterior nos fornece uma delimitação inferior para a similaridade de Q-gramas entre duas cadeias (d_H, r)-vizinhas. Note que o limitante independe das cadeias em si, mas somente de seu comprimento e do valor de r. Novamente, temos um teorema que nos dá base para o desenvolvimento de um filtro para a busca aproximada de padrões, através da determinação de uma condição que deve ser satisfeita por cadeias (d_H, r)-vizinhas entre si. Logo, posições do texto que não fazem parte de segmentos que satisfaçam essa condição podem ser descartadas com segurança. Infelizmente, diferente do que vimos antes, neste caso não temos uma condição baseada em um fórmula em forma fechada, mas sim recorrente. Porém, com certa dose de engenhosidade podemos determinar a satisfazibilidade da condição de modo relativamente eficiente; apenas relativamente eficiente, pois apesar de em termos práticos o tempo de execução do algoritmo ser tolerável, ao menos em se tratando da busca em textos estáticos, seu consumo de tempo não é polinomialmente delimitável no tamanho da entrada, como veremos. É importante notar que a etapa de pré-processamento não é concluída com o cálculo do limiar de filtragem. Lembrando que o foco dessa classe de algoritmos é a busca em textos estáticos, ainda nos resta construir a estrutura de índices que será adotada para processar rapidamente os padrões recebidos.

O Cálculo do Limiar Ótimo de Filtragem

Vamos chamar a delimitação inferior na desigualdade do teorema anterior de $l_Q(m, r)$. Isto é,

$$s_Q(X, Y_{j-m+1..j}) \geq l_Q(m, r) = \min_{M\subseteq\{1,..,m\},|M|=m-r} |\{i\colon Q_i \subseteq M\}|.$$

A recorrência a seguir vai nos permitir calcular $l_Q(m, r)$ através de programação dinâmica:

Definição 51. Seja Q um formato, m e k inteiros e $M \subseteq \{1, \ldots, \text{cob}(Q)-1\}$ um conjunto de inteiros tal que $|M| \geq \text{cob}(Q) - 1 - k$. Vamos definir a recorrência a seguir:

$$l_Q(m, r, M) = \min_{\substack{M' \subseteq \{1,\ldots,m\}, |M'|=m-r \\ (M' \ominus m-c+1) \cap \{1\ldots,c-1\}=M}} |\{i \in \{1, \ldots, m-c+1\} \colon Q_i \subseteq M'\}|.$$

A relação entre $l_Q(m, r)$ e $l_Q(m, r, M)$, é explicitada no que segue:

Lema 52. *Para todo para de inteiros m e r,*

$$l_Q(m, r) = \min\{l_Q(m, r, M) \colon M \subseteq \{1, \ldots, \textit{cob}(Q) - 1\} \textit{ e } |M| \geq \textit{cob}(Q) - 1 - r\}.$$

Vamos olhar com mais atenção para o problema de se calcular $l_Q(m, r, M)$. Seja $I_a = (m_a, r_a, M_a)$ uma instância do problema e seja $I_b = (m_a - 1, r_b, M_b)$ uma outra instância. Considere $T_a = \{1, \ldots, m_a - c + 1\}$ e T_b definido de modo análogo. Seja M'_a o conjunto que minimiza $|\{i \in T_a : Q_i \subseteq M'_a\}|$ e seja M'_b o conjunto que minimiza $|\{i \in T_b : Q_i \subseteq M'_b\}|$.

Se $c < m_a$, então $Q_i \nsubseteq T_a, \forall i \in T_a$ e, portanto, $l_Q(m_a, r_a, M_a) = 0$. Agora, vamos supor $c \geq m$. Devemos levar em conta duas possibilidades

- $c - 1 \in M$: Aqui, teremos $m \in M'$. Observe, que $l_Q(m_a, r_a, M_a)$ pode ser obtido a partir de $l_Q(m_a - 1, r_a - 1, M_b)$, onde $M_b \cap \{2, \ldots, c - 1\}$ equivale ao conjunto $M_a \backslash \{c - 1\}$ quando deslocado uma posição à direita. Note, também, que não podemos determinar, de antemão, se $1 \in M_b$ ou não; precisamos considerar ambas as possibilidades e optar pela ótima. Logo,

$$M_b = \begin{cases} (M_a \cup \{0\} \backslash \{c - 1\}) \oplus 1 \\ (M_a \backslash \{c - 1\}) \oplus 1 \end{cases}$$

- $c-1 \notin M$: Neste caso, $l_Q(m_a, r_a, M_a)$ pode ser obtido a partir de $l_Q(m_a-1, r_a, M_b)$, onde $M_b \cap \{2, \ldots, c-1\}$ equivale ao conjunto M_a deslocado uma posição à direita. Novamente, não podemos determinar, de antemão, se $1 \in M_b$ e duas possibilidades devem ser consideradas:

$$M_b = \begin{cases} (M_a \cup \{0\}) \oplus 1 \\ M_a \oplus 1 \end{cases}$$

Ainda, resta-nos fazer uma pequena observação: em ambas as possibilidades, se $1 \in M_b$ e $Q \subseteq (M_a \cup \{0\})$ então $Q_{m_a-c+1} \subseteq M'_a$ e $l_Q(m_a, r_a, M_a) = l_Q(m_a - 1, r_b, M_b) + 1$.

Então, obtemos a recorrência que segue:

$$l_Q(i,j,M) = \begin{cases} 0 & \text{se } i < c, \\ \min \left\{ \begin{array}{l} l_Q(i-1,j,(M \cup \{0\}) \oplus 1) + [Q \subseteq (M \cup \{0\})] \\ l_Q(i-1,j,M \oplus 1) \end{array} \right\} & \text{se } c-1 \notin M, \\ \min \left\{ \begin{array}{l} l_Q(i-1,j-1,(M \cup \{0\} \backslash \{c-1\}) \oplus 1) + [Q \subseteq (M \cup \{0\})] \\ l_Q(i-1,j-1,(M \backslash \{c-1\}) \oplus 1) \end{array} \right\} & \text{se } c-1 \in M. \end{cases}$$

O algoritmo a seguir determina $l_Q(m,r)$ com base na recorrência fornecida.

CALCULA-LIMIAR(Q, m, r)

1 **para** $i \leftarrow 1$ **até** $c-1$ **faça** $\quad O(n)$
2 $\quad D_i[0\,..\,r] \leftarrow 0$
3 **para** $i \leftarrow c$ **até** m **faça**
4 $\quad$ **para** $j \leftarrow 0$ **até** r **faça**
5 $\quad\quad$ **para todo** $M \subseteq \{1,\ldots,c-1\} : |M| \leq j$ **faça**
6 $\quad\quad\quad$ **se** $c-1 \in M$
7 $\quad\quad\quad\quad$ **então** $D_i[j][M] \leftarrow \min \left\{ \begin{array}{l} D_{i-1}[j][M \cup \{0\} \oplus 1], \\ D_{i-1}[j][M \oplus 1]. \end{array} \right.$ $\quad \triangleright\ O(n)$
8 $\quad\quad\quad\quad$ **senão** $D_i[j][M] \leftarrow \min \left\{ \begin{array}{l} D_{i-1}[j-1][M \cup \{0\} \oplus 1], \\ D_{i-1}[j-1][M \oplus 1]\}. \end{array} \right.$

Teorema 53. *$l_Q(m,r)$ pode ser calculado em tempo $O(mf(r,c))$, onde*

$$f(r,c) = \sum_{k=0}^{r} \binom{c-1}{k} (r-k+1).$$

Demonstração. O consumo de tempo do algoritmo é dominado pelo consumo das linhas 5 à 8, que, por sua vez, é delimitado superiormente por

$$\begin{aligned} (m-c+1) \sum_{j=0}^{r} \sum_{k=0}^{j} \binom{c-1}{k} &= (m-c+1) \sum_{k=0}^{r} \binom{c-1}{k} (m-k+1) \\ &\leq m \sum_{k=0}^{r} \binom{c-1}{k} (m-k+1) \\ &= O(mf(r,c)). \end{aligned}$$

□

Note que $O(mf(r,c)) = O(m(r+1)2^c)$. Logo, o problema é polinomialmente tratável para parâmetros fixos. Os autores apresentam uma delimitação mais justa para alguns casos particulares. Essa observação é muito importante, pois para as aplicações práticas que tratamos nesta dissertação de mestrado é razoável supor valores de c limitados superiormente por uma constante. De modo semelhante, o consumo de espaço do algoritmo é $O(f(r,c))$, onde

$$f(r,c) = \sum_{k=0}^{r} \left(\begin{array}{c} c-1 \\ k \end{array} \right) (r-k+1).$$

Aqui, é importante fazer uma pequena observação: o cálculo do limiar ótimo de filtragem é feito levando-se em conta um determinado formato. Logo, espera-se que o limiar não seja o mesmo para formatos distintos. Mas ainda, o limiar leva em conta um determinado comprimento de padrão. Portanto, para cada formato e para cada comprimento de padrão faz-se necessário determinar um novo limiar de filtragem.

5.4.4 Determinação do Formato Ideal

Além do algoritmo para a determinação do limiar de filtragem ideal para um dado formato, os autores esboçaram uma heurística para a determinação de um formato "bom", isto é, o algoritmo fornece um formato porém não oferece nenhuma garantia quanto à eficiência de filtragem do mesmo. O algoritmo baseia-se no seguinte lema:

Lema 54. *Para todo par de formatos Q' e Q e inteiros m e r, se $Q' \subseteq Q$ então $l'_Q(m,k) \geq l_Q(m,r)$.*

Com base nesse lema, o algoritmo constrói, a partir de um conjunto de formatos relevantes de comprimento $q-1$ e cobertura c, um conjunto de formatos relevantes de comprimento q e cobertura c. São relevantes os formatos que possuem limiar de filtragem positivo. A estratégia é simples: seja Q^* um conjunto de formatos relevantes de comprimento $q-1$ e cobertura c. Vamos particionar Q^* de forma que todo par de formatos que pertençam a uma mesma parte venham a diferir, entre si, apenas no penúltimo elemento; isto é, para cada dois formatos $Q' = i_1 \ldots i_{q-1}$ e $Q'' = j_1 \ldots j_{q-1}$

pertencentes a uma mesma parte, teremos $i_k = j_k, \forall k \neq q-2$. Encontrar tais pares é trivial; basta ordenar Q^* lexicograficamente. Agora, a partir dessa partição vamos obter, para cada par de formatos, um novo formato $Q = Q' \cup Q'' = \{i_1, i_2, .., i_{q-2}, j_{q-2}, i_{q-1}\}$. É óbvio que Q terá comprimento q, e, em virtude do lema, possuirá limiar de filtragem positivo. Não é dito, no artigo original, como deve ser gerado o conjunto de formatos inicial.

5.4.5 Construção das Estruturas de Índices

A estrutura de dados adotada pelo algoritmo BURKHARDT-KÄRKKÄINEN é extremamente semelhante à adotada pelo algoritmo JOKINEN-UKKONEN e pouco podemos acrescentar ao que já foi dito, exceto por uma pequena observação: no caso das Q-gramas, não é possível calcular a função hash apresentada de modo incremental; isto é, obter a codificação inteira, na base $|\Sigma|^q$, da cadeia $X[Q_i]$ a partir da codificação da cadeia $X[Q_{i-1}]$, como foi feito para as q-gramas. Em virtude desse fato, o tempo de construção da tabela de busca de palavras sobe para $O(nq + |\Sigma|^q)$ e o tempo consumido pela etapa de filtragem para $O(mq)$. O consumo de espaço permanece $O(n + |\Sigma|^q)$, e pode ser reduzido, como já comentado, adotando as técnicas propostas por Karp e Rabin [KR87].

5.5 Extensões do Algoritmo de Burkhardt e Kärkkäinen

Em 2005, Kucherov, Noé e Roytberg [KNR04] estenderam a idéia apresentada no algoritmo de Burkhardt e Kärkkäinen e publicaram um algoritmo de filtragem baseado no uso conjunto de dois ou mais formatos. Segundo os autores, essa abordagem já havia sido considerada anteriormente, porém com o foco em algoritmos de filtragem com perda. Assim como ocorre com os demais algoritmos que vimos, a seleção dos formatos e a determinação do critério de filtragem adotados representam etapas cruciais da execução do algoritmo, porém, assim como no trabalho de Burkhardt e Kärkkäinen e na literatura de modo geral, os avanços promovidos no sentido de determinar a melhor família, ou mesmo o melhor formato, são modestos.

Kucherov, Noé e Roytberg apresentam algumas idéias nesse sentido, entre as quais a construção de uma família de formatos a partir de um formato inicial, e tal que a

distribuição dos caracteres curingas, em cada um dos formatos gerados, possui regularidade. Porém, os autores não adotam essa abordagem nos testes experimentais cujos resultados são apresentados ao final do artigo; em vez disso, a família de formatos adotada nos testes é gerada através de uma abordagem mista, baseada em algoritmos genéticos e programação dinâmica. A escolha pode ter se dado em razão de uma possível ineficiência de filtragem dos formatos gerados: segundo Burkhardt e Kärkkäinen formatos regulares estão entre os piores formatos possíveis no que tange sua eficiência de filtragem.

É evidente que esse algoritmo requer uma etapa de processamento bastante custosa, visto que o limiar de filtragem é calculado não apenas para um único formato, mas para um conjunto deles, o que restringe ainda mais a sua viabilidade apenas à busca em textos estáticos.

Dadas duas cadeias $X = x_1 \dots x_n$ e $Y = y_1 \dots y_n$, chamamos de **semelhança** entre X e Y à cadeia $W = w_1 \dots w_n$ sobre o alfabeto binário $\{0, 1\}$ tal que $w_i = 1$ se, e somente se, $x_i = y_i$.

> `Exemplo:` As cadeias são "`cactcgt`" e "`cacactt`". A semelhança entre ambas é a cadeia "`1110101`".

Dado um formato Q, dizemos que Q **detecta** uma semelhança W, entre duas cadeias X e Y, se existe algum posição i de W tal que a cadeia $W[Q_i]$ contenha apenas o caractere 1.

> `Exemplo:` O formato $\#\# - \#$ detecta a semelhança "`1110101`" na posição 2.

Considere uma instância para a busca aproximada de padrões sob a distância de Hamming onde deseja-se encontrar as ocorrência de uma cadeia $X = x_1 \dots x_m$ em uma cadeia $Y = y_1 \dots y_m$ com distância no máximo r.

Seja $F = \langle Q_1, Q_2, \dots, Q_L \rangle$. Vamos chamar F de **família de formatos**, ou simplesmente família. Dizemos que um formato Q resolve uma dada instância da busca aproximada de padrões se cada possível similaridade de comprimento m e com r zeros é detectada por Q. De modo análogo, dizemos que uma família F resolve uma dada instância da busca aproximada de padrões se cada possível similaridade de comprimento

m e com r zeros é detectada por ao menos um formato Q em F. O limiar ótimo de uma família F para essa mesma instância do problema é o maior número $l_F(m,k)$ tal que há pelo menos $l_F(m,k)$ ocorrências de formatos de F em toda e qualquer similaridade de comprimento m e com r zeros. A recorrência para a determinação desse limiar ótimo é uma extensão da ocorrência apresentada na seção anterior para a determinação do limiar ótimo para o algoritmo de Burkhardt e Kärkkäinen e que apresentamos a seguir:

$$l_F(i,j,W[1\,..\,n]) = \begin{cases} l_F(c,j,W[1\,..\,n]) & \text{se } i < \text{c}, \\ l_F(i-1,j-1,1.W[1\,..\,n-1]) & \text{se } W[n]=0 \\ l_F(i,j,1.W[1\,..\,n-1]) + \sum_{k=1}^{|F|} \text{sufixo}(Q_k,W), & \text{se } n=c \\ \min\{l_F(i,j,W[1\,..\,n], l_F(i,j,W[1\,..\,n]\} & \text{se } |\{i:W[i]=0\}|<j \\ l_F(i,j,1.W[1\,..\,n]) & \text{se } |\{i:W[i]=0\}|=j \end{cases}$$

Um ano após apresentarem à comunidade o filtro baseado no uso de formatos com intervalos para a busca aproximada sob a distância de Hamming, Stefan Burkhardt e Juha Kärkkäinen publicaram uma variante desse algoritmo para atacar o problema sob a distância de Levenshtein. Nessa variante, os formatos adotados possuem apenas um intervalo de comprimento unitário: isto é, todo formato Q pode ser particionado em duas partes Q' e Q'' tais que $Q' = \{i : minQ' \leq i \leq maxQ'\}$ e $Q'' = \{i : minQ'' \leq i \leq maxQ''\}$ e $maxQ' = minQ'' - 1$. Quando essa restrição é apÍicada, a recorrência empregada para calcular o limiar de filtragem ótimo no caso mais geral também pode ser empregada para calcular o limiar ótimo desse tipo de formato quando aplicado ao problema sob a distância de Levenshtein. Apesar da modificação em relação aos formato convencional (contínuo) ser pequena, o formato com intervalo unitário representa um incremento significativo na capacidade de filtragem desse tipo de formato.

Capítulo 6

Conclusão

Nesta dissertação de mestrado estudamos algoritmos para a busca e extração de padrões em cadeias. Os algoritmos e estruturas de dados abordados são genéricos o suficiente para serem aplicados em uma grande gama de situações, porém procuramos dar uma maior ênfase a sua aplicação em problemas da biologia computacional. Em outros termos, em virtude do tamanho das entradas tratadas por esse tipo de aplicação — imenso, se considerarmos o poder computacional disponível nos dias de hoje — , procuramos abordar algoritmos e estruturas de dados cujo consumo de tempo e espaço cresça de forma tímida em relação ao comprimento das cadeias fornecidas, e não descartamos de nosso estudo algoritmos e estruturas cujo consumo de recursos aumenta consideravelmente conforme o crescimento do alfabeto, visto que no referente à biologia computacional o tamanho do alfabeto é constante e também pequeno. E também, seguindo uma tendência da área bastante natural, em razão do que já foi dito sobre o tamanho das entradas, é parte dessa dissertação de mestrado um estudo sobre algoritmos de filtragem.

Ainda há muito por ser estudado e diversos caminhos prometem novas descobertas. Além dos aqui cobertos, muitos dos tópicos que foram deixados de lado, nos quais podemos incluir os citados na introdução desse trabalho, são bastante promissores e têm sido alvo de frutífera investigação. No que dá conta o capítulo 2, o cacto dos sufixos é uma estrutura versátil, que absorve parte de funcionalidade da árvore dos sufixos e do vetor dos sufixos; certamente é merecedora de um estudo aprofundado. Uma análise detalhada dos diversos algoritmos para a construção do vetor dos sufixos

é um tópico que consideramos muito interessante e talvez um dos que mais lamentamos ter de excluir desse texto. Apesar do tema ter sido recentemente explorado por Puglisi [PST07], ainda há material para novos estudos.

Referente ao capítulo 3, a pesquisa centrada em algoritmos bit-paralelos para o processamento de cadeias e outros problemas encontra-se no momento em um estágio bastante ativo, como demonstra o surgimento de trabalhos recentes sobre o tema [BGMR06, BHMR07, GF08, HFN06]. A técnica tem sido adotada para resolver uma grande gama de problemas e no momento seria oportuno um estudo inteiramente dedicado ao assunto. Também seria bastante atraente uma análise probabilística detalhada do problema da busca aproximada de padrões; até o momento essa abordagem não tem fornecido bons resultados e muitas perguntas encontram-se em aberto [Nav01, KM97]. Um estudo original poderia vir a ser de grande valia para uma análise do consumo de tempo e eficiência de algoritmos de filtragem que reflita satisfatoriamente o desempenho de tais algoritmos na prática, o que dependeria da escolha de uma distribuição conveniente.

Como foi destacado no decorrer do texto, a determinação de formatos eficientes para a filtragem da busca aproximada de padrões cresceu e tornou-se um tópico de pesquisa independente e extremamente ativo: em uma breve busca na Internet fomos capazes de encontrar mais de 35 trabalhos sobre o tema publicados nos últimos 5 anos, dos quais 15 surgiram nos últimos 24 meses. Há bastante material para servir como base a uma eventual resenha sobre o tema[1]. Também há espaço para um estudo com forte inclinação prática sobre a aplicação das estruturas de dados e algoritmos citados em problemas relacionados a sistemas de busca na Internet.

[1]Na ocasião da escrita desse texto, Laurent Noé [Noé08] mantinha uma página na Internet listando os artigos publicados sobre o tema, na qual constavam 75 trabalhos publicados desde 2001.

Apêndice A

Demonstração do Filtro de Ukkonen

A seguir apresentamos uma demonstração detalhada da correção do algoritmo de filtragem de Ukkonen, baseada nos invariantes do algoritmo.

Demonstração. Para demonstrar a correção do algoritmo, diremos que uma posição i de Y é **boa** se $V[i] \leq 2qr$, **não-boa** se $V[i] > 2qr$ e **ruim** se for não boa e $j - i + 1 > m + r$, onde j e a menor posição boa tal que $i \leq j \leq n$.

Os seguintes invariantes valem no início de cada iteração do laço **para** da linha 4.

(I1) $|Y_{i...j}| \leq m + r$

(I2) Toda posição da cadeia $Y_{i...j-1}$ é não-boa

(I3) Y' é a subseqüência de $Y_{1...i-1}$ obtida após a remoção de toda posição ruim da cadeia.

[Prova de (I1)] No início da primeira iteração do laço temos $|Y_{i=1...j=m}| = m \leq m + r$. Agora, vamos supor que estamos em uma iteração do laço, diferente da última, para a qual o invariante é verdadeiro. Vamos demonstrar que ele permanecerá válido na iteração seguinte. Seja i' e j', respectivamente, os valores de i e j para a próxima iteração. Se $V[j] \leq 2qr$, teremos $|Y_{i'...j'}| = |Y_{j+1...j+1}| = 1$; as linhas 6 e 7 nos garantem isso. Por outro lado, se $V[j] > 2qr$ há duas possibilidades:

1. $|Y_{i...j}| < m + r$. Nesse caso, $Y_{i'...j'} = Y_{i...j+1}$. Logo, $|Y_{i'...j'}| = |Y_{i...j}| + 1 \leq m + r$;

2. $|Y_{i\ldots j}| = m + r$. Aqui, em virtude da linha 9, temos $Y_{i'\ldots j'} = Y_{i+1\ldots j+1}$. Logo, $|Y_{i'\ldots j'}| = m + r$.

O que conclui a nossa demonstração da validade do invariante.

[Prova de (I2)] O invariante vale no início da primeira iteração do laço; basta observar que $V[j] = \infty$ para todo $1 \leq j \leq m - 1$. Vamos considerar uma iteração, que não seja a última. Podemos supor que o invariante vale para a iteração atual. Agora, mostraremos que o mesmo permanece verdadeiro para a iteração seguinte. Seja i' e j', respectivamente, os valores de i e j na próxima iteração. Se $V[j] \leq 2qr$, temos $Y_{i'\ldots j'-1} = \epsilon$; as linhas 6 e 7 nos garantem isso. Agora, se $V[j] > 2qr$ então, independente da execução, ou não, da linha 9, teremos $Y_{i'\ldots j'-1} \subseteq Y_{i\ldots j}$. Como toda posição de $Y_{i\ldots j}$ é não-boa, segue imediatamente a veracidade do invariante.

[Prova de (I3)] O invariante vale, trivialmente, no início da primeira iteração do laço. Vamos supor, por um instante, uma iteração qualquer do algoritmo e assumir que o invariante vale nesse momento. Seja i' e j', respectivamente, os valores de i e j na próxima iteração. Se $V[j] \leq 2qr$ então, em virtude de (I1) não há posição ruim em $Y_{i\ldots j}$, e as linhas 6 e 7 garantem a validade do invariante na iteração seguinte. Agora, se $V[j] > 2qr$ então há duas possibilidades:

1. $|Y_{i\ldots j}| < m + r$. Nesse caso, $i' = i$ e Y' permanece inalterada. Logo, o invariante continua válido.

2. $|Y_{i\ldots j}| = m + r$. Nesse caso, como toda posição em $Y_{i\ldots j}$ e não boa (I2), podemos assumir que i é ruim. Portanto, é seguro afirmar que Y' é a subseqüência de $Y_{1\ldots i-1}.y_i = Y_{1\ldots i}$ obtida após a remoção de toda posição ruim da cadeia. Logo, mesmo após a execução da linha 9, o invariante continuará valendo para a próxima iteração.

A demonstração do algoritmo não está completa: ainda precisamos olhar para a última iteração, onde $j = n + 1$. Façamos isso: vamos supor que na penúltima iteração as linhas 6 e 7 tenham sido executadas. Nesse caso, é evidente que $Y_{1\ldots i-1=n} = Y$. Agora, caso as linhas não tenham sido executadas, basta ver que toda posição em $Y_{i\ldots n}$ é não-boa. Logo, não podem ser posições finais de uma (d, r)-ocorrência do padrão. Portanto, em ambas as possibilidades, todas as (d, r)-ocorrências do padrão estarão contidas na cadeia $Y_{1\ldots i-1}$. Observe, ainda, que posições ruins não podem fazer parte

de uma ocorrência. Em razão de (I3), sabemos que Y' é a subseqüência de $Y_{1...i-1}$ obtida após a remoção das posições ruins da cadeia. Em vista do exposto, não é difícil nos convencermos que o algoritmo de filtragem de Ukkonen é um filtro sem perda para a busca aproximada de padrões. □

Referências Bibliográficas

[AGM+90] Stephen Frank Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman, *Basic local alignment search tool*, Journal of Molecular Biology **215** (1990), número 3, 403–410.

[AK02] Ezekiel Femi Adebiyi and Michael Kaufmann, *Extracting common motifs under the Levenshtein measure: Theory and experimentation.*, Workshop on Algorithms in Bioinformatics, 2002, páginas 140–156.

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch, *Replacing suffix trees with enhanced suffix arrays*, Journal of Discrete Algorithms **2** (2004), número 1, 53–86.

[AOK02] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz, *Optimal exact strring matching based on suffix arrays*, Proceedings of the 11th International Conference on String Processing and Information Retrieval, SPIRE 2002, Springer-Verlag, 2002, páginas 31–43.

[AP03] Alberto Apostolico and Laxmi Parida, *Compression and the wheel of fortune.*, Proceedings of the Data Compression Conference, DCC 2003, 2003, páginas 143–152.

[Apo85] Alberto Apostolico, *The myriad virtues of subword trees*, Combinatorial Algorithms on Words (Alberto Apostolico and Zvi Galil, eds.), NATO ASI Series F: Computer and System Sciences, volume 12, Springer-Verlag, 1985, páginas 85–96.

[AS04] Julien Allali and Marie-France Sagot, *The at most k-deep factor tree*, Relatório Técnico 2004-03, Instituto Gaspard Monge, Universidade Marne la Vallé, 2004.

[BCF+99] Stefan Burkhardt, Andreas Crauser, Paolo Ferragina, Hans-Peter Lenhof, Eric Rivals, and Martin Vingron, *q-gram based database searching using a suffix array (QUASAR)*, ACM Press, 1999.

[BGMR06] Jérémy Barbay, Alexander Golynski, J. Ian Munro, and S. Srinivasa Rao, *Adaptive searching in succinctly encoded binary relations and tree-structured documents*, Proceedings of the 17th Annual Symposium on Combinatorial Pattern Matching, CPM 2006, 2006, páginas 24–35.

[BHMR07] Jérémy Barbay, Meng He, J. Ian Munro, and S. Srinivasa Rao, *Succinct indexes for strings, binary relations and multi-labeled trees*, Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2007, 2007, páginas 680–689.

[BK01] Stefan Burkhardt and Juha Kärkkäinen, *Better filtering with gapped q–grams*, Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching, CPM 2001, Lecture Notes in Computer Science, volume 2089, Springer-Verlag, 2001, páginas 73–85.

[BK03a] ———, *Better filtering with gapped q–grams*, Fundamenta Informaticae **56** (2003), número 1,2, 51–70.

[BK03b] ———, *Fast lightweight suffix array construction and checking*, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, CPM 2003, Lecture Notes in Computer Science, volume 2676, Springer-Verlag, 2003, páginas 55–69.

[BM93] Jon Louis Bentley and Malcolm Douglas McIlroy, *Engineering a sort function*, Software Practice and Experience **23** (1993), número 11, 1249–1265.

[BT02] Jeremy Buhler and Martin Tompa, *Finding motifs using random projections.*, Journal of Computer and System Sciences **9** (2002), número 2, 225–242.

[Bur02] Stefan Burkhardt, *Filter algorithms for approximate string matching*, Tese de Doutorado, Universidade de Saarlandes, 2002.

[BW94] Michael Burrows and David John Wheeler, *A block-sorting lossless data compression algorithm.*, Relatório Técnico 124, Digital Equipment Corporation, 1994.

[BY89] Ricardo Baeza-Yates, *Efficient text searching*, Tese de Doutorado, Universidade de Waterloo, 1989.

[BYG89] Ricardo A. Baeza-Yates and Gaston H. Gonnet, *A new approach to text searching*, Proceedings of the 12th International Conference on Research and Development in Information Retrieval, ACM Press, 1989, páginas 168–175.

[BYRN99] Ricardo A. Baeza-Yates and Berthier A. Ribeiro-Neto, *Modern information retrieval*, ACM Press, 1999.

[Car04] Alexandra M. Carvalho, *Efficient algorithms for structured motif extraction in DNA sequences*, Dissertação de Mestrado, Universidade Técnica de Lisboa, 2004.

[CCI+02] Emilios Cambouropoulos, Maxime Crochemore, Costas S. Iliopoulos, Laurent Mouchard, and Yoan José Pinzón Ardila, *Algorithms for computing approximate repetitions in musical sequences.*, International Journal of Computer Mathematics **79** (2002), número 11, 1135–1148.

[CIR98] Tim Crawford, Costas S. Iliopoulos, and R. Raman, *String matching techniques for musical similarity and melodic recognition*, Computing in Musicology **11** (1998), 73–100.

[CKW85] Martin Campbell-Kelly and Michael Roy Williams, *The Moore School Lectures: Theory and Techniques for Design of Electronic Digital Computers*, MIT Press, 1985.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald Rinn Rivest, and Clifford Stein, *Introduction to algorithms*, segunda edição, MIT Press, 2001.

[CM94] William I. Chang and Thomas G. Marr, *Approximate string matching and local similarity*, Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching, CPM 1994, Lecture Notes in Computer Science, volume 807, Springer-Verlag, 1994, páginas 259–273.

[CMS05] Young H. Cho and William H. Mangione-Smith, *A pattern matching coprocessor for network security*, Proceedings of the 42th Annual Conference on Design Automation, DAC 2005, Association for Computing Machinery, 2005, páginas 234–239.

[CR94] Maxime Crochemore and Wojciech Rytter, *Text algorithms*, Oxford University Press, 1994.

[CS01] Maxime Crochemore and Marie-France Sagot, *Motifs in sequences: localization and extraction*, Handbook of Computational Chemistry, Marcel Dekker, Inc., 2001.

[Dam64] Fred J. Damerau, *A technique for computer detection and correction of spelling errors*, Communications of the ACM **7** (1964), número 3, 171–176.

[Daw85] Clinton Richard Dawkins, *The blind watchmaker*, Norton, 1985.

[dLS03] Alair Pereira do Lago and Imre Simon, *Tópicos em algoritmos sobre sequências*, Colóquio Brasileiro de Matemática, 2003.

[Far97] Martin Farach, *Optimal suffix tree construction with large alphabets*, Proceedings of the 38th Annual Symposium on Foundations of Computer Science, FOCS 1997, IEEE Computer Society Press, 1997, páginas 137.

[FGN06] Michael R. Fellows, Jens Gramm, and Rolf Niedermeier, *On the parameterized intractability of motif search problems*, Combinatorica **26** (2006), número 2, 141–167.

[FMN06] Kimmo Fredriksson, Veli Mäkinen, and Gonzalo Navarro, *Flexible music retrieval in sublinear time*, International Journal of Foundations of Computer Science **17** (2006), número 6, 1345–1364.

[FN04] Kimmo Fredriksson and Gonzalo Navarro, *Average-optimal single and multiple approximate string matching*, ACM Journal of Experimental Algorithms **9** (2004), 1–4.

[Fon03] Paulo G. S. Fonseca, *Índices completos para casamento de padrões e inferência de motifs*, Dissertação de Mestrado, Universidade Federal de Pernambuco, 2003.

[Fre60] Edward Fredkin, *Trie memory*, Communications of the ACM **3** (1960), número 9, 490–499.

[GF08] Szymon Grabowski and Kimmo Fredriksson, *Bit-parallel string matching under hamming distance in $o(n\lceil m/w \rceil)$ worst case time*, Information Processing Letters **105** (2008), número 5, 182–187.

[GK97] Robert Giegerich and Setefan Kurtz, *From Ukkonen to McCreight and Weiner: A unifying view of linear-time suffix tree construction*, Algorithmica **19** (1997), 331–353.

[Gre05] T. Ryan Gregory, *Animal genome size database*, 2005, http://www.genomesize.com.

[Gus97] Dan Gusfield, *Algorithms on strings, trees, and sequences: computer science and computational biology*, Cambridge University Press, 1997.

[Hag98] Torben Hagerup, *Sorting and searching on the word RAM*, Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1998, Springer-Verlag, 1998, páginas 366–398.

[HFN04] Heikki Hyyrö, Kimmo Fredriksson, and Gonzalo Navarro, *Increased bit-parallelism for approximate string matching*, Proceedings of the 3rd Workshop on Efficient and Experimantal Algorithms, WEA 2004, Lecture Notes in Computer Science, volume 3059, Springer-Verlag, 2004, páginas 285–298.

[HFN06] H. Hyyrö, K. Fredriksson, and G. Navarro, *Increased bit-parallelism for approximate and multiple string matching*, ACM Journal of Experimental Algorithms **10** (2006), número 2.6, 1–27.

[Hyy03] Heikki Hyyro, *Practical methods for approximate string matching*, Tese de Doutorado, Universidade de Tampere, Dezembro 2003.

[IKMV00] Costa S. Iliopoulos, M. Kumar, L. Mouchard, and S. Venkatesh, *Motif evolution in polyphonic musical sequences*, Proceedings of the 11th Australasian Workshop on Combinatorial Algorithms, AWOCA 2000, 2000, páginas 53–66.

[IMP+05] Costas S. Iliopoulos, James A. M. McHugh, Pierre Peterlongo, Nadia Pisanti, Wojciech Rytter, and Marie-France Sagot, *A first approach to finding common motifs with gaps.*, International Journal of Foundations of Computer Science **16** (2005), número 6, 1145–1154.

[JU91] Petteri Jokinen and Esko Ukkonen, *Two algorithms for approximate string matching in static texts*, Proceedings of the 19th Symposium on Mathematical Foundations of Computer Science, MFCS 1991 **91** (1991), número 16, 240–248.

[KA03] Pang Ko and Srinivas Aluru, *Space efficient linear time constructions of suffix arrays*, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, CPM 2003, Lecture Notes in Computer Science, volume 2676, Springer, 2003, páginas 200–210.

[Kar93] Ricardo Ueda Karpischeck, *O autômato dos sufixos*, Dissertação de Mestrado, Universidade de São Paulo, 1993.

[KM97] Stefan Kurtz and Eugene W. Myers, *Estimating the probability of approximate matches*, Proceedings of the 8th Annual Symposium on Combinatorial Pattern Matching, CPM 1997, Lecture Notes in Computer Science, Springer-Verlag, 1997, páginas 52–64.

[KMP77] Donald Ervin Knuth, J. Morris, and Vaughan Ronald Pratt, *Fast pattern matching in strings*, SIAM Journal of Commputing **6** (1977), número 2, 127–146.

[KMR72] Richard Manning Karp, Raymond E. Miller, and Arnold L. Rosenberg, *Rapid identification of repeated patterns in strings, trees and arrays*, Proceedings of the 4th annual ACM Symposium on Theory of Computing, STOC 1972, ACM Press, 1972, páginas 125–136.

[KNR04] Gregory Kucherov, Laurent Noé, and Mikhail Roytberg, *Multi-seed lossless filtration - extended abstract*, Proceedings of the 15th Annual Symposium on Combinatorial Pattern Matching, CPM 2004, Lecture Notes in Computer Science, volume 3109, Springer, 2004, páginas 297–310.

[Knu98] Donald Ervin Knuth, *The art of computer programming, volume 3: sorting and searching*, segunda edição, Addison-Wesley, 1998.

[KP02] Uri Keich and Pavel A. Pevzner, *Finding motifs in the twilight zone*, Bioinformatics **18** (2002), número 10, 1374–1381.

[KR87] Richard Manning Karp and Michael Oser Rabin, *Efficient randomized pattern-matching algorithms*, IBM Journal of Research and Development **31** (1987), número 2, 249–260.

[KS03] Juha Kärkkäinen and P. Sanders, *Simple linear work suffix array construction*, Proc. 13th International Conference on Automata, Languages and Programming, Springer, 2003.

[KSPP03] D. K. Kim, J. S. Sim, Hyungju Park, and Kunsoo Park, *Linear time constructions of suffix arrays*, Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching, CPM 2003, Lecture Notes in Computer Science, volume 2676, Springer, 2003, páginas 200–210.

[Kur99] Stefan Kurtz, *Reducing the space requirement of suffix trees*, Software Practice and Experience **29** (1999), número 13, 1149–1171.

[LLM+99] K. Lanctot, M. Li, B. Ma, S. Wang, and L. Zhang, *Distinguish string search problems*, Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1999 (1999), 633–642.

[LMW99] M. Li, B. Ma, and L. Wang, *Finding similar regions in many strings*, Proceedings of the 31th Annual ACM Aymposium on Theory of computing, STOC 1999 (1999), 473–482.

[LMW02] Ming Li, Bin Ma, and Lusheng Wang, *On the closest string and substring problems*, Journal of the ACM **49** (2002), número 2, 157–171.

[LS99] N. Jesper Larsson and Kunihiko Sadakane, *Faster suffix sorting*, Relatório Técnico 99-214, Departamento de Ciência da Computação, Universidade Lund, 1999.

[Mau46] John William Mauchly, *Sorting and collating*, Theory and Techniques for the Design of Electronic Digital Computers (George William Patterson, ed.), volume 3, Universidade da Pensilvânia, 1946, Re-impressão: [CKW85], páginas 22.8–22.9.

[McC76] Edward M. McCreight, *A space-economical suffix tree construction algorithm*, Journal of the ACM **23** (1976), número 2, 262–272.

[MM93] Udi Manber and Eugene W. Myers, *Suffix arrays: a new method for on-line string searches*, Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 1990, Society for Industrial and Applied Mathematics, 1993, páginas 319–327.

[MNU05] Veli Mäkinen, Gonzalo Navarro, and Esko Ukkonen, *Transposition invariant string matching*, Journal of Algorithms **56** (2005), número 2, 124–153.

[Mor68] Donald R. Morrison, *PATRICIA practical algorithm to retrieve information coded in alphanumeric*, Journal of the ACM **15** (1968), número 4, 514–534.

[MP80] William J. Masek and Michael Stewart Paterson, *A faster algorithm for computing string edit distances*, Journal of Computer and System Sciences **20** (1980), 18–31.

[MP07] Michael A. Maniscalco and Simon J. Puglisi, *An efficient, versatile approach to suffix sorting*, ACM Journal of Experimental Algorithms **12** (2007), 1.2.

[MS00a] Laurent Marsan and Marie-France Sagot, *Algorithms for extracting structured motifs using a suffix tree with application to promoter and regulatory site consensus identification*, Journal of Computational Biology **7** (2000), número 3-4, 345–362.

[MS00b] ———, *Extracting structured motifs using a suffix tree. Algorithms and application to consensus identification*, Proceedings of the 4th Annual International Conference on Computational Molecular Biology, RECOMB 2000, 2000, páginas 210–219.

[MW94] Idi Manber and Sun Wu, *An algorithm for approximate membership checking with application to password security*, Information Processing Letters **50** (1994), número 4, 191–197.

[Mye99] Eugene W. Myers, *A fast bit-vector algorithm for approximate string matching based on dynamic programming*, Journal of the ACM **46** (1999), número 3, 395–415.

[Nav98] Gonzalo Navarro, *Approximate text searching*, Tese de Doutorado, Universidade do Chile, Dezembro 1998.

[Nav01] ———, *A guided tour to approximate string matching*, ACM Computing Survey **33** (2001), número 1, 31–88.

[Noé08] Laurent Noé, *Spaced seeds bibliography*, 2008, http://www2.lifl.fr/~noe/seeds.html.

[NR05] François Nicolas and Eric Rivals, *Hardness of optimal spaced seed design*, Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching, CPM 2005, Lecture Notes in Computer Science, volume 3537, Springer, 2005, páginas 144–155.

[NR07] ———, *Hardness of optimal spaced seed design*, Journal of Computer and System Sciences **11** (2007), número 5, 831–849.

[PCGS03a] Nadia Pisanti, Maxime Crochemore, Roberto Grossi, and Marie-France Sagot, *Bases of motifs for generating repeated patterns with don't cares*, Relatório Técnico TR-03-02, Departamento de Informática, Universidade de Pisa, Janeiro 2003.

[PCGS03b] ———, *A basis of tiling motifs for generating repeated patterns and its complexity for higher quorum.*, Proceedings of the 28th Symposium on Mathematical Foundations of Computer Science, MFCS 2003, 2003, páginas 622–631.

[Pet06] Pierre Peterlongo, *DNA sequence filtration for finding long multiple approximate repetitions*, Tese de Doutorado, Universidade Marne-la-Vallé, 2006.

[Pev00] Pavel A. Pevzner, *Computational molecular biology*, Bradford Book, 2000.

[Pis02] Nadia Pisanti, *Segment-based distances and similarities in genomic sequences*, Tese de Doutorado, Universidade de Pisa, Abril 2002.

[PL88] William R. Pearson and David J. Lipman, *Improved tools for biological sequence comparison*, Procedeedings of the National Academy of Sciences of the USA **85** (1988), 2444–8.

[PPBS05] Pierre Peterlongo, Nadia Pisanti, Frédéric Boyer, and Marie-France Sagot, *Lossless filter for finding long multiple approximate repetitions using a new data structure, the bi-factor array.*, Proceedings of the 12th International Conference on String Processing and Information Retrieval, SPIRE 2005, 2005, páginas 179–190.

[PS00] Pavel A. Pevzner and Sing-Hoi Sze, *Combinatorial approaches to finding subtle signals in DNA sequences.*, Proceedings of the 8th Annual International Conference on Intelligent Systems for Computational Biology, ISMB 2000, 2000, páginas 269–278.

[PST05] Simon J. Puglisi, William F. Smyth, and Andrew Turpin, *The performance of linear time suffix sorting algorithms*, Proceedings of the Data Compression Conference, DCC 2005, IEEE Computer Society Press, 2005, páginas 358–367.

[PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin, *A taxonomy of suffix array construction algorithms*, ACM Computing Survey **39** (2007), número 2, 4.

[RBH05] Sanguthevar Rajasekaran, Sudha Balla, and Chun-Hsi Huang, *Exact algorithms for planted motif challenge problems.*, Proceedings of the 3rd Asian-Pacific Bioinformatics Conference, APBC 2005, 2005, páginas 249–259.

[RKI06] Ramaswamy Ramaswamy, Lukas Kencl, and Gianluca Iannaccone, *Approximate fingerprinting to accelerate pattern matching*, Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement, ICM 2006, Association for Computing Machinery, 2006, páginas 301–306.

[Sag98] Marie-France Sagot, *Spelling approximate repeated or common motifs using a suffix tree*, Lecture Notes in Computer Science **1380** (1998), 374.

[SBM03] Simona Santini, Jeffrey L. Boore, and Axel Meyer, *Evolutionary conservation of regulatory elements in vertebrate hox gene clusters*, Genome Research **13** (2003), número 6, 1111–1122.

[Sel74] Peter H. Sellers, *On the theory and computation of evolutionary distances*, SIAM Journal of Applied Mathematics **26** (1974), 787–793.

[Sew00] Julian Seward, *On the performance of BWT sorting algorithms*, Proceedings of the Data Compression Conference, DCC 2000, IEEE Computer Society Press, 2000, páginas 173.

[Sew07] ———, *bzip2: A program and library for data compression*, 2007, http://www.bzip.org.

[SK83] David Sankoff and Joseph Bernard Kruskal, *Time warps, string edits, and macromolecules: The theory and practice of sequence comparison*, Addison-Wesley, 1983.

[SM98] Marie-France Sagot and Eugene W. Myers, *Identifying satellites in nucleic acid sequences*, Proceedings of the 2nd Annual International Conference on Computational Molecular Biology, RECOMB 1998, ACM Press, 1998, páginas 234–242.

[SNH04] Berend Snel, Vera Van Noort, and Martijn A. Huynen, *Gene co-regulation is highly conserved in the evolution of eukaryotes and prokaryotes*, Nucleic Acids Ressearch **32** (2004), número 16, 4725–4731.

[SW03] Marie-France Sagot and Yoshiko Wakabayashi, *Pattern inference under many guises*, Recent Advances in Algorithms and Combinatorics, Springer-Verlag, 2003, páginas 245–287.

[Ukk92] Esko Ukkonen, *Approximate string-matching with q-grams and maximal matches*, Theoretical Computer Science **92** (1992), número 1, 191–211.

[Ukk93] ———, *Approximate string matching over suffix trees*, Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching, CPM 1993, Lecture Notes in Computer Science, no. 684, Springer-Verlag, 1993, páginas 228–242.

[Ukk95] ———, *On-line construction of suffix trees*, Algorithmica **14** (1995), número 3, 249–260.

[vEBKZ76] Peter van Emde Boas, R. Kaas, and E. Zijlstra, *Design and implementation of an efficient priority queue*, Theory of Computing Systems **10** (1976), número 1, 99–127.

[Wei73] Peter Weiner, *Linear pattern matching algorithms*, Proceedings of the 14th Annual Symposium on Switching and Automata Theory, 1973, páginas 1–11.

[Wri94] Alden H. Wright, *Approximate string matching using within-word parallelism*, Software Practice and Experience **24** (1994), número 4, 337–362.

[ZL77] Jacob Ziv and Abraham Lempel, *A universal algorithm for sequential data compression*, IEEE Transactions on Information Theory **23** (1977), número 3, 337–343.

Lista de Figuras

2.1 Árvore dos sufixos da cadeia `abracadabra` 19

3.1 Alguns dos mais populares algoritmos para a busca aproximada de padrões. O consumo de tempo do algoritmo de Ukkonen refere-se ao consumo de tempo de pior caso e de caso médio. Ademais, leia-se: PD: Programação Dinâmica; BP: Bit Paralelismo; H: Heurística; 4R: Método dos Quatro Russos; NFA: Autômato não-determinístico; SA: Autômato dos Sufixos; ST: Árvore dos Sufixos; w: Comprimento da palavra de máquina. 35

3.2 Autômato-Não-Determinístico (AND) que verifica ocorrências (exatas) do padrão `agtacag` na cadeia varrida pelo autômato. Para maior clareza, foram omitidas arestas de retorno ao estado inicial. 37

3.3 Uma célula $[i, j]$ da matriz de programação dinâmica C, recebe valores de células adjacentes e ajuda a compor valores de outras células (a). Essa mesma célula, pode ser vista como um processador, que recebe como entrada os valores $\Delta_v[i, j-1]$, $\Delta_h[i-1, j]$ e $\alpha'(p_i, x_j)$ e devolve como saída os valores $\Delta_v[i, j]$ e $\Delta_h[i, j]$. Para facilitar a visualização é conveniente remover os índices, explicitando ainda mais essa analogia(c). Os valores de entrada e saída são codificados através de vetores binários, permitindo assim o processamento bit-paralelo da matriz (d). A mesma representação após o retorno dos índices (e). 44

3.4 Transdutor que define o resultado da soma de dois inteiros codificados em binário. O rótulo $(a, b)/c$ de cada transição representa um par (a, b) de dígitos dos operandos de entrada, e a saída c fornecida pelo transdutor. O transdutor possui apenas dois estados: o estado "com carry", que indica que uma seqüência de propagação está em andamento, e o estado inicial "sem carry", que indica o oposto. 51

4.1 A figura mostra as cadeias `abraca` e `dabraz`: ambas são motifs comuns no conjunto de cadeias apresentado, para dados $l = 6$, $q = 3$ e $r = 3$. . . . 56

4.2 Árvore dos sufixos generalizada do conjunto de cadeias $\{$`abraca`, `adabra`$\}$ 61

4.3 A figura mostra um motif estruturado composto por dois blocos e comum ao conjunto de cadeias dado, considerando-se os valores $q = 4$, $e_{min} = 3$ e $e_{max} = 6$. 67