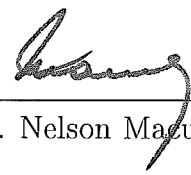
DETECÇÃO DE PALÍNDROMOS APROXIMADOS EM CADEIAS

Alexandre Henrique Lopes Porto

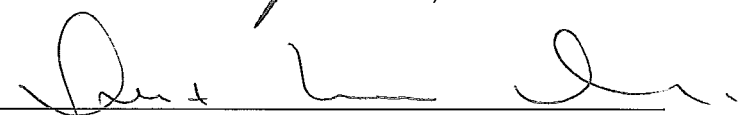
TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.



Prof. Nelson Maculan Filho, D.Sc.



Prof. Luiz Satoru Ochi, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 1999

LOPES PORTO, ALEXANDRE HENRIQUE

Detecção de Palíndromos Aproximados em Cadeias [Rio de Janeiro] 1999

XII, 179 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1999)

Tese – Universidade Federal do Rio de Janeiro, COPPE

1. Detecção de Palíndromos Aproximados
2. Edição de Cadeias
3. Problema de k-diferenças

I. COPPE/UFRJ II. Título (série)

Aos Meus Pais e Irmãs

Agradecimentos

Todo trabalho realizado é um produto de um elevado nível de dedicação, seriedade e estudo. E em cada passo do caminho percorrido, existem aquelas pessoas que através do seu apoio, auxiliaram a tornar possível a realização deste projeto. Pessoas que, ao decorrer da minha vida estiveram sempre ao meu lado, ajudando a me desvencilhar de situações adversas e a seguir sempre em frente em busca de um futuro próximo e também outras pessoas, que ao longo dos meses que decorreram, fizeram parte do que o transformou em um objetivo alcançado. Desejo então dividir o mérito desta realização com cada uma destas pessoas.

Inicialmente, agradeço aos meus pais, por terem me concebido o dom da vida, e às minhas irmãs, Flávia e Cláudia, por dividirem comigo os momentos mais preciosos da minha infância e tantos outros presentes à maioridade. Agradeço também aos meus pais pela minha formação ética e moral, base do que me levou a galgar cada degrau de minha vida acadêmica.

É impossível expressar, em meras e simples palavras, a enorme gratidão ao meu orientador, Valmir Barbosa, que ao longo desta trajetória demonstrou ser ainda mais do que o brilhante, competente e inspirador profissional que é. Que demonstrou ser um amigo inestimável e incentivador, apoiando-me em cada momento crucial.

Agradeço imensamente também, a cada um dos meus amigos da Coppe- Josina, Robson, Gabriel, Eduardo, Rodrigo e Lauro- pelas intermináveis noites de estudo e pelo apoio a cada dia.

Agradeço por fim, a todos aqueles que direta ou indiretamente, também participaram da realização da minha tese, cujos os nomes e rostos jamais serão esquecidos.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

DETECÇÃO DE PALÍNDROMOS APROXIMADOS EM CADEIAS

Alexandre Henrique Lopes Porto

Setembro/1999

Orientador : Valmir Carneiro Barbosa

Programa: Engenharia de Sistemas e Computação

Nesta tese apresentamos um algoritmo para computar todos os palíndromos aproximados em uma cadeia S de tamanho N. Nossa definição de um palíndromo aproximado é baseada no problema de k-diferenças, que é o caso particular do problema de edição de cadeias que limita o número de erros a no máximo k. Adaptamos um algoritmo para o casamento de cadeias aproximado para resolver o problema de k-diferenças, que nos permitiu achar um palíndromo num tempo de $O(k^2)$. Desde que temos $O(N)$ palíndromos na cadeia S, um tempo de $O(k^2N)$ foi suficiente para acharmos todos os palíndromos. Também melhoramos o algoritmo resultante através de duas otimizações que conduziram a melhores tempos na prática. Uma grande quantidade de resultados experimentais foram apresentados.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

DETECTING APPROXIMATE PALINDROMES IN STRINGS

Alexandre Henrique Lopes Porto

September/1999

Advisor: Valmir Carneiro Barbosa

Department: Computing Systems Engineering

In this tesis we give an algorithm for finding all approximate palindromes in a string S of length N. Our definition of an approximate palindrome is based on the problem of k-differences, which is the particular case of the string editing problem that limits the number of erros at no more than k. We have adapted an algorithm for approximate string matching to solve the k-differences problem, which has allowed us to find a palindrome in $O(k^2)$ time. Because we have $O(N)$ palindromes in string S, a time of $O(k^2N)$ suffices for all palindromes to be found. We have also improved the resulting algorithm by two optimizations that lead to better times in practice. Extensive experimental results are presented.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

Nesta tese apresentamos um algoritmo eficiente para calcular os palíndromos aproximados centrados em todas as possíveis posições de uma dada cadeia. A solução pode ser aplicada em vários casos, principalmente quando estamos analisando cadeias de DNA em biologia molecular. Somente estudamos e definimos algoritmos seqüenciais para o problema. Entretanto, descrevemos um mecanismo simples de paralelização do nosso algoritmo.

Para construir o algoritmo proposto e resolver o problema, avaliamos os algoritmos existentes para a descoberta dos palíndromos exatos em uma cadeia, definimos a noção de palíndromo aproximado, e concluímos que a melhor solução para o nosso problema seria adaptar o problema de edição de cadeias ao nosso caso. Foram analisados todos os principais algoritmos para este problema, e foi escolhida uma adaptação de um algoritmo para um problema particular de edição de cadeias (o problema de casamento de cadeias aproximado) descrito em [17, 26]. Desta adaptação, foi construído um algoritmo que foi posteriormente otimizado, obtendo um tempo de execução real melhor do que o do algoritmo original.

Como não parece existir um estudo que descreva o tema do reconhecimento de palíndromos aproximados (somente uma descrição simples em [17]), foi definido o que é um palíndromo aproximado, qual é o melhor palíndromo para uma dada posição da cadeia, e foi descrito como encontrar este melhor palíndromo.

Para a análise de tempo de pior caso, avaliamos todos os algoritmos adaptados ao nosso problema com exemplos que gerassem o maior tempo de execução possível. Além disso, também executamos exemplos baseados em várias cadeias com tamanhos variáveis, com diferentes números de erros máximos permitidos, para que pudéssemos analisar como o tamanho da cadeia, o número máximo de erros, e

o conteúdo de cada cadeia em particular podem variar o tempo de execução em relação ao tempo de pior caso. Ao analisarmos o tempo, foi contabilizado o número de passos executados, pois estavamos interessados em comparar o tempo de execução de uma cadeia em particular com o tempo de pior caso. Para medirmos a eficiência das otimizações propostas pelos dois algoritmos que serão descritos no capítulo 5, usamos o tempo de execução de pior caso de um algoritmo adaptado diretamente do algoritmo para o problema de k-diferenças também descrito neste capítulo.

Para a análise de complexidade de tempo médio, definimos dois modelos probabilísticos, que serão descritos no capítulo 6, baseados no tamanho da cadeia, no número máximo de erros, e no tamanho do alfabeto. Usamos estes modelos para obter o número médio de passos, e o tempo de execução, tanto para um palíndromo centrado em uma dada posição da cadeia, como para todos os palíndromos presentes em uma cadeia. Aplicamos estes modelos em vários exemplos com diferentes valores para o tamanho da cadeia, para o tamanho do alfabeto, e para o número de erros. Depois comparamos estes resultados probabilísticos com os exemplos executados sobre os algoritmos reais para estudarmos as diferenças obtidas pelo modelo probabilístico e por uma execução real. Os exemplos da execução real têm os mesmos parâmetros usados nos exemplos aplicados aos modelos probabilísticos, mas como o modelo probalilístico não depende do conteúdo particular de uma cadeia, as cadeias destes exemplos são seqüências aleatórias de caracteres do alfabeto dado.

1.1 Palíndromos

Um *palíndromo* é uma cadeia simétrica que é lida igualmente pela esquerda ou pela direita. Outra descrição mais exata é dada pela enciclopédia Larousse Cultural [29]: "Diz-se de palavras, números ou frases que podem ser lidos indiferentemente da esquerda para a direita e vice-versa, sempre com o mesmo sentido". Por exemplo, a cadeia xyaayx e a palavra "osso" são palíndromos, a frase "orava o avaro" será um palíndromo se os espaços forem ignorados, e o número 63736 também é um palíndromo.

Em biologia molecular, a definição descrita acima é mais conhecida como *repetição espelhada*, e a mais comumente usada é a de *palíndromo complementar* [17], que é uma cadeia de DNA ou RNA que se tornará um palíndromo se cada caractere de uma das metades da cadeia for trocado pelo seu caractere complementar (a

base complementar). Por exemplo, a cadeia de DNA $AGCTCGCGAGCT$ é um palíndromo complementar segundo esta definição.

$$5'\ TCGACCGCTCGA\ 3'$$
$$3'\ AGCTCGGCGAGT\ 5'$$

Figura 1.1: Um palíndromo em uma visão mais molecular

Uma visão mais molecular [17] é a de que um *palíndromo* é um segmento de uma dupla-hélice de um DNA ou um RNA que será lido igualmente em ambos os segmentos se escolhermos uma direção para esta leitura. Uma outra definição, baseada em palíndromos complementares, é a de que um palíndromo é um segmento de dupla-hélice de um DNA ou um RNA que é simétrico[1] em relação ao eixo horizontal e ao ponto mediano do segmento, ou seja, se concatenarmos as duas cadeias representando o segmento, usando uma das possíveis direções, teremos um palíndromo complementar. No exemplo da figura 1.1, temos um palíndromo segundo esta última definição, pois se concatenarmos os dois segmentos da direção $5'$ para a direção $3'$ teremos um palíndromo complementar. Portanto, o segmento de dupla-hélice de DNA formado por estas duas cadeias é simétrico em relação ao eixo horizontal e ao ponto mediano.

Um *palíndromo separado ou interrompido* é um palíndromo em que os dois lados simétricos que compõe o palíndromo estão separados por uma subcadeia qualquer. Similarmente, podemos definir que um *palíndromo complementar separado ou interrompido* é aquele em que as duas cadeias de DNA ou RNA que o formam estão separadas por uma subcadeia composta por quaisquer outras bases válidas.

Formalmente, seja Σ um alfabeto. Se $P = ww^R$ então P é um *palíndromo par*; se $P = w\alpha w^R$, então P é um *palíndromo ímpar*; e se $P = wvw^R$, então P é um *palíndromo interrompido*, onde $P, v, w \in \Sigma^*$ e $\alpha \in \Sigma$. Por exemplo, se $P = alexxela$ então P é um palíndromo par, se $P = alexaxela$ então P será um palíndromo ímpar, e se $P = alexopqxela$, então P é um palíndromo interrompido, onde a subcadeia opq separa os dois lados do palíndromo. Similarmente, podemos definir as noções de *palíndromo complementar par, ímpar e interrompido* mudando a cadeia w^R nas definições acima pela cadeia $(w^R)^C$.[2] Por exemplo, o palíndromo complementar

[1]Daqui em diante, a simetria em palíndromos complementares significará que os caracteres simétricos em relação ao centro são complementares, ou seja, representam bases que são complementares.

[2]Daqui em diante, S^R indicará a cadeia S revertiva e S^C indicará a cadeia S complementada,

$AGCTCGCGAGCT$ dado anteriormente é um palíndromo complementar par, e o palíndromo $GCCATGCGGC$ é um palíndromo complementar interrompido, onde a subcadeia é igual a $ATGC$.

Dada uma cadeia S de tamanho N, dizemos que há um *palíndromo par* de raio R *centrado* na posição c de S, $1 \leq c < N$, se $S(c-i+1) = S(c+i)$, para $i = 1, \ldots, R$. Similarmente, há um *palíndromo ímpar* de raio R *centrado* na posição c de S, $1 < c < N$, se $S(c-i) = S(c+i)$, para $i = 1, \ldots, R$. O raio R será *máximo* se não existir um palíndromo com raio maior do que R centrado nesta posição, e neste caso, o palíndromo com este raio é chamado de *palíndromo maximal*. Em alguns contextos, quando a posição for omitida, o palíndromo estará centrado no meio da cadeia. Por exemplo, se $S = alexaabbabbaaxela$, então teremos um palíndromo par de raio igual a 1 nas posições 5 e 12, um palíndromo par de raio 2 nas posições 7 e 10, e um palíndromo ímpar de raio 8 na posição 9.

Um palíndromo centrado na posição c será *inicial* se ele terminar na posição inicial da cadeia, isto é, se $R = c$ para o palíndromo par, e se $R = c - 1$ para o palíndromo ímpar. No exemplo acima, o palíndromo ímpar centrado na posição 9 é inicial, pois $R = c - 1 = 9 - 1 = 8$.

Também podemos definir as noções de *palíndromo complementar par e ímpar* de raio R *centrado* na posição c de uma cadeia S, de *palíndromo complementar maximal*, e de *palíndromo complementar inicial* se, nas definições dadas anteriormente, $S(c-i+1) = S^C(c+i)$ para o palíndromo par centrado na posição c, e $S(c-i) = S^C(c+i)$ para o palíndromo ímpar centrado na mesma posição, onde $1 \leq i \leq R$.

A determinação de todos os palíndromos maximais em uma cadeia S de tamanho N pode ser feita em tempo linear em relação ao tamanho da cadeia através do uso das árvores de sufixos, e da busca do último ancestral comum de duas folhas desta árvore, através dos seguintes passos:

1. Obtemos a árvore de sufixos para a cadeia $S^R S$ e a pré-processamos para podermos obter o último ancestral comum.

2. Usando a árvore de sufixos obtemos, em tempo constante, o raio do palíndromo maximal par para um centro c, $1 \leq c < N$, determinando o último ancestral comum dos sufixos $S[1..c]^R$ e $S[c+1..N]$ para descobrir o tamanho do prefixo comum entre estes dois sufixos, o qual será exatamente o raio R do palíndromo

onde trocamos cada base da cadeia de DNA ou RNA por sua base complementar.

maximal. Para obtermos o palíndromo maximal ímpar, basta buscarmos o ancestral para os sufixos $S[1..c-1]^R$ e $S[c+1..N]$.

Para obtermos todos os palíndromos maximais complementares pares e ímpares centrados nas posições da cadeia S acima, devemos, no passo 1, calcular a árvore de sufixos para $S^R S^C$, e no passo 2, calculamos os ancestrais com a cadeia $S[c+1..N]^C$ ao invés da cadeia $S[c+1..N]$.

Existem vários artigos que tratam da procura de palíndromos. Em [31] foi descoberto um algoritmo com tempo de execução linear que acha todos os palíndromos iniciais, e que também calcula todos os palíndromos com raio máximo centrados em todas as possíveis posições de uma cadeia, e em [23] é apresentado outro algoritmo para achar todos os palíndromos iniciais de uma cadeia em tempo linear.

Também existem algoritmos paralelos para a obtenção de todos os palíndromos presentes em uma cadeia. Em [16] é descrito um algoritmo paralelo não ótimo com tempo de execução de $O(\log n)$ que calcula todos os palíndromos iniciais de uma cadeia sobre um alfabeto genérico, e em [3] é descrito um algoritmo ótimo com tempo de $O(\log\log n)$ para o mesmo problema. Em [10] é mostrado que qualquer algoritmo paralelo que ache todos os palíndromos iniciais de uma cadeia sobre um alfabeto genérico requer um tempo de $\Omega(\lceil n/p \rceil + \log\log_{\lceil 1+p/n \rceil} 2p)$ usando p processadores, e é apresentado um algoritmo com tempo de $O(\log\log n)$ para achar todos os palíndromos iniciais. Em [4] são descritos dois algoritmos para descobrir todos os palíndromos de uma cadeia, um com tempo de $O(\log n)$ com n processadores, e outro com tempo de $O(\log\log n)$ e $n\log n/\log\log n$ processadores, ambos para cadeias sobre alfabetos genéricos.

1.2 Palíndromos Aproximados

Vamos agora descrever a definição de palíndromo aproximado que foi criada após um estudo das várias possibilidades de definições para o que seria um palíndromo aproximado. Em [17] é citada uma definição de palíndromo aproximado que somente considera casamentos e descasamentos, mas que é muito simples se comparada à nossa definição, que considera além dos descasamentos, a inserção e deleção de caracteres.

A definição de palíndromo aproximado só é aplicada se definirmos uma posição onde o centro do palíndromo está na cadeia S (podemos considerar o meio da cadeia

como o centro, se este não foi especificado). Assim como no caso de palíndromos exatos, podemos ter palíndromos aproximados pares e ímpares centrados na posição c desta cadeia S.

Sejam N o tamanho da cadeia S e c o centro onde está o palíndromo, seja $S_e = S[1..c]^R$ se desejamos um palíndromo aproximado par, ou $S_e = S[1..c-1]^R$ se desejamos um palíndromo aproximado ímpar, e seja $S_d = S[c+1..N]$. Existirá um *palíndromo aproximado* com no máximo k erros *centrado* em c (*par* ou *ímpar*) com tamanho total de $m+n$, se for possível converter um prefixo de S_e com tamanho igual a m num prefixo de S_d com tamanho igual a n com no máximo k erros, onde um erro ou é uma inserção de um caractere em S_e ou S_d, ou é uma substituição de um caractere de S_e por um outro de S_d. O *palíndromo aproximado maximal* com no máximo k erros neste centro será qualquer dos palíndromos com valor máximo para a soma $m+n$, e com o menor número de erros. Se o prefixo de S_e usado for igual a S_e, então teremos um *palíndromo aproximado inicial* com no máximo k erros.

Assim como no caso de palíndromos exatos, podemos definir palíndromos aproximados pares e ímpares com no máximo k erros para todos os centros (posições) da cadeia, com exceção da posição 1 para o palíndromo aproximado ímpar, e da posição N para os palíndromos aproximados par e ímpar, pois nestes casos teríamos $S_e = \varepsilon$ ou $S_d = \varepsilon$.

Também podemos ter a noção de *palíndromos aproximados interrompidos* se soubermos onde a subcadeia se localiza na cadeia S. Se a subcadeia começa na posição c e termina na posição d, então basta mudarmos as definições de S_e e de S_d para $S_e = S[1..c-1]^R$ e $S_d = S[d+1..N]$.

Podemos também definir as noções acima para os palíndromos complementares, se trocarmos a definição de S_d para $S_d = S[c+1..N]^C$ na definição de palíndromos aproximados, e para $S_d = S[d+1..N]^C$ na adaptação para os palíndromos aproximados interrompidos.

Por exemplo, para a cadeia $S = bbaabab$, se $k = 3$ teremos os palíndromos aproximados pares dados na tabela 1.1, e os palíndromos aproximados ímpares dados na tabela 1.2.

O algoritmo descrito em [17, 26] foi adaptado para o nosso caso particular e tem um tempo de execução de $O(k^2)$ no pior caso para um dado centro c de uma cadeia S, após um tempo de pré-processamento de $O(N)$.[3] O algoritmo para calcular todos

[3] Se estivermos em um modelo de computação RAM, e se o alfabeto for fixo.

Posição na cadeia (centro)	Palíndromo
1	␣␣␣b baab
2	b␣b␣␣ aabab
3	b␣ba abab
4	bbaa* bab␣
5	bbaab ␣a␣b␣
6	aaba ␣b␣␣

Tabela 1.1: Na tabela, temos todos os palíndromos aproximados pares da cadeia *bbaabab* com no máximo 3 erros. O centro do palíndromo é indicado por um espaço, e o caractere "␣" indica uma inserção. O caractere "*" sobre um caractere do lado esquerdo indica que este caractere deve ser substituído pelo caractere do lado direito, e portanto, indica os pontos em que temos uma substituição.

Posição na cadeia (centro)	Palíndromo
2	␣b␣␣ b aaba
3	b␣b␣ a abab
4	bba␣ a bab␣
5	bbaa b a␣b␣
6	baab a b␣␣␣

Tabela 1.2: Na tabela, temos todos os palíndromos aproximados ímpares da cadeia *bbaabab* com no máximo 3 erros. O centro do palíndromo é indicado pelo caractere na posição correspondente da cadeia, e o caractere "␣" indica uma inserção.

os palíndromos aproximados (pares e ímpares) maximais com no máximo k erros centrados em todas as possíveis posições da cadeia S será de $O(k^2N)$. Uma descrição mais detalhada destes dois algoritmos e da otimização que melhora este algoritmo será vista no capítulo 5.

1.3 Aplicações

As aplicações mais comuns de palíndromos e cadeias com a forma de palíndromos são encontradas em biologia molecular, sendo mais comum o uso da noção de palíndromos complementares.

Por exemplo, palíndromos complementares são encontrados em DNAs e RNAs que atuam como reguladores na transcrição de um DNA para um RNA, onde as duas partes do palíndromo complementar se dobram e se emparelham para formar

um "grampo na forma de um laço", e em tRNAs (RNA transportadores) que permitem que a molécula dobre-se em uma estrutura de uma folha de trevo através do emparelhamento das bases complementares. Algumas pequenas estruturas com a forma de palíndromos são usadas, assim como outras pequenas estruturas repetitivas (como vetores de arranjos [17]), para ajudar a dobragem dos cromossomos em uma estrutura mais compacta.

Também são encontrados palíndromos complementares na família Alu [21], que são estruturas repetitivas intercaladas[4] encontradas ao longo dos genomas dos mamíferos, com cerca de 300 nucleotídos, ocorrendo em cópias quase idênticas ao longo de todo o genoma. O interior de uma Alu geralmente consiste de subcadeias com tamanho de até 40 bases, e ambos os lados da seqüência são constituídos por pedaços de repetições arranjadas com tamanho variando de 7 até 10, que usualmente serão palíndromos complementares interrompidos, se somente considerarmos as bases compostas por estas repetições de arranjos.

Um outro exemplo de cadeias de DNA com a forma de palíndromos em biologia são comumente encontradas em pontos de corte por enzimas de restrição, que são subcadeias repetitivas de pequena escala e bem estruturadas[5], com grande importância em biologia molecular, motivando a busca em bases de DNA por repetições comuns com a mesma forma para descobrir candidatos adicionais para pontos de corte por enzimas de restrição desconhecidos. Uma enzima de restrição é uma enzima que reconhece partes específicas nas cadeias de DNA de procariontes e eucariontes, e particiona o DNA em cada lugar onde o padrão que a enzima reconhece ocorre. Existem centenas de enzimas de restrição conhecidas, e o seu uso é crítico em praticamente todos os aspectos da biologia molecular e da tecnologia de recombinação de DNAs. Um exemplo da importância das enzimas de restrição é a descoberta de que os DNAs dos eucariontes contém introns, que são subcadeias de DNA que limitam as regiões de codificação das proteínas.

Um exemplo de um ponto de corte por enzimas de restrição é dado pela enzima de restrição EcoRI que reconhece o palíndromo complementar $GAATTC$, tendo o ponto de corte entre a base G e sua base A adjacente. Outras enzimas de restrição reconhecem palíndromos complementares interrompidos, como por exemplo, a enzi-

[4]Estas estruturas com função e origem menos claras podem ser divididas em duas classes: as SINEs (short interspersed nuclear sequences) da qual a família Alu faz parte, e as LINEs (long interspersed nuclear sequences).

[5]São estruturas cuja função ou origem são pelo menos parcialmente compreendidas.

ma de restrição BgII que reconhece o palíndromo $GCCNNNNNGGC$, onde N pode ser qualquer uma das bases, e neste caso, o ponto de corte está entre os dois últimos Ns da cadeia. Há uma teoria de que nas estruturas com a forma de palíndromos complementares, as duas metades de um palíndromo complementar (interrompidos ou não) podem ser dobradas para formar pares de DNA complementares, e aparentemente esta dobragem seria a responsável pelo reconhecimento ou cortagem pela enzima.

1.4 Nos Próximos Capítulos

No capítulo 2 iremos estudar o problema de edição de cadeias, um algoritmo básico com tempo de execução de $O(mn)$, para obter a distância de edição e o script de edição entre uma cadeia S de tamanho m e uma cadeia P de tamanho n, e apresentaremos casos particulares deste problema, além de aplicações dos problemas em biologia molecular.

No capítulo 3 iremos estudar o que são árvores de sufixos, e apresentaremos o algoritmo de Ukkonen para calcular a árvore de sufixos para uma cadeia S de tamanho n num tempo de $O(n)$. Este algoritmo será usado pelo algoritmo descrito no capítulo 5, para o problema de casamento de cadeias aproximado, que foi adaptado ao nosso problema.

No capítulo 4 vamos estudar um algoritmo para calcular o último ancestral comum de dois nós de uma árvore genérica num tempo constante, e o algoritmo de pré-processamento desta árvore, com um tempo de execução de $O(n)$, onde n é o número de nós da árvore, necessário para que possamos obter este ancestral num tempo constante.

No capítulo 5 iremos mostrar o algoritmo que construímos para calcular todos os palíndromos aproximados num tempo total de $O(k^2N)$, onde N é o tamanho da cadeia S, e k o número máximo de erros permitidos. Para que seja possível a obtenção de cada um dos $O(N)$ palíndromos num tempo de $O(k^2)$, adaptamos um algoritmo com tempo de execução de $O(kn)$ para o casamento de cadeias aproximado, onde k é o número máximo de erros e n é o tamanho do texto, ao problema de k-diferenças, que em seguida foi adaptado para calcular um palíndromo. Em seguida mostramos duas otimizações que reduzem o trabalho realizado, mas não melhoram o tempo de pior caso. Também apresentamos uma paralelização simples, mas não eficiente, do

nosso algoritmo, com tempo de execução total de $O(k)$ e $O(kN)$ processadores.

No capítulo 6 mostramos exemplos da execução de dois algoritmos, cada um implementando uma das otimizações descritas no capítulo 5, mostrando que as otimizações trazem ganhos ao tempo de execução na prática, apesar de não melhorarem o tempo de execução de pior caso do algoritmo. Também definimos um modelo probabilístico com o objetivo de estimar o tempo de execução médio ao usarmos as duas otimizações, para podermos analisar os ganhos médios obtidos por estas otimizações.

No capítulo 7 apresentamos as conclusões que chegamos ao construímos o nosso algoritmo, e os estudos futuros que podem ser feitos neste problema.

No apêndice A apresentamos as tabelas com os tempos de execução para todos os exemplos executados, dados pelos números totais de passos executados pelo algoritmo ao calcular todos os palíndromos pares e ao calcular todos os palíndromos ímpares.

No apêndice B mostramos todos os gráficos dos ganhos para todos os exemplos executados, das duas otimizações em relação ao tempo de execução máximo, e da segunda otimização em relação a primeira, ao calcularmos todos os palíndromos pares, e ao calcularmos todos os palíndromos ímpares.

Capítulo 2

Edição de Cadeias

Existem várias aplicações para o problema de casamento aproximado, que variam da busca de um padrão (por exemplo, uma palavra) num texto permitindo-se a ocorrência de alguns erros, da conversão de uma cadeia numa outra também permitindo-se erros, até as aplicações do problema em biologia molecular, como, por exemplo, ao buscarmos uma seqüência de DNA ou de proteína dentro de uma base de dados que contém todas as proteínas ou DNAs para algum ser vivo, ou até mesclar o DNA (ou a proteína) com outros similares nesta base para obter uma amostra melhor para este segmento de DNA (ou de proteína). No nosso caso, aplicaremos um caso particular do problema de edição de cadeias, o de k-diferenças, para encontrar todos os palíndromos aproximados centrados em todas as posições de uma cadeia. Mais tarde definiremos como é que calculamos um palíndromo aproximado usando os conceitos definidos neste e nos próximos capítulos.

2.1 Conceitos Básicos

Existem várias formas de definirmos a distância entre duas cadeias. Uma definição mais comum, chamada de *distância de edição* [30, 34], é baseada na transformação (ou edição) de uma cadeia na outra, usando um certo número de operações definidas sobre caracteres individuais das cadeias. Seja S uma cadeia com tamanho igual a m. Vamos agora definir três operações sobre esta cadeia S com custo igual a um, chamadas de *operações de edição*:

- *Inserção* de um novo caractere α entre dois caracteres quaisquer consecutivos de S;
- *Deleção* de um caractere α de qualquer posição de S;

- *Substituição* de um caractere α de qualquer posição de S por um caractere β.

Um *script de edição* σ sobre a cadeia S é qualquer seqüência coerente de casamentos e das operações acima, onde todas as operações de edição são viáveis. Mais especificamente, se I representa uma inserção, D uma deleção, S uma substituição e C um casamento, o script será uma cadeia sobre o alfabeto $\Sigma = \{I, D, S, C\}$ que descreverá como devemos transformar a cadeia S em uma outra cadeia P obtida a partir de S após a execução destas operações. Por exemplo, se $S = vintner$ e $\sigma = SICDCDCCI$, então após a execução deste script σ, se na substituição trocarmos o v pelo w, e se na primeira inserção inserirmos um r e na segunda um s, então a cadeia P será igual a $writers$.

Agora, se P é uma cadeia com tamanho igual a n, o *problema de edição de cadeias* para as cadeias S e P é o de achar o *script de edição ótimo* σ' com o menor número de operações dentre os possíveis scripts que transformam a cadeia S na cadeia P. O número de operações executadas no script é o *número de erros* executados ao transformar S em P. Se tivermos mais de um script ótimo que converte a cadeia S na cadeia P, estes scrips serão chamados de *simultaneamente ótimos.*

Neste caso, a execução do script de edição é interpretada da seguinte forma: seja p_S o ponteiro para o caractere atual de S e p_P o ponteiro para o caractere atual de P, e suponha que antes da execução do script estes ponteiros apontem para o início das cadeias. Ao encontramos um casamento ou uma substituição moveremos ambos os ponteiros das cadeias, e para a operação de edição ser válida, o caractere atual da cadeia S deve ser igual ou diferente do caractere atual de P, dependendo da execução de um casamento ou de uma substituição, e no caso de executarmos uma substituição, o caractere atual de S é substituído pelo caractere atual de P. Ao executarmos uma inserção, o caractere atual da cadeia P é inserido antes do caractere atual da cadeia S, e o ponteiro p_P da cadeia P é incrementado, e ao executarmos uma deleção, o caractere atual de S é deletado, e o ponteiro p_S de S é incrementado.

Para obtermos o script simétrico que transforma a cadeia P na cadeia S, basta trocarmos todas as operações I pelas operações D, e todas as operações D pelas operações I, lembrando que o script é agora executado sobre a cadeia P e não sobre a cadeia S.

A *distância de edição* entre as cadeias S e P é igual ao número de erros do script ótimo. Existe um algoritmo básico com tempo de $O(mn)$ para o cálculo deste

script e da distância de edição baseado em programação dinâmica [17, 36], que será descrito posteriormente.

Uma forma alternativa de visualizarmos a transformação de S em P é através da noção de alinhamento: um *alinhamento*[1] de duas cadeias S e P é obtido após a execução dos seguintes passos:

1. Escolhemos quaisquer posições em S, que dependerão do alinhamento, e inserimos espaços nestas posições. Repetimos o mesmo processo acima para a cadeia P.

2. Após a inserção dos espaços, emparelhamos as duas cadeias de tal forma que um espaço em uma cadeia esteja sempre emparelhado com um caractere da cadeia oposta[2].

A noção de alinhamento é mais usada em biologia molecular do que a noção de script de edição. Como exemplo, no alinhamento das cadeias *qacdbd* e *qawxb*, o caractere c descasa com o caractere w, e os caracteres x e os dois ds estão alinhados com espaços na cadeia oposta.

```
qac dbd
qawx b
```

Matematicamente, o relacionamento entre as cadeias S e P descrito pelo script de edição e pelo alinhamento são equivalentes: uma inserção de um caractere de P equivale a alinhar este caractere com um espaço em S, uma deleção de um caractere de S equivale a alinhar o caractere deletado com um espaço em P, e dois caracteres que casam ou descasam em um alinhamento equivalem a um casamento ou uma substituição no script. Então, a distância de edição pode ser obtida pelo alinhamento, minimizando-se os caracteres opostos a espaços e os descasamentos.

Já do ponto de vista da modelagem, um script de edição é bem diferente de um alinhamento: um script de edição descreve o processo, isto é, os eventos de mutação que irão transformar uma cadeia na outra, e um alinhamento descreve o produto, isto é, as relações existentes entre as duas cadeias.

[1] Na verdade, este conceito se refere ao alinhamento global que usa toda a cadeia, em oposição ao alinhamento local onde obtemos um alinhamento entre subcadeias das cadeias [17].

[2] O emparelhamento de um espaço com outro não é válido.

2.2 Cálculo da Distância de Edição Usando a Programação Dinâmica

Vamos agora descrever o algoritmo básico para cálculo da distância de edição de duas cadeias e do script de edição que converte uma em outra, através do uso da programação dinâmica.

Vamos supor que desejemos obter a distância de edição e o script de edição que converte a cadeia S de tamanho igual a m na cadeia P de tamanho igual a n. Se $D(i,j)$ denotar a distância de edição entre o prefixo $S[1..i]$ da cadeia S e o prefixo $P[1..j]$ da cadeia P, então a distância de edição das cadeias S e P será dada precisamente pela célula $D(m,n)$.

Agora, vamos calcular $D(m,n)$ através da resolução do problema mais geral de calcular $D(i,j)$ para i variando de 1 até m e j variando de 1 até n, e a abordagem da programação dinâmica é bem adequada para a resolução deste problema. Na abordagem da programação dinâmica existem três componentes essenciais: *as relações de recorrência, a computação tabular dos subproblemas e a remontagem do caminho.* Portanto, para obtermos um algoritmo para o cálculo da distância de edição entre S e P devemos definir precisamente cada um destes componentes.

2.2.1 A Relação de Recorrência

Na programação dinâmica, a *relação de recorrência* define o relacionamento recursivo entre o valor de $D(i,j)$, $i,j \geq 0$ com os outros valores de D com pares de índices (x,y) menores do que o par de índices (i,j). Quando não existir um par de índices menor do que (i,j), o valor de $D(i,j)$ deve ser declarado explicitamente por condições chamadas de *condições básicas.* No problema de edição de cadeias, definimos as seguintes condições básicas quando $i = 0$ ou $j = 0$:

$$D(i,0) = i;$$
$$D(0,j) = j;$$

Estas condições básicas estão corretamente definidas, pois o par $(i,0)$ significa converter a cadeia $S[1..i]$ na cadeia vazia, e somente podemos conseguir isso se delatarmos todos os caracteres de $S[1..i]$, e o par $(0,j)$ significa converter a cadeia vazia na cadeia $P[1..j]$, e a única forma de conseguir isso é inserindo todos os caracteres de $P[1..j]$.

Agora, a relação de recorrência do nosso problema para $i, j > 0$ é dada abaixo, onde $t(i,j) = 0$ se $S(i) = S(j)$, 1 se $S(i) \neq S(j)$:

$$D(i,j) = \min\{D(i-1,j-1) + t(i,j), D(i,j-1) + 1, D(i-1,j) + 1\}$$

Lema 2.1 *O valor de $D(i,j)$ deve ser ou o valor de $D(i-1,j-1) + t(i,j)$, ou o valor de $D(i,j-1)+1$, ou o valor de $D(i-1,j)+1$. [17, Cap. 11]*

Prova Seja σ o script de edição ótimo que converte a cadeia $S[1..i]$ na cadeia $P[1..j]$ e seja α o último caractere deste script. Como o script σ é definido sobre o alfabeto $\Sigma = \{I, D, S, C\}$, então temos as seguintes possibilidades para o caractere α:

- $\alpha = C$ ou $\alpha = S \Rightarrow$ Neste caso, antes de executarmos a operação α, já devemos ter convertido a cadeia $S[1..i-1]$ na cadeia $P[1..j-1]$ com o número mínimo de erros, pois se isso não ocorresse, α não seria um script ótimo. Como neste caso, para o script ser válido, α deve ser igual a C se $S(i) = S(j)$, e α deve ser igual a S se $S(i) \neq S(j)$, e como o número mínimo de erros para converter $S[1..i-1]$ em $P[1..j-1]$ é dado por $D(i-1,j-1)$, então $D(i,j) = D(i-1,j-1)+t(i,j)$.

- $\alpha = I \Rightarrow$ Antes da execução da operação α, já convertemos a cadeia $S[1..i]$ na cadeia $P[1..j-1]$ com o número mínimo de erros dado por $D(i,j-1)$, pois se não novamente σ não seria um script ótimo. Como a inserção do caractere $P(j)$ incrementa em um o número de erros, então $D(i,j) = D(i,j-1)+1$.

- $\alpha = D \Rightarrow$ Neste caso já convertemos a cadeia $S[1..i-1]$ na cadeia $P[1..j]$ com o número mínimo de erros dado pela célula $D(i-1,j)$. Como neste caso só nos resta deletar o caractere $S(i)$, como essa deleção incrementa o número de erros em um, e como $D(i-1,j)$ deve ser mínimo pois se não σ não seria ótimo, então $D(i,j) = D(i-1,j)+1$. □

Então $D(i,j)$ deve ser igual a $D(i-1,j-1)+t(i,j)$, $D(i,j-1)+1$ ou $D(i-1,j)+1$.

Lema 2.2 $D(i,j) \leq \min\{D(i-1,j-1) + t(i,j), D(i,j-1)+1, D(i-1,j)+1\}$ *[17, Cap. 11]*

Prova O objetivo deste lema é o de mostrar que podemos converter a cadeia $S[1..i]$ na cadeia $P[1..j]$ usando todas as três possibilidades que compõem o mínimo, pois neste caso, o mínimo será viável. Para transformarmos $S[1..i]$ em $P[1..j]$ a partir da conversão de $S[1..i-1]$ em $P[1..j-1]$, simplesmente convertemos estas cadeias

usando o número mínimo de erros dado pela célula $D(i-1, j-1)$, e logo após executamos um casamento, se $S(i) = S(j)$, ou um descasamento, se $S(i) \neq S(j)$. Portanto é possível converter $S[1..i]$ em $P[1..j]$ com $D(i-1, j-1)+t(i, j)$ erros, pois o casamento não aumenta o número de erros, e a substituição aumenta o número de erros em uma unidade. Já para transformarmos $S[1..i]$ em $P[1..j]$ a partir da conversão de $S[1..i]$ em $P[1..j-1]$, simplesmente convertemos as cadeias com o número de erros dado pela célula $D(i, j-1)$, e logo após inserimos o caractere $P(j)$, o que irá aumentar o número de erros em uma unidade. Portanto também é viável converter $S[1..i]$ em $P[1..j]$ com $D(i, j-1)+1$ erros. Similarmente é possível converter $S[1..i]$ em $P[1..j]$ a partir da conversão de $S[1..i-1]$ em $P[1..j]$, seguida da deleção do caractere $S(i)$, e neste caso o número de erros será igual a $D(i-1, j)+1$, e este valor também será viável. Como esgotamos todas as possibilidades para o mínimo, este é valido e o lema está provado. □

Teorema 2.1 *Se* $i, j > 0$, *então* $D(i, j) = \min\{D(i-1, j-1)+t(i, j), D(i, j-1)+1, D(i-1, j)+1\}$. *[17, Cap. 11]*

Prova Do lema 2.1 vem que $D(i, j)$ ou é $D(i-1, j-1)+t(i, j)$, $D(i, j-1)+1$ ou $D(i-1, j)+1$ e do lema 2.2 sabemos que $D(i, j)$ é menor ou igual do que o menor destes valores. Portanto $D(i, j)$ deve ser igual ao menor valor, e o teorema está provado. □

Pelo teorema 2.1, a relação de recorrência que calcula $D(i, j)$ para $i, j > 0$ está correta.

2.2.2 Computação Tabular da Distância de Edição

O segundo componente essencial da programação dinâmica é como devemos usar a relação de recorrência e as condições básicas para computar eficientemente o valor da célula $D(m, n)$. Uma maneira não eficiente de computar este valor é a de usar uma *abordagem top-down* através do uso de um procedimento que computa recursivamente o valor de $D(i, j)$, que é chamado com a entrada $D(m, n)$, e que será ineficiente para valores grandes de m e n, pois o número de vezes que o procedimento recursivo é executado será exponencial.

O processo top-down acima é ineficiente devido ao número massivo de computações redundantes [13], pois temos somente $(m+1) \times (n+1)$ subproblemas

(chamadas recursivas) a serem executados, devido ao tamanho de S ser igual a m e o de P ser igual a n. A idéia para se obter um algoritmo mais eficiente para computar $D(m, n)$ é abandonar a simplicidade do método top-down e usar o método botton-up.

$D(i,j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1							
i	2	2							
n	3	3							
t	4	4							
n	5	5							
e	6	6							
r	7	7							

Figura 2.1: A matriz de programação dinâmica usada para computar a distância de edição entre *vintner* e *writers*. Os valores da linha e da coluna 0 já foram inicializados usando as condições básicas.

Na *abordagem bottom-up* computamos inicialmente os valores de $D(i, j)$ para os menores valores possíveis de i e j, e depois computamos os outros valores de $D(i, j)$ para valores incrementais de i e de j. No nosso caso, a computação botton-up será organizada em uma matriz com $(m + 1) \times (n + 1)$ células tendo os valores para todas as possíveis escolhas de i e j. A cadeia S é representada pela linha vertical da matriz, e a cadeia P é representada pela linha horizontal. Como os valores de i e de j podem ser iguais a zero, a matriz tem uma linha 0 (onde $j = 0$) e uma coluna 0 (onde $i = 0$) que são inicializadas usando as condições básicas definidas. Agora, para computarmos todas as outras $m \times n$ células restantes na matriz, preenchemos uma linha de cada vez, de cima para baixo (para manter a ordem incremental dos índices i), e em uma linha particular calculamos as células da esquerda para a direita (para manter a ordem incremental dos índices j).

Para calcularmos uma célula da matriz vemos, pela figura 2.1, que o valor da célula $D(1, 1)$ pode ser computado após preenchermos a linha e a coluna 0 com as condições básicas, pois para calcularmos esta célula precisamos dos valores de $D(0, 0)$, $D(1, 0)$ e $D(0, 1)$. Para calcular $D(1, 2)$, precisamos de $D(0, 1)$, de $D(1, 1)$ que acabou de ser calculado, e de $D(0, 2)$, e mais genericamente, para calcularmos

$D(1,j)$ precisamos de $D(0,j-1)$, $D(0,j)$ e $D(1,j-1)$ que acabou de ser calculado. Para calcular a célula genérica $D(i,j)$ da linha i, precisamos dos valores de $D(i-1,j-1)$ e $D(i-1,j)$ calculados ao processar a linha $i-1$, e do valor de $D(i,j-1)$ que foi calculado no passo anterior. Portanto, todas as células da matriz serão corretamente calculadas se processamos a matriz linha por linha de cima para baixo, e se em cada linha processarmos as células da esquerda para a direita. Na figura 2.2 apresentamos um exemplo do preenchimento desta matriz.

$D(i,j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	1	2	3	4	5	6	7
v	1	1	1	2	3	4	5	6	7
i	2	2	2	2	2	3	4	5	6
n	3	3	3	3	3	3	4	5	6
t	4	4	4	4	4	*			
n	5	5							
e	6	6							
r	7	7							

Figura 2.2: O exemplo mostra as distâncias de edição já calculadas até a coluna 3 da linha 4, e portanto, a próxima célula a ser computada será a (4, 4) identificada por um "*". O valor de D(4, 4) será igual a 3 desde que D(3, 3) = 3 e S(4) = P(4) = t.

A computação acima também poderia ser alternativamente realizada por colunas, da coluna esquerda para a coluna direita, e dentro de uma coluna, o cálculo das células será executado de cima para baixo. Também poderiamos executar a computação através das anti-diagonais da matriz, onde (i,j) pertence a d se $d = i + j$, e a computação seria executada para valores incrementais de d, e de baixo para cima dentro desta diagonal d.

Ao computarmos todos os valores desta matriz, teremos computado a distância de edição entre a cadeia S e a cadeia P que será igual ao valor da célula (m,n). Então o nosso algoritmo deve exatamente executar o processamento descrito anteriormente. O próximo teorema define qual será o trabalho executado, isso é, o tempo total de execução do algoritmo.

Teorema 2.2 *A matriz de programação dinâmica para calcular a distância de edição entre uma cadeia S de tamanho m e uma cadeia P de tamanho n pode*

ser preenchida com um trabalho de $O(mn)$. Portanto, o algoritmo para calcular a distância de edição $D(m,n)$ terá um tempo de execução de $O(mn)$. [17, Cap. 11]

Prova Ao computarmos o valor da célula (i,j), somente precisamos checar os valores das células $(i-1,j-1)$ para executarmos uma substituição ou um casamento, da célula $(i,j-1)$ para executarmos uma inserção, e da célula $(i-1,j)$ para executarmos uma deleção, ao descobrirmos qual é o valor mínimo. Como a obtenção, a comparação destes valores, e as somas podem ser feitas num tempo constante[3], como o cálculo de uma das células da linha e da coluna 0 a partir das condições básicas também pode ser feito num tempo constante, e como temos $(m+1)\times(n+1)$ células, o trabalho total para computar todas as células será de $O(mn)$, e, portanto, o tempo de execução para o cálculo da distância de edição entre a cadeia S e a cadeia P será de $O(mn)$. □

2.2.3 Remontagem do Caminho

Agora que já obtivemos a distância de edição entre as cadeias S e P devemos saber como podemos obter o script de edição σ associado com esta transformação. Isso pode ser feito através da criação de ponteiros na matriz, na medida em que os valores das células são calculados.

Seja (i,j), $i,j>0$, a célula que acabou de ser calculada. Se $D(i,j)$ for igual a $D(i-1,j-1)+t(i,j)$, criamos um ponteiro da célula (i,j) para a célula $(i-1,j-1)$, se $D(i,j)$ for igual a $D(i,j-1)+1$ criamos um ponteiro da célula (i,j) para a célula $(i,j-1)$, e se $D(i,j)$ for igual a $D(i-1,j)+1$ criamos um ponteiro da célula (i,j) para a célula $(i-1,j)$. Como na maioria dos casos a célula $(0,j)$ representa a conversão da cadeia vazia na cadeia $P[1..j]$, e a célula $(i,0)$ representa a conversão da cadeia $S[1..i]$ na cadeia vazia, teremos um ponteiro de $(0,j)$ para $(0,j-1)$ se $j>0$ e $(i,0)$ para $(i-1,0)$ se $i>0$. A célula $(0,0)$ não terá ponteiros saindo dela.

Agora, para calcularmos o script de edição, seguimos o caminho que vai da célula (m,n) a célula $(0,0)$, e interpretamos cada ponteiro de (i,j) para $(i,j-1)$ como uma inserção do caractere $P(j)$, cada ponteiro de (i,j) para $(i-1,j)$ como uma deleção do caractere $S(i)$, e cada ponteiro de (i,j) para $(i-1,j-1)$ como um casamento se $S(i)=P(j)$, e como uma substituição se $S(i)\neq S(j)$. Podemos facilmente construir o script ótimo σ de trás para frente, inserindo o caractere correto que representa

[3]Em um modelo de computação RAM onde a soma e a comparação de dois valores pode ser feita num tempo constante.

a operação executada por um movimento de uma célula para outra através destes ponteiros.

Se estamos interessados em obter o alinhamento ótimo das cadeias, basta interpretarmos um ponteiro de (i, j) para $(i, j-1)$ como um alinhamento de $P(j)$ com um espaço, um ponteiro de (i, j) para $(i-1, j)$ como um alinhamento de $S(i)$ com um espaço, e um ponteiro de (i, j) para $(i-1, j-1)$ como um descasamento se $S(i) \neq S(j)$, e como um casamento se $S(i) = S(j)$. Assim como no caso da construção do script, o alinhamento será obtido de trás para a frente.

$D(i,j)$			w	r	i	t	e	r	s
		0	1	2	3	4	5	6	7
	0	0	← 1	← 2	← 3	←4	← 5	← 6	← 7
v	1	↑ 1	↖ 1	↖← 2	↖← 3	↖← 4	↖← 5	↖← 6	↖← 7
i	2	↑ 2	↖↑ 2	↖ 2	↖ 2	← 3	← 4	← 5	← 6
n	3	↑ 3	↖↑ 3	↖↑ 3	↖↑ 3	↖ 3	↖← 4	↖← 5	↖← 6
t	4	↑ 4	↖↑ 4	↖↑ 4	↖↑ 4	↖ 3	↖← 4	↖← 5	↖← 6
n	5	↑ 5	↖↑ 5	↖↑ 5	↖↑ 5	↑ 4	↖ 4	↖← 5	↖← 6
e	6	↑ 6	↖↑ 6	↖↑ 6	↖↑ 6	↑ 5	↖ 4	↖← 5	↖← 6
r	7	↑ 7	↖↑ 7	↖ 6	↖←↑ 7	↑ 6	↑ 5	↖ 4	←5

Figura 2.3: A matriz de programação dinâmica já preenchida com os ponteiros necessários para descobrirmos o script. Para uma célula (i, j), a seta ↖ aponta para a célula $(i-1, j-1)$, a seta ← aponta para a célula $(i, j-1)$, e a seta ↑ aponta para a célula $(i-1, j)$.

No exemplo dado na figura 2.3, onde $S = vintner$ e $P = writers$, existem três remontagens do caminho da célula $(7, 7)$ para a célula $(0, 0)$. Estes caminhos são idênticos na porção da célula $(7, 7)$ até a célula $(3, 3)$, e após este ponto teremos duas possibilidades (para cima ou diagonalmente) a serem escolhidas. Neste caso temos três possíveis scripts de edição para as cadeias S e P: $SSSCDCCI$, $SICDCDCCI$ e $ISCDCDCCI$. Os alinhamentos correspondentes a estes scripts são dados abaixo:

vintner	v intner	vintner
writ ers	wri t ers	wri t ers

Se há mais de um ponteiro saindo da célula (m, n) então escolhemos um destes ponteiros, pois qualquer um destes caminhos será um caminho da célula (m, n) até a célula $(0, 0)$, e em seguida continuamos o processamento para a próxima célula. Mais genericamente, se estamos na célula (i, j) e temos mais de um caminho saindo

desta célula, escolhemos um destes caminhos, seguimos o ponteiro e continuamos o processamento com a próxima célula. Portanto, um caminho da célula (m, n) até a célula $(0, 0)$ pode ser obtido por um dos caminhos que saem da célula (m, n), e de todas as outras células intermediárias que tenham mais de um caminho possível. Como somente a célula $(0, 0)$ não tem ponteiros saindo dela, o caminho sempre chegará ao ponto $(0, 0)$. Desde que qualquer caminho descrito pelos ponteiros do ponto (m, n) ao ponto $(0, 0)$ define um script ou alinhamento ótimo, enunciamos o seguinte teorema:

Teorema 2.3 *Uma vez que já tenhamos computado a matriz de programação dinâmica com os respectivos ponteiros entre as células desta matriz, um script de edição ótimo pode ser obtido num tempo de* $O(m + n)$.

Na verdade, apesar de os ponteiros no caminho da célula (m, n) até a célula $(0, 0)$ nos permitirem obter todos os scrips ótimos que convertem a cadeia S na cadeia P, recuperamos somente um script porque o tempo de recuperação de todos estes scripts seria exponencial, pois para cada célula no caminho podemos ter no pior caso três caminhos saindo dela, e um destes scripts simultaneamente ótimos pode ser obtido no limite de tempo definido pelo teorema 2.3. Com base nesta observação podemos enunciar o seguinte teorema:

Teorema 2.4 *Qualquer caminho do ponto* (m, n) *ao ponto* $(0, 0)$ *obtido através do uso dos ponteiros entre as células criados ao computar a matriz especificam um script de edição com o número mínimo de operações, e, reciprocamente, um script de edição ótimo deve ser representado por um destes caminhos. Além disso, como um script é descrito por um único caminho, a relação entre scripts de edição e caminhos é de um-para-um.*

2.3 Grafo de Edição

Dada uma cadeia S de tamanho igual a m e uma cadeia P de tamanho igual a n, um *grafo de edição ponderado* é um grafo acíclico com $(m + 1) \times (n + 1)$ nós, cada um rotulado por um par distinto (i, j). O significado das arestas do grafo dependem do problema sobre as cadeias, e no caso do problema de edição de cadeias, teremos uma aresta de (i, j) para $(i, j + 1)$, se $j < n$, com custo igual a 1 representando a inserção do caractere $P(j + 1)$, uma aresta de (i, j) para $(i + 1, j)$, se $i < m$, com

custo igual a 1 representando a deleção do caractere $S(i+1)$, e uma aresta de (i,j) para $(i+1,j+1)$, se $i < m$ e $j < n$, com custo igual a 1 se $S(i+1) \neq P(j+1)$, e 0 se $S(i+1) = P(j+1)$, representando, respectivamente, uma substituição e um casamento. Como exemplo, na figura 2.4, vemos um grafo de edição para as cadeias CAN e ANN.

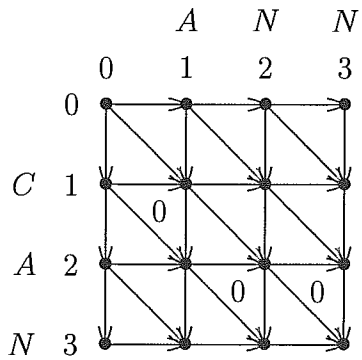


Figura 2.4: Grafo de edição para as cadeias CAN e ANN. Todas as arestas, exceto as indicadas na figura, têm peso igual a um.

A principal propriedade do grafo de edição é a de que qualquer caminho mais curto da fonte (o nó $(0,0)$) ao sumidouro (o nó (m,n)) representa um script de edição ótimo com o número mínimo de operações de edição, ou o alinhamento ótimo com o menor custo (o número de espaços e descasamentos), e com isso, teremos um mapeamento de um para um entre os scripts de edição e os caminhos mais curtos. Com base nas afirmações acima, podemos enunciar o teorema e o colorário a seguir:

Teorema 2.5 *Um script de edição para as cadeias S e P terá o número mínimo de operações, ou equivalentemente, um alinhamento terá o menor custo, se e somente se ele corresponder a um dos caminhos mais curtos da fonte ao sumidouro do grafo de edição.*

Colorário 2.1 *O conjunto formado por todos os caminhos mais curtos da fonte ao sumidouro do grafo representa exatamente todos os scripts de edição ou alinhamentos ótimos sobre as cadeias S e P (e vice-versa).*

No grafo de edição, o script de edição ótimo pode ser obtido a partir do caminho mais curto da fonte ao sumidouro, interpretando cada aresta horizontal como uma inserção, cada aresta vertical como uma deleção, e cada aresta diagonal de (i,j)

para $(i+1, j+1)$ como uma substituição se $S(i+1) \neq P(j+1)$, e um casamento se $S(i+1) = P(j+1)$. O alinhamento ótimo pode ser obtido de forma similar.

Além disso, devemos notar também que um script de edição ótimo (ou um alinhamento) sobre os prefixos $S[1..i]$ e $P[1..j]$ é representado pelo caminho mais curto da fonte ao nó (i, j), e que todos os scripts de edição ótimos sobre estes prefixos são representados por caminhos mais curtos da fonte ao nó (i, j).

A visão do problema de edição de cadeias como um grafo de edição é freqüentimente útil, pois existem várias ferramentas para descobrir e representar compactamente os caminhos mais curtos em um grafo.

2.4 Casos Particulares

Existem diversas variações do problema de edição de cadeias básico. Estas variações podem ser casos particulares do problema de edição de cadeias básico, ou casos em que cada erro poderá ter um custo diferente, dependendo ou do tipo da operação, ou dos caracteres envolvidos nesta operação.

Os seguintes casos particulares do problema de edição de cadeias são mais comumente estudados: o problema de k-diferenças, o problema de k-descasamentos, e o problema de casamento de cadeias aproximado.

Se somente desejamos obter um script de edição de S em P com no máximo k erros, isto é, somente estamos interessados na conversão se esta gera menos do que k erros, teremos um caso particular de edição de cadeias chamado de *k-diferenças* [25, 33]. Se as inserções e as deleções não são permitidas, teremos um problema de *k-descasamentos* [25]. Se usarmos a noção de alinhamento, este problema geralmente será chamado de *alinhamento global com k diferenças* [15, 28].

Se desejamos procurar um padrão P de tamanho m num texto T de tamanho n, e assim como no caso de k-diferenças estamos interessados em conversões com no máximo k erros, e levando em conta que neste caso não devemos contar as inserções de caracteres do texto no inicio do padrão, teremos agora um caso particular do problema de k-diferenças chamado de *casamento de cadeias aproximado* ou *problema de casamento inexato com k diferenças* [39]. Neste caso, uma das condições básicas deve ser redefinida como $D(0, j) = 0$, ao criarmos os ponteiros entre as células, e ao calcular a matriz não teremos um ponteiro de $(0, j)$, $j > 0$, para $(0, j-1)$, e o script de edição pode começar em qualquer célula $(0, j_1)$, $j_1 \geq 0$, e terminar em

qualquer célula (m, j_2), $j_2 \geq 0$, e representa a conversão de P em $T[j_1..j_2]$. No grafo de edição, não existem arestas verticais dos pontos $(0, j)$ para $(0, j+1)$ e de (m, j) para $(m, j+1)$, $j \geq 0$, e um caminho mais curto de qualquer ponto $(0, j_1)$ para qualquer ponto (m, j_2), $j_1, j_2 \geq 0$ representa um script que transforma o padrão na porção do texto que começa em j_1 e termina em j_2.

Das variações do problema de edição de cadeias descrita acima, a mais estudada é a de casamento de cadeias aproximado. Existem várias propostas para reduzir o tempo básico de $O(mn)$. Em [40] é apresentado um algoritmo com tempo de execução de $O(n)$ após uma fase de pré-processamento custosa, e é apresentado outro algoritmo com tempo esperado de $O(kn)$. Em [17, 26] é apresentado um algoritmo com tempo de pior caso de $O(kn)$, usando árvores de sufixos e a busca do último ancestral comum. Em [7, 44] a visão da matriz de programação dinâmica como um autômato é usada para otimizar o algoritmo básico de $O(mn)$ para $O(n)$ e $O(kn)$, respectivamente. Em [7, 12, 17] o particionamento é usado para reduzir a porção de texto analisada pelo algoritmo. E em [17] é apresentado um algoritmo com tempo de execução de $O(kn)$ para o problema de k-diferenças, e um algoritmo otimizado baseado no paradigma dos quatro russos [6].

Dos algoritmos descritos acima para o casamento de cadeias aproximado, somente adaptaremos os algoritmos descritos no capítulo 5, pois a maior parte dos algoritmos existentes para casamento de cadeias aproximado aproveita o fato de o tamanho do padrão ser (muito) menor do que o tamanho do texto.

Também existem algoritmos paralelos para o problema de edição de cadeias e os casos particulares descritos acima. Em [8] são discutidas formas de paralelização do algoritmo básico com aceleração ótima (com $PT = O(mn)$, onde P é o número de processadores, e T é o tempo de execução). Em [26] é apresentada uma variação paralela do algoritmo seqüencial descrito no artigo com tempo de execução de $O(k)$ e $O(n)$ processadores, e em [2] são descritos dois algoritmos para o caso de edição de cadeias genérico, um com tempo de $O(\log a \log b)$ e $O(ab/\log a)$ processadores para uma CREW PRAM, e outro com tempo de $O(\log b(\log\log a)^2)$ e $O(ab/\log\log a)$ processadores para uma CRCW PRAM, onde $a = \min(m, n)$ e $b = \max(m, n)$.

Também podemos considerar a possibilidade de o custo dos erros variarem de acordo com as operações ou com os caracteres envolvidos, e nestes casos, teremos um peso, custo ou débito para cada operação ou caractere envolvido.

No caso do custo variar de acordo com as operações envolvidas, teremos um custo

diferente para cada operação, isto é, a inserção terá um custo igual a i', a deleção um custo igual a d, a substituição um custo igual a s, e em alguns casos, também pode ser atribuído um custo c ao casamento. Neste caso, o problema de converter a cadeia S na cadeia P será chamado de *problema de edição de cadeias com operações ponderadas.* As condições básicas serão dadas por:

$$D(i,0) = i \times d;$$
$$D(0,j) = j \times i';$$

A relação de recorrência da computação da distância de edição fica com a seguinte aparência, onde agora $t(i,j) = s$ se $S(i) \neq P(j)$ e $t(i,j) = c$ se $S(i) = P(j)$ (se o casamento não é contabilizado, então $c = 0$).

$$D(i,j) = \min\{D(i-1,j-1) + t(i,j), D(i,j-1) + i', D(i-1,j) + d\}$$

Já no grafo de edição, as arestas horizontais terão um custo de i', as verticais um custo igual a d, e a aresta diagonal do ponto (i,j) para o ponto $(i+1,j+1)$ terá um custo de s se $S(i+1) \neq P(j+1)$, ou de c, se $S(i+1) = P(j+1)$.

No caso do custo depender dos caracteres, teremos um custo diferente para cada caractere envolvido nas operações de inserção e deleção, e para cada par de caracteres envolvidos na operação de substituição. Neste caso teremos o *problema de edição de cadeias com alfabeto ponderado*, ou simplesmente, *edição de cadeias ponderada.* Se o custo de inserir um caractere α for de $I(\alpha)$, o custo de deletar o caractere α for de $D(\alpha)$, se o custo de substituir um caractere α por um caractere β for igual a $Su(\alpha,\beta)$, e se $Su(\alpha,\alpha)$ representar o custo de um casamento (0 se não for contabilizado), então as condições básicas serão dadas por:

$$D(i,0) = \sum_{k=1}^{i} D(S(k));$$
$$D(0,j) = \sum_{k=1}^{j} I(P(k));$$

A relação de recorrência terá a seguinte aparência:

$$D(i,j) = \min\{D(i-1,j-1) + Su(\alpha,\beta), D(i,j-1) + I(\beta), D(i-1,j) + D(\alpha)\},$$

onde $\alpha = S(i)$ e $\beta = P(j)$, e no grafo de edição, a aresta horizontal do ponto (i,j) para o ponto $(i,j+1)$ terá o custo de $I(P(j+1))$, a aresta vertical do ponto (i,j) para o ponto $(i+1,j)$ terá um custo de $D(S(i+1))$, e o custo da aresta diagonal do ponto (i,j) para o ponto $(i+1,j+1)$ terá um custo de $Su(S(i+1),P(j+1))$. A matriz com os pesos para cada caractere do alfabeto é chamada de *matriz ponderada.*

Nos casos acima, o *custo* do script σ será igual a soma de todos os custos das operações executadas neste script, e o custo de todos os casamentos se estes forem contabilizados, e o problema de edição de cadeias para as cadeias S e P será o de achar o script de edição σ' com o menor custo possível dentre os possíveis scripts que transformam a cadeia S na cadeia P. A distância de edição entre as cadeias S e P é definida, neste caso, como este custo mínimo, e é chamada de *distância de edição com operações ponderadas* no primeiro caso, e de *distância de edição com o alfabeto ponderado*, ou simplesmente, *distância de edição ponderada* para o último caso.

As aplicações da distância de edição ponderada geralmente estão ligadas a comparação de proteínas em biologia molecular, onde freqüentimente o termo distância de edição se refere a distância de edição ponderada, e o peso (ou custo) de cada operação é geralmente conhecido como um débito[4]. Há uma grande quantidade de estudo sobre quais devem ser os pesos usados para as operações sobre caracteres de seqüências de aminoácidos [1, 14, 19, 22].

Já ao compararmos seqüências de cadeias de DNA, é mais comum o uso da distância de edição (não ponderada), e da distância de edição com operações ponderadas, mas também é interessante o uso da edição de cadeias com alfabeto ponderado, e esquemas de débitos baseados em alfabetos já foram propostos [20]. Em biologia molecular, a matriz ponderada é conhecida como matriz de substituição de aminoácidos e matriz de substituição de nucleotídios e neste contexto, o termo matriz ponderada tem um significado bem diferente do seu significado usual.

[4]Em biologia molecular é mais comum o uso deste termo do que do termo peso ou custo.

Capítulo 3

Árvore de Sufixos

As árvores de sufixos são estruturas de dados que expõem de forma mais profunda a estrutura interna das cadeias. Podem ser usadas para resolver o problema de casamento exato num tempo linear, mas as maiores aplicações são em problemas de processamento de cadeias mais complexos, que também poderão ser resolvidos num tempo linear através do uso das árvores de sufixos. Outra característica importante das árvores de sufixos é que podem ser consideradas como uma ponte entre os problemas de casamento exato e inexato [17].

Um dos problemas mais comuns onde a árvore de sufixos pode ser aplicada é o problema da subcadeia: dado um texto T de tamanho m e uma cadeia S desconhecida, devemos procurar por uma ocorrência desta cadeia no texto. Se for achada, retornamos esta ocorrência; se não, retornamos que a cadeia não está no texto. Se o tamanho de uma cadeia particular S é de n, este problema pode ser resolvido num tempo de $O(n)$ para cada cadeia S, após uma fase de pré-processamento de $O(m)$, onde a árvore de sufixos para o texto T é construída.

Outros exemplos podem ser citados, como uma generalização do problema acima onde o texto é composto por um conjunto de cadeias, e queremos saber se a cadeia S é uma subcadeia de uma destas. Em [17, Cap. 7] são dadas várias aplicações em biologia molecular do uso de árvores de sufixos. O uso da árvore de sufixos e da busca do último ancestral comum de dois sufixos, que será descrito no próximo capítulo, é essencial para a obtenção dos limites de tempo alegados em [17, 26], sendo de grande importância ao nosso problema, e para vários outros problemas descritos em [17].

Existe um algoritmo de fácil entendimento, mas não otimizado, que obtém uma árvore de sufixos para uma cadeia S de tamanho n num tempo de $O(n^2)$. O primeiro

algoritmo linear existente para a construção de uma árvore de sufixos foi descrito por Weiner [42], e, logo após, um outro algoritmo com complexidade de espaço mais eficiente foi dado por McCreight [32]. Além destes dois algoritmos, conhecidos por terem uma descrição muito complexa, foi criado um outro algoritmo conceitualmente diferente por Ukkonen [41], com todas as vantagens do algoritmo de McCreight, mas com uma descrição muito mais clara. Neste capítulo, descreveremos o algoritmo básico e o algoritmo de Ukkonen.

3.1 Conceitos Básicos

Dada uma cadeia S de tamanho n, a *árvore de sufixos* T para a cadeia S é uma árvore com exatamente n folhas, que são numeradas de 1 até n, onde a folha i representa o sufixo que começa na posição i na cadeia S. Cada aresta da árvore de sufixos é rotulada por uma subcadeia da cadeia S. Cada nó interno da árvore de sufixos, com exceção da raiz, deve obrigatoriamente ter dois ou mais filhos, e cada rótulo de uma aresta emergente de um nó deve começar por um caractere diferente. A característica chave da árvore de sufixos é que a concatenação dos rótulos no caminho da raiz até a folha i soletra exatamente o sufixo $S[i..n]$ da cadeia. Com isso, as arestas emergentes de um nó devem ter caracteres diferentes, o que garantirá um único caminho até uma folha para um sufixo, e que um caminho até uma folha exista para um sufixo. Já a obrigação de um nó interno ter dois ou mais filhos é devido ao fato de um nó interno com um filho não ser útil à estrutura da árvore.

Na figura 3.1 é mostrada a árvore de sufixos para a cadeia *xabxac*. Pela figura, além de vermos as características descritas na definição, vemos que o caminho da raiz à folha 1 soletra exatamente a cadeia (o sufixo que começa na posição 1 é a própria cadeia), e que o caminho da raiz até a folha 5 soletra exatamente o sufixo *ac* que começa na posição 5 da cadeia.

A definição de árvore de sufixos não garante a existência da árvore de sufixos para qualquer cadeia S genérica. Pela definição da árvore de sufixos, o problema ocorrerá quando algum sufixo da cadeia S for prefixo de algum outro sufixo, pois neste caso, o caminho que soletra este sufixo não terminaria em uma folha, pois ele seria uma parte do caminho do sufixo do qual ele é prefixo, e terminaria ou em uma aresta, ou num nó interno. Por isso, como foi descrita, a árvore de sufixos impede que um sufixo da cadeia seja um prefixo de outro. Por exemplo, se a cadeia da figura

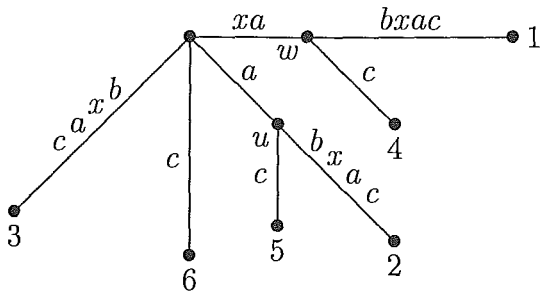


Figura 3.1: Árvore de sufixos para a cadeia *xabxac*.

3.1 fosse *xabxa* ao invés de *xabxac*, o sufixo *xa* seria um prefixo do sufixo *xabxa*, e seu caminho não terminaria em uma folha, pois neste caso, o caminho terminaria no nó interno w.

Mas isso não é realmente problemático; basta assumirmos que o último caractere da cadeia não aparece em nenhuma outra posição desta cadeia, porque se isso ocorrer, nenhum sufixo poderá ser prefixo de um outro sufixo. Portanto, para que possamos construir árvores de sufixos para cadeias genéricas, vamos considerar a adição à cadeia S de um "caractere terminador \$" ao final da cadeia, que não pertence ao alfabeto Σ do qual se originaram os caracteres da cadeia. Com isso, cada cadeia S será estendida com o caractere terminador \$ que impedirá que um sufixo seja prefixo de outro, e daqui em diante vamos supor que todas as cadeias são estendidas com este terminador.

O *rótulo de um caminho* proveniente da raiz de T que termina em um nó v é a concatenação, em ordem, das subcadeias que rotulam as arestas do caminho da raiz até este nó v. O *nome do caminho* de um nó v é o rótulo do caminho da raiz de T até este nó v. Dado um nó v da árvore T, a *profundidade de cadeia* deste nó é o número de caracteres do nome do nó v. Se um caminho termina no meio da aresta (u, v) que liga os nós u e v, ele dividirá o rótulo desta aresta (u, v) no ponto onde este caminho termina dentro da aresta. O rótulo deste caminho será definido como o nome do caminho do nó u concatenado com os caracteres da aresta (u, v) até o ponto definido pela divisão do rótulo. Por exemplo, na figura 3.1, a cadeia *xa* é o nome do nó interno w, a cadeia *a* nomeia o nó u, e a cadeia *xabx* rotula um caminho que termina dentro da aresta $(w, 1)$, isto é, dentro da aresta incidente à folha 1.

Uma *árvore de sufixos implícita* para a cadeia S é obtida através da árvore de

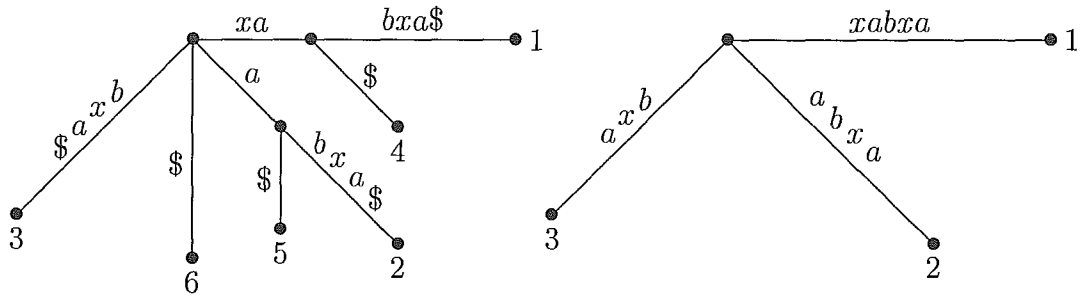


Figura 3.2: Na esquerda temos a verdadeira árvore de sufixos para a cadeia $xabxa\$$, e na direita temos a árvore de sufixos implícita para a mesma cadeia.

sufixos da cadeia $S\$$ pela remoção de todos os símbolos terminais \$ dos rótulos das arestas desta árvore, e após este processamento, de todas as arestas que tenham rótulos nulos juntamente com os nós filhos ligados a estas arestas, e de todos os nós internos que tenham menos do que dois filhos, juntando as duas arestas em uma única aresta após a deleção de um destes nós.

Devido à retirada de todos os símbolos terminais, a árvore de sufixos implícita para a cadeia S irá ter menos nós do que a verdadeira árvore de sufixos para a cadeia $S\$$, se e somente se um dos sufixos desta cadeia S for um prefixo de outro sufixo, pois a adição do símbolo terminal \$ ocorreu exatamente para evitar esta situação, e neste caso, nem todos os sufixos da árvore irão terminar em uma folha. Caso o último caractere da cadeia não ocorra nas outras posições da cadeia, teremos uma folha para cada um dos sufixos, e, neste caso, a árvore de sufixos implícita será uma verdadeira árvore de sufixos.

Mesmo quando a árvore de sufixos implícita para a cadeia S não tiver folhas para todos os sufixos existentes, ela irá codificar todos os sufixos de S. Assim como na verdadeira árvore de sufixos, cada sufixo será soletrado pelos caracteres de algum caminho da raiz da árvore, mas, neste caso, se um caminho não terminar em uma folha, não existirá um marcador para indicar o fim deste caminho. Devido a isso, as árvores de sufixos implícitas são consideradas menos informativas do que as verdadeiras árvores de sufixos.

Como exemplo das noções acima, considere a árvore de sufixos para a cadeia $xabxa\$$ mostrada na parte esquerda da figura 3.2. Como o sufixo xa é um prefixo do sufixo $xabxa$, como o sufixo a é um prefixo de $abxa$, e como há uma folha com nome de caminho igual a \$, as arestas incidentes às folhas 4, 5, e 6 são rotuladas

com o símbolo terminador \$. Então, para obtermos a árvore de sufixos implícita dada na parte direita da figura 3.2, devemos deletar as folhas 4, 5 e 6, as arestas incidentes a estas folhas, e os nós pais das folhas 4 e 5, pois somente os pais destas folhas passarão a ter apenas um filho.

Para a cadeia *xabxac*, cuja árvore de sufixos foi mostrada na figura 3.1 sem os caracteres terminadores, como o caractere c não aparece em nenhuma outra posição da cadeia, a árvore de sufixos da figura para esta cadeia será também a árvore de sufixos implícita da cadeia.

3.2 O Algoritmo Básico

Agora vamos apresentar um algoritmo direto para a construção da árvore de sufixos com complexidade de tempo de $O(n^2)$ para uma cadeia S de tamanho igual a n. Neste algoritmo, para a construção da árvore de sufixos, primeiramente inserimos a aresta para o sufixo $S[1..n]\$$ representando a cadeia $S\$$, na árvore que está vazia. Então, nos passos sucessivos, inserimos os sufixos $S[i..n]\$$ na árvore para i variando de 2 até n. Seja N_i uma designação para a árvore de sufixos com todos os sufixos de 1 até i de S inseridos na árvore.

Usando-se esta designação, N_1 é a árvore com somente uma aresta rotulada por $S\$$ da raiz até a folha rotulada com 1, representando a cadeia S. Agora iremos mostrar como construir a árvore N_{i+1} a partir da árvore N_i. Começando da raiz, procuramos o maior caminho da raiz cujo rótulo seja um prefixo do sufixo $S[i+1..n]\$$. Para descobrimos este caminho, casamos sucessivamente os caracteres de um único caminho iniciado na raiz com os caracteres do sufixo $S[i+1..n]\$$ até não ocorrerem mais casamentos. Após executarmos estes casamentos, o caminho será único devido à propriedade da árvore de sufixo de que nenhum nó pode ter duas ou mais arestas emergentes com o rótulo começando com o mesmo caractere. Agora, se ao encontrarmos um descasamento pararmos no meio de uma aresta (u, v), criamos um novo nó w, e dividimos a aresta (u, v) em duas arestas: (u, w) rotulada com o prefixo do rótulo de (u, v) com os caracteres que casaram, e (w, v) rotulada com o resto do rótulo da aresta (u, v). Caso paremos em um nó, seja w a designação deste nó. Agora, criamos uma folha rotulada com $i+1$, e criamos uma aresta $(w, i+1)$ do nó w para esta folha, rotulada com o restante do sufixo $S[i+1..n]\$$ que não casou ao percorremos o caminho. Após isso, o sufixo $S[i+1..n]\$$ estará inserido na árvore,

pois teremos um único caminho da raiz até a folha nomeado pelo sufixo.

Todos os nós da árvore gerada terão as arestas emergentes com os rótulos iniciados por caracteres diferentes, pois se criamos um novo nó w teremos duas arestas emergindo dele: a aresta (w, v) surgida da divisão da aresta (u, v), e a nova aresta $(w, i+1)$ criada, e estas arestas terão os caracteres iniciais dos seus rótulos diferentes, pois o rótulo da aresta surgida da divisão da aresta (u, v) será iniciado pelo caractere que descasou. No caso de o nó w já existir, paramos devido a um descasamento, e portanto, o caractere inicial do rótulo da nova aresta não pode ser igual ao das arestas já existentes.

Se supusermos que o alfabeto é constante, o tempo requerido para descobrir qual é a aresta a seguir será constante. Agora, como no pior caso percorremos toda a cadeia com exceção do terminador, e como as operações de divisão da aresta e criação dos nós podem ser feitas em tempo constante, a construção da árvore N_{i+1} a partir da N_i tem um tempo total de $O(n)$. Como temos $O(n)$ árvores para computar, o tempo total para computar a árvore de sufixo será de $O(n^2)$.

3.3 O Método de Ukkonen

O algoritmo que será descrito nesta seção, que foi desenvolvido por Esto Ukkonen, é um algoritmo para a construção de árvores de sufixo com tempo de execução linear ao tamanho da cadeia, que é considerado o algoritmo mais fácil de ser compreendido. Este algoritmo usa melhor o espaço do que o algoritmo de Weiner, e tem uma propriedade "on-line" que pode ser útil em algumas situações. Não iremos descrever esta propriedade "on-line", mas vamos enfatizar que uma das principais características do algoritmo de Ukkonen é a facilidade de sua explicação, de sua prova, e de sua análise de tempo. Esta facilidade de explicação vem do fato de que o algoritmo pode ser visto como um método ineficiente para a construção da árvore de sufixos, seguido por alguns truques de implementação que reduzem o trabalho realizado e levam ao limite de tempo alegado, e será desta forma que o algoritmo será descrito nesta seção.

De forma geral, o algoritmo de Ukkonen funciona da seguinte forma: ele irá construir uma seqüência de árvores de sufixos implícitas, onde uma árvore será obtida a partir da anterior, e depois converteremos a última árvore gerada para uma verdadeira árvore de sufixos, ou seja, a nossa ferramenta básica são as árvores

de sufixos implícitas.

Uma árvore de sufixos implícita para um prefixo $S[1..i]$ de uma cadeia S com tamanho igual a n pode ser obtida de forma similar a árvore de sufixos implícita para a cadeia S: construímos a árvore de sufixos para a cadeia $S[1..i]\$$ e deletamos todos os símbolos \$, e todas as arestas e nós desnecessários, como no caso da construção da árvore implícita para S. Considere que esta árvore de sufixos implícita para $S[1..i]$ seja denotada por I_i, $1 \leq i \leq n$.

Uma descrição mais detalhada de alto nível do algoritmo de Ukkonen é a seguinte: o algoritmo de Ukkonen construirá uma árvore de sufixos implícita I_i para cada prefixo $S[1..i]$ de S, começando de I_1 e incrementando i por um até a árvore I_n ter sido construída. A verdadeira árvore de sufixos para a cadeia S será obtida a partir da árvore de sufixos implícita I_n, e o tempo de execução total do algoritmo será de $O(n)$. Seguindo a idéia de apresentar o algoritmo de Ukkonen como um método ineficiente que depois será refinado com alguns truques para atingirmos o tempo de execução alegado de $O(n)$, iremos agora descrever um algoritmo com tempo de execução de $O(n^3)$ que irá construir todas as árvores de sufixos implícitas I_i, $1 \leq i \leq n$, e logo após, as otimizações que irão culminar no algoritmo com tempo de execução de $O(n)$.

3.3.1 O Algoritmo de Alto Nível

Seja S uma cadeia com tamanho igual a n para a qual desejamos construir a árvore de sufixos. O algoritmo é dividido em n fases. Na fase $i+1$, a árvore I_{i+1} será computada a partir da árvore I_i. Para podermos computar a árvore atual, a fase $i+1$ será dividida em $i+1$ extensões, onde cada extensão representa um sufixo de $S[1..i+1]$. Na extensão j, iremos inserir o sufixo $S[j..i+1]$ na árvore, e para isso, primeiramente encontramos o final do sufixo $S[j..i]$ na árvore atual. Depois, verificamos se o próximo caractere pode ser igual a $S(i+1)$. Se este próximo caractere não existir, ou se não puder ser igual a $S(i+1)$, inserimos o caractere $S(i+1)$ na árvore. Com isso, na fase $i+1$, inserimos a subcadeia $S[1..i+1]$ na extensão 1, $S[2..i+1]$ na extensão 2, ..., e na extensão $i+1$ da fase $i+1$ iremos inserir o sufixo vazio de $S[1..i]$, ou seja, iremos colocar a subcadeia composta por um simples caractere $S(i+1)$ na árvore caso ela ainda não esteja presente. A árvore inicial I_1 será uma árvore com somente uma aresta rotulada pelo caractere $S(1)$. A seguir, temos a forma procedural do que foi descrito.

Descrição procedural do Algoritmo

Construir a árvore I_1
Para i **de** 1 **até** $n-1$ **faça**
  Começo { da fase $i+1$ }
    Para j **de** 1 **até** $i+1$ **faça**
      Começo { da extensão j }
        Achar o final do caminho da raiz rotulado pela subcadeia $S[j..i]$ na árvore atual. Se for necessário, estendemos este caminho através da adição do caractere $S(i+1)$, o que deverá garantir que a subcadeia $S[j..i+1]$ está na árvore.
      Fim
  Fim

Na descrição acima não foi dito como descobrimos se o caractere $S(i+1)$ está na árvore, e como proceder caso este caractere não ocorra na árvore. Portanto, devemos especificar exatamente como executar a extensão do sufixo. As regras que definem como esta extensão deve ser realizada são chamadas de *regras de extensão de sufixo*. Seja $S[j..i] = \beta$ uma subcadeia da cadeia S. Na extensão j da fase $i+1$, primeiramente procuramos pelo final de β na árvore de sufixos. Após acharmos o final de β, devemos estender β para garantir que a subcadeia $\beta S(i+1)$ está inserida na árvore. A extensão é feita de acordo com uma das três regras de extensão dadas a seguir:

1. Na árvore atual, o caminho β termina em uma folha, ou seja, o caminho da raiz rotulado pela subcadeia β termina em uma folha. Neste caso, para atualizarmos a árvore, inserimos o caractere $S(i+1)$ ao final do rótulo da aresta incidente a esta folha.

2. Nenhum dos possíveis caminhos ao final do caminho rotulado por β começa por $S(i+1)$. Neste caso, criamos uma nova folha rotulada pelo número j, e uma nova aresta incidente a esta folha com o rótulo $S(i+1)$. Se o caminho β termina em um nó w, a aresta será emergente deste nó. Mas se o caminho β termina dentro de uma aresta, criamos um novo nó w, e dividimos a aresta onde o caminho terminou em duas arestas: uma aresta incidente ao nó w com todos os caracteres da aresta original até o final do caminho, e outra aresta emergindo do nó w com os caracteres restantes da aresta original. Neste caso a aresta incidente à folha criada também será emergente do nó w.

3. Um dos possíveis caminhos começam pelo caractere $S(i+1)$. Neste caso, a subcadeia $\beta S(i+1)$ já está na árvore de sufixo, e como numa árvore de sufixos implícita nem todos os sufixos precisam terminar em uma folha, não executamos nenhuma operação ao aplicar esta regra.

Como exemplo da aplicação das regras de extensão de sufixo acima, considere a árvore de sufixos implícita para a cadeia $axabx$ descrita na parte esquerda da figura 3.3. Como podemos ver pela figura, os quatro primeiros sufixos terminam em uma folha, e o último sufixo termina dentro de uma aresta. Ao inserirmos um sexto caractere b, os quatro primeiros sufixos serão estendidos usando-se a regra de extensão 1, o quinto usando a regra de extensão 2, e o último sufixo (a inserção da cadeia formada somente por b) será inserido usando a regra de extensão 3. A árvore de sufixos implícita resultante após a inserção é dada na parte direira da figura 3.3.

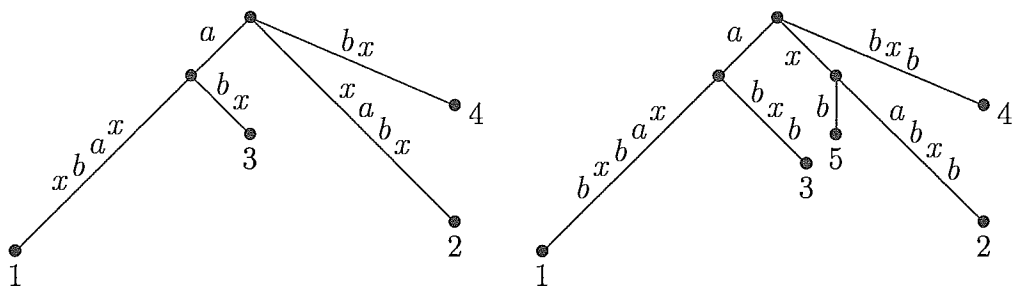


Figura 3.3: Na esquerda temos a árvore de sufixos implícita para a cadeia $axabx$, e na parte direita, temos a árvore resultante após a inserção do caractere b.

Usando as regras de extensão acima, uma vez que tenhamos achado o final do sufixo β da cadeia $S[1..i]$, precisamos somente de um tempo constante para executar as regras de extensão acima, que garantem que o sufixo $\beta S(i+1)$ esteja na árvore.

Então a questão chave do algoritmo de Ukkonen é como acharmos o final de todos os $i+1$ sufixos de $S[1..i]$ de forma eficiente. Uma forma simples de achar o final de qualquer sufixo β num tempo de $O(|\beta|)$ é através do caminhamento da raiz da árvore usando o sufixo β como um guia no caminhamento. Neste caso, a extensão j da fase $i+1$ será executada num tempo de $O(i-j+1)$, e portanto, a árvore I_{i+1} pode ser calculada a partir de I_i num tempo de $O(i^2)$, e com isso, I_n será calculada com um tempo total de $O(n^3)$.

Agora, para reduzirmos o tempo de execução do algoritmo de Ukkonen de $O(n^3)$ para $O(n)$, iremos aplicar alguns truques de implementação que são heurísticas para

reduzir o tempo de execução do algoritmo, mas que não irão necessariamente reduzir o tempo de execução se não forem aplicadas conjuntamente. A mais importante delas é proveniente do uso de conexões de sufixo.

3.3.2 O Uso de Conexões de Sufixo

Seja $x\alpha$ uma cadeia arbitrária, onde x denota um único caractere e α denota uma subcadeia que pode ser vazia. Para um dado nó interno v com nome de caminho igual a $x\alpha$, se há outro nó $s(v)$ com um nome de caminho igual a α, então um ponteiro de v para $s(v)$ é chamado de *conexão de sufixo*. Se a cadeia α for vazia, então a conexão de sufixo de um nó interno com nome de caminho igual a $x\alpha$ irá terminar na raiz da árvore. Apesar de a definição de conexão de sufixo não implicar que todos os nós internos têm uma conexão de sufixo, todos estes nós podem ter uma conexão de sufixo, o que será provado no colorário 3.1 enunciado abaixo. Somente a raiz, que não é considerada um nó interno, não poderá ter uma conexão de sufixo emergente dela.

Em alguns casos, inclusive durante as explicações neste capítulo do algoritmo de Ukkonen, uma conexão de sufixo do nó v para o nó $s(v)$ pode ser representada pelo par $(v, s(v))$. Por exemplo, na parte esquerda da figura 3.2, seja v o nó com nome de caminho xa, e seja $s(v)$ um outro nó com nome de caminho igual ao caractere a. Neste caso, há uma conexão de sufixo do nó v para o nó $s(v)$, e α é uma cadeia com um único caractere igual a a.

Lema 3.1 *Se um nó interno v com um nome de caminho $x\alpha$ for adicionado na extensão j da fase $i+1$, então ou já existe um nó interno $s(v)$ com nome de caminho igual a α, ou este nó será criado, através da aplicação das regras de extensão, na extensão $j+1$ da mesma fase $i+1$. [17, Cap. 6]*

Prova Um novo nó v será criado somente quando a regra de extensão 2 for aplicada. Isso significa que na extensão j o caminho rotulado por $x\alpha$ continua com algum caractere diferente de $S(i+1)$, digamos, o caractere y. Após executarmos a extensão $j+1$, existirá um caminho rotulado por α na árvore atual, e com certeza ele continuará no mínimo com o caractere y. Se o caminho continuar com outros caracteres adicionais além do caractere y, então α já terminava no nó interno $s(v)$, e caso contrário, o nó $s(v)$ deve ter sido criado pela aplicação da regra de extensão 2. Portanto, se um nó v com nome de caminho $x\alpha$ for criado na extensão j, um nó $s(v)$ com nome de caminho α existirá ao final da extensão $j+1$. □

Colorário 3.1 *No algoritmo de Ukkonen, qualquer nó criado em uma extensão terá uma conexão de sufixo emergente dele até o final da próxima extensão. [17, Cap. 6]*

Prova A prova é baseada em indução e será verdadeira para a árvore I_1, pois esta árvore não tem nós internos. Vamos supor então que a hipótese é verdadeira para a árvore I_i criada ao final da fase i, ou seja, que ela contém todas as conexões de sufixo. Pelo lema 3.1, quando um nó v for criado na extensão j da fase $i+1$, o correto nó $s(v)$ será criado até o final da extensão $j+1$ da mesma fase. Como a regra de extensão 2 não é usada para inserir o sufixo composto pelo simples caractere $S(i+1)$ na árvore, pois neste caso somente as regras 1 ou 3 são aplicadas, então ao final da fase $i+1$ todos os nós internos criados terão conexões de sufixos emergindo deles, e com isso, a árvore de sufixos implícita I_{i+1} também terá todas as conexões de sufixo. □

O colorário 3.1 declara uma característica importante das árvores de sufixos implícitas, e em última análise, das árvores de sufixos. Esta característica é descrita no colorário 3.2 que redefine o colorário 3.1 para enfatizar esta característica.

Colorário 3.2 *Dada qualquer árvore de sufixos implícita I_i, se há um nó interno v com nome de caminho igual a $x\alpha$, então há um outro nó interno $s(v)$ com nome de caminho α na mesma árvore.*

Pelo colorário 3.1, vemos que todos os nós internos da árvore em alteração pelo algoritmo terão conexões de sufixo emergindo deles, com exceção do nó mais recentimente criado, que terá sua conexão de sufixo definida até o final da próxima extensão. Agora vamos mostrar como usamos as conexões de sufixo para acelerar o algoritmo.

Para vermos a aplicação das conexões de sufixo no algoritmo, vamos lembrar que na extensão j da fase $i+1$, primeiramente o algoritmo localiza o sufixo $S[j..i]$ de $S[1..i]$. Como já dissemos, isso pode ser conseguido de forma não ótima através do casamento da cadeia $S[j..i]$ ao longo de um caminho que começa na raiz da árvore. Vamos usar as conexões de sufixo para encurtar o caminhamento até a posição final da cadeia $S[j..i]$, reduzindo o tempo de execução de uma extensão. Para tornar a explicação mais clara, primeiramente iremos mostrar como usar as conexões de

sufixo para as extensões 1 e 2, e depois mostraremos como usar as conexões de sufixo numa extensão j genérica.

Na extensão 1, procuramos pelo final da cadeia $S[1..i]$, que deve terminar em uma folha, pois este será o maior sufixo representado pela árvore de sufixos implícita I_i atual, e portanto, aplicaremos a regra de extensão 1 ao estendê-lo. Devido à forma de construção das árvores, todos os prefixos de S em todas as fases irão terminar nesta mesma folha, e iremos sempre aplicar a regra de extensão 1 neste caso. Portanto, se mantivermos um ponteiro para a folha que representa o maior sufixo da fase atual, a extensão 1 da qualquer fase será um caso particular e poderá sempre ser executada num tempo constante.

Seja $x\alpha = S[1..i]$, onde x representa um caractere, e α representa uma cadeia que pode ser vazia, e seja $(v, 1)$ a aresta incidente à folha 1 que representa o maior sufixo $S[1..i]$ da última árvore I_i já computada. Na extensão 2 da fase $i+1$, o algoritmo deve agora encontrar o final da cadeia $S[2..i] = \alpha$ na árvore I_i. No método simples, simplesmente procuramos o final do caminho da raiz com um rótulo que casa com α, mas podemos otimizar isso usando as conexões de sufixo. Se o nó v for a raiz, usamos o método simples, mas se v não é a raiz (v é um nó interno), pelo colorário 3.2 v terá uma conexão de sufixo para um nó $s(v)$, e como $s(v)$ terá um nome de caminho que é um prefixo da cadeia α, então α deve terminar na subárvore definida por este nó, e poderemos simplesmente começar o caminhamento pelo nó $s(v)$. Mais especificamente, se γ denotar o rótulo da aresta $(v, 1)$, então podemos usar o seguinte processo para obter o final da cadeia $S[2..i]$ na extensão 2: caminhamos da folha 1 para o nó v; se o nó v não for a raiz, seguimos a conexão de sufixo do nó v para o nó $s(v)$; e por último, caminhamos para baixo, a partir do nó atual, o caminho rotulado por γ na subárvore definida por este nó. Após este processo, teremos achado o final do caminho rotulado pela cadeia α, como pode ser visto pela figura 3.4, e portanto, basta que agora apliquemos as regras de extensão de sufixo para garantir que o sufixo $S[2..i+1]$ esteja na árvore I_{i+1}.

Para estendermos qualquer cadeia $S[j..i]$ para $S[j..i+1]$ numa extensão $j > 2$ da fase $i+1$, usamos a mesma idéia da extensão 2, com uma pequena diferença que será explicada. Dado o final do caminho rotulado pela cadeia $S[j-1..i]$ na árvore I_i, que foi obtido na extensão $j-1$, se estivermos em um nó que tem uma conexão de sufixo, simplesmente seguimos a conexão de sufixo deste nó que nos levará ao final do caminho rotulado pela cadeia $S[j..i]$. Caso este nó não tenha uma conexão de

sufixo, ou tenhamos parado dentro de uma aresta, caminhamos para cima na árvore I_i até o primeiro nó v, e seja γ a cadeia que rotula o caminho deste nó v até o final da cadeia $S[j-1..i]$. Se o nó v atual não for a raiz, seguimos a conexão de sufixo para o nó $s(v)$. Após isso, caminhamos para baixo na subárvore definida pelo nó atual o caminho rotulado por γ, o que nos levará ao final do caminho rotulado pela cadeia $S[j..i]$. Não precisaremos caminhar para cima por mais de um nó, pois se este não for a raiz, ele com certeza terá uma conexão de sufixo, mesmo se o final da cadeia $S[j-1..i]$ terminar em um nó que foi recentemente criado (usamos a regra de extensão 2 na extensão $j-1$), pois pelo colorário 3.1 o seu pai já deverá ter uma conexão de sufixo. Após determinado o final do caminho da cadeia $S[j..i]$ na árvore atual, simplesmente aplicamos as regras de extensão para estendermos este sufixo e terminar a execução da extensão j desta fase.

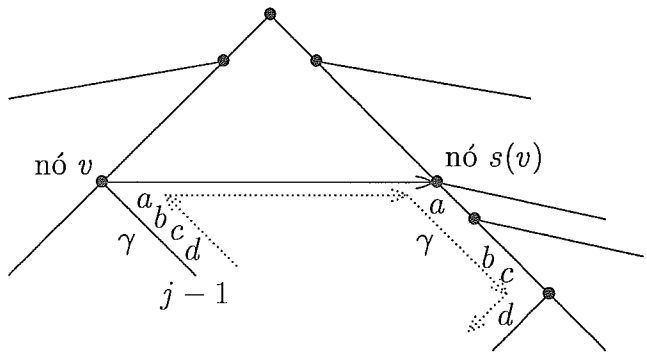


Figura 3.4: Extensão $j > 1$ da fase $i+1$. Caminhamos para cima por no máximo uma aresta (rotulada por γ) a partir do final do caminho rotulado pela cadeia $S[j-1..i]$ para o nó v, seguimos a conexão de sufixo para o nó $s(v)$, e então caminhamos para baixo pelo caminho rotulado pela subcadeia γ. Agora que achamos o final da cadeia $S[j..i]$, aplicamos a regra de extensão apropriada para inserir a cadeia $S[j..i+1]$.

Como vimos, a primeira extensão de qualquer fase pode ser feita num tempo constante sem nenhum caminhamento na árvore, se mantivermos um ponteiro para o maior sufixo, e neste caso sempre aplicaremos a regra de extensão 1. Então, quando usamos as conexões de sufixo, temos o seguinte algoritmo descrito a seguir para executar uma extensão $j \geq 2$ numa fase $i+1$:

Algoritmo para executar uma extensão j

Começo

1. Achar o primeiro nó v no final ou acima do final do caminho rotulado por $S[j-1..i]$ que tenha uma conexão de sufixo emergindo dele. Este caminhamento para cima irá requerer, como vimos, o caminhamento por no máximo uma aresta na árvore atual. Seja γ a cadeia, que pode ser vazia, entre este nó v e o final deste caminho.

2. Se o nó v não for a raiz, seguimos a conexão de sufixo do nó v para o nó $s(v)$. Se seguimos a conexão de sufixo, seguimos (para baixo) o caminho rotulado por γ a partir da subárvore definida por $s(v)$. Se v é a raiz, seguimos o caminho rotulado por $S[j..i]$ a partir da raiz, como no método simples.

3. Usando as regras de extensão de sufixo, estendemos o sufixo $S[j..i]$ para garantir que $S[j..i+1]$ está na árvore.

4. Se criamos um novo nó w com nome de caminho $x\alpha = S[j-1..i]$ ao final da extensão anterior $j-1$, então, pelo lema 3.1, a cadeia $\alpha = S[j..i]$ deve terminar em um nó $s(w)$ após a execução do passo 3. Então, neste caso, devemos criar uma conexão de sufixo $(w, s(w))$.

Fim

O uso das conexões de sufixo nos caminhamentos das extensões é claramente um melhoramento sobre o caminhamento a partir da raiz executado pelo algoritmo simples, mas o uso das conexões de sufixo não irá reduzir o tempo de pior caso do algoritmo de $O(n^3)$. Na próxima subseção iremos mostrar o primeiro truque de implementação que irá permitir, juntamente com as conexões de sufixo, a redução do tempo de pior caso de $O(n^3)$ para $O(n^2)$.

3.3.3 O Truque de Salto/Soma

No passo 2 do algoritmo descrito na subseção anterior, na extensão j, o algoritmo irá caminhar para baixo na subárvore definida ou pelo nó $s(v)$, ou pela raiz por um caminho rotulado por γ, que deve existir na árvore. De forma simples, este caminhamento pode ser feito num tempo de $O(|\gamma|)$,[1] o número de caracteres do rótulo deste caminho. O truque descrito a seguir, chamado de *truque de salto/soma*, irá não só permitir que o tempo de execução seja proporcional ao número de nós deste caminho, como irá reduzir, o que será provado no teorema 3.1, o tempo de

[1] Se pudermos obter a próxima aresta num tempo constante, ou seja, se o alfabeto for constante.

execução de uma fase de $O(n^2)$ para $O(n)$, reduzindo o tempo de pior caso de $O(n^3)$ para $O(n^2)$.

O truque de salto/soma funciona da seguinte maneira: seja g o tamanho da cadeia γ que rotula o caminho. Como todos os rótulos das arestas que emergem do nó atual (v ou $s(v)$) devem começar por caracteres diferentes, e como deve existir o caminho γ a partir deste nó, então o primeiro caractere de γ deve aparecer no rótulo de alguma das arestas emergentes deste nó. Seja g' o número de caracteres do rótulo desta aresta. Se $g \leq g'$, então simplesmente saltamos para o caractere da posição g nesta aresta, ou para o nó que a aresta incide se $g = g'$, e terminamos, pois já atingimos o final do caminho. Agora se $g > g'$, seja $h = g' + 1$. Então simplesmente saltamos para o nó destino da aresta atual, e verificamos qual a próxima aresta a seguir usando o caractere dado pela posição h em γ, e fazemos $g = g - g'$. Agora se $g \leq g'$, onde agora g' é o número de caracteres da nova aresta atual, novamente saltamos para a posição g na aresta ou para o nó destino desta aresta se $g = g'$, e o processo irá parar, pois já processamos toda a cadeia γ. Se $g > g'$, fazemos $h = h + g'$, $g = g - g'$, e repetimos o processo descrito, saltando para o próximo nó, procurando pela próxima aresta que começa pelo caractere na posição h da cadeia γ, e alterando g' para o número de caracteres desta aresta. Na figura 3.5 é ilustrado como este processo funciona.

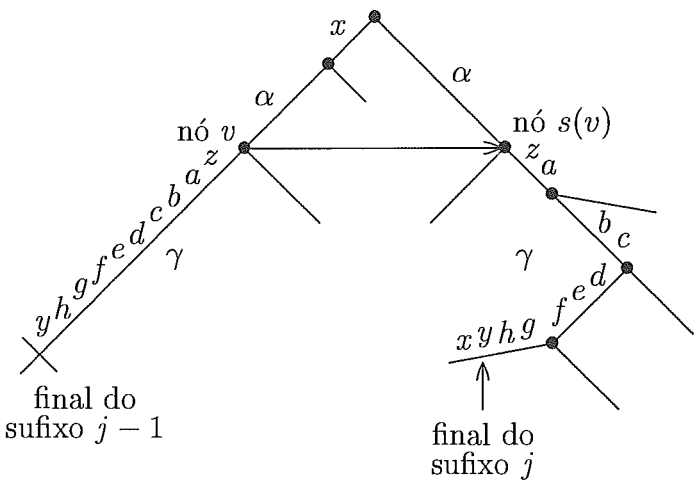


Figura 3.5: O truque de salto/soma. Na fase $i+1$ representada na figura, a subcadeia γ tem tamanho igual a 10. Há uma cópia da subcadeia γ saindo do nó $s(v)$; o seu final é achado três caracteres abaixo da última aresta, após quatro saltos de nó terem sido executados.

Se supusermos que as operações de extração de caracteres da cadeia e de soma e subtração[2] podem ser feitas num tempo constante, o truque de salto/soma irá reduzir o tempo de caminhamento para um tempo proporcional ao número total de nós deste caminho, pois a movimentação entre os nós (através das arestas) do caminho será agora feito num tempo constante. O próximo teorema irá provar que de fato este truque reduz o tempo de execução de uma fase de $O(n^2)$ para $O(n)$. Para podermos demonstrar as provas, vamos definir a *profundidade* de um nó u como sendo o número de nós no caminho da raiz até este nó u, e a *profundidade do nó atual* como sendo a profundidade do último nó visitado pelo algoritmo.

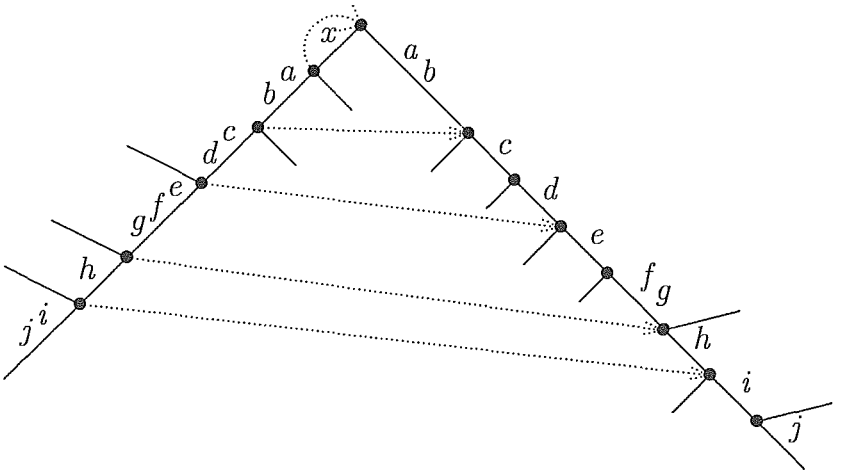


Figura 3.6: Nesta figura, para cada nó v sobre o caminho $x\alpha$ é dado a sua conexão de sufixo $s(v)$ sobre o caminho α. Como podemos ver neste exemplo, a profundidade de um nó $s(v)$ pode ser uma unidade menor do que a do nó v, pode ser igual, ou pode ser maior. Na figura, o nó rotulado por xab tem profundidade de nó igual a 3, e o nó ab tem profundidade de nó igual a 2. Já o nó com nome de caminho $xabcdefg$ tem profundidade de nó igual a 5, enquanto que a profundidade do nó de $abcdefg$ é igual a 6.

Lema 3.2 *Seja $(v, s(v))$ uma conexão de sufixo atravessada durante o algoritmo de Ukkonen. Em um dado instante, a profundidade do nó v é no máximo uma unidade maior do que a profundidade do nó $s(v)$. [17, Cap. 6]*

Prova Quando atravessamos a conexão de sufixo $(v, s(v))$, qualquer ancestral interno de v com nome de caminho $x\beta$ terá uma conexão de sufixo para um nó que tem um nome de caminho igual a β, e como $x\beta$ é um prefixo do nome do caminho

[2]Se estivermos em um modelo de computação RAM.

do nó v, e β é o prefixo do nome do caminho do nó $s(v)$, que será interno se β não for vazia, segue que uma conexão de qualquer ancestral de v vai para um ancestral de $s(v)$. Agora, cada um dos ancestrais de v deve ter uma conexão de sufixo para um ancestral distinto de $s(v)$, porque dois ancestrais de v não podem ter a mesma profundidade de nó. Com isso, a profundidade de $s(v)$ é no mínimo igual a um (a raiz) mais o número de ancestrais de v com nomes de caminho com dois ou mais caracteres. O único ancestral que v pode ter a mais é um nó nomeado por um caractere x cuja conexão de sufixo levará à raiz da árvore. Portanto, se há uma conexão de sufixo de v para $s(v)$, a profundidade de v pode ser maior por no máximo uma unidade do que a profundidade de $s(v)$. Na figura 3.6 vemos um exemplo mostrando a propriedade que acabamos de provar. □

Teorema 3.1 *Se for usado o truque de salto/soma descrito anteriormente, juntamente com as conexões de sufixo, qualquer fase do algoritmo de Ukkonen será executada num tempo de $O(n)$. [17, Cap. 6]*

Prova Existem $i + 1 \leq n$ extensões no algoritmo de Ukkonen numa fase $i + 1$. Em cada extensão, subimos através somente de uma aresta para um nó, seguimos a conexão de sufixo deste nó se o nó não for a raiz, descemos por um certo número de nós, aplicamos as regras de extensão, e se necessário criamos uma nova conexão de sufixo. Como vimos que todas as operações, com exceção do caminhamento para baixo, podem ser executadas num tempo constante, iremos agora analisar como a profundidade do nó atual varia entre as extensões. O caminhamento para cima em qualquer extensão decrementa a profundidade do nó atual em uma unidade, e a travessia da conexão de sufixo decrementa a profundidade também por no máximo uma unidade devido ao lema 3.2. Com isso, durante toda a fase atual, a profundidade do nó atual irá ser decrementada no máximo por $2n$. Como nenhum dos nós pode ter uma profundidade maior do que $O(n)$ (a árvore de sufixo tem $O(n)$ nós), e como cada aresta atravessada num caminhamento para baixo irá incrementar a profundidade do nó atual em uma unidade, o número de incrementos será então limitado por $3n$ durante toda a fase. Com isso, o número de arestas atravessadas também será limitado por $3n$ na fase atual. Se usarmos o truque de salto/soma, o tempo para atravessar uma aresta particular será constante, e com isso, o tempo total de todos os caminhamentos para baixo, e portanto, da fase, será de $O(n)$. □

Como temos n fases no algoritmo de Ukkonen, o seguinte colorário é imediato:

Colorário 3.3 *O algoritmo de Ukkonen pode rodar num tempo de $O(n^2)$ se for implementado com conexões de sufixo e o truque de salto/soma.*

Devido ao teorema 3.1 somente descrever a complexidade de tempo para uma fase, tivemos que fazer uma análise imperfeita da complexidade de tempo no colorário 3.3 multiplicando o tempo máximo de execução de uma fase pelo número de fases. Mas até agora, a descrição nos deu uma boa base conceitual, e somente com uma otimização de espaço e mais dois truques, estaremos aptos a descrever um algoritmo com tempo de execução de $O(n)$, e para fazer uma análise de tempo que cruza o limite de uma fase. Isso será feito nas três próximas subseções.

3.3.4 Uso dos Rótulos de Aresta Comprimidos

Agora só nos resta descrever dois truques a mais que irão nos permitir executar $O(n)$ extensões no pior caso, onde iremos gastar um tempo total acumulado de $O(n)$ para executá-las. Entretanto, com a estrutura atual da árvore, temos um impedimento para realizar esta redução de tempo: no pior caso, a árvore de sufixos pode requerer um espaço de $O(n^2)$, pois o número de caracteres totais das arestas pode ser maior do que $O(n)$, e como sabemos, o tempo de execução de um algoritmo não pode ser menor do que a saída que este algoritmo gera. Como um exemplo deste efeito, considere a cadeia $S = abcdefghijklmnopqrstuvwxyz$ com 26 caracteres. Nesta cadeia, cada sufixo irá começar por um caractere diferente, e com isso, teremos 26 arestas emergentes da raiz, cada uma rotulada com todo o sufixo que esta representará. Com isso, precisaremos de $26 \times 27/2$ caracteres. Se tivéssemos uma cadeia S com tamanho de n toda diferente, o espaço requerido seria de $n(n+1)/2 = O(n^2)$ caracteres. Mesmo quando as cadeias têm um tamanho maior do que do alfabeto de onde elas se originaram, ainda podemos gastar um espaço total maior do que $O(n)$, pois as cadeias são genéricas.

Portanto, para podermos criar um algoritmo com tempo de execução de $O(n)$, devemos ter um esquema alternativo de armazenamento dos rótulos das arestas que gaste um espaço constante na representação dos rótulos, pois o número de nós (e portanto, de arestas) em uma árvore de sufixos é de $O(n)$.

Num esquema alternativo e simples, chamado de *compressão dos rótulos das arestas*, ao invés de escrevermos a subcadeia $S[p..q]$ que rotula a aresta explicitamente, escrevemos um par de índices (p, q) sobre esta aresta, que especificam o começo e o fim daquela subcadeia na cadeia S. Como o algoritmo tem uma cópia da cadeia

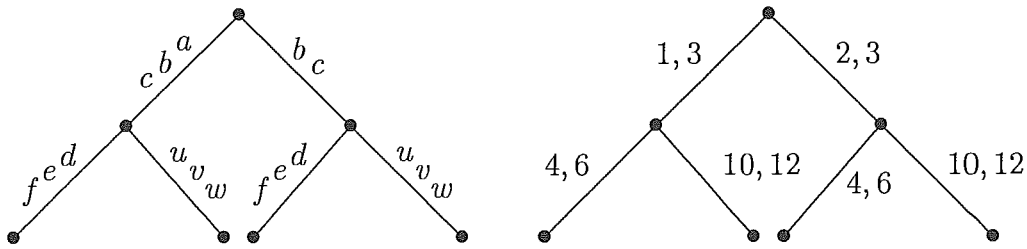


Figura 3.7: A árvore da esquerda é um fragmento da árvore de sufixos para a cadeia *abcdefabcuvw*, com os rótulos das arestas escritos explicitamente. Já na árvore da direita, os rótulos das arestas estão comprimidos. Note que as arestas rotuladas pelo par de índices $(2, 3)$ poderiam também ser rotuladas pelo rótulo $(8, 9)$.

S e como cada caractere da cadeia pode ser localizado e recuperado num tempo constante[3] se dermos a sua posição, qualquer algoritmo para árvores de sufixos que represente o rótulo explicitamente pode também representar o rótulo na forma compacta sem nenhum acréscimo no tempo de execução, com a vantagem de que neste último caso precisaremos de um espaço constante (para armazenar os dois índices) ao representarmos um rótulo de uma aresta. Para um exemplo concreto, veja a figura 3.7 onde mostramos um fragmento de uma árvore de sufixos com os rótulos das arestas não compactados, e o mesmo fragmento com os rótulos comprimidos.

Para compreendermos melhor que esta compactação não influencia o nosso algoritmo, vamos explicar como duas das fases anteriores devem ser implementadas: ao usarmos o truque de salto/soma, agora $g' = f - i + 1$, onde (i, f) é o par de índices da aresta. E agora, ao usarmos as regras de extensão de sufixo, devemos proceder da seguinte maneira: ao aplicarmos a regra 1, em que acrescentamos um caractere $S(i + 1)$ em uma aresta incidente à uma folha com rótulo igual a (p, q), basta que incrementemos o último índice, o que irá gerar o rótulo $(p, q + 1)$ para esta aresta, e neste caso é fácil ver que q deve ser igual a i. E ao aplicarmos a regra 2 inserindo um caractere $S(i + 1)$, e dividindo a aresta (p, q) num ponto e, onde a cadeia $S[j..i]$ terminou, numa extensão j, nomeamos a aresta incidente à nova folha j com o rótulo $(i + 1, i + 1)$, o que deverá ser feito também se o caminho terminou em um nó, nomeamos a aresta incidente ao novo nó com (p, e), e a outra emergente do novo nó com $(e + 1, q)$.

Com este esquema, devemos escrever somente um par de índices (dois números) como rótulo de uma aresta, e como o número de nós de uma árvore de sufixos é de

[3]Novamente assumindo que estamos num modelo de computação RAM.

no máximo $2n = O(n)$, agora a árvore de sufixos irá usar somente $O(n)$ símbolos, e portanto, irá requerer um espaço total de $O(n)$. Com isso, agora poderemos construir um algoritmo com tempo total de execução de pior caso de $O(n)$.

3.3.5 Dois Truques de Otimização

Vamos agora mostrar mais dois truques de implementação baseados em duas observações sobre como as extensões interagem numa mesma fase e entre fases distintas. Com mais estes dois truques, iremos mostrar, na próxima subseção, como construir o algoritmo com tempo de execução de pior caso de $O(n)$.

A primeira observação que pode ser notada é a de que a extensão em que a regra 3 é aplicada pode ser a última extensão executada nesta fase. Isso ocorre porque quando a regra 3 for aplicada a uma extensão j da fase $i+1$ (indicando que $S(i+1)$ já está após o final do caminho de $S[j..i]$) ela será aplicada também nas extensões de $j+1$ até $i+1$, pois este caractere $S(i+1)$ também deve estar após o final do caminho destes outros sufixos existentes na árvore I_i, que serão estendidos pela aplicação das regras de extensão, devido à estrutura da árvore. Além disso, como somente devemos criar uma conexão de sufixo um passo além do passo onde aplicamos a regra de extensão 2 (onde um nó interno é criado), não é necessário nenhum trabalho ao executarmos as extensões de $j+1$ até $i+1$.

Com base na observação acima, podemos definir o *primeiro truque de implementação*: terminar a fase $i+1$ sempre quando a regra de extensão 3 for aplicada em uma extensão j, e neste caso, as extensões dos sufixos $S[k..i+1]$, $j < k \leq i+1$ serão feitas implicitamente. Se não aplicarmos a regra 3 na fase $i+1$, terminamos normalmente esta fase na última extensão. Este truque com certeza é uma boa heurística para redução do trabalho realizado, mas ainda não será suficiente para reduzir o tempo de pior caso.

Devido a algumas extensões não serem, a partir de agora, de fato executadas, chamaremos de *extensões explicítas* aquelas que foram explicitamente computadas pelo algoritmo, e de *extensões implícitas* aquelas que não foram executadas pelo algoritmo, ou seja, que foram implicitamente calculadas, como as extensões de $j+1$ até $i+1$ após a execução da regra 3 numa extensão j, se usarmos o truque descrito acima.

Agora, iremos descrever outra observação notada sobre a interação entre as extensões. Após esta observação, e do truque baseado nela, seremos capazes de des-

crever o algoritmo de Ukkonen com tempo de execução linear.

Observando a forma como as extensões são executadas podemos notar o seguinte: uma vez que uma folha rotulada por j, representando o sufixo que começa nesta posição, for criada em uma extensão j (pela regra de extensão 2), ela será sempre uma folha em todas as árvores sucessivamente criadas pelo algoritmo. Isso é verdade pois nenhuma das regras de extensão permitem que uma aresta incidente à uma folha seja estendida além desta folha, o que transformaria esta folha em um nó interno. Com isso, após a folha ter sido criada em uma extensão j, iremos sempre aplicar a regra de extensão 1 a esta mesma extensão j em todas as outras fases do algoritmo.

A folha 1 representando a cadeia S (no final do algoritmo) será criada na extensão 1 da fase 1. Com isso, em qualquer fase i do algoritmo, há uma seqüência de extensões consecutivas (começando pela extensão 1 para a folha 1) onde iremos aplicar a regra 1 ou 2. Seja j_i a última extensão desta seqüência. Como a regra 2 sempre cria uma nova folha, e como pela observação anterior, nas fases sucessivas sempre iremos aplicar a regra 1 à extensão que criou esta folha, devemos ter que $j_i \leq j_{i+1}$, significando que a seqüência de extensões não pode encurtar de uma fase para a outra. Além disso, em todas as extensões de 1 até j_i na fase $i+1$, a regra de extensão 1 deve ser aplicada, pois ou esta regra já era aplicada anteriormente em uma extensão, ou passará a ser aplicada agora, após a criação da folha pela regra 2 nesta extensão na fase anterior. Baseado nestas observações, o truque de implementação descrito a seguir irá computar todas as extensões de 1 até j_i de forma implícita na fase $i+1$.

Para uma melhor compreensão deste truque lembremos que qualquer aresta rotulada pela subcadeia $S[p..q]$ em uma árvore de sufixos (implícita) pode ter o seu rótulo compactado com o par de índices (p, q). E lembremos também que para qualquer aresta incidente à uma folha da árvore I_i criada ao final da fase i, o índice q deve ser igual a i e será incrementado (pela regra 1) para $i+1$ na fase $i+1$.

Com isso, podemos descrever o *segundo truque de implementação*: na fase $i+1$, quando criarmos pela primeira vez uma folha, ao invés de rotularmos a aresta incidente a esta folha com o par de índices $(i+1, i+1)$ (representando a adição do caractere $S(i+1)$ pela regra 2) rotulamos esta folha com o par de índices $(i+1, e)$ onde e será um índice global do algoritmo denotando a fase atual que está sendo executada. Agora, como na fase $i+1$, o algoritmo sabe que a regra de extensão 1 será aplicada a todas as extensões de 1 até j_i, se usarmos este símbolo global e ao

rotularmos a aresta incidente à folha quando esta foi criada, poderemos executar todas estas extensões de forma implícita simplesmente incrementando o índice global e a cada fase, de tal forma que se estivermos na fase $i+1$, $e = i+1$. Com este truque, somente as extensões após a extensão j_i devem ser executadas de forma explícita.

3.3.6 O Algoritmo Linear de Ukkonen

Através do uso dos truques descritos na subseção anterior, na fase $i+1$ somente precisaremos executar explicitamente as extensões de $j_i + 1$ até $j_{i+1} + 1$, onde j_i é a última extensão usando a regra 1 ou 2 na fase i, e j_{i+1} é a última extensão usando as mesmas regras na fase $i+1$ (as extensões antes da regra 3 ser aplicada). As outras extensões serão calculadas de forma implícita. Com isso, podemos definir o seguinte algoritmo para a execução de uma fase:

Algoritmo para uma fase i + 1

Começo

1. Incrementar o índice e para $i+1$, o que nos permitirá executar as extensões de 1 até j_i de forma implícita através do uso do segundo truque.
2. Computamos explicitamente as extensões de $j_i + 1$ até a extensão j^*, que será a primeira extensão onde a regra de extensão 3 é executada, e que deverá ser inicializada com $i+2$ se não aplicarmos a regra 3 até o final da fase $i+1$. Usamos o algoritmo dado na subseção do truque de salto/soma para executar cada uma destas extensões. Como vimos anteriormente, se usarmos o primeiro truque, isso irá calcular de forma implícita as extensões de $j^* + 1$ até $i+1$, se executarmos a regra de extensão 3.
3. Fazer $j_{i+1} = j^* - 1$ para podermos começar na extensão correta na próxima fase.

Fim

A característica chave do algoritmo acima é que a fase $i+2$ irá começar a computar suas extensões explícitas a partir da extensão j^*, que é a última extensão explícita executada na fase $i+1$ anterior, se executamos a regra de extensão 3, e que será igual a $i+2$ caso não executemos a regra 3. Com isso, duas fases consecutivas compartilham no máximo este índice j^* onde executaremos uma extensão explícita, como pode ser visto pela figura 3.8 em que temos 4 fases, e onde as extensões explicitamente executadas estão indicadas. Além disso, se executarmos a regra 3 na fase anterior, a fase $i+1$ termina sabendo a posição final da cadeia $S[j^*..i+1]$, que foi

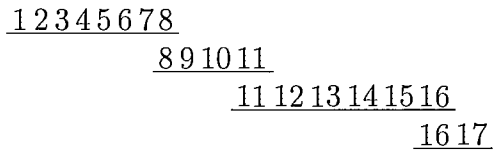


Figura 3.8: Desenho de uma possível execução do algoritmo de Ukkonen. Na figura, cada linha representa uma fase, e um número representa uma extensão explícita. Neste caso particular de execução, temos 4 fases e 17 extensões explícitas, sendo que as três que estão nos limites que delimitam as fases são compartilhadas. Com isso, somente uma extensão é compartilhada entre duas fases diferentes.

computada após a execução da última extensão explícita j^*, e com isso, esta extensão j^* na fase $i+2$ pode ser estendida num tempo constante, sem caminhamentos ou travessia de uma conexão de sufixo. Isso ocorrerá também se a regra 3 não for executada na fase $i+1$, pois neste caso, se for necessário, criaremos uma aresta emergente da raiz rotulada pelo caractere $S(i+2)$. Para implementarmos o algoritmo de Ukkonen, basta executarmos o algoritmo dado acima para as fases de 1 até n, onde n é o tamanho da cadeia S.

Teorema 3.2 *Usando as conexões de sufixo, o truque de salto/soma, e os dois truques dados na subseção anterior, o algoritmo de Ukkonen irá construir as árvores de sufixos implícitas de I_1 até I_n num tempo total de $O(n)$. [17, Cap. 6]*

Prova O tempo para computar todas as extensões implícitas será constante e portanto, igual a $O(n)$ para todo o algoritmo. Agora, quando o algoritmo está executando as extensões explícitas, seja $\bar{\jmath}$ o índice da extensão explícita atualmente executada. Como $\bar{\jmath}$ não é decrementado entre duas fases consecutivas, como $\bar{\jmath}$ é limitado por n, e como duas fases compartilham no máximo uma extensão, então serão executadas no máximo $2n$ extensões explícitas. Como usamos o algoritmo dado na subseção que apresenta o truque de salto/soma, sabemos que o tempo de execução de uma extensão é uma constante mais um tempo proporcional ao número de saltos executados durante um caminhamento para baixo na árvore ao executarmos a extensão. Para descobrirmos o tempo total de execução destes caminhamentos em todas as fases, vamos considerar a variação da profundidade do nó atual (como foi feito na prova do algoritmo não ótimo dada pelo teorema 3.1), só que agora iremos considerar também como esta profundidade varia entre as extensões de duas fases diferentes. Um detalhe bem importante é o de que a profundidade do nó atual não

irá variar do final de uma fase, após executarmos a última extensão explícita, até o início da próxima fase, pois a primeira extensão desta fase é executada sem caminhamentos e sem a travessia de uma conexão de sufixo. Agora, como no teorema 3.1 descrito anteriormente, em uma extensão, a profundidade do nó atual é decrementada no máximo por 2, e incrementada pelo número de nós no caminhamento para baixo. Como temos no máximo $2n$ extensões, e como a profundidade do nó atual não varia de uma fase para outra, a profundidade deste nó será decrementada por no máximo $2n$ em todo o algoritmo. Agora, como a profundidade de um nó é de no máximo n, e como é incrementada a cada aresta atravessada, o número de incrementos da profundidade, e portanto, o número de arestas atravessadas, será de no máximo $3n$ para todo o algoritmo. Como já contabilizamos todo o trabalho realizado, o tempo de execução total será portanto de $O(n)$. □

Agora, para construir a verdadeira árvore de sufixos para a cadeia S com tamanho igual a n a partir da árvore I_n, basta adicionar o caractere terminador \$ ao final da cadeia S, e continuar o algoritmo de Ukkonen para este caractere (até a fase $n+1$). Após a execução desta fase adicional (o que não aumentará o tempo de execução total do algoritmo), nenhum sufixo irá ser prefixo de outro, e a árvore será uma verdadeira árvore de sufixos. Além disso, devemos ainda trocar todos os índices globais e nas arestas incidentes às folhas pelo índice n. Isso pode ser facilmente feito num tempo de execução de $O(n)$ através de um caminhamento na árvore, onde todos os índices globais e nas arestas incidentes às folhas serão trocados por n. Com isso podemos enunciar o seguinte teorema:

Teorema 3.3 *O algoritmo de Ukkonen constrói, para uma cadeia S de tamanho igual a n, uma árvore de sufixos com todas as conexões de sufixo num tempo de $O(n)$.*

O algoritmo de Ukkonen acima terá um tempo de execução linear somente se o alfabeto for constante. Caso o alfabeto seja variável, seja σ o tamanho do alfabeto. Ao buscarmos pela próxima aresta no truque de salto/soma, iremos gastar um tempo de $O(x)$ ao invés de $O(1)$, onde x é o tempo necessário para acharmos a aresta cujo rótulo começa por um certo caractere. Se usarmos uma estrutura para armazenar as arestas onde a busca e a inserção de novos caracteres (que serão as chaves) podem ser feitas num tempo logarítmico (por exemplo, se usarmos árvores AVL [13, 38, 43]), o

tempo de execução desta busca, e da inserção de novas arestas será de $O(\log \sigma)$ no pior caso, e como na prova do teorema 3.2 vimos que o maior tempo de execução do algoritmo de Ukkonen é proveniente dos caminhamentos, então com alfabetos variáveis, o algoritmo de Ukkonen pode ser implementado para rodar num tempo de $O((\log \sigma)n)$.

Capítulo 4

Último Ancestral Comum

Como vimos na introdução do capítulo 3 descrito anteriormente, as árvores de sufixos, juntamente com a busca do último ancestral comum, nos permitirão executar vários algoritmos de processamento de cadeias de forma muito mais eficiente.

Apresentaremos, neste capítulo, um algoritmo para o pré-processamento de uma árvore genérica com n nós num tempo de $O(n)$, que permitirá identificar o último ancestral comum de dois nós da árvore num tempo constante. Sua aplicação nos problemas de cadeias vem do fato de que se obtivermos o último ancestral comum dos nós que representam dois sufixos i e j de uma cadeia S numa árvore de sufixos T, o caminho da raiz até este ancestral será igual para os dois sufixos, e logo após, cada um dos caminhos seguirá por uma aresta diferente emergente deste nó. Devido à estrutura da árvore de sufixos, vemos facilmente que o nome de caminho deste nó dará exatamente o maior prefixo comum destes dois sufixos.

Portanto, o algoritmo para identificar o último ancestral comum de dois nós num tempo constante, será muito importante em problemas de manipulação de cadeias, pois, como poderemos obter o maior prefixo comum entre duas cadeias num tempo constante, se construirmos a árvore de sufixos adequada para a cadeia S e pré-processá-la, teremos muitas identificações deste tipo presentes em vários algoritmos de processamento de cadeias. Para o nosso problema, em que usamos uma adaptação do algoritmo descrito em [17, 26], os algoritmos apresentados aqui serão muito importantes, pois como executamos muitas destas identificações para descobrirmos o maior prefixo comum de duas cadeias, construir uma árvore de sufixos para uma cadeia particular (que será descrita no próximo capítulo), executar o algoritmo de pré-processamento, e usar o algoritmo para identificar o ancestral num tempo constante, nos permitirá atingir os limites de tempo alegados para os algoritmos descritos

nesta tese.

4.1 Conceitos Básicos

Antes de descrevermos formalmente o problema que os algoritmos propostos devem resolver, vamos relembrar dois conceitos básicos, mas importantes, referentes a uma árvore genérica: um nó u será um *ancestral* de um nó v se ele estiver dentro do único caminho da raiz até este nó v. Além disso, se $v \neq u$, u será um *ancestral próprio* de v. Agora, o *problema da busca do último ancestral comum* (*lca*, da palavra original Lowest Commom Ancestor) entre dois nós u e v de uma árvore T, é o de achar o último nó w na parte comum dos únicos caminhos da raiz aos nós u e v. Na figura 4.1, o *lca* dos nós 6 e 10 é o nó 5, e o *lca* dos nós 6 e 3 é a raiz rotulada com o número 1.

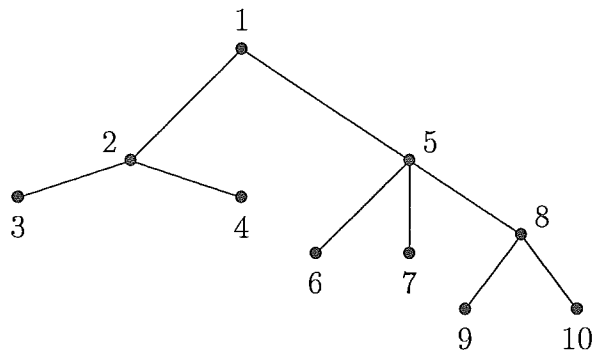


Figura 4.1: Uma árvore genérica T, onde os nós são numerados através do caminhamento pré-ordem.

Na próxima seção, apresentaremos um algoritmo de pré-processamento com tempo de execução de $O(n)$, onde n é o número de nós de uma árvore T, que permitirá obter o último ancestral comum entre dois nós genéricos desta árvore num tempo constante. Se não executássemos este pré-processamento na árvore, o tempo de execução de pior caso para a obtenção deste ancestral de dois nós poderia ser de $\Theta(n)$, o que prova a grande utilidade dos algoritmos deste capítulo. Vamos chamar o problema de identificação do último ancestral comum de dois nós de uma árvore de *perguntas lca.*

Os algoritmos para a resolução de perguntas *lca* foram primeiramente descritos

por Harel e Tarjan [18], e mais tarde foram simplificados por Schieber e Vishkin [35]. Basearemos a nossa descrição nos algoritmos descritos em [35]. Apesar dos resultados de perguntas *lca* serem grandemente usados em processamento de cadeias, estes algoritmos são geralmente vistos como "caixas pretas", não sendo muito conhecidos porque não são muito triviais (são baseados em operações sobre bits), e são geralmente considerados como algoritmos teóricos, apesar do algoritmo que iremos descrever ser bem prático.

Para podermos descrever corretamente os algoritmos e as provas dadas nas próximas seções, devemos descrever para qual modelo de máquina o nosso algoritmo apresentará os limites de tempo alegados. Os algoritmos são descritos sobre um modelo RAM de custo unitário. Neste modelo, dois números com $O(\log n)$ bits podem ser lidos, escritos e usados para acessar uma posição da memória num tempo constante. Além disso, as operações de comparação, adição, subtração, multiplicação e divisão sobre dois números com $O(\log n)$ bits, também podem ser feitas num tempo constante. Se os números tiverem mais do que $O(\log n)$ bits, não poderemos executar operações sobre estes números num tempo constante. Além destas operações, que estão definidas neste modelo, também precisamos executar as operações booleanas de XOR, OR e AND sobre dois números de $O(\log n)$ bits num tempo constante, deslocar uma máscara de bits para esquerda ou para a direita por $O(\log n)$ bits num tempo constante, criar uma máscara composta por uma seqüência de bits em 1 num tempo constante, e obter a posição do bit em 1 mais à esquerda, ou mais à direita também num tempo constante. Para vários modelos de máquinas reais e linguagens de programação de alto nível, estas hipóteses são razoáveis[1].

Vamos agora apresentar o algoritmo para a busca do último ancestral comum de dois nós em uma árvore binária completa num tempo constante, e como devemos pré-processar esta árvore para que este algoritmo possa ser executado. Estes conceitos serão importantes ao explicarmos, nas próximas seções, como pré-processar uma árvore genérica T, e como usar as informações obtidas neste pré-processamento no algoritmo que responde a uma pergunta *lca*, para descobrimos o ancestral comum de dois nós num tempo constante.

Seja B uma árvore binária completa com p folhas, isto é, com $n = 2p - 1$ nós. Como a árvore é binária e completa, cada nó (que não é uma folha) desta árvore

[1]Em [17, Cap. 8] temos maiores informações de como implementar as operações acima num modelo RAM de custo unitário.

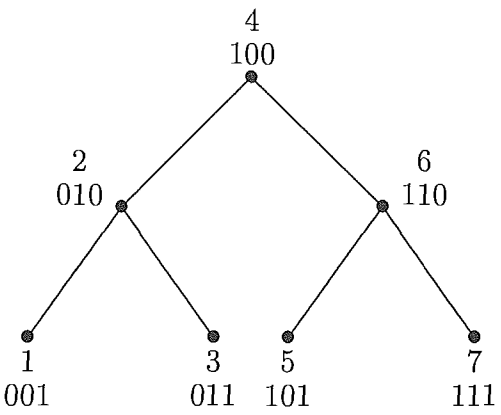


Figura 4.2: Uma árvore binária B com quatro folhas ($p = 4$, e $n = 2 \times 4 - 1 = 7$). Os números de caminhos atribuídos aos nós após um caminhamento inorder são mostrados nas bases dez e binária.

terá exatamente dois filhos, e o número de arestas num caminho da raiz até qualquer folha será de $d = \log_2 p$. Um conceito que será importante é o da *altura* de um nó v de B, que é igual ao número de nós no caminho deste nó até uma das folhas. A altura das folhas será sempre igual a 1, e a altura da raiz será igual a altura da árvore. Na figura 4.2, a altura dos nós 2 e 6 é igual a 2, e a altura da raiz é igual a 3, e a altura das folhas é igual a 1.

Na fase de pré-processamento, iremos nomear cada nó da árvore B com um número binário de $d+1$ bits codificando o caminho da raiz até aquele nó, chamado de *número de caminho*. Contando da esquerda para a direita[2], o i-ésimo bit representa a i-ésima aresta deste caminho. Se este bit for igual a 0, a aresta emergente do nó i será incidente ao filho esquerdo, e se for igual a 1, a aresta será incidente ao filho direito. Se o número resultante após a codificação do caminho tiver menos do que $d+1$ bits, preenchemos o resto do número, da esquerda para a direita, com um bit 1 seguido de quantos zeros forem necessários para termos um número com $d+1$ bits. Chamamos os bits que compõem este número de *bits de caminho*. Daqui em diante, os nós da árvore B serão referenciados por seus números de caminho.

Por exemplo, se caminhamos, a partir da raiz, para a direita, para a esquerda, e mais duas vezes para direita, o número de caminho do nó que atingimos no caminhamento será igual a 1011. Se $d = 6$, devemos estender o caminho com um bit 1 e dois bits 0s, obtendo o número 1011100 para este caminho. Como não caminha-

[2]Contrariando a numeração dos bits em um número binário que é da direita para a esquerda.

mos para atingir a raiz, ela será representada como um valor de $d+1$ bits em que somente o bit mais à esquerda (na posição $d+1$) será igual a um. Na figura 4.2, mostramos uma árvore binária completa onde $d=3$ com todos os nós rotulados por seus números de caminho.

Existe uma outra propriedade interessante da numeração proposta acima (que pode ser vista pela figura 4.2): os números de caminho são os números inorder, ou seja, eles serão os números obtidos ao numerarmos os nós da árvore ao executarmos um caminhamento inorder, onde rotulamos, recursivamente, todos os nós do filho esquerdo, a raiz, e recursivamente todos os nós do filho direito. Iremos usar a notação dada acima, pois sabemos que o número de caminho de um nó v descreve o caminho da raiz até este nó v. Portanto, podemos numerar os nós da árvore através de um caminhamento inorder, e a fase de pré-processamento pode ser executada num tempo de $O(n)$.

Agora, vamos mostrar como o ancestral de dois nós com os números de caminho iguais a i e j será calculado num tempo constante. Seja $lca(i,j)$ o número de caminho do último ancestral comum de i e j. Vamos considerar o caso em que nem i é ancestral de j, e nem j é ancestral de i. Se um dos nós for ancestral do outro, o ancestral poderá ser facilmente detectado, através do seguinte método: seja k_i a posição do bit 1 mais à direita de i, e k_j a posição deste mesmo bit em j, e seja m o número de bits de i e j. Agora, vamos compor uma máscara em que somente os $m-k$ bits mais significativos são iguais a 1, onde $k=\max(k_i,k_j)$. Seja agora x_i o "e"[3] de i com esta máscara, e similarmente x_j o "e" de j com a máscara. Agora, se o valor do "ou exclusivo" de x_i e x_j for igual a zero, o nó com o bit 1 mais à direita na maior posição será o pai do nó com este bit na menor posição. Se este "ou exclusivo" não for igual a zero, então não há possibilidade de i ser ancestral de j, e nem de j ser ancestral de i. Neste caso, deveremos usar o algoritmo descrito abaixo para descobrir o último ancestral comum de i e j.

Seja x_{ij} o "ou exclusivo" entre o números i e j. Agora, seja k [4] o bit mais à esquerda igual a 1 de x_{ij}. Como, em uma operação de "ou exclusivo", teremos um bit 0 como resultado se os dois operandos forem iguais, isso significará que os $k-1$ bits mais significativos dos números i e j são iguais, e que uma divergência

[3]No nosso modelo de computação, esta operação booleana, assim como as outras que serão usadas posteriormente, podem ser executadas num tempo constante, pois os operandos tem $O(\log n)$ bits.

[4]Numeraremos, neste caso, os bits na palavra da esquerda para a direita.

ocorreu exatamente na posição k. Como este número representa o caminho da raiz até cada um dos nós, e como este caminho é codificado da esquerda para a direita, isso indicará que os dois caminhos são idênticos por $k-1$ arestas até a aresta k, onde um dos caminhos segue pelo filho esquerdo, enquanto que o outro segue pelo filho direito. Portanto, o número composto, da esquerda para a direita, destes $k-1$ bits de i (ou j), seguido por um bit 1, e seguido por $d+1-k$ bits 0 irá codificar o caminho para o nó onde há o ponto de divergência entre os dois caminhos, que será exatamente o último ancestral comum dos nós i e j. Como todas as operações podem ser feitas num tempo constante no nosso modelo proposto descrito anteriormente, podemos enunciar o seguinte teorema:

Teorema 4.1 *Numa árvore binária completa, após um pré-processamento linear onde numeramos os nós da árvore com os seus números de caminho, uma pergunta lca para dois nós desta árvore pode ser respondida num tempo constante.*

Como dissemos anteriormente, descrevemos como calcular o *lca* em uma árvore binária completa devido a esta ser usada em nossas descrições do algoritmo para árvores genéricas. Estes conceitos serão usados, de forma geral, da seguinte forma: mapearemos os nós da árvore genérica T nos nós da árvore binária completa B, de tal forma que possamos usar a busca dos ancestrais em B para ajudar a resolver rapidamente estas perguntas sobre a árvore genérica T. Na próxima seção iremos descrever o algoritmo de pré-processamento e como este mapeamento será usado, e na última seção deste capítulo, iremos descrever o algoritmo que responderá as perguntas *lca* num tempo constante.

4.2 O Pré-processamento da Árvore

Nesta seção iremos descrever o algoritmo que processa uma árvore genérica T com $O(n)$ nós num tempo total de $O(n)$. O primeiro passo executado pelo algoritmo é o de numerar os nós da árvore usando o caminhamento pré-ordem, gerando uma numeração dos nós chamada de *pré-ordem* ou *numeração do mais profundo primeiro.* Um exemplo desta forma de numeração pode ser visto na figura 4.1. Uma característica importante desta numeração é a de que os números de pré-ordem dos nós de uma subárvore definida por um nó v de T serão consecutivos, onde o menor dos números será atribuído ao nó v, ou seja, se esta subárvore tiver q nós, e se o número

de pré-ordem do nó v for k, os números estarão no intervalo $[k, k+q-1]$. Este passo inicial terá um tempo de execução de $O(n)$.

Antes de descrevermos os próximos passos, vamos assumir que quando falamos de um nó v, estaremos falando tanto deste nó, como do número de pré-ordem atribuído a ele. Como vamos usar a árvore binária completa B em nossas descrições, devemos evitar confundir os números descritos para esta árvore, com os números descritos para uma árvore genérica T. Em seguida vamos definir algumas funções essencias para o pré-processamento da árvore.

Dado um número binário k, $h(k)$ dará a posição[5] do bit 1 mais a direita de k, ou alternativamente, $h(k)$ é o número de bits 0 antes da primeira posição com um bit em 1 mais 1. Por exemplo, $h(16) = 5$, $h(8) = 4$, e $h(5) = h(3) = 1$, pois como 5 e 3 são ímpares, os primeiros bits destes números serão iguais a 1. De forma geral, $h(p) = 1$ se p for um número ímpar.

O lema abaixo estabelece a relação entre a altura de um nó v de uma árvore binária completa B e seu número de caminho.

Lema 4.1 *Para qualquer nó v de B com número de caminho k, $h(k)$ será igual a altura deste nó na árvore B.*

Por exemplo, na figura 4.2, o valor de h dos nós internos será igual a 2, o das folhas será igual a 1, e o da raiz será igual a 3.

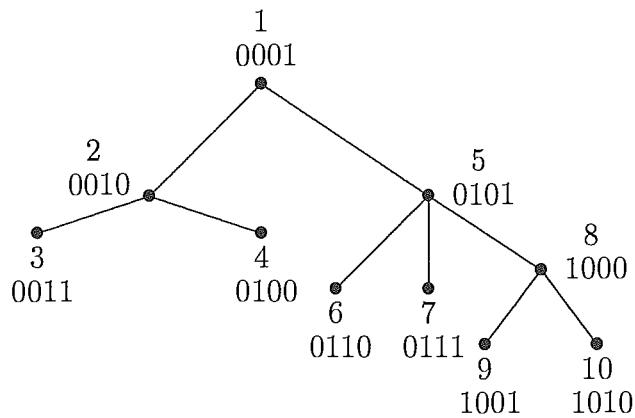


Figura 4.3: Numeração dos nós de uma árvore genérica T, para ilustrar a definição de $I(v)$, e como podemos calculá-lo. Os números de pré-ordem atribuídos aos nós ao executarmos o caminhamento pré-ordem são mostrados nas bases binária e dez.

[5] Agora usando a ordem de numeração padrão.

Para um dado nó v da árvore genérica T, definimos $I(v)$ como o nó w (número de pré-ordem) cujo valor de $h(w)$ é máximo sobre todos os outros valores dos nós que pertencem a subárvore definida por este nó v. Se olharmos para a representação binária dos números de pré-ordem, podemos dizer que $I(v)$ é o nó w cujo número (binário) de pré-ordem tem o maior número consecutivo de zeros iniciais do que todos os números de quaisquer outros nós presentes nesta subárvore. Na figura 4.3, mostramos os números de pré-ordem para cada nó nas bases dez e binária. Pela figura, notamos que $I(1) = I(5) = I(8) = 8$, $I(2) = I(4) = 4$, e que $I(v) = v$ para os outros nós não citados anteriormente. Se um nó v é um ancestral de um nó w, então pela definição devemos ter $h(I(v)) \geq h(I(w))$, pois $I(v)$ será igual ao nó com o valor máximo para h, que poderá ser $I(w)$, ou outro nó pertencente a subárvore definida pelo nó v. Portanto, o valor de $h(I(v))$ não pode decrementar ao caminharmos para cima em uma árvore.

Outra propriedade importante para o algoritmo de *lca*, que será provada pelo próximo lema, é a de que o valor de $I(v)$ para um dado nó v é único, ou seja, somente existe um w na subárvore definida por v tal que $I(v) = w$, e $h(w)$ é máximo. Com isso, enunciamos o seguinte lema:

Lema 4.2 *Para qualquer nó v em T, existe um único nó w na subárvore definida por v, para o qual o valor de $h(w)$ é máximo sobre todos os outros valores h dos outros nós desta subárvore. [17, Cap. 8]*

Prova Suponha que além do nó w exista um nó u tal que $h(w) = h(u) > h(q)$ para qualquer outro nó q na subárvore definida pelo nó v, e que $h(w) = i$. Seja m o número de bits de u e w.[6] Como u é diferente de w, estes valores devem diferir em algum bit à esquerda da posição i, pois a direita todos os bits de u e w devem ser iguais a 0. Então, seja k a posição mais a esquerda onde ocorre esta diferença, e sem perda de generalidade, seja $u > w$ (o caso em que $u < w$ é similar ao explicado a seguir). Agora, seja N o número formado, da esquerda para a direita, dos $m - k$ bits mais significativos de u (ou w), de um bit 1, e de $k - 1$ bits 0. Como podemos ver pela figura 4.4, o valor de N será maior do que w, e N será menor do que u, ou seja, $w < N < u$. Com isso, o nó numerado por N também está na subárvore de v, devido à propriedade dos números de pré-ordem em uma subárvore serem consecutivos. Mas como $h(N) = k > i$, isso irá contradizer a hipótese de

[6] Caso um deles seja menor do que o outro, estendemos o menor para m bits adicionando bits 0 à direita até que o seu tamanho seja igual a m.

que $I(v) = w$ (ou u), pois $h(N) > h(u) = h(w)$. Portanto, como chegamos a uma contradição, não existe este nó u, e o valor de $I(v)$ será único. □

	m	k		i		
u		1		10	$\cdots$	0
w		0		10	$\cdots$	0
N		10	$\cdots$			0

Figura 4.4: Representação gráfica dos números w, u, e N usados na prova do lema 4.2. Os bits de m até $k+1$ (os $m-k$ bits mais significativos) são iguais nestes três números.

Com base no lema acima, podemos resumir a unicidade do valor de $I(v)$ com o seguinte colorário:

Colorário 4.1 *A função $v \to I(v)$ é bem definida.*

Podemos calcular os valores de $I(v)$ num tempo de $O(n)$ através de um caminhamento de baixo para cima linear no tempo da seguinte forma: inicializamos todos os valores de $I(v)$ das folhas para $I(v) = v$. Para um dado nó interno v, $I(v) = v$ se $h(v)$ for maior do que $h(I(w))$ para cada filho w de v. Caso contrário, $I(v) = I(w)$, onde w é o filho de v com o maior valor para $h(I(w))$. O caminhamento será linear, pois cada nó será visitado somente uma vez.

Como já dissemos anteriormente, ao mapearmos os nós de T para os nós de B, queremos armazenar um número suficiente de informações das relações ancestrais de T, para que possamos resolver perguntas *lca* em T através das relações ancestrais entre os nós de B, e do uso do cálculo do *lca* para dois nós desta árvore. A unicidade de $I(v)$ será importante ao mapearmos o nó v de T para um nó na árvore B.

Vamos particionar os nós de T em subconjuntos cujo valor de $I(v)$ é o mesmo, como pode ser visto pela figura 4.5. Um subconjunto contendo todos os nós v de T com o mesmo valor de $I(v)$ é chamado de uma *sucessão*. Com isso, para que dois nós v e w estejam em uma mesma sucessão, $I(v) = I(w)$. Como cada sucessão forma um caminho para cima na árvore T, e como os valores de $h(I(v))$ nunca decrementam dentro desta sucessão, podemos enunciar o seguinte lema:

Lema 4.3 *Para qualquer nó v em T, o nó $I(v)$ é o nó mais profundo dentro da sucessão contendo o nó v.*

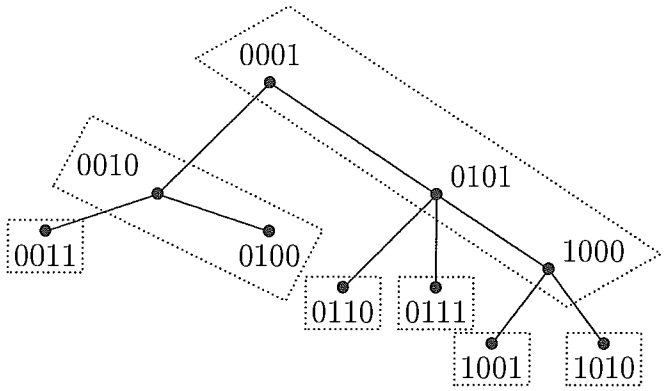


Figura 4.5: Nesta figura, mostramos a divisão dos nós da árvore T da figura 4.3 em subconjuntos com o mesmo número para I, chamados de sucessões. Neste caso, existem sete sucessões distintas.

Definimos a *cabeça* de uma sucessão como o nó desta sucessão mais próximo da raiz. Na figura 4.5, o nó 1 é a cabeça da sucessão de tamanho 3, o nó 2 da sucessão com tamanho igual a 2, e os outros nós são as cabeças das sucessões somente compostas por eles.

Agora, podemos definir o mapeamento (conceitual) dos nós de T para os nós de B: o *mapeamento da árvore* é um mapeamento dos nós da árvore genérica T para os nós de uma árvore binária completa B com altura igual a $d = \lceil \log n \rceil - 1$, onde cada nó v, com número de pré-ordem v, de T é mapeado para o nó $I(v)$ de B, com número de caminho igual a $I(v)$. Esta definição de mapeamento é bem definida, pois $I(v)$ será um número de $d+1$ bits, e com isso, cada um dos nós da árvore binária B serão numerados por um valor distinto de $d+1$ bits. Como mapeamos um nó v pelo valor de seu $I(v)$, todos os nós pertencentes a uma mesma sucessão serão mapeados para o mesmo nó $I(v)$ em B. Pela figura 4.6, onde mostramos o mapeamento para a árvore da figura 4.5, vemos que nem todos os nós de B serão usados pelo mapeamento, e portanto, na figura 4.6, somente mostraremos os números de caminho dos nós de B que foram usados no mapeamento.

Agora, vamos dar a descrição procedural detalhada do algoritmo de pré-processamento da árvore genérica T, que permitirá o uso do algoritmo da próxima seção para responder a uma pergunta *lca* num tempo constante.

Algoritmo para o pré-processamento de T

Início

1. Executar uma travessia pré-ordem na árvore T para atribuir números de pré-ordem aos nós durante este percusso. Ao atribuir o número de pré-ordem a um nó, calcular o valor de $h(v)$, e além disso, criar um ponteiro deste nó para o seu pai.

2. Executar o algoritmo botton-up descrito anteriormente para calcular os valores de $I(v)$ para cada nó v. Para cada um dos possíveis valores k tais que $I(v) = k$ para algum nó v, inicializar $L(k)$ para apontar para a cabeça da sucessão contendo este nó k. Isso pode ser feito ao calcularmos os valores de $I(v)$ da seguinte forma: um nó v será a cabeça de sua sucessão ou se ele for a raiz, ou se $I(v) \neq I(w)$, onde w é o seu pai.

3. Seja B uma árvore binária completa com altura igual a $d = \lceil \log n \rceil - 1$. Mapeamos cada nó v (com número de pré-ordem igual a v) de T para o correspondente nó $I(v)$ (com número de caminho igual a $I(v)$) de B.

4. Para cada nó v em T, criamos um número com $O(\log n)$ bits chamado de A_v, onde o bit i deste número será igual a 1 se e somente se v tiver um ancestral u que é mapeado para um nó $I(u)$ de B com altura igual a i, ou seja, se $h(I(u)) = i$. Este valor, para um nó v, pode ser facilmente calculado por uma travessia na árvore da seguinte forma: se v não for a raiz, seja $A = A_u$ onde u é o pai de v, e seja $A = 0$ caso contrário. Então fazemos $A_v = A$, e logo após ligamos o bit i de A_v, onde $i = h(I(v))$.

Fim

Após a execução do passo 2, poderemos obter a cabeça da sucessão da qual um nó v pertence, simplesmente calculando o valor de $I(v)$ e usando o array L, que guarda as cabeças das sucessões, para obter, em tempo constante, a cabeça da sucessão a qual o nó v pertence, dada por $L(I(v))$.

O mapeamento executado pelo passo 3 é útil, pois ele preservará as informações (não todas) sobre as relações ancestrais de T, mas é interessante sabermos, para um nó v, onde os ancestrais de v são mapeados em B. Estas são exatamente as informações codificadas nos valores de A_v calculados no passo 4. Pela definição de A_v, notamos que o número de bits 1 deste valor será exatamente o número de sucessões da raiz até este nó v. Por exemplo, para o nó 3 (em binário 0011), A_3 será igual a 1101 pois este nó será mapeado para o nó de B com altura igual a 1, o nó ancestral 2 será mapeado para um nó com altura igual a 3, e o nó ancestral 1 será mapeado para um nó com altura igual a 4.

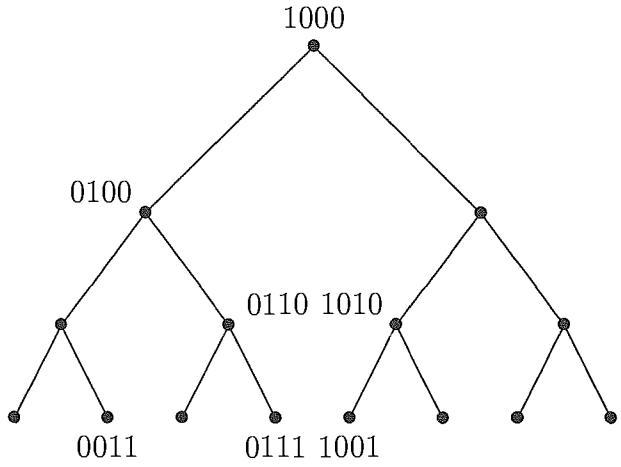


Figura 4.6: A árvore binária completa B usada no mapeamento dos nós da árvore T da figura 4.5. Somente mostramos os números de caminho, na base binária, dos nós desta árvore que tenham pelo menos um nó da árvore genérica T mapeado para ele.

O tempo de execução total do algoritmo será de $O(n)$, pois somente executamos travessias na árvore, e operações sobre números de $O(\log n)$ bits que são executadas num tempo constante em nosso modelo. Como este é um mapeamento de muitos para um, ele não preservará todas as informações ancestrais entre os nós de T,[7] mas estas informações serão suficientes, o que poderá ser visto na próxima seção, onde mostraremos o algoritmo para responder uma pergunta *lca* num tempo constante, usando as informações obtidas por este pré-processamento.

Lema 4.4 *Se um nó u é um ancestral de um nó v, então $I(u)$ é um ancestral de $I(v)$ na árvore B, ou alternativamente, se u for um ancestral de v na árvore T, ou u e v estão em uma mesma sucessão em T, ou $I(u)$ é um ancestral próprio de $I(v)$. [17, Cap. 8]*

Prova A prova é trivial quanto $I(u) = I(v)$ (u e v estão na mesma sucessão). Portanto, vamos supor que $I(u) \neq I(v)$. Como o nó u é ancestral de v, $h(I(u)) > h(I(v))$ pela definição de I (a igualdade vale quando $I(u) = I(v)$). Como os valores de h dão as alturas dos nós $I(u)$ e $I(v)$ na árvore B, o nó $I(u)$ está numa altura maior do que o nó $I(v)$. Se $h(I(u)) = i$, nós alegamos que os bits de $I(u)$ e $I(v)$ à esquerda de i são iguais, pois se isto ocorrer, os caminhos de $I(u)$ e $I(v)$ até a aresta

[7] Isso nem seria possível, pois a altura de B é de $O(\log n)$, e a de T pode ser, no pior caso, igual a $O(n)$.

$i-1$ serão iguais, e o caminho de $I(v)$ irá continuar para baixo, pois devemos ter algum bit à direita de i igual a 1 para que $h(I(u)) > h(I(v))$. Com isso, $I(u)$ seria um ancestral próprio de $I(v)$. Vamos supor que nossa alegação não é verdadeira, e que $k > i$ é a posição mais à esquerda da divergência entre $I(u)$ e $I(v)$, e seja m o tamanho de $I(u)$ e $I(v)$. Além disso, vamos supor, sem perda de generalidade, que este bit na posição k é igual a 1 em $I(u)$, e 0 em $I(v)$. Como k é a posição de divergência mais a esquerda, os $m-k$ bits mais significativos de $I(u)$ e $I(v)$ serão iguais, e com isso, $I(u) > I(v)$. Agora, como $I(u)$ e $I(v)$ são números de pré-ordem, como estão na subárvore definida por u, e como os números de pré-ordem nesta subárvore são consecutivos[8], teremos nós com números de pré-ordem entre $I(v)$ e $I(u)$ nesta subárvore. Seja N o número formado, da esquerda para a direita, pelos $m-k$ bits mais significativos de $I(u)$ (ou $I(v)$), seguido por um bit em 1, seguido por suficientes bits 0 para tornar o tamanho do número de pré-ordem de N igual ao de $I(u)$ e $I(v)$. Podemos ver facilmente que $I(v) < N < I(u)$, e portanto, N pertence a subárvore definida por u. Mas acontece que $h(N) = k > i = h(I(u))$, contradizendo a definição de I. Portanto, $I(u)$ e $I(v)$ devem ser iguais nos bits à esquerda de i, e o lema está provado. □

4.3 O Algoritmo para Calcular o *lca*

Vamos agora mostrar o algoritmo para calcular o último ancestral comum z de dois nós u e v de uma árvore genérica T. A descrição dada posteriormente irá explicar como calcular z a partir de $h(I(z))$, e das informações dadas pelo pré-processamento, e o teorema 4.2 descrito abaixo irá provar que podemos obter $h(I(z))$ usando o valor de $I(u)$, de $I(v)$, e as mesmas informações obtidas por este pré-processamento. Portanto, é essencial que o pré-processamento seja executado antes de executar o algoritmo que irá responder a uma pergunta *lca* para dois nós u e v de T. O pré-processamento só precisará ser executado uma vez, antes de começarmos a responder as perguntas *lca*.

Teorema 4.2 *Seja $i = h(w)$, onde w é o último ancestral comum de $I(u)$ e $I(v)$ em B, e seja $j \geq i$ a menor posição onde temos um bit 1 em A_u e A_v. Então, a altura do nó $I(z)$, para o qual z foi mapeado, será igual a j, ou seja, $h(I(z)) = j$. [17, Cap. 8]*

[8]Devido a interessante propriedade da numeração pré-ordem.

Prova Suponha que $h(I(z)) = k$. Como z é um ancestral de u e v, então A_u e A_v devem ter um bit 1 na posição k, e como pelo lema 4.4 $I(z)$ é um ancestral de $I(u)$ e de $I(v)$, $k \geq i$ ($I(z)$ ou será igual a w, ou será um ancestral próprio de w) e portanto, $I(z)$ terá altura maior ou igual do que a de w. Como, por definição, $j \geq i$, então $k \geq j$, garantindo a existência de uma posição mais à direita $j \geq i$ onde um bit 1 ocorrerá tanto em A_u como em A_v. Como A_u tem um bit 1 na posição j, então u terá um ancestral u' em T que será mapeado para o nó $I(u')$ de B. Como $j \geq i$, $I(u')$ será um ancestral de w em B, pois $I(u')$ estará numa altura maior do que w. Da mesma forma, A_v terá um bit 1 na posição j, e então v terá um ancestral v' em T mapeado para um nó $I(v')$ de B que também será um ancestral de w. Agora, como os nós $I(u')$ e $I(v')$ são ancestrais de um mesmo nó w em B, e como têm a mesma altura, $I(u') = I(v')$. Portanto u' e v' estarão em uma mesma sucessão em T, e um deles será ancestral do outro. Sem perda de generalidade, seja u' ancestral de v'. Então, como z é o último ancestral comum de u e v, u' ou será z, ou será um ancestral de z. Portanto, se z é mapeado para $I(z)$, e u' para $I(u')$, pelo lema 4.4 temos que $h(I(u')) \geq h(I(z))$, ou seja, $j \geq k$. Mas como já mostramos acima que $k \geq j$, então $k = j$, e o lema está provado. □

Agora mostraremos como obter, em tempo constante, último ancestral comum z de u e v, a partir de $h(I(z))$, e das informações coletadas no passo de pré-processamento. Considere a sucessão em que o nó z está. Então, o caminho de u até z deverá entrar nesta sucessão num nó u', e similarmente, o caminho de v até z deverá entrar nesta sucessão no nó v'. Então, z será o nó que estiver mais próximo da raiz que será, pela numeração pré-ordem, o nó com o menor número de pré-ordem. Como u' e v' representam tanto os nós como seus números de pré-ordem, então z será u' se $u' < v'$, e v' caso contrário. Por exemplo, na figura 4.5, se $u = 9$ (binário 1001) e $v = 6$ (em binário 0110), então $u' = 8$ (1000), e $z = v' = 5$ (0101).

Devido ao fato de z poder ser computado se tivermos u' e v', vamos mostrar como calcular estes dois valores usando o valor de $h(I(z))$, e as informações obtidas pelo pré-processamento. Explicaremos somente como computar u', pois v' poderá ser calculado da mesma forma, se substituirmos todas as ocorrências, nas descrições abaixo, de u por v, u' por v', A_u por A_v, e de $I(u)$ por $I(v)$. Seja m o número de bits de $I(u)$ e $I(v)$.

Seja $j = h(I(z))$, ou seja, j é a altura do nó $I(z)$ em B. Como z é um ancestral de u, $I(z) \geq I(u)$ pelo lema 4.4. Se $I(u) = I(z)$, então u e z estão na mesma sucessão, e $u' = u$. Agora, se $I(z) > I(u)$, z e u não estão na mesma sucessão, e existirá um nó w no caminho de z até o nó u que será filho do nó u' que estamos procurando. Como w não está na mesma sucessão que z, e está na subárvore definida por este nó, $I(w) < I(z)$ pelo lema 4.4. Como A_u codifica as alturas dos nós para os quais os ancestrais de u foram mapeados, poderemos descobrir o valor de $h(I(w))$ simplesmente procurando pela maior posição $k < j$ de A_u com um bit 1, pois w será o nó com a maior altura (com o maior valor para $h(I(v))$) no caminho de u até o nó w.

Vamos agora calcular $I(w)$ a partir de $h(I(w))$, e depois w a partir de $L(I(w))$. O nó w ou será igual a u (e neste caso $I(w) = I(u)$), ou será um ancestral próprio de u. Em ambos os casos, como w é um ancestral de u, como $h(I(w)) = k$, e como $I(w)$ é um ancestral de $I(u)$ pelo lema 4.4, os números de caminho para os nós $I(u)$ e $I(w)$ devem ser iguais nos $m - k$ bits mais significativos (devido à codificação dos números de caminho), e portanto, poderemos construir $I(w)$ a partir de $I(u)$. O número de caminho de $I(w)$ será composto, da esquerda para a direita, pelos $m - k$ bits mais significativos de $I(u)$, seguido por um bit 1, seguido por quantos bits 0 forem necessários para tornar o tamanho de $I(w)$ igual ao de $I(u)$. Agora, como o nó w é filho de um nó u' que está em uma sucessão diferente, w será a cabeça de sua sucessão, isto é, $w = L(I(w))$. Como w é filho de u', basta usarmos o ponteiro colocado na árvore pelo pré-processamento para acessar o nó u'.

Depois de calcular u', repetimos o processamento acima para calcular v'. Após isso, z poderá ser calculado a partir de u' e v' pelo processo explicado acima. Com base nas explicações acima, podemos enunciar o seguinte teorema, que resume o que foi feito acima.

Teorema 4.3 *Seja z o último ancestral comum dos nós u e v. Dado o valor de $h(I(z))$, poderemos, após a fase de pré-processamento anterior, achar z num tempo constante.*

Em seguida, damos a descrição procedural do algoritmo descrito nesta seção:

Busca do *lca* num tempo constante para dois nós u e v, onde z = lca(u, v)

Início

1. Achar o último ancestral comum t de $I(u)$ e $I(v)$ usando o algoritmo para cálculo do *lca* em árvores binárias completas descrito anteriormente.

2. Achar a menor posição $j \geq h(t)$ em que A_u e A_v têm um bit 1. Pelo teorema 4.2, $h(I(z)) = j$.

3. Achar o nó u' mais próximo do nó u, que está na mesma sucessão de z, através dois seguintes passos:

   - Achar a posição k do bit 1 mais à direita de $I(u)$.
   - Se $k = j$, então u' será igual a u.
   - Agora, se $k < j$, então devemos achar a posição $l < j$ mais à esquerda para a qual temos um bit 1 em A_u. Construir, da esquerda para a direita, o número composto pelos $m - l$ bits mais significativos de $I(u)$, seguido por um bit 1, seguido pelo número necessário de bits 0 para completar o tamanho correto de $I(w)$. Agora, usando o vetor L computado pelo pré-processamento, $w = L(I(w))$. O nó u' será o pai de w.

4. Achar o nó v' mais próximo de v, que está na mesma sucessão de z, usando o mesmo método ao calcular o nó u'.

5. Se $u' < v'$, então $z = u'$, caso contrário, $z = v'$.

Fim

Como no algoritmo acima somente executamos as operações definidas pelo nosso modelo sobre números com $O(\log n)$ bits, poderemos, após o pré-processamento da árvore genérica T responder, em tempo constante, a uma pergunta $lca(u, v)$ para dois nós u e v de T.

Capítulo 5

Detecção de Palíndromos Aproximados

Neste capítulo, iremos mostrar o algoritmo básico para resolver o nosso problema de calcular todos os palíndromos aproximados em uma cadeia S de tamanho igual a N, que poderá usar qualquer um dos algoritmos descritos neste capítulo para o problema de k-diferenças, para obter um palíndromo aproximado para um centro particular, e quais os tempos de execução de pior caso obtidos. Mostraremos também duas otimizações que podem reduzir o tempo na prática. Para finalizar, mostraremos como usar a paralelização descrita no paper [26] para obtermos uma simples paralelização (não eficiente) do nosso algoritmo.

Para podermos calcular um destes palíndromos, iremos estudar dois algoritmos baseados na restrição ao número máximo de erros no problema de edição de cadeias, um para o caso de casamento de cadeias aproximado descrito em [17, 26], que foi primeiramente descrito em [27, 33], que usa árvores de sufixos e a busca do último ancestral comum para obter um tempo de execução de $O(kn)$, onde k é o número máximo de erros, e n é o tamanho do texto T. Iremos também estudar um algoritmo bem simples com tempo de execução de $O(km)$ descrito em [17], onde m é o tamanho da cadeia S que queremos transformar em uma cadeia P de tamanho n, e mostraremos como adaptar o algoritmo descrito para o problema de casamento de cadeias aproximado para obtermos um algoritmo, para o problema de k-diferenças, com tempo de execução de $O(k^2 + m + n)$. Em [11, 37] é apresentada uma ampla pesquisa de métodos para os problemas particulares de edição de cadeias descritos neste capítulo.

Os algoritmos descritos neste capítulo são somente válidos para o caso em que o custo de uma inserção, de uma deleção e de uma substituição são iguais a 1, e em

que o casamento não é contabilizado, que é o caso mais comum onde o problema de edição de cadeias e seus casos particulares são usados.

5.1 O Algoritmo para o Casamento de Cadeias Aproximado

Nesta seção, vamos considerar um algoritmo para a resolução do problema do casamento de cadeias aproximado baseado nas restrições impostas por este problema. Neste problema, é dado um texto T de tamanho igual a n e um padrão P de tamanho igual a m (onde geralmente m é bem menor do que n), e desejamos obter todas as posições no texto T onde começam (ou terminam) todas as ocorrências aproximadas do padrão no texto com no máximo k erros. Devemos também lembrar de que, neste algoritmo, as inserções de caracteres do texto antes e depois do padrão não contam como erros, o que significa que a linha 0 da matriz deve ser toda preenchida com 0, e que um script poderia começar em qualquer posição $(0, j_1)$ e terminar em qualquer posição (m, j_2), onde $0 \leq j_1 \leq j_2 \leq n$. No algoritmo dado a seguir, acharemos todas as posições de término destas ocorrências do padrão, mas como podemos obter os scripts de edição de todas as ocorrências num tempo total igual ao do algoritmo, se desejarmos poderemos obter a posição inicial após a computação do script.

A grande diferença do algoritmo descrito a seguir em relação ao algoritmo básico, dado na seção 2.2, e dos outros algoritmos existentes para este e outros problemas de edição de cadeias, é a de que ele será baseado nas diagonais da matriz ao invés de nas células da matriz como no algoritmo básico. Uma *diagonal* d é composta por todas as células (i, j) para as quais a diferença do índice j e do índice i é igual a d. A diagonal 0, composta por todas as células da forma (i, i), $0 \leq i \leq \min(m, n)$, é chamada de *diagonal principal* da matriz. Devido à definição de como uma diagonal é composta, as diagonais abaixo da diagonal principal serão numeradas de -1 até $-m$, e as diagonais acima da diagonal principal serão numeradas de 1 até n. Com isso, uma diagonal d começará na célula $(0, d)$ se $d \geq 0$, e na célula $(|d|, 0)$ se $d < 0$. Na figura 5.1, mostramos todas as diagonais da matriz dada, rotuladas com os seus respectivos números.

O uso das diagonais é importante porque elas definem uma relação entre as células, que permitem saltar de uma célula $(a, a + d)$ para uma célula $(b, b + d)$ em uma diagonal d, se $P[a + 1..b]$ casar com $T[d + a + 1..d + b]$, pois saberemos, pela

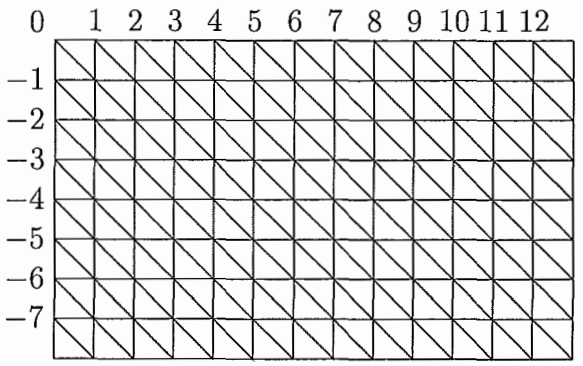


Figura 5.1: As diagonais da matriz de programação dinâmica onde o tamanho m do padrão P é igual a 7, e o tamanho n do texto T é igual a 12. Também é mostrado os números atribuídos a cada diagonal.

relação de recorrência definida na seção 2.2, que os valores das células $(l, l+d)$, $a < l \leq b$, deverão ser iguais ao da célula $(a, a+d)$ devido a este casamento. Este salto da célula $(a, a+d)$ para a célula $(b, b+d)$ faz com que não seja necessário o cálculo das células que saltamos, o que reduz o trabalho a ser executado pelo algoritmo. Os saltos nas diagonais são importantes por permitirem calcularmos a conversão do padrão numa porção do texto, calculando o menor número possível de células da matriz de programação dinâmica.

Um *caminho-e* é um caminho com e erros (inserções, deleções e substituições) que pode começar em qualquer célula $(0, d)$, $0 \leq d \leq n$, ou seja, no começo de qualquer uma destas diagonais d. Um caminho-e é o *último a chegar* a uma diagonal d, se ele for um caminho-e que termina na diagonal d, e se o índice da linha i em que este caminho termina é maior ou igual ao da linha do término de qualquer outro caminho-e que termina nesta diagonal. Graficamente, um caminho-e será o último a chegar a uma diagonal d, se nenhum outro caminho-e terminar num ponto desta diagonal mais abaixo do ponto em que este caminho termina.

O algoritmo, com tempo de $O(kn)$, consistirá de $k+1$ passos, onde cada passo irá executar num tempo de $O(n)$. Em cada passo e, $0 \leq e \leq k$, iremos calcular o final de um caminho-e que é o último a chegar a uma diagonal d, para d variando de $-\min(e, m)$ até n.[1] O último caminho-e a chegar a uma diagonal d será obtido

[1] O min nos limites inferiores desta seção é para evitar que usemos uma diagonal que esteja fora da faixa máxima de diagonais que varia de $-m$ até n, como pode ser visto na figura 5.1, onde $m = 8$ e $n = 13$.

a partir dos últimos caminhos-$(e-1)$ que chegaram, no passo $e-1$, nas diagonais $d-1$, d e $d+1$.

Se um caminho-e terminar na última célula de uma diagonal d, e se esta célula estiver na linha m, então a coluna desta célula, dada pelo valor $d+m$, será a posição de término de uma ocorrência do padrão P no texto T com exatamente e erros. Como será explicado abaixo, cada passo será implementado num tempo de $O(k+n) = O(n)$, pois o último caminho-e a chegar à diagonal d será obtido dos últimos caminhos-$(e-1)$ a chegarem às diagonais $d-1$, d, e $d+1$ num tempo constante.

Para podermos calcular o último caminho-e a chegar a uma diagonal d, devemos ser capazes de obter o tamanho do maior prefixo comum de um sufixo que começa na posição i de P, e de um sufixo que começa na posição j de T num tempo constante. Podemos fazer isso através dos seguintes passos:

1. Se ainda não construímos a árvore de sufixos, obtemos a árvore de sufixos para a cadeia $PT\$$, num tempo de $O(m+n)$.[2] Caso contrário, saltamos para o passo 4.

2. Pré-processamos a árvore num tempo de $O(m+n)$ para atribuir a cada nó o tamanho do nome do caminho da raiz até este nó;

3. Pré-processamos novamente a árvore para podermos obter o último ancestral comum de dois nós quaisquer num tempo constante.

4. Agora, obteremos o último ancestral comum para os sufixos i e $j+m$ (as folhas que os representam) da árvore de sufixos usando o algoritmo que responde a uma pergunta *lca* num tempo constante, se nenhum dos sufixos é vazio. O nome de caminho deste ancestral dará o maior prefixo comum entre estes dois sufixos, pois os dois caminhos da raiz até cada um dos sufixos são iguais da raiz até este ancestral, e após este nó, cada caminho seguirá por uma aresta diferente, e portanto, exatamente após este ponto existirá um descasamento.

5. Como cada nó tem o tamanho do seu nome de caminho, seja q o tamanho do nome de caminho do último ancestral comum dos sufixos i e $j+m$. Então, o tamanho do maior prefixo comum entre os sufixos i de P e j de T será igual

[2]Este limite de tempo só é válido se o alfabeto for fixo. Se o alfabeto for variável, já vimos que o tempo deve ser multiplicado por $\log \sigma$, onde σ é o tamanho do alfabeto.

a $\min(q, m - i + 1)$, onde o min da fórmula é para evitar que o tamanho do prefixo seja maior do que o tamanho do sufixo i de P. Se um dos sufixos for vazio, então o tamanho do maior prefixo comum entre os dois sufixos será igual a zero.

Agora, no passo $e = 0$, a faixa de diagonais d a serem calculadas será de 0 até n. Como neste passo procuramos pelo último caminho-0 a chegar a cada diagonal, e como não podemos cometer erros, ele deverá começar na célula inicial $(0, d)$ para cada diagonal d. Isso quer dizer que a linha do último caminho-0 a chegar a uma diagonal d será igual ao tamanho do maior prefixo comum entre P, e o sufixo que começa na posição $d + 1$ de T. Como vimos anteriormente, isso poderá ser obtido num tempo constante. Com isso, o tempo total do primeiro passo será igual a $O(n)$.

Para um passo genérico e, $0 < e \leq k$, o último caminho-e a chegar a uma diagonal d, $-\min(e, m) \leq d \leq n$, poderá ser obtido a partir dos últimos caminhos-$(e-1)$ a chegarem às diagonais $d - 1$, d e $d + 1$. Ele será, dentre os caminhos R_1, R_2 e R_3 descritos abaixo, aquele que terminar na célula com o maior índice possível para a sua linha, como podemos ver graficamente pela figura 5.2. Usaremos o método descrito anteriormente para obter o maior prefixo comum entre um sufixo de P e um sufixo de T. Os caminhos abaixo irão existir somente se o caminho-$(e-1)$ correspondente estiver definido.

- O caminho chamado de R_1 com e erros consistirá do último caminho-$(e-1)$ a chegar sobre a diagonal $d - 1$, seguido por uma inserção que estenderá o caminho até d, seguido por um certo número c de casamentos. Se x for a linha de término do caminho- $(e-1)$ na diagonal $d-1$, descoberta no passo anterior, c será igual ao tamanho q do maior prefixo comum entre os sufixos $P[x+1..m]$ e $T[x + 1 + d..n]$, ou seja, $c = \min(q, m - x)$. Este caminho só será válido se pudermos inserir um caractere do texto, ou seja, se $x + d - 1 < n$, se $x < m$, pois as inserções após o final do padrão não nos interessam, e se um último caminho-$(e-1)$ na diagonal $d - 1$ existir.

- O caminho chamado de R_2 com e erros é composto pelo último caminho-$(e-1)$ a chegar à diagonal d, seguido por uma substituição[3], seguido novamente por um número c de casamentos dados pelo tamanho q do maior prefixo comum de $P[x + 2..m]$ e de $T[x + 2 + d..n]$, onde x é a linha onde o caminho-$(e-1)$

[3] O caminho-$(e-1)$ deve terminar ou no último ponto da diagonal, ou antes de uma substituição.

termina em d, ou seja $c = \min(q, m - x - 1)$. Este caminho só existirá se for possível executarmos uma substituição de um caractere do padrão por um caractere do texto, ou seja, se $x < m$ e $x + d < n$.

- O caminho chamado de R_3 com e erros é composto pelo último caminho-$(e-1)$ a chegar à diagonal $d+1$, seguido por uma deleção que estenderá o caminho até a diagonal d, seguido por um número c de casamentos que será dado, se x for a linha em que o caminho-$(e-1)$ termina em $d+1$, por $c = \min(q, m-x-1)$ onde q é o tamanho do maior prefixo comum entre $P[x+2..m]$ e $T[x+2+d..n]$. Somente existirá este caminho se pudermos fazer uma deleção de um caractere do padrão, ou seja, se $x < m$, e se um último caminho-$(e-1)$ existir na diagonal $d+1$.

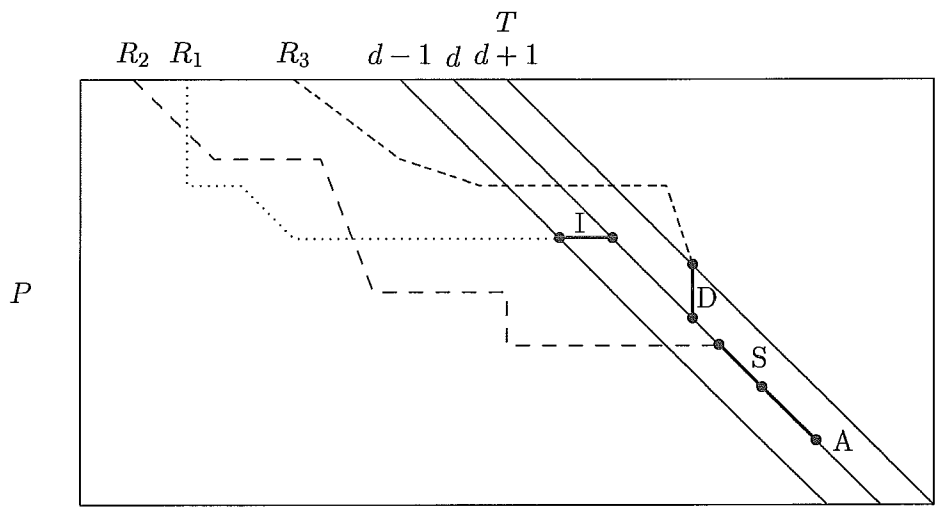


Figura 5.2: Cálculo do último caminho-e a chegar sobre a diagonal d. Na figura, mostramos os caminhos-$(e-1)$ usados pelos caminhos R_1, R_2 e R_3 que terminam, respectivamente, nas diagonais $d-1$, d e $d+1$, e os pontos de entrada destes caminhos na diagonal d após serem estendidos usando uma operação que aumentará o número de erros de $e-1$ para e, e que dependerá da diagonal de origem deste caminho. Como podemos ver pela figura, o caminho R_2 entrará na diagonal pela última vez no maior ponto, e portanto, ao final de sua extensão, dada pelo ponto A, teremos o final do último caminho-e a chegar sobre d.

Como pode ser visto pela figura 5.2, a computação do último caminho-e a chegar em d pode ser calculada obtendo a extensão do último caminho-$(e-1)$ que termina em $d-1$, d, ou $d+1$ que terminar, após a extensão, no maior ponto possível, seguida

por uma extensão de casamentos, pois após os caminhos dados acima entrarem em d, executamos para todos os caminhos uma extensão de casamentos que deverá, no máximo, parar exatamente no ponto de término do último caminho.

Já a importância do último caminho-e a chegar a uma diagonal d vem do modo de execução do nosso algoritmo, e da definição de distância de edição: como em cada passo do algoritmo calculamos os caminhos com um certo número de erros que será aumentado nos passos posteriores, o uso do último caminho nos permitirá omitir a computação das células $(l, l+d)$, $a < l < b$, se no passo $e-1$ o último caminho-$(e-1)$ terminou na célula $(a, a+d)$ da diagonal d, e se a extensão do último caminho-$(e-1)$ escolhido terminar na célula $(b, b+d)$. Além disso, se o último caminho-e terminar na célula $(c, c+d)$, não calcularemos também todas as células $(l, l+d)$, $b < l \leq c$, devido à extensão de casamentos que foi executada.

O teorema a seguir irá provar que o último caminho-e a chegar a uma diagonal d no passo e será, como foi descrito no parágrafo anterior, R_1, R_2 ou R_3.

Teorema 5.1 *Como vimos, os caminhos R_1, R_2 e R_3 serão caminhos-e que terminam na diagonal d. O último caminho-e a chegar à diagonal d será, dentre estes caminhos, o que atingir esta diagonal na maior linha possível. [17, Cap. 12]*

Prova Cada um dos caminhos-e R_1, R_2 e R_3 descritos anteriormente são extensões de caminhos-$(e-1)$, e como adicionamos um erro ao estendermos estes caminhos, serão todos caminhos-e. Portanto, o último caminho-e a chegar à diagonal d deve ser ou um destes caminhos, ou um outro caminho. Seja R^* o último caminho-e a chegar em d, e seja R' o último caminho-$(e-1)$ a chegar em d. Seja x a posição após o término de R'. O passo necessário para passarmos do ponto final de R' para o ponto x deve ser uma substituição, pois se não R' não seria o último a chegar em d. Agora, como R^* termina na diagonal d, deve existir um ponto em que ele entra nesta diagonal pela última vez. Suponha que este caminho entre pela última vez acima do ponto do término de R'. Então, ao chegar ao ponto x, o caminho deverá ter e erros, pois se não o caminho R' não seria novamente o último a chegar em d. Portanto, neste caso, R^* chegaria à mesma célula de R_2, pois eles teriam o mesmo número de erros logo após o final de R'. Se R^* entra pela última vez em d abaixo do final de R', ele deverá novamente ter e erros após entrar em d, pois se não R' novamente não seria o último caminho a chegar em d. Agora, R^* deve ter vindo ou da diagonal $d-1$, ou da diagonal $d+1$. Suponha que ele tenha vindo da

diagonal $d-1$. Como o caminho R_1 é uma extensão do último caminho-$(e-1)$ a chegar sobre $d-1$, o caminho R^* deverá entrar no mesmo ponto, ou acima do ponto em que R_1 entrou em d. Portanto, R^* e R_1 devem terminar no mesmo ponto após os casamentos. De forma similar, se R^* entrar em d da diagonal $d+1$, ele deverá terminar no mesmo ponto do que R_3. Portanto, o último caminho-e a chegar sobre d no passo e será, dos possíveis caminhos R_1, R_2 e R_3, o que terminar em d num ponto na maior linha possível. □

Como vimos anteriormente, a computação do último caminho-e de uma diagonal d pode ser feita, num tempo constante, a partir dos pontos de término dos caminhos-$(e-1)$ que são aplicáveis para o passo e e a diagonal d atual. Agora, como computamos $n+1$ diagonais no passo 0, $n+2$ no passo 1, e genericamente, $n+1+e$ num passo e, $0 \leq e \leq k$, o número total de caminhos a serem computados será igual a $(n+1)+(n+2)+(n+3)+\cdots+(n+1+k) = O(kn)$.[4] Como cada um destes caminhos podem ser computados num tempo constante, o tempo total de execução da computação dos caminhos será de $O(kn)$. Como o tempo de criação da árvore de sufixos e dos pré-processamentos é de $O(m+n)$, então o tempo total de execução do algoritmo será de $O(kn+m+n) = O(kn)$.

Se desejamos obter os scripts de edição que convertem o padrão P nas porções do texto que terminam nos pontos dados pelo algoritmo anterior, devemos guardar todos os valores dos pontos de término dos últimos caminhos-e que terminam em uma diagonal d para todos os $k+1$ passos, junto com as diagonais e os passos em que as ocorrências aproximadas do padrão terminaram. Com isso, todos os scripts de edição, juntamente com as posições iniciais do casamento aproximado, poderão ser obtidos também num tempo de $O(kn)$. Como temos $O(k+n)$ diagonais, como a árvore de sufixos e as estruturas para o cálculo do *lca* gastam um espaço de $O(m+n) = O(n)$, e como o espaço necessário para guardar o valor da linha de término de um caminho é constante, assim como o espaço para guardar o passo e a diagonal para uma ocorrência aproximada do padrão particular, o espaço necessário será de $O(kn)$. Podemos então enunciar o seguinte teorema:

Teorema 5.2 *Todas as posições de início e de término das ocorrências aproximadas de um padrão P de tamanho igual a m num texto T de tamanho igual a n, juntamente com os scripts de edição que convertem o padrão nas porções do texto*

[4] Esta é a soma máxima, para o caso em que $k \leq m$.

limitadas por estas posições, podem ser obtidas num tempo e espaço total de $O(kn)$.

Se estamos somente interessados nos pontos de término no texto das ocorrências aproximadas do padrão, precisaremos somente dos pontos de término dos caminhos do passo anterior, e com isso, o espaço necessário será de $O(k+n)$. Com isso, podemos enunciar o seguinte teorema:

Teorema 5.3 *Todas as posições de término das ocorrências aproximadas do padrão P de tamanho m num texto T de tamanho n podem ser obtidas num tempo de $O(kn)$, com um espaço total de $O(k+n)$.*

5.2 Os Algoritmos para o Problema de k-Diferenças

Para começarmos a descrição destes algoritmos, devemos lembrar que, no problema de k-diferenças, queremos obter um script de edição σ ou alinhamento que converta a cadeia S de tamanho igual a m na cadeia P de tamanho igual a n, que tenha no máximo k erros. Se isso não for possível, não podemos converter a cadeia S na cadeia P com menos do que $k+1$ erros. A idéia básica do primeiro algoritmo será então a de reduzir o tempo de execução da computação da matriz de programação dinâmica de $O(mn)$ para $O(km)$, usando esta restrição ao número de erros imposta a este problema. O segundo algoritmo irá adaptar o algoritmo da seção anterior ao problema de k-diferenças.

5.2.1 O Algoritmo Baseado em Faixas

Para descrevermos a idéia do algoritmo, vamos usar o conceito de diagonal principal da matriz de programação dinâmica descrito na seção anterior. Como qualquer caminho no problema de k-diferenças começa na célula $(0,0)$, um script de edição ótimo σ que converta a cadeia S na cadeia P deve começar na diagonal principal e terminar numa célula desta diagonal, em uma célula abaixo desta diagonal, ou em uma célula à direita desta diagonal. Com isso, um caminho descrito pelos ponteiros de remontagem de um script de edição σ com k ou menos erros não pode passar pelas células $(i, i-l)$ e $(i, i+l)$ onde $l > k$, pois seriam necessários no mínimo l erros (deleções para atingir a primeira célula, e inserções para atingir a segunda

célula) para atingirmos estas células, devido a este caminho começar na célula $(0,0)$. Portanto a idéia, ilustrada na figura 5.3, é a de ao invés de computarmos todas as $O(mn)$ células da matriz de programação dinâmica, computarmos somente as células dentro de uma faixa que conterá as células cuja distância à diagonal principal é menor ou igual a k. Com isso, em cada linha teremos no máximo $2k+1$ células, e o tempo total de execução será de $O(km)$. Se todas as células abaixo ou à direita da diagonal principal estiverem a uma distância menor do que k, todas elas serão computadas pelo algoritmo, e neste caso, não teremos uma faixa inferior ou uma faixa superior.

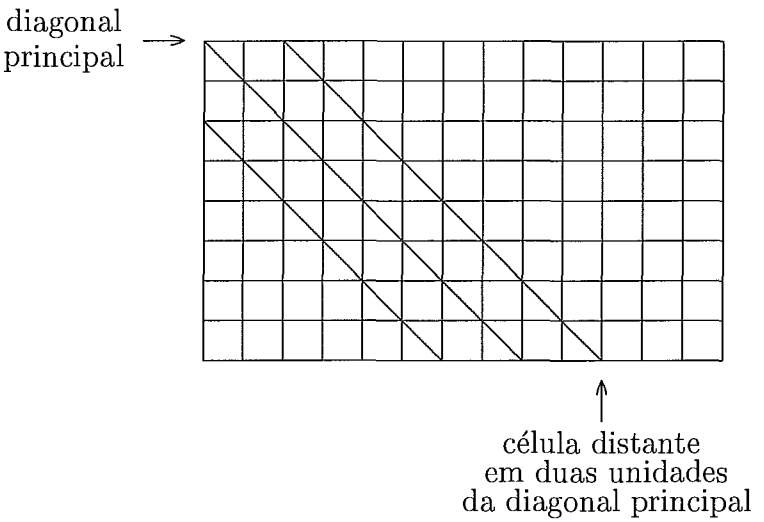


Figura 5.3: A diagonal principal da matriz de programação dinâmica, e duas faixas que estão a uma distância de $k = 2$ espaços da diagonal principal. Portanto, esta figura define a porção da matriz que será calculada, que será composta pelas células dentro do limite definido pelas faixas.

Ao calcularmos uma célula que não esteja em um dos limites da faixa definida pelas células distantes em k posições da diagonal principal, usamos a mesma relação de recorrência descrita no capítulo 2 para calcular o valor desta célula, e o mesmo método para atribuir os ponteiros usados pela remontagem do caminho. Caso a célula esteja no limite inferior da faixa, devemos ignorar, na relação de recorrência, a parte responsável pela inserção de um caractere, e o ponteiro correspondente se esta opção for um dos mínimos, ou seja, como a célula será da forma $(i, i-k)$, $1 \leq i \leq \min(m, n+k)$, devemos ignorar o termo $D(i, i-k-1)+1$ do min da relação de recorrência, e não criar um ponteiro da célula $(i, i-k)$ para a célula $(i, i-k-1)$ caso esta opção seja um dos mínimos. Caso a célula esteja no limite superior,

devemos ignorar, na relação de recorrência, a parte responsável pela deleção de um caractere, e o ponteiro relacionado gerado se esta opção for um dos mínimos, isto é, como a célula é da forma $(i, i+k)$, $1 \leq i \leq \min(m, n-k)$, devemos ignorar o termo $D(i-1, i+k)+1$, e um possível ponteiro da célula $(i, i+k)$ para a célula $(i-1, i+k)$.

Devido às células fora desta faixa serem atingíveis somente com mais do que k erros, a célula (m, n) deverá estar dentro da faixa para convertermos a cadeia S na cadeia P com k ou menos erros, ou seja $|m-n| \leq k$. Como nós só calculamos as células dentro desta faixa, os valores das células somente terão valores corretos se forem menores ou iguais a k. Se o valor de uma célula for maior do que k, ele somente estará correto se o caminho da célula $(0,0)$ até esta célula estiver contido dentro da faixa. Com isso, se o valor da célula (m, n) for maior do que k, isso indicará que precisamos mais do que k erros para converter a cadeia S na cadeia P, mas não necessariamente o valor dado pela célula (m, n). Como todas as células atingíveis com k ou menos erros estão dentro desta faixa, então todos os caminhos que reconstituem os scripts de edição da cadeia S na cadeia P com $d \leq k$ erros devem estar dentro da faixa. Com isso, podemos enunciar o seguinte teorema:

Teorema 5.4 *Existirá um script de edição (ou um alinhamento) da cadeia S de tamanho m na cadeia P de tamanho n com k ou menos erros, se e somente se a célula (m, n) estiver dentro da faixa, e se o valor da célula (m, n) for menor ou igual a k. Com isso, o problema de k-diferenças pode ser resolvido num tempo de $O(km)$ com um espaço também igual a $O(km)$.*

Este algoritmo pode ser usado mesmo no problema em que não há um limite de erros, onde a distância de edição das cadeias S e P é igual a k^*, onde k^* é desconhecido. Se usássemos o algoritmo básico, o tempo de execução seria de $O(mn)$, mas podemos reduzir o tempo de execução para $O(k^*m)$, usando o seguinte algoritmo: fazemos $k=1$, e verificamos se é possível converter S em P com no máximo um erro. Caso seja possível, terminamos o algoritmo, caso contrário, dobramos o valor de k ($k=2$) e verificamos novamente se é possível converter S em P com no máximo dois erros. De forma geral, quando não é possível converter a cadeia S na cadeia P com no máximo k erros, dobramos o valor de k, e tentamos novamente até conseguirmos converter S em P com no máximo k erros, isto é, até a distância de edição de S e P ser menor ou igual a k. O tempo de execução, assim como o espaço gasto, será

de $O(k^*m)$ como será provado pelo teorema abaixo:

Teorema 5.5 *Através da dobragem sucessiva de k até termos um script de edição (ou alinhamento) que converta a cadeia S de tamanho igual a m, na cadeia P de tamanho igual a n com no máximo k erros, a distância de edição e um dos scripts poderá ser obtido num tempo e espaço de $O(k^*m)$, onde k^* é a distância de edição entre S e P. [17, Cap. 12]*

Prova Seja k' o maior valor de k usado pelo algoritmo descrito acima. Como $k'/2 < k^*$, ou seja, $k' < 2k^*$, e como o trabalho total do algoritmo, assim como o espaço total, é de $O(k'm + k'm/2 + k'm/4 + \cdots + m) = O(k'm)$, então o tempo e o espaço gasto serão de $O(k^*m)$. □

5.2.2 A Adaptação do Algoritmo da Seção 5.1

Agora, mostraremos como adaptar o algoritmo para o problema de casamento de cadeias aproximado para um texto T de tamanho igual a n, e um padrão P de tamanho igual a m, dado na seção 5.1, com tempo de execução de $O(kn)$ se usarmos árvores de sufixos e a busca do último ancestral comum, tendo um consumo de espaço de $O(kn)$ se desejamos obter um script de edição, e de $O(k+n)$ caso contrário.

Assim como no caso anterior, para podermos converter, usando a adaptação descrita abaixo, a cadeia S na cadeia P, devemos ter a célula (m, n) em uma diagonal que esteja dentro da faixa usada, ou seja, como esta célula está na diagonal $d = n - m$, esta diagonal d deverá ser usada em algum passo do algoritmo, devendo estar presente dentro da faixa máxima descrita a seguir. Caso esta diagonal não esteja nesta faixa, a conversão da cadeia S na cadeia P irá gastar mais do que k erros.

No problema de casamento de cadeias aproximado, a faixa de diagonais calculadas em um passo e é de $-\min(m, e)$ até n, porque como no problema de casamento de cadeias aproximado as inserções antes do padrão não contam, podemos começar em qualquer diagonal de 0 até n no passo 0, e portanto, podemos atingir qualquer uma destas diagonais em qualquer passo.

Mas como no nosso problema estas inserções contam, as diagonais com $d > k$ serão agora atingíveis com mais do que k erros, e com isso, devemos restringir esta faixa para $-\min(m, k)$ até $\min(n, k)$ no último passo[5]. Com isso, se estivermos num

[5]Novamente, o min nos limites é necessário para evitar que usemos diagonais inexistentes na matriz.

passo p em que somente estamos interessados nas diagonais que possam ser atingidas com p erros, a faixa ao invés de ser de $-\min(p,m)$ até n, será de $-\min(p,m)$ até $\min(p,n)$. Com isso, como $p \leq k$, o número de passos totais, e portanto, o tempo de execução de pior caso será de $1+3+5+\cdots+(2k+1) = O(k^2)$ passos[6]. Além disso, devemos parar no passo x onde encontramos a distância de edição das cadeias S e P que será igual a x, pois após este passo, não é necessária a realização de nenhum trabalho adicional.

Agora, como usaremos a árvore de sufixos e a busca pelo último ancestral comum, após um tempo de $O(m+n)$ onde construímos a árvore de sufixos para a cadeia $SP\$$ e depois a pré-processamos, o tempo total será de $O(k^2)$, pois cada passo poderá ser executado num tempo constante. Portanto, o tempo total do algoritmo será de $O(k^2+m+n)$, e o espaço total será de $O(k^2+m+n)$, se desejamos obter um script de edição, e de $O(k+m+n)$ caso contrário.

Para obtermos o script de edição, devemos executar os seguintes passos dados abaixo, após executarmos o algoritmo descrito acima que irá computar a distância de edição de S e P, onde as informações dos pontos de término dos caminhos armazenadas serão usadas:

1. Como a distância de edição entre S e P nos dá o último passo executado pelo algoritmo, seja e o valor desta distância de edição, e seja $d = n-m$ a diagonal que contém o ponto (m,n).

2. Se terminamos no primeiro passo do algoritmo ($e = 0$), então nenhum erro foi cometido ao convertemos a cadeia S na cadeia P. Isso significa que as duas cadeias não só são idênticas, como os seus tamanhos são iguais ($m = n$). Então, neste caso, a cadeia que representa o script de edição[7] será composta por m (ou n) caracteres C.

3. Caso não tenhamos terminado no primeiro passo, construiremos a cadeia que representa o script através dos passos dados abaixo. A cadeia é inicializada com uma cadeia vazia.

4. Seja i o ponto de entrada do caminho-e na diagonal d, que será o ponto de entrada da extensão do caminho-$(e-1)$ que termina num ponto com a maior

[6]Esta é a maior soma possível, para o caso em que $k \leq m$ e $k \leq n$.

[7]Usaremos o alfabeto $\Sigma = \{C, D, I, S\}$ do capítulo 2 na construção da cadeia.

linha, e seja f o ponto de término deste caminho em d. Adicionamos $f - i$ símbolos de casamento C, no início da cadeia atual[8].

5. Após isso, determinamos qual dos possíveis caminhos-$(e-1)$ foi estendido, o que poderá ser obtido pelos pontos de término dos caminhos-$(e-1)$ das diagonais $d-1$, d e $d+1$, descobertos no passo $e-1$ anterior. Se este caminho for uma extensão do caminho-$(e-1)$ da diagonal $d-1$ (o caminho R_1), adicionamos o símbolo I no começo da cadeia. Da mesma forma, adicionaremos um símbolo S no início da cadeia se o caminho-$(e-1)$ estiver na diagonal d (o caminho R_2), ou um símbolo D se este caminho veio da diagonal $d+1$ (o caminho R_3).

6. Caso seja possível partimos de mais de um caminho-$(e-1)$ para atingirmos o mesmo ponto i em d no passo anterior, devemos, assim como no capítulo 2, escolher qualquer um dos caminhos, pois como vimos, é necessário um tempo exponencial para obter todos os scripts de edição.

7. Seja x a diagonal em que o caminho-$(e-1)$ escolhido nos passos anteriores termina. fazemos $d = x$, $e = e - 1$ e saltamos para o passo 4 se o novo valor de e é maior do que 0. Caso este valor seja igual a 0, d deverá ser igual a 0, pois devemos necessariamente começar o script na célula $(0, 0)$ da diagonal 0, e o passo 0 não permite a ocorrência de erros. Para terminarmos a descrição do script, se f for o ponto de término deste caminho-0, inserimos f símbolos de casamento C no início da cadeia. Após isso, terminamos o algoritmo, pois já obtemos a cadeia representando o script que converte a cadeia S na cadeia P.

Como a busca pelo último caminho-$(e-1)$ que foi estendido para gerar o último caminho-e a chegar a uma dada diagonal d num passo e, nos passos de 3 até 6 acima, é constante, pois somente reconstituimos um dos scripts de edição, e como temos no máximo $O(k)$ possíveis passos distintos, o tempo total para reconstruir o script será de $O(k)$, e não influênciará o tempo de execução de $O(k^2)$. Portanto, o tempo total de execução ainda será de $O(k^2 + m + n)$.

[8]Para podermos manter o tempo de execução de $O(k)$ na implementação podemos indicar, usando um caractere especial, o número de casamentos após este caractere ao invés de representá-los explicitamente. Isso também permitirá que a execução do script seja feita num tempo total de $O(k)$.

5.3 Os Algoritmos para Calcular os Palíndromos Aproximados

Nesta seção vamos descrever os dois algoritmos construidos para resolver o problema de obter todos os palíndromos aproximados pares e ímpares centrados em todas as possíveis posições da cadeia, usando uma variação do algoritmo para o problema de k-diferenças baseado em árvores de sufixos e na busca pelo último ancestral comum descrito na seção anterior. Também podemos usar o algoritmo descrito no começo da seção anterior.

Começaremos a nossa descrição apresentando um algoritmo de alto nível que poderá usar algum dos algoritmos descritos posteriormente nesta seção para obter, dadas duas cadeias S de tamanho igual a m e P de tamanho igual a n, uma conversão de um prefixo de S num prefixo de P com no máximo k erros, com o maior valor possível para a soma $i + j$, onde i é o tamanho do prefixo de S e j é o tamanho do prefixo de P.[9] Se existir mais de uma conversão com o mesmo valor para a soma $i + j$, a melhor será aquela com o menor número de erros. Vamos chamar esta conversão de a *melhor conversão* de um prefixo de S num prefixo de P. Os algoritmos dados posteriormente irão retornar estes tamanhos i e j, juntamente com o número $e \leq k$ de erros necessário para converter este prefixo de S no prefixo de P.

Agora, o objetivo da tese é o de encontrar todos os palíndromos aproximados maximais (pares e ímpares) com no máximo k erros centrados em todas as possíveis posições de uma cadeia S de tamanho igual a N. Lembrando da definição de palíndromos aproximados e de palíndromos aproximados maximais, isso significa obter, usando um dos algoritmos descritos posteriormente nesta seção, a melhor conversão de um prefixo de S_e com tamanho igual a i num prefixo de S_d com tamanho igual a j, com no máximo k erros, com o maior valor possível para a soma $i + j$. Lembremos também que se desejamos obter um palíndromo par centrado em uma posição c de S, $S_e = S[1..c]^R$ [10] e $S_d = [c + 1..N]$, e que se desejamos obter um palíndromo ímpar centrado nesta mesma posição c, $S_e = S[1..c - 1]^R$ e $S_d = [c + 1..N]$. Se desejarmos obter palíndromos complementares aproximados, basta usarmos a cadeia complementar de S_d, em que trocamos todos os caracteres que representam as bases pelos caracteres complementares.

[9] Estas também serão as posições de término dos prefixos nas cadeias.

[10] Lembre-se de que S^R é a cadeia S revertida.

Além disso, como existem palíndromos aproximados pares somente nas posições de 1 até $N-1$ de S, e ímpares somente nas posições de 2 até $N-1$,[11] podemos usar o seguinte algoritmo para obter todos os palíndromos aproximados pares e ímpares centrados em todas as possíveis posições da cadeia S, num tempo de $O(xN)$, onde x é o tempo de execução das adaptações dos algoritmos de k-diferenças descritos posteriormente, pois podemos reverter a cadeia S antes de começarmos o algoritmo dado num tempo de $O(N)$, para calcularmos S_e a partir de S^R da seguinte forma: se $S_e = S[1..c]^R$, então $S_e = S^R[N-c+1..N]$, e se $S_e = S[1..c-1]^R$, então $S_e = S^R[N-c+2..N]$. Além disso, ao contrário da seção anterior, deveremos sempre executar os algoritmos, e não somente quando o ponto (m,n) estiver dentro dos limites definidos pelos algoritmos.

Algoritmo para obter todos os palíndromos aproximados em uma cadeia

Começo
    Para c de 1 até $N-1$ faça
        /* Obtém primeiramente o palíndromo aproximado par na posição c */
        $S_e = S[1..c]^R$
        $S_d = S[c+1..N]$
        Calcular a melhor conversão, com no máximo k erros, entre os prefixos de S_e e de S_d.
        Seja i a posição do término do prefixo de S_e usado.
        Seja j a posição do término do prefixo de S_d usado.
        Seja e o número de erros desta conversão.
        Imprimir "Há um palíndromo aproximado par centrado em ", c
        Imprimir " com ", e, " erros, começando na posição ", $c-i+1$
        Imprimir " e terminando na posição ", $c+j$, " de S."
        /* Obtém o palíndromo aproximado ímpar centrado em c, se $c > 1$ */
        Se $c > 1$ então
            $S_e = S[1..c-1]^R$
            Calcular a melhor conversão, com no máximo k erros, entre os prefixos de S_e e de S_d.
            Seja i a posição do término do prefixo de S_e usado.
            Seja j a posição do término do prefixo de S_d usado.
            Seja e o número de erros desta conversão.
            Imprimir "Há um palíndromo aproximado ímpar centrado em ", c
            Imprimir " com ", e, " erros, começando na posição ", $c-i$
            Imprimir " e terminando na posição ", $c+j$, " de S."
        Fim
    Fim
Fim

[11] As posições da cadeia S não usadas gerariam palíndromos aproximados com um dos lados totalmente vazios, pois ou $S_e = \varepsilon$, ou $S_d = \varepsilon$.

Se desejarmos obter palíndromos complementares aproximados ao invés de palíndromos normais, basta mudarmos, no algoritmo acima, a definição de $S_d = S[c+1..N]$ para $S_d = S[c+1..N]^C$.[12] Assim como na reversão da cadeia S, a complementação desta cadeia pode ser executada num tempo de $O(N)$ antes da execução do algoritmo.

Vamos agora descrever como adaptar os algoritmos descritos na seção anterior para o nosso problema. Como para podermos obter um palíndromo aproximado em uma posição c da cadeia S de tamanho igual a N devemos obter a melhor conversão de um prefixo de S_e num prefixo de S_d, deveremos adaptar os algoritmos para obterem esta melhor conversão. Seja m o tamanho de S_e e n o tamanho de S_d, ou seja, seja $m = c$ se o palíndromo for par e $m = c - 1$ se o palíndromo for ímpar, e seja $n = N - c$.

No primeiro algoritmo dado na seção anterior, que terá um tempo de execução de pior caso de $O(kN)$, pois no pior caso $m = O(N)$, devemos, após calcular cada célula (i, j), verificar se o número de erros necessários para a conversão do prefixo $S_e[1..i]$ no prefixo $S_d[1..j]$ é menor ou igual a k. Se isso ocorrer, devemos verificar se a soma $i + j$ é melhor do que a melhor soma atual encontrada, guardada em uma estrutura *melhor* juntamente com o número de erros z desta melhor soma, e dos valores x e y que geraram esta soma. Se o valor da soma for igual, e se o número de erros da conversão atual for maior ou igual a z, ignoramos esta conversão, pois ou ela simplesmente será uma outra possível melhor conversão até o momento, ou não será uma melhor conversão. Se o valor da soma for maior, ou se ele for igual e o número de erros da conversão for menor do que z, atualizamos a estrutura *melhor* com o valor desta soma, e com os valores de i, j e e, onde $e = D(i, j)$, pois esta será agora a melhor conversão até o momento. O valor da estrutura *melhor* é inicializado com o valor 0 para o campo *soma*. Após terminarmos o processamento do algoritmo, a estrutura *melhor* terá os tamanhos dos prefixos que geraram a melhor conversão. Retornamos então os tamanhos i e j dados na estrutura que indicarão, respectivamente, as posições de término dos prefixos de S_e e S_d, juntamente com o número de erros e desta conversão.

Usando a adaptação dada acima, o tempo total para calcular todos os palíndromos aproximados pares e ímpares centrados em todas as possíveis posições de c será, considerando que m varia de 1 até $N - 1$ para os palíndromos pares, e de

[12] S^C representa a cadeia S complementada.

1 até $N-2$ para os palíndromos ímpares, de $O(k+2k+\cdots+(N-1)k) = O(kN^2)$ para os palíndromos aproximados pares, e de $O(k+2k+\cdots+(N-2)k) = O(kN^2)$ para os palíndromos aproximados ímpares. Portanto, o tempo total será de $O(kN^2)$.

Se desejarmos obter o script de edição σ, basta remontar o caminho da célula (i,j) onde encontramos a melhor conversão até a célula $(0,0)$, no primeiro algoritmo da seção anterior, ao invés da célula (m,n). Se desejarmos guardar este script para cada palíndromo, o espaço total será de $O(kN+N^2) = O(N^2)$,[13] pois como o tamanho do script pode ser de $O(N)$ [14] no pior caso, o espaço necessário para um centro (posição da cadeia) será de $O(N)$ para um palíndromo aproximado. Entretanto, se somente desejarmos guardar as informações da posição inicial, da posição final, e do número de erros, o espaço total será de $O(kN+N) = O(kN)$, pois é necessário um espaço constante para armazenarmos três valores inteiros.

Ao interpretarmos o script de edição σ no nosso caso, é mais conveniente interpretar o caractere D do script como uma inserção de um caractere em $S_d[1..j]$ ao invés de uma deleção de um caractere de $S_e[1..i]$, onde i e j são os tamanhos dos prefixos da melhor conversão[15], ou seja, se no processamento atual estivermos na posição x de $S_e[1..i]$ e y de $S_d[1..j]$, inserimos o caractere $S_e(x)$ antes do caractere $S_d(y)$. Com isso, após a execução do script, os prefixos $S_e[1..i]$ de S_e, e $S_d[1..j]$ de S_d serão convertidos para uma cadeia S' que dará origem a um palíndromo exato. Se o script é obtido ao construirmos um palíndromo aproximado par, obteremos um palíndromo exato $S'^R S'$ com o tamanho igual a $2 * |S'|$, e se o script é obtido ao construirmos um palíndromo aproximado ímpar, obteremos o palíndromo exato $S'^R S(c) S'$ com o tamanho igual a $2 * |S'| + 1$.

Para adaptarmos o último algoritmo descrito na seção anterior, vamos novamente definir uma estrutura *melhor* que guardará a melhor soma achada até o momento, juntamente com os tamanhos dos prefixos de S_e e de S_d que geraram esta soma, e do número de erros e necessários para converter um prefixo no outro. Agora, após calcularmos o valor do ponto máximo que podemos atingir em uma diagonal d num passo e (o ponto onde o último caminho-e a chegar termina), devemos verificar se a soma dos tamanhos dos prefixos de S_e e S_d para esta conversão, que são iguais a i e $i+d$ respectivamente[16], se i for o ponto máximo, é maxima, ou seja, se o valor dado

[13] Não há sentido termos $k > N$, pois $m < N$ e $n < N$.

[14] Na implementação, usar a idéia descrita anteriormente irá reduzir este tamanho para $O(k)$.

[15] Essa medida faz com que o tamanho do palíndromo exato também seja máximo.

[16] Lembre-se de que uma célula (i,j) pertence a diagonal $d = j - i$.

por $2 * i + d$ é maior do que o valor da melhor soma atual dada na estrutura *melhor*. Neste caso, devemos ignorar a conversão se a soma for menor ou igual, pois caso a soma seja igual, o número de erros da conversão deverá ser maior ou igual do que o da melhor conversão atual, pois esta melhor conversão foi gerada ou neste passo, ou num passo anterior. Portanto, se o valor desta soma for maior, devemos atualizar a estrutura *melhor* com o valor da soma, e com os valores de i, $i + d$ e e. Assim como no algoritmo anterior, após o término da execução, a estrutura *melhor* terá os tamanhos dos prefixos da melhor conversão, e terá também o número de erros usados nesta conversão, e portanto, devemos retornar estes valores como resultado da execução do algoritmo.

O tempo de execução deste algoritmo seria de $O(k^2+N)$, pois o tempo necessário para construir a árvore de sufixos para a cadeia $S_e S_d\$$, e pré-processá-la para obter o último ancestral comum de dois nós é de $O(m+n)$, onde no nosso problema, $m = |S_e|$ e $n = |S_d|$. Com isso, o tempo de execução total do nosso algoritmo para este caso seria de $O(k^2N + N^2)$. Mas ao contrário do problema genérico de k-diferenças, em que as cadeias S e P são genéricas, todas as cadeias S_e serão sufixos da cadeia S^R, assim como todas as cadeias S_d serão sufixos da cadeia S (ou da cadeia S^C, no caso de buscarmos por palíndromos complementares). Para obtermos o tempo alegado de pior caso de $O(k^2N)$, devemos executar duas simples alterações no algoritmo descrito na seção anterior, baseadas nesta propriedade do nosso problema, o que reduzirá o tempo de computação de um palíndromo de $O(k^2 + N)$ para $O(k^2)$. Para tornar a explicação mais clara, deve-se notar que se uma cadeia S' terminar na posição x de S, S'^R comecará na posição $N - x + 1$ de S^R. Devemos executar as seguintes alterações:

1. Construir, antes de calcular os palíndromos, uma árvore de sufixos para a cadeia $S^R S\$$, e executar os mesmos pré-processamentos que seriam executados se construíssemos a árvore de sufixos ao calcular cada palíndromo. Se desejarmos calcular palíndromos complementares aproximados, basta construir a árvore para a cadeia $S^R S^C\$$ ao invés da cadeia $S^R S\$$. A determinação do reverso da cadeia S e a sua complementação, caso desejemos obter palíndromos complementares, pode ser feita num tempo total de $O(N)$ antes da construção da árvore de sufixos.

2. Ao buscarmos o ancestral comum de dois sufixos, ao invés de buscarmos o

último ancestral comum de i e de $j+m$ para os sufixos $S_e[i..m]$ e $S_d[j..n]$, procuramos pelo último ancestral comum de $N-c+i$ e de $c+j+N$, se procuramos por um palíndromo aproximado par maximal, e pelo ancestral de $N-c+i+1$ e de $c+j+N$, se procuramos por um palíndromo aproximado ímpar maximal. Além disso q, o tamanho do nome do caminho deste ancestral, deverá ser limitado pelo tamanho do sufixo de S_e, pois como construímos uma árvore de sufixos para $S^RS\$$, novamente esta limitação é para evitarmos considerar caracteres além do final de S^R onde S_e deve terminar. Não precisaremos limitar este valor pelo tamanho de S_d, pois esta cadeia terminará no final da cadeia S que dá o final dos sufixos da cadeia S^RS. Então, o tamanho do maior prefixo comum destas cadeias será igual a $\min(q, m-i+1)$. Portanto, o algoritmo de alto nível descrito no início desta seção deverá passar para este algoritmo, além de S_e e de S_d, o tamanho N da cadeia S, o centro atual c, as estruturas necessárias a execução do *lca* num tempo constante, e se o palíndromo que desejamos é par ou ímpar.

Para podermos obter os scripts de edição, basta executarmos os passos descritos na seção anterior para a diagonal d em que a melhor conversão foi encontrada, e iniciarmos no passo e em que obtivemos esta melhor conversão, ao invés da diagonal e do passo em que atingimos o ponto (m,n). O significado do script é o mesmo descrito anteriormente, e o seu tamanho máximo para um palíndromo também será de $O(N)$.

Além disso, deveremos sempre executar o algoritmo, ao contrário da seção anterior em que não executávamos o algoritmo se o ponto (m,n) não estivesse dentro da faixa de diagonais, e devemos parar a execução do algoritmo após atingirmos o ponto m da diagonal que contém a célula (m,n), pois como $m+n$ é a maior soma possível, não encontraremos melhores conversões do que a atual nos próximos passos do algoritmo.

Com isso, podemos enunciar o seguinte teorema:

Teorema 5.6 *Todos os palíndromos aproximados maximais (pares e ímpares) com no máximo k erros centrados em todas as possíveis posições de uma cadeia S de tamanho igual a N podem ser obtidos num tempo total de $O(k^2N)$, e num espaço de $O(k+N)$ se não desejamos os scripts, e de $O(k^2+N^2)$, se desejamos todos os scripts de edição.*

Agora vamos descrever duas otimizações simples do algoritmo descrito anteriormente, que apesar de não reduzirem a complexidade de pior caso de $O(k^2N)$, podem reduzir o tempo de execução em muitos casos, como poderá ser visto nos exemplos dados no próximo capítulo. Nas duas próximas subseções, vamos mostrar duas otimizações aplicadas ao último algoritmo descrito nesta seção, e nas descrições das subseções a seguir, m é o tamanho da cadeia S da qual é obtido um prefixo que será convertido no prefixo obtido da cadeia P com tamanho igual a n, com o número de erros limitado por k.

5.3.1 Primeira Otimização

A primeira otimização, que gerou o primeiro algoritmo que construímos, é baseada no fato de que após atingirmos o ponto máximo de uma diagonal d, esta diagonal não poderá gerar mais melhores conversões, pois mesmo quando a sua soma for máxima, se outros caminhos a atingirem no ponto máximo em um passo posterior, estes não poderão gerar melhores conversões, pois o número de erros destas conversões será maior do que da primeira conversão para a qual atingimos o último ponto desta diagonal d.

Além disso, no passo seguinte ao atingirmos este ponto máximo, um caminho de d atingirá o ponto máximo ou de sua vizinha $d+1$ ser for possível executar uma inserção, ou de sua vizinha $d-1$ se for possível executar uma deleção, e com isso, esta diagonal d também não precisará ser mais usada nos passos posteriores como a origem de uma extensão de um caminho, como pode ser visto pela figura 5.4. Com isso, após atingirmos o ponto máximo de uma diagonal d num passo e, só a usaremos no passo posterior. Chamaremos a diagonal que atingiu o seu ponto máximo de *diagonal invalidada*. O ponto máximo de uma diagonal pode ser determinado pela fórmula dada abaixo:

$$PontoMaximo(d) = \begin{cases} m & \text{se } m+d \leq n \\ n-d & \text{se } m+d > n \end{cases}$$

Se algum caminho computado pela primeira vez em um passo posterior passar pelo último ponto de uma diagonal invalidada, ele deverá ter passado por este ponto exatamente com o número de erros igual ao número do passo em que a diagonal foi invalidada, pois se este caminho passasse por este ponto máximo num passo posterior, a porção deste caminho até este ponto teria mais erros, e o valor da célula (i,j) em que ele termina não teria o número mínimo de erros para converter $S[1..i]$

em $P[1..j]$, o que é absurdo, pois esta célula deve ter o número mínimo de erros para converter este prefixo de S no prefixo de P.

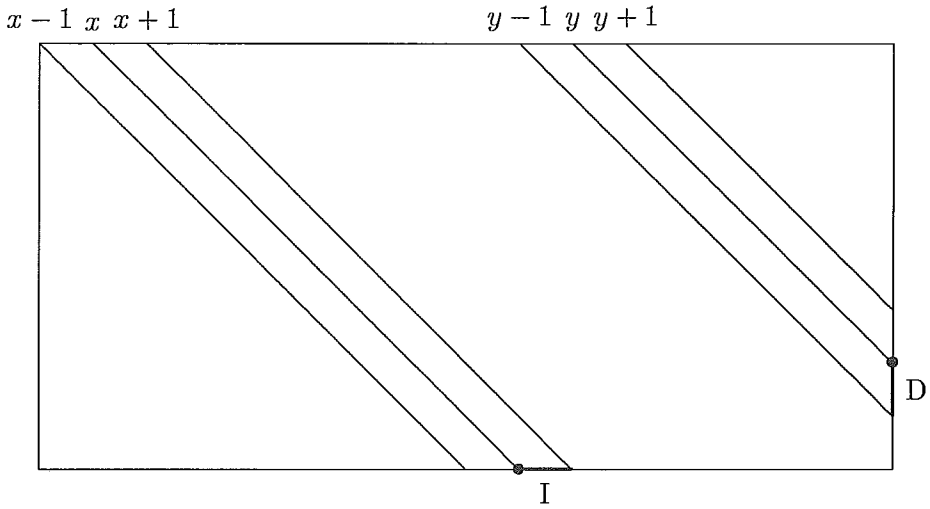


Figura 5.4: Como atingimos os pontos finais das diagonais x e y, elas não serão usadas, após o próximo passo, por nenhum caminho que gere uma conversão. Além disso, como no próximo passo atingiremos o ponto máximo de $x+1$ e de $y-1$, estas diagonais também serão invalidadas, e não dependerão mais de x e y, respectivamente. Portanto, após invalidarmos as diagonais x e y, não precisaremos mais calcular os últimos caminhos-e a chegar a elas, ou seja, podemos reduzir o trabalho realizado não executando mais nenhuma computação sobre estas diagonais.

Com isso, podemos executar a seguinte alteração no algoritmo descrito anteriormente: ao invés de termos um laço em que computamos todas as diagonais de $-\min(m,e)$ até $\min(n,e)$ para um passo e, teremos uma lista duplamente ligada que conterá as diagonais que deverão ser computadas neste passo, ordenadas pelos números das diagonais. A ordenação das diagonais, que será usada pelo próximo algoritmo, não causará nenhum problema, pois ao passarmos do passo e ao passo $e+1$ ($e < k$), devemos incluir as diagonais $-e-1$ e $e+1$, que não estavam presentes nos passos anteriores. Para manter a ordenação, basta inserir a diagonal $-e-1$ no início da lista, se $e+1 \leq m$, e a diagonal $e+1$ no final da lista, se $e+1 \leq n$. Agora, se após computarmos o último caminho-e a chegar a uma diagonal d num passo e atingirmos o ponto máximo desta diagonal, retiramos esta diagonal da lista, pois como vimos, não precisaremos calcular os outros últimos caminhos a atingirem d nos passos posteriores. A retirada desta diagonal pode ser feita num tempo constante, pois podemos retirar um elemento de qualquer posição de uma lista duplamente

ligada num tempo constante.

5.3.2 Segunda Otimização

O próximo algoritmo é baseado no fato de que se existem várias diagonais invalidadas após a execução de um passo, podemos restringir, como podemos ver pela figura 5.5, a computação das diagonais dentro de uma faixa limitada pela última diagonal invalidada L_i (com o maior número possível) que atinge o limite inferior da matriz de programação dinâmica (atinge o limite na maior coluna), e pela primeira diagonal L_f (com o menor número possível) a atingir o limite direito desta matriz de programação dinâmica (atinge este limite na maior linha). Pela figura 5.5, vemos que todas as diagonais d fora da faixa terão, para todos os seus pontos (x, y), uma soma $x + y$ menor do que a soma dada pelas coordenadas do ponto A da diagonal L_i, isto é, $x + y < 2 * m + L_i$, se $d < L_i$, e menor do que a soma das coordenadas do ponto B da diagonal L_f, ou seja $x + y < 2 * n - L_f$, se $d > L_f$. Portanto, ao alterarmos o valor da melhor conversão neste passo ou num passo posterior, a diagonal com a melhor conversão deverá estar dentro desta faixa. Além disso, como as diagonais sobre o limite estão invalidadas, nenhum caminho de fora da faixa poderá influenciar os caminhos de dentro da faixa num passo posterior.

Portanto, no próximo passo, não precisamos computar nenhuma das diagonais que estão fora da faixa, ou seja, nenhuma diagonal d para a qual $d < L_i$ ou $d > L_f$, além dos limites que também estão invalidados, e também não precisaremos inserir, no próximo passo p, a diagonal $-p$, se $p \leq m$, pois ela teria o menor número, e portanto, seria menor do que L_i, assim como não precisamos inserir a diagonal p, se $p \leq n$, pois claramente seu número seria maior do que L_f.

Se não existir o limite inferior, ou seja, se ou não for possível atingir o limite inferior da matriz, isto é, se não existem diagonais terminando neste limite, ou se não existirem diagonais invalidadas terminando sobre este limite, não teremos o limite inferior, e neste caso, não deveremos excluir nenhuma das diagonais que terminam sobre o limite inferior, e nem deixar de inserir, no próximo passo p, a diagonal $-p$, se $p \leq m$. Da mesma forma, se não existir um limite superior, ou seja, se não existir uma diagonal que termine no limite direito da matriz, ou se todas as diagonais que terminam sobre este limite não estão invalidadas, não deveremos excluir da execução as diagonais que terminam sobre o limite direito, e nem deixar de inserir, no próximo passo p, a diagonal p, se $p \leq n$.

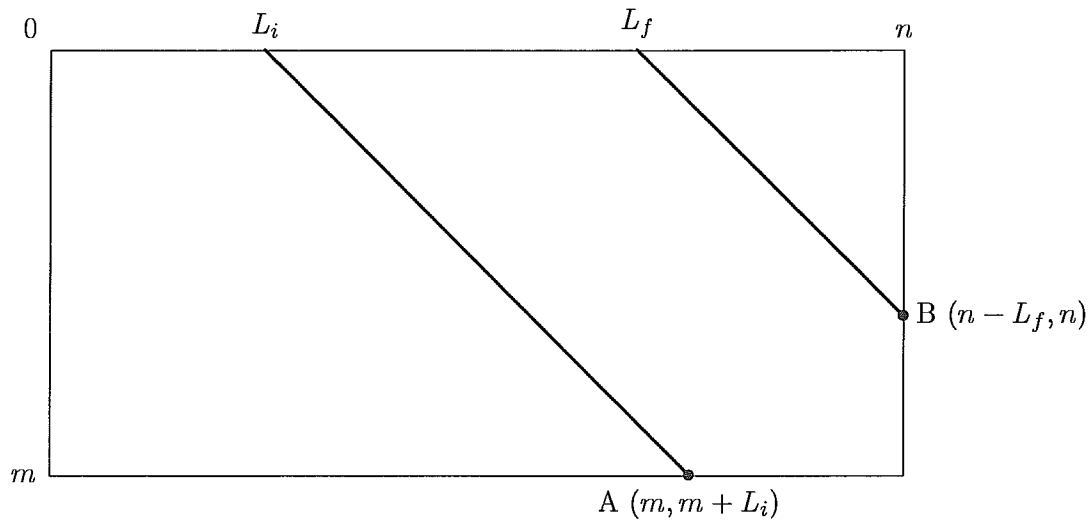


Figura 5.5: Como podemos ver pela figura, todas as diagonais d, para as quais ou $d < L_i$, ou $d > L_f$, terão a soma de todos os seus pontos (x, y) menores do que a da melhor conversão atual, pois esta soma será no mínimo igual a maior das somas dos pontos máximos dos limites, e a soma máxima dos pontos (x, y) de d será menor do que a soma das coordenadas do ponto máximo A de L_i, se $d < L_i$, ou do ponto máximo B de L_f, se $d > L_f$.

Neste caso, a retirada das diagonais da lista duplamente ligada, que não serão computadas no passo posterior, pode ser feita num tempo proporcional ao tamanho da lista, que é igual a $O(k)$ no pior caso, através do caminhamento pelos nós desta lista, onde eliminamos o nó atual (o que pode ser feito num tempo constante) se o número da diagonal d que este nó representa for menor ou igual do que o limite inferior L_i, se este existir, ou maior ou igual do que o limite superior L_f, também se este existir. Este processo, que é executado após o final do passo p, pode ser executado no mesmo tempo do que o da computação dos pontos em que terminam os últimos caminhos-p das diagonais neste passo p, que é de $O(p) = O(k)$ no pior caso, pois $p \leq k$. Para que este esquema seja mais eficiente, a lista deve estar ordenada pelos números das diagonais, mas como vimos na descrição da otimização na subseção anterior, a ordenação não causará nenhum problema, pois podemos manter a lista ordenada sem nenhum custo adicional. A ordenação facilitará a deleção dos nós da lista, se mantivermos um ponteiro para os nós que correspondem aos limites, se estes existirem.

5.4 Métodos para Paralelizar o Algoritmo

Em [26] é apresentado um método para paralelizar o algoritmo para o casamento de cadeias aproximado com tempo de execução seqüencial de $O(kn)$, e tempo de pré-processamento de $O(n+m) = O(n)$, onde n é o tamanho do texto T, m é o tamanho do padrão P, e k é o número máximo de erros permitido.

Na fase de pré-processamento, construiremos a árvore de sufixos em pararelo num tempo de $O(\log n)$ com $O(n)$ processadores[17], usando o algoritmo descrito em [5, 24]. A fase de pré-processamento da árvore de sufixos, para permitir a resolução de perguntas *lca* num tempo constante, usará o algoritmo descrito em [35], que irá pré-processar a árvore de sufixos com $O(n)$ nós num tempo de $O(\log n)$ usando $O(n/\log n)$ processadores. Após esta fase de pré-processamento, poderemos resolver uma pergunta *lca* num tempo de $O(1)$ usando um processador.

Agora, notemos que ao computarmos os pontos de término dos últimos caminhos-e a chegarem sobre as diagonais d, $-\min(e,m) \leq d \leq n$, num passo e, $e \leq k$, somente precisamos dos valores dos pontos de término dos caminhos-$(e-1)$ em $d-1$, d e $d+1$ se existirem, calculados no passo $e-1$ anterior. Com isso, em um passo e, não há dependência entre as computações dos pontos de término dos caminhos que terminam sobre as diagonais. Então a idéia é a de que, após a fase de pré-processamento, que poderá ser executada num tempo total de $O(\log n)$ e $O(n)$ processadores, computemos todas as diagonais de um passo em paralelo, o que permitirá a execução deste passo num tempo constante.

Cada processador ficará, em todos os passos, responsável pela computação de uma diagonal d, $-\min(m,k) \leq d \leq n$. Suponha que, no passo e, o processador que fica responsável pela diagonal d tenha todas as informações necessárias para computá-la, ou seja, os pontos de término dos últimos caminhos-$(e-1)$ a chegarem em $d-1$, d e $d+1$. O processador então computa, usando o algoritmo para responder perguntas *lca* num tempo constante e os últimos caminhos-$(e-1)$, o ponto de término do último caminho-e a chegar em d. Após isso, como os processadores que calculam $d+1$ e $d-1$ precisarão deste ponto de término no próximo passo, enviar para estes processadores o valor deste ponto de término. Se $d < 0$, o processador responsável pelo cálculo da diagonal d só deverá se tornar ativo no passo $|d|$, pois antes deste passo d, temos menos do que d erros, e a diagonal d é atingível no mínimo com d

[17]Este limite de tempo vale tanto para o alfabeto constante, como para o alfabeto variável.

erros.

Como todas as diagonais de um passo são computadas em paralelo, o tempo de computação de um passo será constante. Com isso, o tempo de execução será reduzido de $O(kn)$ para $O(k)$. Como a faixa de diagonais calculadas varia de $-k$ até n no máximo, serão necessários $O(k+n) = O(n)$ processadores para executar um passo num tempo constante. Com isso, o algoritmo poderá ser executado num tempo total de $O(k + \log n)$ com $O(n)$ processadores.

Agora, no nosso algoritmo, sendo N o tamanho da cadeia S da qual desejamos obter todos os palíndromos aproximados, podemos, usando os mesmos algoritmos paralelos do problema de casamento de cadeias aproximado, construir a árvore de sufixos para a cadeia $S^R S\$$ (ou $S^R S^C\$$ se desejamos obter palíndromos complementares aproximados), e a pré-processá-la para respondermos as perguntas *lca* num tempo constante, num tempo total de $O(\log N)$ e $O(N)$ processadores. Assim como antes, esta fase de pré-processamento será executada antes do cálculo dos palíndromos aproximados. Além disso, como num passo p, a faixa de diagonais, no pior caso, será de $-p$ até p, e como os limites desta faixa são atingíveis somente neste passo p, o processador responsável pelo cálculo de uma diagonal d somente se tornará ativo do passo $|d|$ em diante.

Ao calcularmos um palíndromo, a faixa máxima de diagonais a serem calculadas é de $-k$ até k, e portanto, precisaremos de $O(k)$ processadores para permitir que um passo seja computado num tempo constante. Se desejamos obter o script de edição σ, podemos usar um processador a mais que, a cada passo e, receberá as informações dos pontos de término dos últimos caminhos-e a chegar a todas as diagonais dos outros processadores, e que após o termino da computação da melhor conversão irá calcular, num tempo de $O(k)$, este script de edição, como no algoritmo seqüencial.

Com isso, o tempo de execução de um passo do nosso algoritmo será de $O(k)$. Agora, como a computação de um palíndromo numa posição c da cadeia S não depende da computação de um outro palíndromo numa outra posição d de S, e como temos $N-1$ palíndromos pares e $N-2$ palíndromos ímpares, podemos calcular todos estes palíndromos num tempo total de $O(k)$, o tempo necessário para calcular uma melhor conversão, se usarmos $O(k)$ processadores para calcular cada um dos $O(N)$ palíndromos em paralelo.

Com isso, podemos calcular todos os palíndromos aproximados presentes em uma cadeia S de tamanho igual a N com no máximo k erros, num tempo total de

$O(k+\log N)$, usando $O(kN)$ processadores. O espaço gasto será, no total, o mesmo do caso anterior: $O(k+N)$ se não desejamos obter os scrips de edição, e $O(k^2+N^2)$ se desejamos obter os scrips de edição.

A primeira otimização pode ser facilmente implementada da seguinte forma: ao calcular o ponto de término do último caminho-e a chegar a uma diagonal d, o processador responsável por computá-la, após enviar o valor deste ponto aos processadores responsáveis pelas diagonais $d+1$ e $d-1$, deve verificar se esta diagonal está invalidada. Caso esta diagonal esteja invalidada, ele se tornará inativo, e não calculará mais esta diagonal nos passos posteriores, e neste caso, os processadores responsáveis pelas diagonais $d-1$ e $d+1$ deverão saber que este processador se tornou inativo para não esperarem mais por mensagens vindas deste processador. Caso exista um processador responsável pelo cálculo do script, ele deverá saber se o valor do ponto de término do último caminho atual a chegar em d é o ponto máximo da diagonal, pois se esta diagonal acabou de ser invalidada, ele não receberá mais os pontos de término dos últimos caminhos a chegarem a esta diagonal d nos passos posteriores ao da invalidação, pois o processador que calcula esta diagonal se tornou inativo, e ele não receberá mais mensagens deste processador.

A segunda otimização pode ser implementada da seguinte maneira: após um processador responsável pelo cálculo da diagonal d terminar a computação e enviar as mensagens aos vizinhos, ele deve verificar se esta diagonal está invalidada. Caso esta diagonal esteja invalidada, e termine sobre o limite inferior, ela será um possível limite inferior da faixa de diagonais, e com isso, este processador e todos os processadores responsáveis pelos cálculos das diagonais x, $x < d$, devem estar inativos nos passos posteriores. Portanto, neste caso, o processador enviará uma mensagem para estes processadores informando que eles devem ficar inativos até o final da computação. Caso a diagonal invalidada termine no limite direito, ela poderá ser um possível limite superior, e neste caso, o processador, além de se tornar inativo, deverá enviar uma mensagem para os processadores responsáveis pelo cálculo da diagonal x, $x > d$, informando que eles estarão inativos até o final da computação.

Portanto, ao usarmos faixas de diagonais para limitar a computação de um passo, o processador responsável pela computação da diagonal d somente irá computá-la após o passo atual se esta diagonal não estiver invalidada, e se não receber nenhuma mensagem mandando ele se tornar inativo até o final da computação. Todos os processadores que receberem pelo menos uma mensagem deverão se tornar inativos,

inclusive aqueles que ainda não se tornaram ativos até o passo atual[18]. Além disso, se existir um processador responsável pela computação do script de edição, ele deverá saber, ao final de um passo e, qual é a faixa de diagonais ativas que existirá no próximo passo, para saber quais diagonais estarão ativas no próximo passo, e com isso, saber de quais processadores ele irá receber informações dos pontos de término dos caminhos.

[18]Isso evitará a inserção das novas diagonais no próximo passo.

Capítulo 6

Análise do Tempo de Execução

Neste capítulo, iremos verificar como o tempo de execução das otimizações descritas no capítulo anterior variam, em relação ao tempo de execução máximo, de acordo com os vários exemplos executados, que variam tanto no tipo da cadeia, como no tamanho do alfabeto em que baseamos estes exemplos. Também computaremos o tempo de execução para uma cadeia toda diferente, que dará o maior tempo de execução possível, e para uma cadeia toda igual, que dará o menor tempo de execução possível. O objetivo desta análise, que será vista na seção 6.1, é o de mostrar que apesar de o tempo de execução de pior caso não ser reduzido pelas nossas otimizações, teremos reduções de tempo num caso genérico, que podem ser significativas.

Além disso, definimos um modelo probabilístico que dará o número médio de passos executados, e a área média esperada para uma dada posição da cadeia S, um dado tamanho de alfabeto, e um dado número de erros. Usaremos este modelo para calcular o número médio de passos, e a área média esperada para toda a execução do algoritmo, para avaliar os ganhos médios, e depois compararemos este resultado com a área e o número de passos obtidos em uma execução real do algoritmo, para determinar o grau de precisão do nosso modelo.

6.1 Análise do Tempo Esperado de Execução

Nesta seção, iremos verificar a eficiência das nossas duas otimizações em reduzirem o tempo de execução ao serem aplicadas sobre exemplos concretos. Antes de começarmos a descrição dos gráficos e sua análise, vamos descrever quais os tipos de exemplos que foram usados pelas execuções reais, e como é verificada a eficiência destas otimizações. Também mostraremos visualmente a redução do tempo de com-

putação de um palíndromo em uma dada posição da cadeia S.

Para analisarmos a redução do tempo de execução das duas otimizações em relação a execução sem o uso de nenhuma das duas otimizações, dada pela conversão direta do algoritmo de k-diferenças aplicada ao nosso caso, iremos contar o número total de passos ao executarmos estas duas otimizações, e compará-lo com o número de passos máximo necessário para calcular todos os palíndromos desta cadeia, ao usarmos esta conversão direta, sem pararmos ao invalidarmos a diagonal que contém o ponto com a soma máxima. Esta forma de medição do tempo foi escolhida porque ela facilita a análise da complexidade, e devido ao fato de alguns exemplos demorarem muito para serem executados, o que nos obrigou a deixá-los rodando em background. Vamos chamar esta forma de medição do tempo de *área total* ao calcularmos os palíndromos, para diferenciá-la do tempo de execução real obtido ao executarmos os algoritmos.

Como usamos a noção de passos, na seção 5.1 do capítulo anterior, como o número de erros dos últimos caminhos a chegarem que estão sendo atualmente calculados pelos algoritmos, vamos chamar de *área* o número de últimos caminhos a chegarem calculados em todos os passos, ao calcularmos um palíndromo, pois cada um destes caminhos poderá ser calculado num tempo constante. Com isso, a área total será a soma de todas as áreas ao computarmos os palíndromos em cada uma das possíveis posições da cadeia S.

Para podermos analisar a influência da escolha de calcularmos os palíndromos pares ou os palíndromos ímpares, analisaremos a área total para cada um destes casos separadamente, ao invés da área total para calcularmos todos os palíndromos, que poderá facilmente ser obtida através da soma das áreas totais para cada um destes casos. Para os palíndromos pares, a área total máxima, dada pelo tempo de execução do algoritmo sem o uso das otimizações, pode ser facilmente calculada através do seguinte método: calcular, para todos os palíndromos pares centrados nos centros c da cadeia S de tamanho igual a N, $1 \leq c < N$, a soma dos valores $A(m, n, k)$ para cada um destes centros, usando a fórmula para A dada abaixo, onde $m = c$, $n = N - c$, e k é o número máximo de erros permitidos. Para calcular a área total máxima para todos os palíndromos ímpares, basta substituirmos o valor dado para m por $m = c - 1$, e lembrar que não temos um palíndromo ímpar na posição 1 da cadeia S no nosso problema.

$$A(m,n,k) = \begin{cases} (k+1)^2 & \text{se } m \geq k \text{ e } n \geq k \\ (k^2-n^2+3k+2kn+n+2)/2 & \text{se } m \geq k \text{ e } n < k \\ (k^2-m^2+3k+2km+m+2)/2 & \text{se } m < k \text{ e } n \geq k \\ (2k+m+n+2km+2kn-m^2-n^2+2)/2 & \text{se } m < k \text{ e } n < k \end{cases}$$

Como para os centros extremos (1 e $N-1$) temos $O(k^2)$ passos $(1+2+\cdots+k+1)$, como temos no máximo $(k+1)^2$ passos, como para estes centros temos o menor número de passos possíveis, e além disso como calculamos $N-1$ palíndromos pares e $N-2$ palíndromos ímpares, a área total máxima para calcular todos os palíndromos pares e ímpares será de $O(k^2N)$. Com isso, a área total máxima para calcular todos os palíndromos será de $O(k^2N)$.

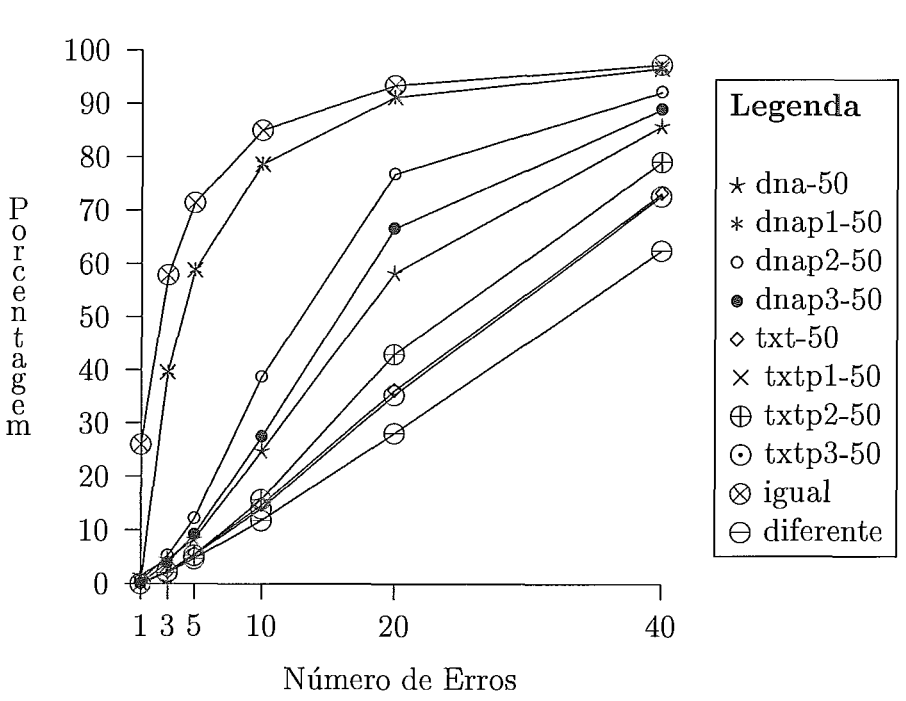


Figura 6.1: Ganho ao calcularmos todos os palíndromos pares para as cadeias com tamanho igual a 50, usando a primeira otimização. As linhas para as cadeias dnap1-50 e txtp1-50 são iguais, pois ambas são cadeias periódicas com período igual a 2.

Executamos os dois algoritmos que implementam as duas otimizações descritas no capítulo 5 sobre dez exemplos, com o tamanho da cadeia S igual a N sendo igual a 50, 100, 500, 1000, 2500 e 5000, com o número de erros k sendo calculado com base no tamanho N das cadeias, sendo igual ao teto de 1%, 5%, 10%, 20%, 40% e

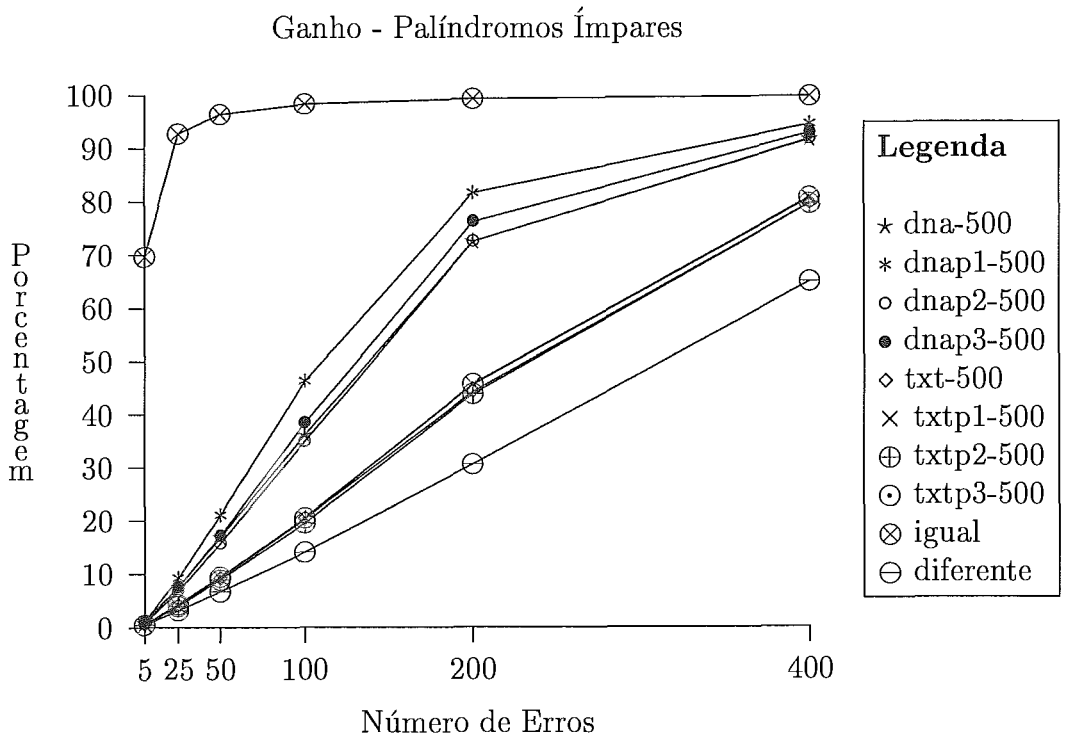


Figura 6.2: Ganho ao calcularmos todos os palíndromos ímpares para as cadeias com tamanho igual a 500, usando a primeira otimização.

80% de N. Teremos quatro cadeias compostas por partes de seqüências de cadeias de DNA, sendo que a primeira é totalmente obtida de uma seqüência, enquanto que as outras três cadeias são cadeias períodicas, onde o período foi obtido de uma cadeia que representa uma seqüência de DNA. Os tamanhos dos períodos são iguais ao piso de 5%, 25% e 50% do tamanho N destas cadeias. Nos gráficos, estas cadeias serão nomeadas pelo prefixo "dna-" para a primeira cadeia, e "dnap1-", "dnap2-", e "dnap3-" para as três cadeias periódicas, onde os prefixos são usados de acordo com o tamanho do período. Após este prefixo, teremos o tamanho destas cadeias. Outras quatro cadeias serão formadas por partes obtidas de textos em inglês ou português, e serão construídas pelo mesmo método usado para formar as cadeias de DNA descrito anteriormente, ou seja, teremos uma cadeia normal e três cadeias periódicas. Os prefixos usados, neste caso, serão "txt-", "txtp1-", "txtp2-" e "txtp3-". Finalmente, teremos uma cadeia com todos os caracteres iguais nomeada por "igual", e uma cadeia com todos os caracteres diferentes nomeada por "diferente".

Mostraremos somente os gráficos dos ganhos em relação a área máxima necessária para calcular um conjunto de palíndromos (pares ou ímpares), e o gráfico do ganho

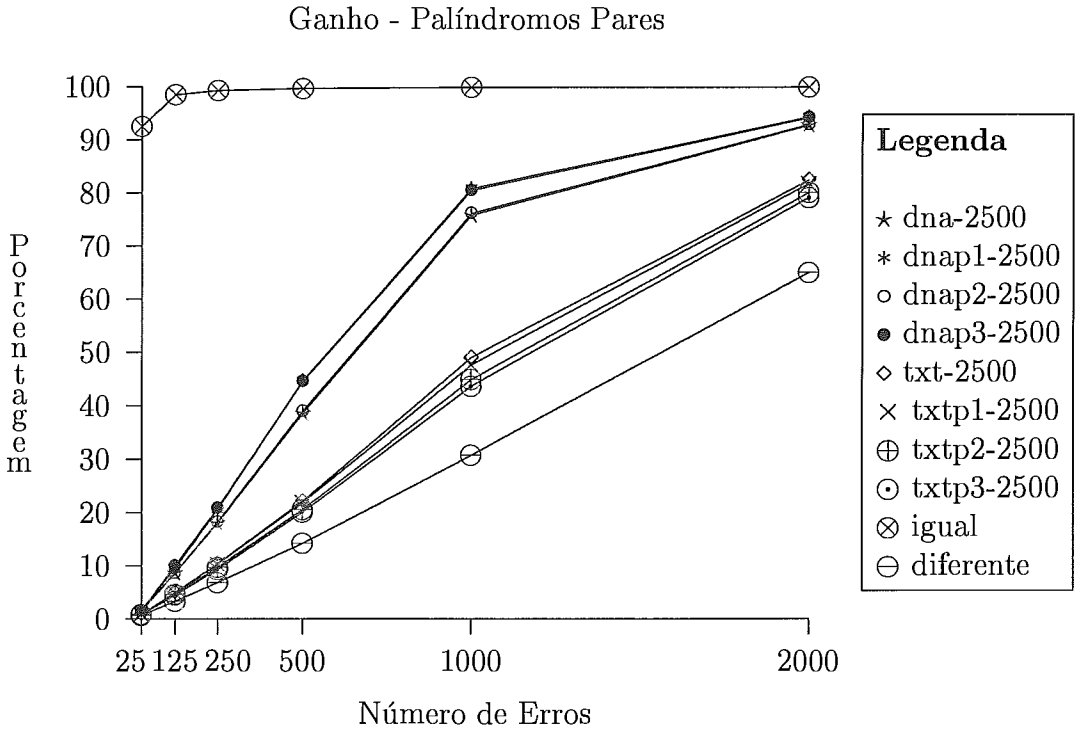


Figura 6.3: Ganho ao calcularmos todos os palíndromos pares para as cadeias com tamanho igual a 2500, usando a primeira otimização.

ao aplicarmos o algoritmo que implementa a segunda otimização em relação a aquele que implementa a primeira otimização. Entretanto, não mostraremos as tabelas com as áreas totais neste capítulo, mas estas tabelas podem ser vistas no apêndice A. Além disso, como temos muitos gráficos, mostraremos neste capítulo somente os três casos mais interessantes para cada ganho, ou seja, três gráficos para o ganho do algoritmo que implementa a primeira otimização, três gráficos para o ganho do algoritmo que implementa a segunda otimização, e três gráficos do ganho da segunda otimização em relação a primeira. Todos os gráficos dos ganhos podem ser vistos no apêndice B.

Uma das primeiras coisas a serem notadas nos gráficos das figuras de 6.1 a 6.6 é a de que o ganho é sempre maior para a cadeia igual, e sempre menor para a cadeia diferente. Isso ocorre porque a cadeia igual invalida as diagonais no passo em que elas são inicializadas, e a cadeia diferente invalidará as diagonais, se for possível, mais tarde do que qualquer outra cadeia. Como ambas as otimizações são baseadas na invalidação de diagonais, quanto mais casamentos ocorrerem sobre uma diagonal, mais cedo esta diagonal será invalidada pelos algoritmos, e maior será a redução da

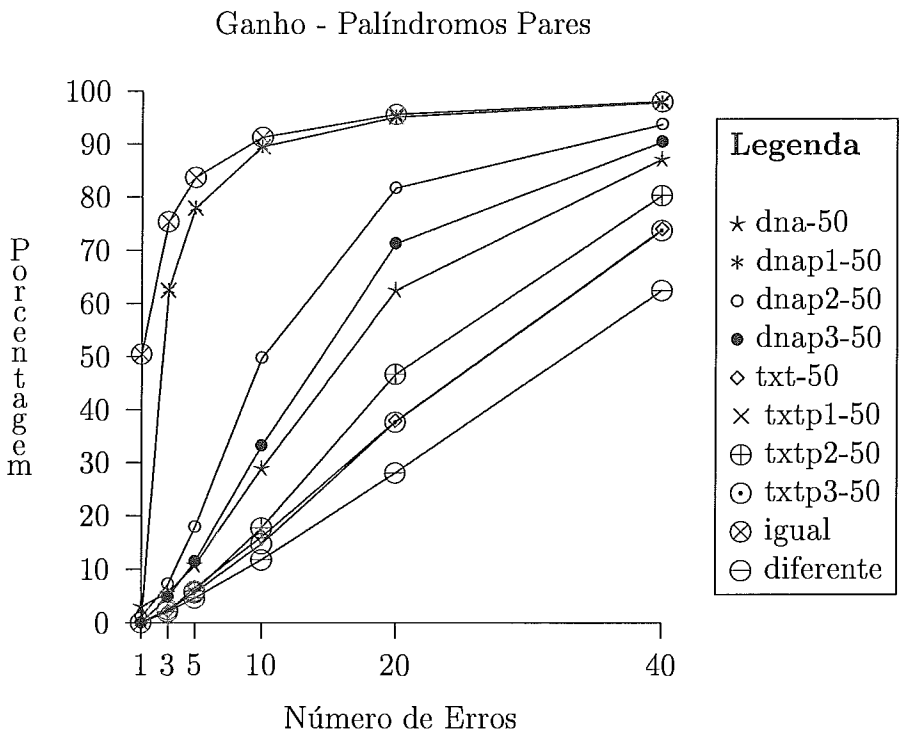


Figura 6.4: Ganho ao calcularmos todos os palíndromos pares para as cadeias com tamanho igual a 50, usando a segunda otimização. As linhas para as cadeias dnap1-50 e txtp1-50 são iguais, pois ambas são cadeias periódicas com período igual a 2.

área. O ganho para a cadeia igual aumenta na medida em que o tamanho da cadeia aumenta, pois nos exemplos que executamos, os números de erros são baseados no tamanho da cadeia, e portanto, a faixa máxima de uso de uma diagonal será maior ao aumentarmos o tamanho da cadeia. Não existem ganhos, como podemos ver pelas figuras de 6.7 a 6.9, para a cadeia diferente, da segunda otimização sobre a primeira, o que nos diz que na ausência de casamentos, o uso de faixas não colabora na redução do tempo de execução.

Também podemos ver, pelas figuras de 6.7 a 6.9, que os ganhos do segundo algoritmo em relação ao primeiro reduzem na medida em que o tamanho da cadeia cresce, com exceção da cadeia igual, provavelmente devido à invalidação direta das diagonais, logo assim que elas são inicializadas. Isso ocorre porque a redução da freqüência com que as diagonais são invalidadas reduzirá a criação de faixas que limitarão a computação de um passo, pois quanto maior o tamanho da cadeia, mais pontos estão presentes em uma diagonal, o que reduz a invalidação de diagonais.

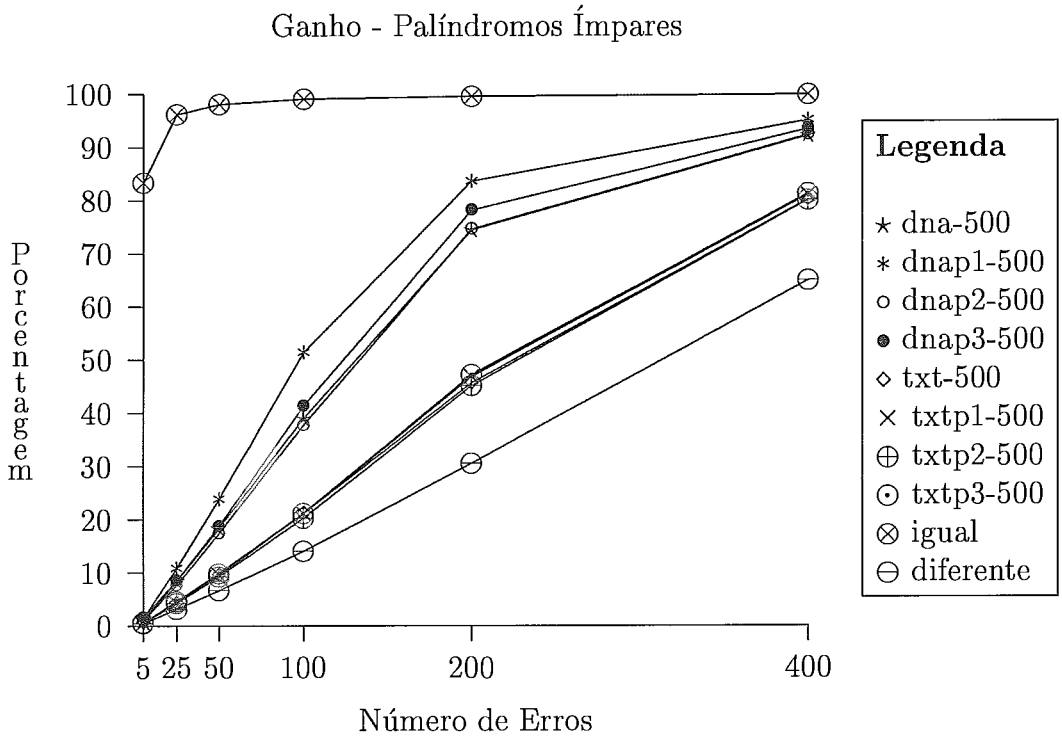


Figura 6.5: Ganho ao calcularmos todos os palíndromos ímpares para as cadeias com tamanho igual a 500, usando a segunda otimização.

Como também podemos ver pelas figuras de 6.1 a 6.9, há um aumento nos ganhos ao aumentarmos o número de erros, para um dado caso em particular, devido ao aumento da faixa máxima de diagonais e do número máximo de passos, com exceção dos ganhos do segundo algoritmo em relação ao primeiro para a cadeia igual, em que temos uma redução dos ganhos a partir de um certo número de erros, devido provavelmente ao término precoce do algoritmo, ao atingirmos a diagonal que contém o ponto com a soma máxima, num passo em que o uso de faixas ainda não produz um ganho significativo, e com exceção dos ganhos para as cadeias dnap1-50 e txtp1-50, dados pela figura 6.7, como veremos posteriormente.

Os ganhos para as cadeias genéricas para o primeiro e o segundo algoritmo em relação ao original, como podemos ver pelas figuras de 6.1 a 6.6, se mantém mais ou menos os mesmos, apesar de existir algum ganho ao aplicarmos a segunda otimização, como podemos ver pelas figuras de 6.7 a 6.9, pois nestes casos a quantidade de casamentos, e portanto, dos pontos na execução em que as diagonais são invalidadas, dependem da cadeia e do número de erros em particular, o que indica que provavelmente a escolha do número de erros como uma porcentagem do tamanho

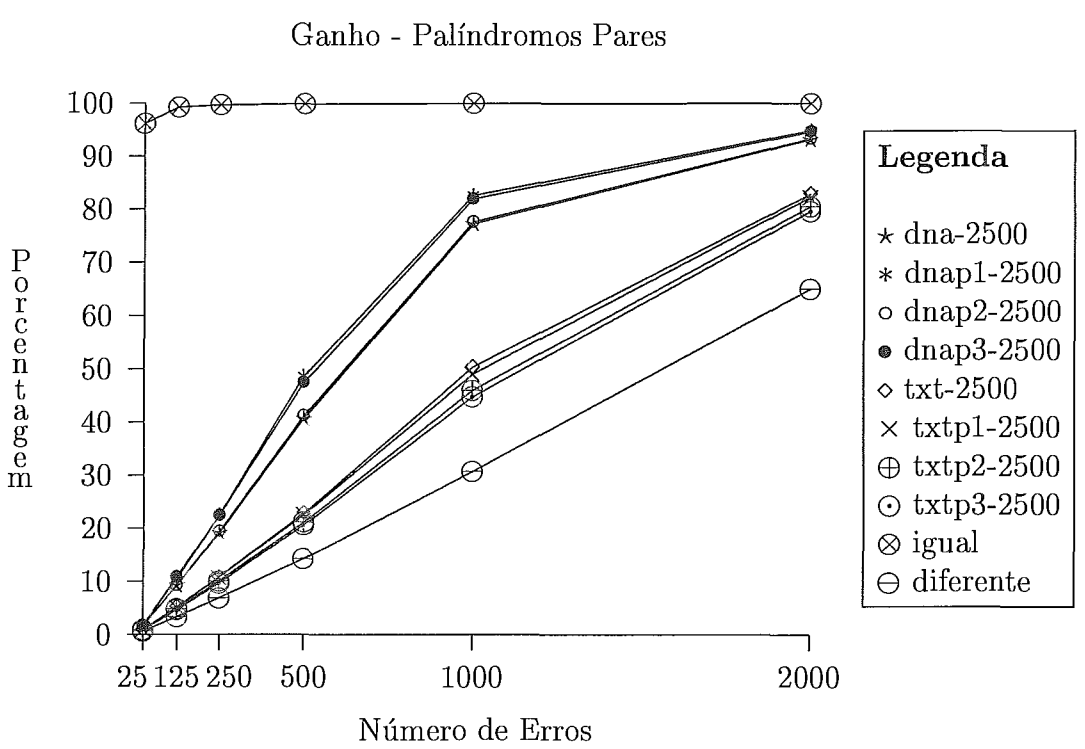


Figura 6.6: Ganho ao calcularmos todos os palíndromos pares para as cadeias com tamanho igual a 2500, usando a segunda otimização.

da cadeia é que foi a responsável por ganhos próximos para cada um dos gráficos mostrados. Também notamos que os ganhos para as cadeias provenientes de cadeias de DNA são maiores do que os ganhos das cadeias provenientes de cadeias de texto, pois no primeiro caso as freqüências de casamentos são maiores, o que aumenta a invalidação de diagonais nas duas otimizações, e portanto, aumenta a criação de faixas na segunda otimização.

Para terminar a análise dos gráficos, vamos dar uma olhada nas figuras 6.1, 6.4 e 6.7, que representam os gráficos dos ganhos para as cadeias com tamanho igual a 50. Como podemos ver em cada uma destas figuras, as linhas para as cadeias dnap1-50 e txtp1-50 são idênticas. Isso ocorre porque como elas são cadeias periódicas com período igual a 2, ambas serão compostas por uma seqüência alternada de dois caracteres. A única diferença é que para a cadeia dnap1-50 estes caracteres são provenientes de um alfabeto de DNA, e para a cadeia txtp1-50 eles são provenientes de um alfabeto ASCII. Também podemos notar que o ganho do algoritmo que implementa a segunda otimização em relação ao algoritmo que implementa a primeira otimização é melhor para estes casos do que o caso da cadeia igual, se $k > 4$. Isso

é devido ao fato de que em ambos os gráficos de ganho dos algoritmos em relação a área máxima, as linhas para estas cadeias estão bem próximas das linhas para as cadeias iguais, e devido ao fato de a linha para segunda otimização se aproximar mais da linha da cadeia igual do que a linha para a primeira otimização, o que indica provavelmente que os ganhos do algoritmo para as cadeias com caracteres alternados é próximo aos ganhos da cadeia igual, e que há mais ganho ao usarmos a segunda otimização. Devido também a esta propriedade, o ganho é reduzido a partir de um certo número de erros, assim como ocorre com a cadeia igual.

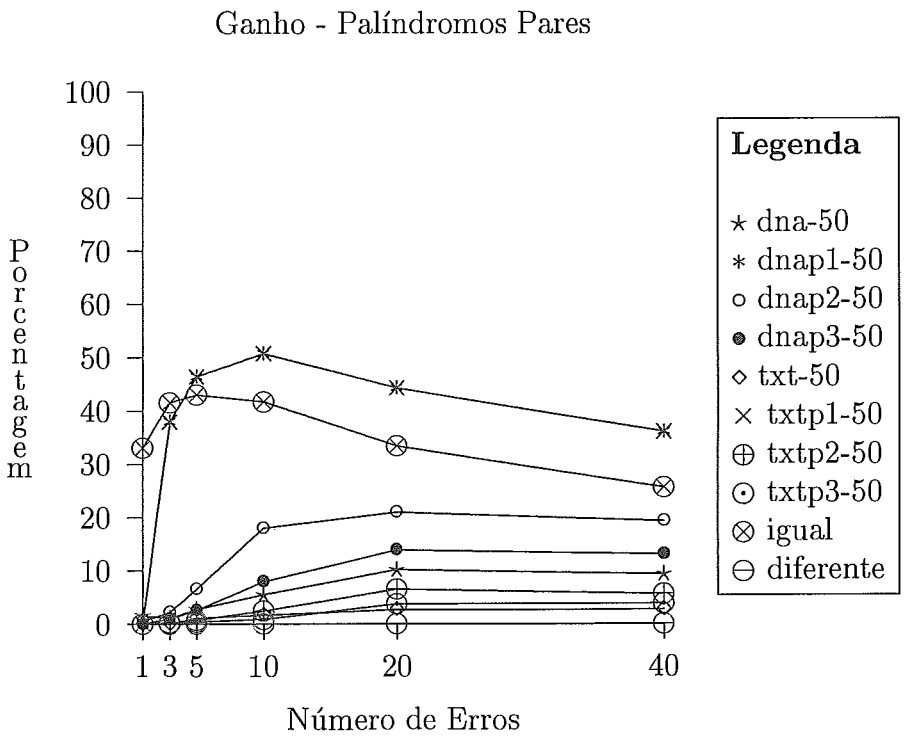


Figura 6.7: Ganho ao calcularmos todos os palíndromos pares para as cadeias com tamanho igual a 50, da segunda otimização em relação a primeira. As linhas para as cadeias dnap1-50 e txtp1-50 são iguais, pois ambas são cadeias periódicas com período igual a 2. Além disso, o ganho é maior para estas cadeias, para $k > 4$, devido ao fato de, nos gráficos das figuras 6.1 e 6.4, a linha de ganho para estes casos ser próxima da linha para a cadeia igual.

Agora vamos analisar os gráficos que mostram a redução do tempo de execução ao computarmos um palíndromo num centro (posição) da cadeia S, onde mostraremos o passo inicial e o final em que calculamos os últimos caminhos a chegarem para todas as diagonais, dentro da maior faixa de diagonais possível, a serem calculadas para este centro. Através destes gráficos, poderemos ter uma boa idéia de como os

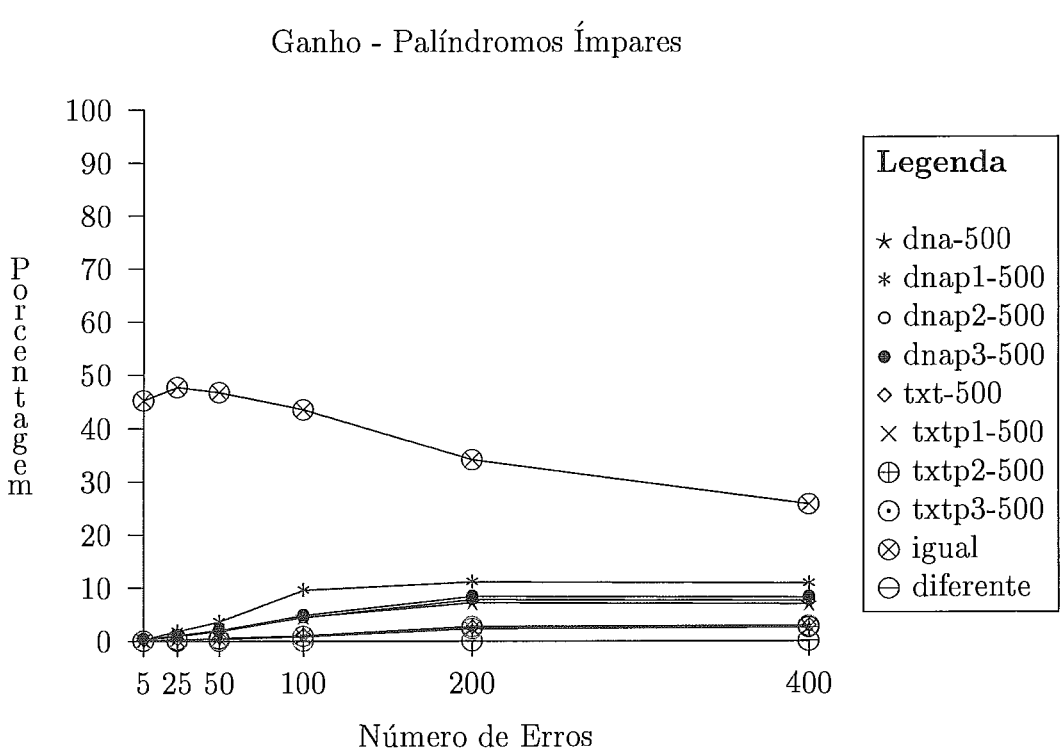


Figura 6.8: Ganho ao calcularmos todos os palíndromos ímpares para as cadeias com tamanho igual a 500, da segunda otimização em relação a primeira.

algoritmos reduzem o tempo de execução máximo para calcular um palíndromo num centro em particular, que seria representado por um gráfico com o triângulo todo preenchido no pior caso, se usarmos a visualização proposta pelas figuras de 6.10 a 6.13. Também poderemos, através destes gráficos, ter uma idéia de como a segunda otimização melhora o tempo de execução da primeira otimização.

Os exemplos dados a seguir foram obtidos pela execução do algoritmo com 200 erros no máximo, para as cadeias dna-1000 e txt-1000, ambas com o tamanho igual a 1000, e para as cadeias dna-500 e txt-500 com tamanho igual a 500. Iremos analisar a área ao calcular o palíndromo ímpar centrado na posição 99 ($99I$), e o palíndromo par centrado na posição 227 ($227P$) das cadeias dadas.

Das figuras analisadas, iremos somente mostrar aquelas que tenham as propriedades mais interessantes. Na tabela 6.1 são dadas as áreas para todos os casos estudados. Como podemos notar pelas figuras 6.10 e 6.11 (e também pela tabela 6.1), há uma redução na área ao calcularmos o palíndromo par na posição 227 da cadeia dna-1000 em relação a área máxima, ao usarmos os algoritmos que implementam a primeira e a segunda otimização, devido à invalidação das diagonais antes

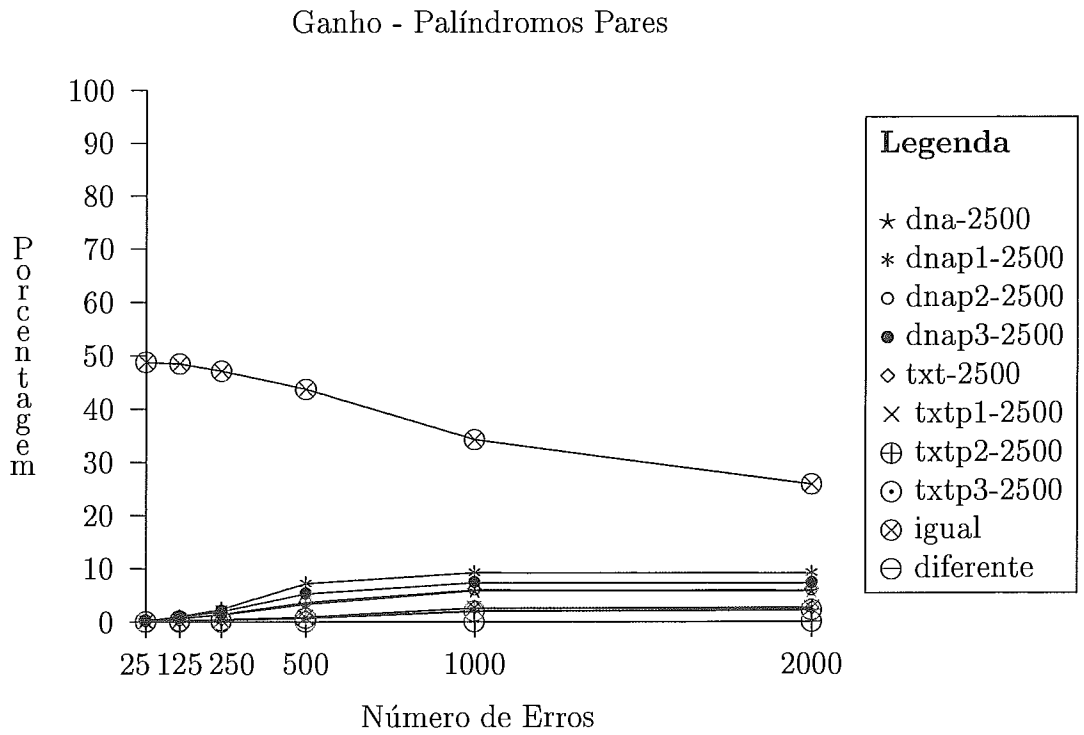


Figura 6.9: Ganho ao calcularmos todos os palíndromos pares para as cadeias com tamanho igual a 2500, da segunda otimização em relação a primeira.

do último passo, e do término precoce do algoritmo ao atingirmos o ponto com a soma máxima. Neste caso, ao compararmos as figuras geradas pelas duas otimizações, podemos notar que a segunda otimização impôs somente um limite inferior na computação das diagonais, pois neste caso (assim como para o $99I$) não há um limite superior, pois como $m = 227$, $n = 773$, e $k + m = 427 < n$, todas as 401 diagonais deste caso terminarão sobre o limite inferior.

Podemos ver pela tabela 6.1 que a área necessária para calcular estes palíndromos para a cadeia txt-1000 é maior do que a daquela para a cadeia dna-1000, o que é devido à maior probabilidade da ocorrência de casamentos da cadeia dna-1000, pois esta cadeia é proveniente de um alfabeto com tamanho igual a 4, enquanto que a outra cadeia é proveniente do alfabeto ASCII, com tamanho igual a 128.

Pela figura 6.12, que dá a área necessária para calcularmos o palíndromo ímpar na posição 99 da cadeia dna-1000, vemos que a redução da área será bem maior, pois como esta posição da cadeia está mais próxima do início e é menor do que 200, o número máximo de erros, as diagonais de -200 a -99 não estarão dentro da faixa máxima de cálculo do algoritmo, e além disso, as diagonais terão menos pontos,

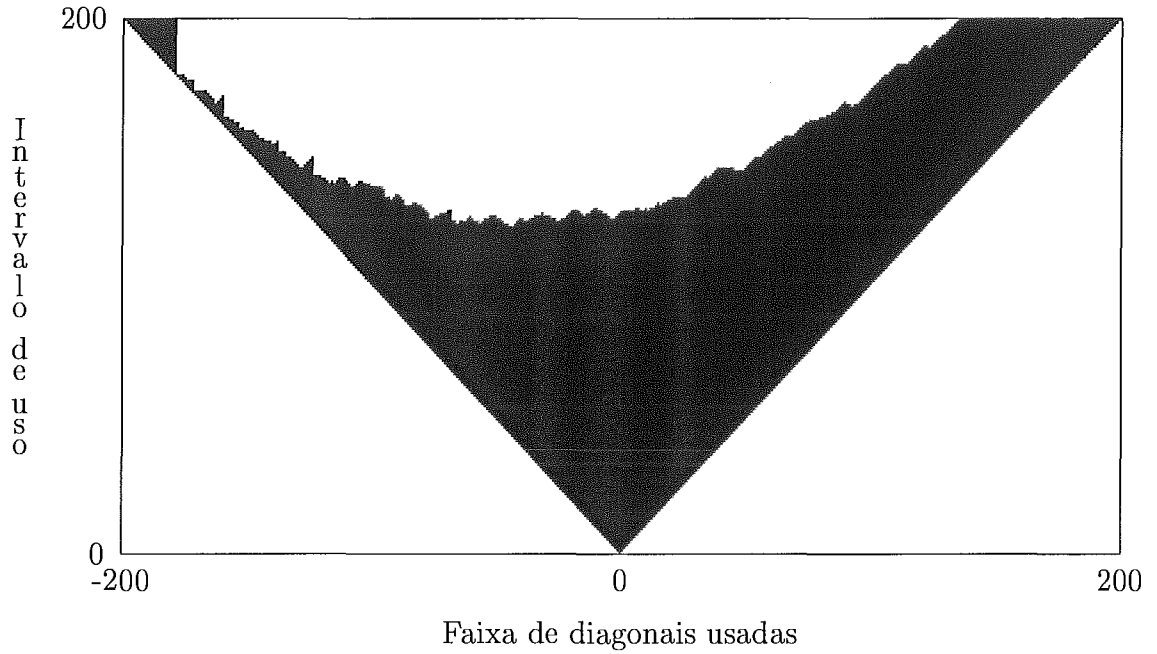


Figura 6.10: Aréa ao calcular o palíndromo par na posição 227 da cadeia dna-1000, usando a primeira otimização.

Exemplos / Centros	Área usada para calcular um centro			
	Primeira otimização		Segunda otimização	
	99I	227P	99I	227P
dna-500	6529	18309	6065	16991
dna-1000	5310	23690	4763	22113
txt-500	14352	38329	14093	37166
txt-1000	14747	36844	14494	35548
Área Total	35148	40401	35148	40401

Tabela 6.1: Tabela com as áreas ao calcularmos o palíndromo par na posição 227 e o palíndromo ímpar na posição 99, para as cadeias dna-500, dna-1000, txt-500 e txt-1000, usando os algoritmos que implementam as duas otimizações. Como podemos ver por esta tabela, todas as áreas geradas ao aplicarmos a segunda otimização são menores do que as geradas ao aplicarmos a primeira otimização.

o que aumenta a chance de invalidação. Da mesma forma, a área para a cadeia txt-1000 é maior do que a da cadeia dna-1000, novamente devido à maior freqüência de casamentos da primeira cadeia.

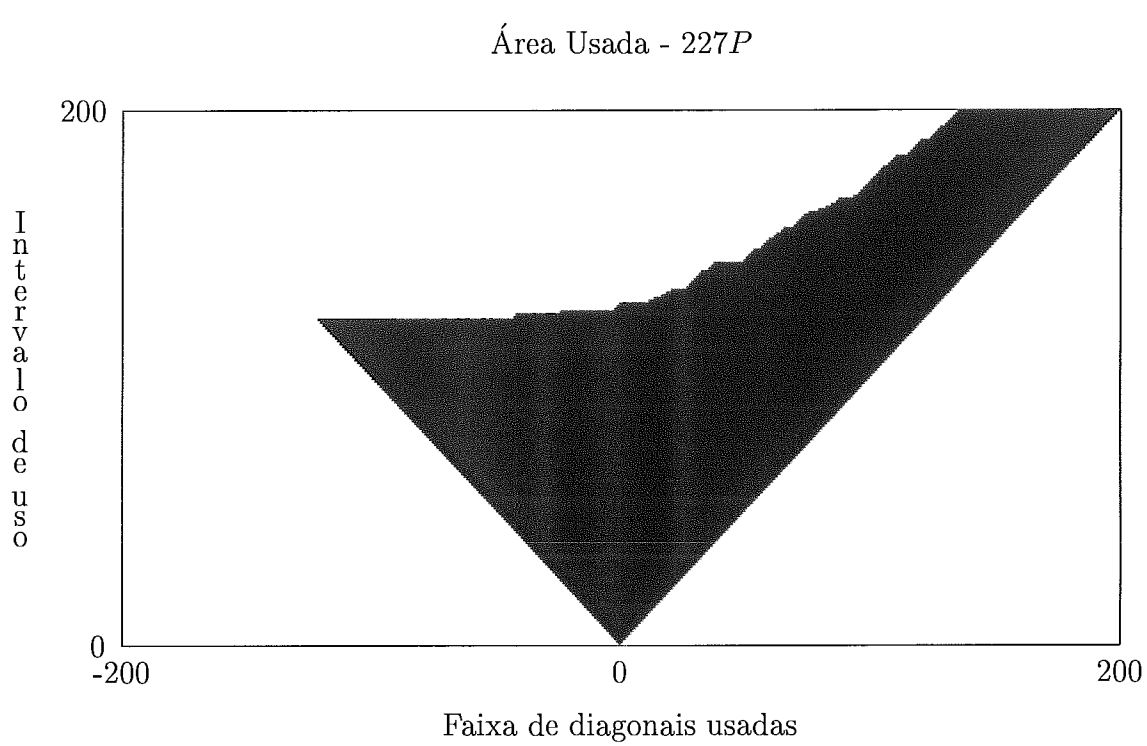


Figura 6.11: Aréa ao calcular o palíndromo par na posição 227 da cadeia dna-1000, usando a segunda otimização. Neste caso, podemos visualmente, se compararmos esta figura com a figura 6.10, verificar que o algoritmo impôs um limite inferior na computação.

Também existem casos em que nenhuma das otimizações conseguem reduzir a área total necessária para calcularmos o palíndromo em uma dada posição da cadeia S, como o cálculo do palíndromo par na posição 500, para as cadeias dna-1000 e txt-1000.

Pela figura 6.13, podemos ver que há uma maior redução da área ao calcularmos o palíndromo par na posição 227 para a cadeia dna-500, em relação a cadeia dna-1000, mas pela tabela 6.1, vemos que há um aumento da área ao calcularmos o palíndromo ímpar na posição 99 desta cadeia. Já para a cadeia txt-500 ocorre o oposto, ou seja, há um aumento da área ao calcularmos o palíndromo na posição 227, e uma redução desta área ao calcularmos o palíndromo na posição 99, em relação a cadeia txt-1000. Isso provavelmente ocorre devido à cadeia dna-500 ter maior probabilidade de casamentos entre os caracteres para o cálculo do palíndromo na posição 227, enquanto que a cadeia txt-500 tem maior probabilidade de casamentos entre os caracteres ao calcularmos o palíndromo na posição 99, ou seja, isso provavelmente

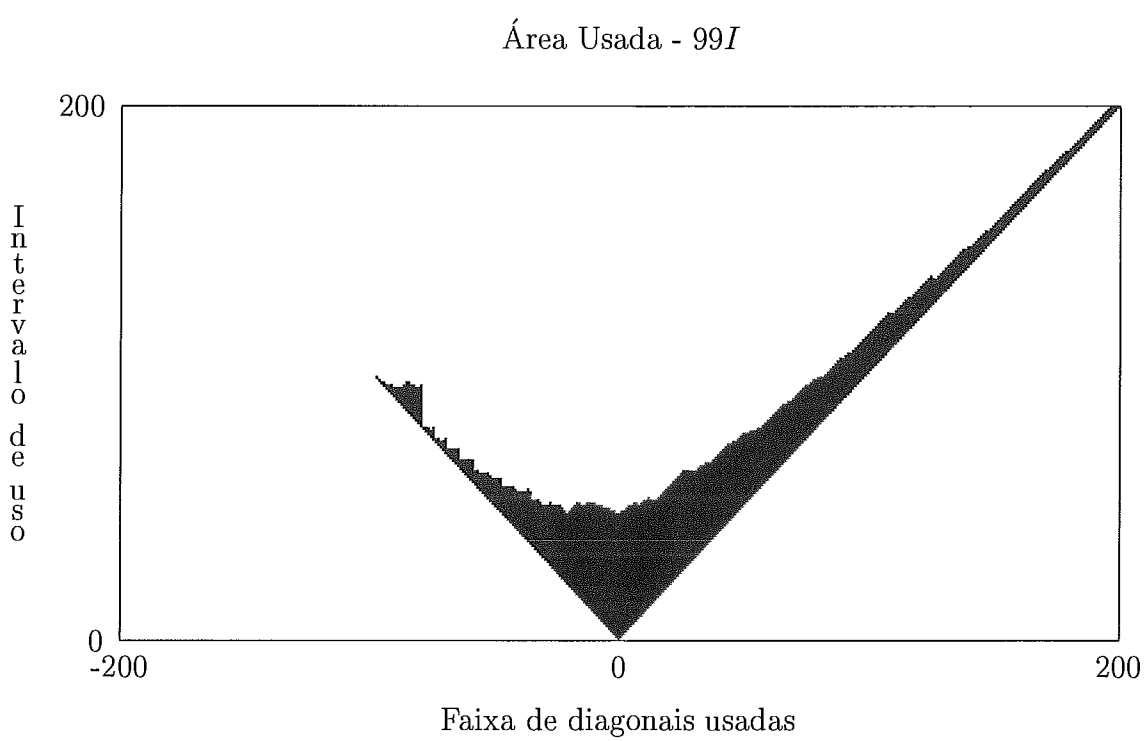


Figura 6.12: Aréa ao calcular o palíndromo ímpar na posição 99 da cadeia dna-1000, usando a primeira otimização. Neste caso, as diagonais de -200 a -99 não são usadas.

é devido à distribuição dos caracteres nestas cadeias, que variam de acordo com a formação desta cadeia a partir do seu alfabeto de origem.

6.2 Análise Probabilística

Nesta seção iremos apresentar a análise do tempo de execução médio do nosso algoritmo através da definição de um modelo probabilístico na primeira subseção, e compararemos os resultados retornados pelo nosso modelo, na próxima subseção, com os resultados obtidos com as execuções reais dos algoritmos que implementam as duas otimizações descritas no capítulo anterior, para mostrarmos a precisão do nosso modelo ao determinar o tempo médio de execução em ambos os algoritmos. Além disso, analisaremos os gráficos dos ganhos obtidos para as médias, através da aplicação dos modelos probabilísticos, para as duas otimizações.

Os modelos probabilísticos serão usados para calcularmos o *número médio de passos*, que é o número médio de passos externos do algoritmo básico, em que varia-

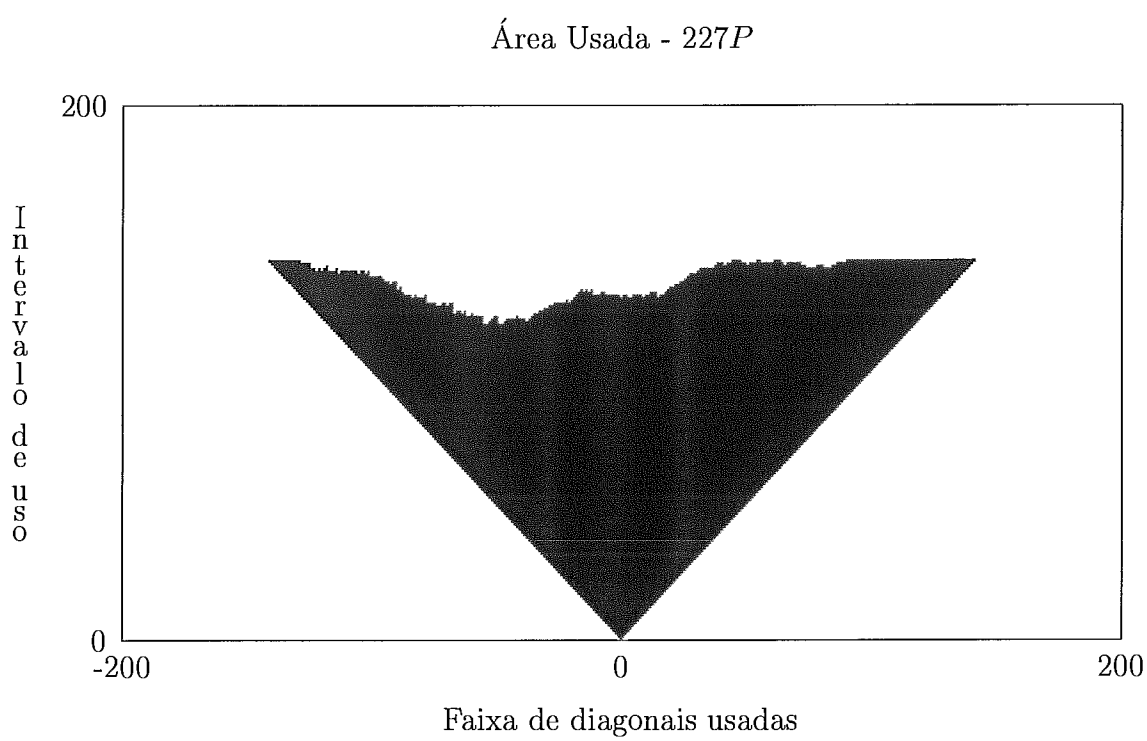


Figura 6.13: Aréa ao calcular o palíndromo par na posição 227 da cadeia dna-500, usando a primeira otimização. Neste caso, a área total para calcular este palíndromo será menor do que a do cálculo com a cadeia dna-1000, como pode ser visto ao compararmos esta figura com a figura 6.10.

mos o número de erros dos caminhos calculados, e a *área média* que dará o número médio de últimos caminhos a chegarem, computados ao calcularmos um palíndromo em uma dada posição da cadeia, para todos os passos externos executados.

6.2.1 O Modelo Probabilístico

O nosso modelo probabilístico será representado como um grafo acíclico orientado, onde os vértices representam os possíveis estados em que o algoritmo pode estar em um dado passo, e as arestas emergentes de um estado S_i gerado num passo p representam todos os possíveis estados atingíveis a partir deste estado, ou seja, todos os possíveis estados gerados após o término do passo $p+1$ usando o estado S_i como um estado de entrada. Para melhorar o entendimento do grafo, na primeira otimização, podemos agrupar todos os estados gerados em um passo p numa mesma coluna, o que nos permitirá compreender melhor a execução do algoritmo para um dado centro, pois saberemos quais estados deverão ser usados como entrada para

um dado passo.

Agora, seja S uma cadeia de tamanho igual a N para a qual desejamos obter um palíndromo aproximado par ou ímpar em uma dada posição c desta cadeia, e seja k o número máximo de erros permitidos para este palíndromo. Um estado S_i gerado ao final de um passo $p \leq k$ será um conjunto de $2k+1$ valores, cada um representando o final do último caminho-p a chegar a uma diagonal d, $-k \leq d \leq k$, ordenados de acordo com o número das diagonais, ou seja, de acordo com a relação $d = j - i$ existente entre as células (i, j) sobre esta diagonal d. As diagonais não atingíveis neste passo, assim como aquelas nunca atingíveis pelo algoritmo (se existirem), serão representadas pelo valor -1 no estado, e serão chamadas de *diagonais não inicializadas*. Se após o final de um passo p atingirmos o ponto máximo de uma diagonal d, ela será chamada de *diagonal invalidada*, como já vimos no capítulo anterior.

Então se X_d, $-k \leq d \leq k$, representar o valor deste ponto de término para uma diagonal d, se usarmos a primeira otimização, e se $L_i = \max(-p, -m)$ e $L_s = \min(p, n)$ representarem os limites da faixa de diagonais já inicializadas, onde $n = N - c$, e $m = c$ se estamos calculando um palíndromo par, e $m = c - 1$ se estamos calculando um palíndromo ímpar, então para as diagonais d, $-k \leq d < L_i$ e $L_f < d \leq k$, $X_d = -1$, pois estas diagonais ainda não são atingíveis. O estado S_i terá a seguinte aparência, se forem usadas as notações descritas acima:

$$S_i = (\underbrace{-1, -1, \ldots, -1}_{k-|L_i| \text{ vezes}}, X_{L_i}, X_{L_i+1}, \ldots, X_{-1}, X_0, X_1, \ldots, X_{L_s-1}, X_{L_s}, \underbrace{-1, \ldots, -1, -1}_{k-L_s \text{ vezes}})$$

Os possíveis estados S_j atingíveis a partir do estado S_i num passo p serão determinados a partir do *estado mínimo* S_i^* atingível a partir do estado S_i, obtido antes de executarmos as extensões de casamentos nas diagonais, ou seja, após determinarmos quais das extensões dos últimos caminhos-p a chegarem sobre uma diagonal terminam no maior ponto, que será dado pelo ponto máximo do término destas extensões, como vimos no capítulo anterior. Este estado mínimo representará o caso em que não executamos nenhuma extensão de casamentos no passo $p+1$, e será usado também ao calcularmos a probabilidade de transição de um estado S_i para cada um dos possíveis estados destinos S_j. Se cada um dos possíveis valores neste estado S_i^* for representado por X_d^M, $L_i \leq d \leq L_s$, onde $X_{L_i-1}^M$ e $X_{L_s+1}^M$ representam as duas diagonais que podem ser inicializadas neste passo, temos que:

$$S_i^* = (\underbrace{-1,\ldots,-1}_{k-|L_i|-1 \text{ vezes}}, X^M_{L_i-1}, X^M_{L_i}, \ldots, X^M_{-1}, X^M_0, X^M_1, \ldots, X^M_{L_s}, X^M_{L_s+1}, \underbrace{-1,\ldots,-1}_{k-L_s-1 \text{ vezes}})$$

Como a diagonal $-p-1$ é inicializada no passo $p+1$ se $p+1 \leq m$, então $X^M_{-p-1} = X^M_{L_i-1} = X_{-p} + 1$ se for possível executarmos a deleção de um caractere, ou seja, se $X_{-p} \neq pf(-p)$, onde $pf(d)$ dado abaixo, dará o ponto máximo em uma diagonal d. Se não for possível executarmos uma deleção, então $X^M_{-p-1} = X^M_{L_i-1} = p+1$. Similarmente, a diagonal $p+1$ também deverá ser inicializada se $p+1 \leq n$, então $X^M_{p+1} = X^M_{L_s+1}$ será inicializado com o valor de X_p se for possível a execução de uma inserção, ou seja, se $X_p + p < n$, e com 0 se não for possível executar esta inserção. Se alguma das diagonais não for usada, isso indicará que elas não serão usadas em nenhum outro passo posterior do algoritmo, e por isso, deverão reter o valor -1 indicando que elas ainda não foram inicializadas.

$$pf(d) = \begin{cases} m & \text{se } d+m \leq n \\ n-d & \text{se } d+m > n \end{cases}$$

Agora para uma dada diagonal d, $L_i \leq d \leq L_s$, o valor do ponto de término X^M_d do último caminho-$(p+1)$ a chegar a esta diagonal d deste estado mínimo, somente será calculado se a diagonal não estiver invalidada, ou seja, se $X_d < pf(d)$. Caso a diagonal esteja invalidada, ela irá reter o valor de seu ponto máximo. Caso contrário, o valor de X^M_d para a diagonal d será o maior dos seguintes valores:

- $X_d + 1$, representando a substituição de um caractere;
- X_{d-1} se for possível executar uma inserção, ou seja, se $X_{d-1} + d - 1 < n$, e se $d-1 \geq L_i$, ou -1 caso contrário;
- $X_{d+1} + 1$ se for possível executar uma deleção, isto é, se $X_{d+1} < m$, e se $d+1 \leq L_s$, ou -1 caso contrário.

Seja L'_i igual a $L_i - 1$ se $p+1 \leq m$ e igual a L_i caso contrário, e L'_s igual a $L_s + 1$ se $p+1 \leq n$ e igual a L_s caso contrário. Se usarmos as fórmulas descritas abaixo, podemos definir que, neste estado mínimo, $X^M_d = -1$ se $d < L'_i$ ou $d > L'_s$, que $X^M_d = \max(ins(d-1), sub(d), del(d+1))$, se $L_i \leq d \leq L_s$, que $X^M_{L'_i} = \max(p+1, del(-p))$, se $p+1 \leq m$, e que $X^M_{L'_s} = \max(0, ins(p))$, se $p+1 \leq n$, para d variando de $-k$ até k.

$$del(d) = \begin{cases} X_d + 1 & \text{se } X_d < m \\ -1 & \text{se } X_d = m \end{cases}$$

$$ins(d) = \begin{cases} X_d & \text{se } d + X_d < n \\ -1 & \text{se } d + X_d = n \end{cases}$$

$$sub(d) = \begin{cases} X_d + 1 & \text{se } X_d < m \text{ e } d + X_d < n \\ pf(d) & \text{caso contrário} \end{cases}$$

Agora, para obtermos todos os estados S_j atingíveis pelo estado S_i, basta aplicarmos todas as possíveis extensões de casamentos a todas as diagonais não invalidadas e já inicializadas definidas após o cálculo do estado mínimo. Se um estado S_j particular for igual a:

$$S_j = (\underbrace{-1, -1, \ldots, -1}_{k-|L'_i| \text{ vezes}}, X'_{L'_i}, X'_{L'_i+1}, \ldots, X'_{-1}, X'_0, X'_1, \ldots, X'_{L'_s-1}, X'_{L'_s}, \underbrace{-1, \ldots, -1, -1}_{k-|L'_s| \text{ vezes}}),$$

a determinação de todos estes estados poderá ser feita através da variação, para cada uma destas diagonais d, do valor de X'_d de X^M_d até $pf(d)$, variando o número de casamentos na extensão de casamentos de 0 até $pf(d) - X^M_d$. Com isso, o número de estados atingíveis a partir de um estado S_i será igual ao produto da diferença $pf(d) - X^M_d + 1$ para todas estas diagonais d.

O *estado inicial* do nosso grafo será um estado onde $X_d = -1$, $-k \leq d \leq k$, representando o fato de que todas as diagonais devem estar não inicializadas se ainda não executamos nenhum passo do nosso algoritmo. Um estado S_i será um *estado final* ou se ele for gerado no final do último passo k do algoritmo, ou se atingirmos o ponto máximo da diagonal $d = n - m$ que contém o ponto (m, n), pois como já vimos no capítulo anterior, neste último caso não existirão melhores conversões nos passos posteriores ao passo em que atingimos este ponto (m, n).

O cálculo do estado mínimo S^*_i gerado a partir do estado S_i, e dos possíveis estados S_j destinatários deste estado, se usarmos a segunda otimização, pode ser facilmente feito adaptando-se o que já foi descrito anteriormente, determinando-se os corretos valores para L_i e L_s. Para determinarmos estes limites, que darão a faixa de execução no passo $p + 1$ ao usarmos o estado S_i, deveremos primeiramente obter o valor da diagonal d_m que contém o ponto com soma máxima (m, n). Para que tenhamos um limite inferior L_i, $d_m > -k$ e neste caso, L_i será a diagonal invalidada com o maior valor possível para o seu número d, com a restrição de que $d < d_m$. Caso o limite inferior não exista, fazemos $L_i = \max(-p, -m)$. Para existir o limite

superior L_s, deveremos ter que $d_m < k$ e neste caso, L_s será a diagonal invalidada com o menor valor possível para o seu número d, com a restrição de que $d > d_m$. Da mesma forma, se o limite superior não existir, $L_s = \min(p, n)$.

Agora, a diagonal $-p-1$ deverá ser inserida ao criarmos o estado mínimo S_i^* a partir do estado S_i se $L_i = -p$, e se a diagonal L_i não estiver invalidada, ou seja, se não existir um limite inferior, e neste caso também faremos $L_i' = L_i - 1$. Se não inicializarmos a diagonal $-p-1$, ou seja, se existir um limite inferior, então $L_i' = L_i$. A diagonal $p+1$ deverá ser inicializada se não existir um limite superior, ou seja, se $L_s = p$, e se a diagonal L_s não estiver invalidada, e neste caso, $L_s' = L_s + 1$. Caso exista um limite superior, $L_s' = L_s$. Usamos o mesmo processo descrito anteriormente para calcular as diagonais d dentro dos limites L_i e L_s, onde $L_i \leq d \leq L_s$, e para calcular todos os estados S_j atingíveis por S_i a partir do estado S_i^*.

Vamos a seguir mostrar como construir o grafo com todos os possíveis estados para um problema em particular. Isso poderá ser feito, da seguinte maneira: a partir do estado inicial, geramos todos os possíveis estados atingíveis a partir do estado mínimo deste estado inicial[1]. Em seguida, usamos os estados não finais gerados pelo primeiro passo como entrada para o segundo passo, para gerar todos os possíveis estados criados após o final deste passo, e assim por diante, ou seja, após o final de um passo $p < k$, pegamos todos os estados não finais gerados ao final deste passo, e aplicamos as regras descritas anteriormente para gerar todos os possíveis estados criados após o término do passo $p+1$, descartando neste processamento todos os estados repetidos que já foram gerados.

Como um exemplo de construção de um grafo, na figura 6.14 mostramos o grafo para uma cadeia de tamanho $N = 4$, para um palíndromo par centrado na posição 2 desta cadeia, com $k = 2$ erros no máximo, usando a primeira otimização, e na figura 6.15 mostramos o mesmo grafo para o mesmo exemplo, mas agora usando a segunda otimização. As setas de um estado para outros estados nas figuras indicam os estados gerados por este estado como entrada para o passo atual (dado acima deste estado na figura 6.14), sendo que um "*" sobre a seta indica que o estado mínimo é o destino desta seta. Como podemos ver pelas figuras, os estados destinos de um estado S_i foram gerados a partir da aplicação de extensões de casamentos sobre as diagonais dentro dos limites L_i' e L_s' do estado S_i^*.

[1]Neste caso, somente devemos inicializar a diagonal 0, ou seja, $X_0^M = 0$ e $X_d^M = -1$, se $d \neq 0$.

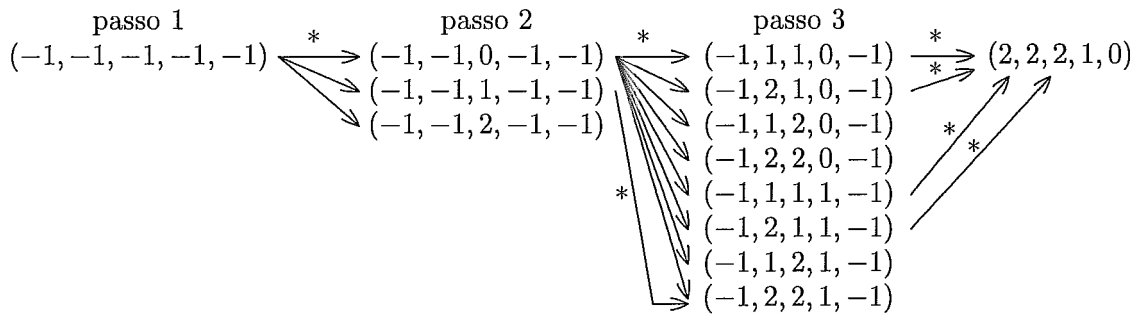


Figura 6.14: Exemplo da construção do grafo para $N = 4$ e $k = 2$, ao procurarmos por um palíndromo par na posição 2 desta cadeia, usando a primeira otimização descrita no capítulo anterior. No grafo, agrupamos todos os estados gerados em um passo numa mesma coluna, sendo que o número do passo acima de um grupo de estados dá o passo em que estes estados são usados. O "*" sobre as setas indicam qual é a transição de um estado S_i para o estado S_i^*.

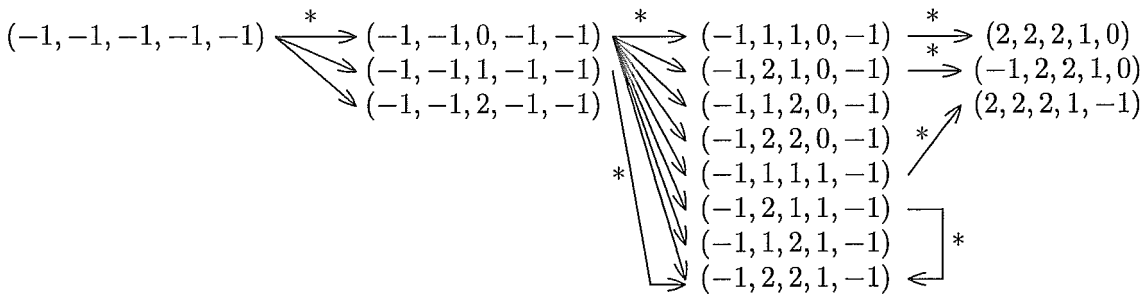


Figura 6.15: Exemplo da construção do grafo para $N = 4$ e $k = 2$, ao procurarmos por um palíndromo par na posição 2 desta cadeia, usando a segunda otimização descrita no capítulo anterior. O "*" sobre as setas indicam qual é a transição de um estado S_i para o estado S_i^*.

Após construirmos o grafo, devemos agora usá-lo para calcular o número médio de passos e a área média para um dado problema em particular, o que poderá ser feito a partir de um procedimento recursivo como veremos adiante, mas antes é necessário calcularmos a probabilidade de transição de um estado não final S_i para cada um dos possíveis estados S_j gerados a partir do estado mínimo S_i^* de S_i. Como veremos adiante, esta probabilidade dependerá tanto do estado S_i^* e do estado S_j, como do caminho que seguimos no grafo a partir do estado inicial para atingir o estado S_i.

Seja $S_e = S[1..c]^R$ se desejamos obter um palíndromo par, $S_e = S[1..c-1]^R$ se

desejamos obter um palíndromo ímpar, e seja $S_d = S[c+1..N]$. Pelas definições dadas anteriormente, $|S_e| = m$ e $|S_d| = n$. Seja C uma matriz $m \times n$ onde a célula $c_{i,j}$ irá representar qual é a relação existente entre o caractere $S_e(i)$ e o caractere $S_d(j)$. Chamaremos esta matriz C de *matriz de condições*, e ela irá definir quais relações foram implicadas entre os caracteres das cadeias para atingirmos o estado S_i por algum caminho a partir do estado inicial. O valor da célula $c_{i,j}$ terá os possíveis significados:

- $0 \Rightarrow$ Não há uma relação definida entre os caracteres $S_e(i)$ e $S_d(j)$;
- $1 \Rightarrow$ O caractere $S_e(i)$ deve casar com o caractere $S_d(j)$, ou seja, $S_e(i) = S_d(j)$;
- $2 \Rightarrow$ O caractere $S_e(i)$ deve ser diferente do caractere $S_d(j)$, ou seja, $S_e(i) \neq S_d(j)$.

As relações entre os caracteres serão geradas conforme nós passamos de um estado para um outro, e isso significa que quanto mais caminhamos no grafo em direção a um estado final a partir do estado inicial, mais condições serão formadas entre os caracteres, e com isso restringiremos cada vez mais o número de possíveis cadeias do número total σ^N, onde σ é o tamanho do alfabeto, ao passarmos de um estado para um outro. As condições adicionadas à matriz ao passarmos de um estado S_i para um estado S_j dependerão destes estados, e do estado mínimo S_i^*. O número de casos possíveis para atingirmos o estado S_i dependerá do caminho em particular seguido do estado inicial até este estado S_i, assim como a matriz de condições para este estado. Como veremos adiante que a probabilidade dependerá do número de casos e da matriz de condições que usamos para atingir o estado S_i, teremos uma probabilidade de transição diferente de S_i para S_j para cada um dos possíveis caminhos do estado inicial até o estado S_i.

Seja C_i uma das matrizes de condições que impõe as relações entre os caracteres de S_e e S_d necessárias para atingirmos o estado S_i por algum caminho a partir do estado inicial, e seja $x \leq \sigma^N$ o número de casos que satisfazem as condições impostas pela matriz. A probabilidade de transição P entre o estado S_i e o estado S_j será igual a y/x onde y é o número de casos que satisfazem as relações entre os caracteres impostas pela matriz de condições C_j para atingirmos o estado S_j através deste caminho usado para chegar ao estado S_i.

A matriz C_j irá ser obtida da matriz C_i através da adição de relações entre os caracteres, usando os seguintes passos descritos a seguir, para cada diagonal d não invalidada em S_i^*, $L_i' \leq d \leq L_s'$:

1. Seja $l = X_d' - X_d^M$. Se $l > 0$, como a célula (a, b) indicará a conversão do prefixo $S_e[1..a]$ no prefixo $S_d[1..b]$, se $a = X_d^M$ e $b = a + d$, então a extensão de casamentos deverá começar na posição $a + 1$ de S_e e $b + 1$ de S_d. Com isso, esta extensão de casamentos irá impor uma relação de igualdade entre os caracteres das subcadeias $S_e[a+1..a+l]$ e $S_d[b+1..b+l]$. Para atualizarmos a matriz com estas novas relações impostas entre os caracteres, basta que todas as células $c_{a+q,b+q}$ de C_j sejam iguais a 1, onde $1 \leq q \leq l$. Se alguma destas células tiver o valor 2, isso indicará que a transição do estado S_i para o estado S_j é impossível, e neste caso além de não existir uma matriz C_j consistente, o número de casos y e a probabilidade P serão iguais a 0, pois não existirá uma cadeia S atendendo a estas condições.

2. Se $X_d' \neq pf(d)$, precisaremos executar uma substituição para garantir que o número de casamentos é exatamente igual a l. Para isso, devemos executar uma substituição após o ponto de término da extensão de casamentos, ou seja, se $a = X_d' + 1$ e $b = a + d$, então $S_e(a) \neq S_d(b)$, e com isso, devemos atribuir o valor 2 a célula $c_{a,b}$ da matriz C_j. Se esta célula for igual a 1, novamente teremos uma inconsistência, e assim como no item anterior, não teremos uma matriz C_j definida, e teremos que $y = 0$ e $P = 0$.

Na figura 6.16 mostramos a construção da matriz de condições C_j para o estado $S_j = (-1, 2, 1, 0, -1)$ a partir da matriz do estado $S_i = (-1, -1, 0, -1, -1)$, onde estes estados são provenientes do grafo construído na figura 6.14.

A soma de todas as probabilidades P de todas as possíveis transições que partem do estado S_i deve ser igual a 1, pois na verdade, a adição de condições a matriz C_i existe para dividir os x casos possíveis neste caminho particular até S_i em vários subcasos, um para cada uma das possíveis transições que partem do estado S_i. Com isso, se não adicionarmos condições à matriz C_j, deveremos ter somente uma transição de S_i para um certo estado S_j, e teremos neste caso que $y = x$ e $P = 1.0$.

A matriz de condições para o estado S_i pode ser obtida usando-se o processo descrito acima para cada um dos estados intermediários neste caminho particular do estado inicial até o estado S_i, considerando que a matriz do estado inicial é a

Diagonais	Início em S_e	Início em S_d	Condições geradas
-1	a	c	$a = c$
0	a	d	$a \neq d$
1	b	d	$b \neq d$

(a)

Matriz para o estado S_i Matriz para o estado S_j

	c	d
b	2	0
a	0	0

	c	d
b	2	2
a	1	2

(b)

Figura 6.16: Nesta figura, calculamos a matriz de condições para o estado $S_j = (-1, 2, 1, 0, -1)$ a partir da matriz do estado $S_i = (-1, -1, 0, -1, -1)$ do grafo construído na figura 6.14. Neste caso, o estado mínimo será $S_i^* = (-1, 1, 1, 0, -1)$. Se a cadeia $S = abcd$, então $S_e = ba$ e $S_d = cd$. Na parte (a) da figura, mostramos quais as novas condições que devem ser adicionadas à matriz C_i de S_i para construirmos a matriz C_j de S_j, determinadas pelo processo de cálculo desta matriz a partir da matriz C_i e dos estados. Na parte (b) da figura, mostramos a matriz C_i para o estado S_i, e a matriz C_j para o estado S_j, após a adição das condições necessárias para passarmos do estado S_i para o estado S_j.

matriz nula, pois não existem relações entre os caracteres de S_e e de S_d antes de começarmos a execução do algoritmo.

Em seguida iremos mostrar como usar as matrizes C_i e C_j para obter x e y, e como detectar outras transições entre o estado S_i e o estado S_j onde não existem cadeias, ou seja, onde $y = 0$ e $P = 0$, devido a inconsistências nas relações entre os caracteres de S_e, e entre os caracteres de S_d, originárias das relações entre os caracteres de S_e e de S_d definidas pela matriz de condições C_j.

Agora vamos mostrar como obter a restrição ao número de casos ao impormos as relações entre os caracteres das cadeias S_e e S_d, e portanto, entre os caracteres da cadeia S, dadas pela matriz de condições. Se um caractere estiver na posição i de S_e, este caractere estará na posição $c+1-i$ de S se desejamos obter um palíndromo par na posição c da cadeia, ou $c - i$ se desejamos obter um palíndromo ímpar nesta mesma posição, e se um caractere estiver na posição j de S_d ele estará na posição $c + j$ de S.

Se construirmos um grafo em que cada vértice representa um grupo de caracteres iguais da cadeia S, definidos pelas relações de igualdade dadas pela matriz, e se

definirmos que cada aresta entre estes vértices representam que o grupo de caracteres de um vértice é diferente do grupo de caracteres do outro vértice, que serão definidas pelas relações de desigualdade desta mesma matriz, então o nosso problema pode ser visto como um problema de coloração de um grafo [9] com σ cores onde dois vértices vizinhos, que são conectados por uma aresta, não podem ter a mesma cor, sendo que ao atribuirmos uma certa cor a um vértice, todos os caracteres que este representa serão iguais a um dos possíveis σ caracteres do alfabeto. Como uma cor representa que um grupo de caracteres é igual a um certo caractere do alfabeto, e como uma aresta indica um descasamento entre grupos de caracteres, a coloração de vértices vizinhos com a mesma cor indicaria que os caracteres destes grupos representados por estes vértices são iguais e diferentes ao mesmo tempo, o que é um absurdo. Vamos chamar este grafo que codifica o número de casos e a matriz de condições de *grafo de relações.*

Portanto, o número de casos será o número de formas diferentes de colorirmos o grafo de relações com σ cores, dada a restrição de que dois vértices vizinhos não tenham a mesma cor, o que poderá ser feito através do cálculo do polinômio cromático [9] para este grafo. Como já sabemos qual é o valor do número de cores, não precisaremos realmente calcular este polinômio, e sim, usar a idéia de seu cálculo para obter o número de colorações diferentes deste grafo, que dará o número de casos, se existirem, sujeitos as restrições dadas pela matriz de condições.

Devido ao fato de agruparmos todos os caracteres iguais em um mesmo vértice, todas as implicações indiretas entre os caracteres de S geradas pela matriz de condições que gerem inconsistências poderão ser detectadas ao construirmos o grafo, se primeiramente agruparmos todos os caracteres iguais num mesmo vértice, e somente depois criarmos as arestas entre os grupos de caracteres diferentes, pois neste caso, se existir uma inconsistência indireta, tentaremos, em algum ponto da criação das arestas do grafo, criar uma aresta de um vértice para ele próprio, o que claramente indicará uma inconsistência, e neste caso, abortaremos a construção do grafo, pois já saberemos que o número de casos é igual a 0.

Não teremos 0 casos somente ao acontecerem inconsistências, pois as relações entre os caracteres podem forçar que o tamanho σ do alfabeto seja limitado inferiormente, e neste caso, qualquer valor menor do que o limite inferior irá indicar um caso impossível devido à falta de caracteres diferentes para que as relações dadas pela matriz sejam satisfeitas. Este caso será detectado ao calcularmos o polinômio

cromático, pois como o número de cores será igual ao tamanho σ do alfabeto, será impossível a coloração do grafo com este número de cores, de tal forma que dois vizinhos não possuam a mesma cor.

Agora vamos mostrar como iremos construir o grafo de relações descrito anteriormente a partir da matriz de condições, e depois mostraremos um algoritmo para calcular o número possível de colorações, dado o tamanho σ do alfabeto e o grafo, através do cálculo do polinômio cromático para este alfabeto.

Seja Map um vetor com N posições, cada uma representando o caractere da cadeia S na mesma posição. Este vetor indicará, para cada caractere, um número identificando o vértice que representa o grupo de caracteres que são iguais a este caractere. Vamos inicializar este vetor com um valor igual a zero para cada posição, indicando inicialmente que nenhum dos caracteres estão associados a algum vértice do grafo. Seja na o número atual disponível para numerar um vértice, inicializado com 1. Vamos analisar primeiramente as relações de igualdade dadas pela matriz, para agruparmos os caracteres iguais no mesmo vértice, e depois analisar as relações de desigualdade. Seja $c_{i,j}$ uma célula da matriz que indica uma igualdade, ou seja, $c_{i,j} = 1$. Então, se $a = c+1-i$ se estamos interessados em obter um palíndromo par e $a = c - i$ se estamos interessados em obter um palíndromo ímpar, e se $b = c + j$, teremos as seguintes possibilidades, dependendo dos valores de $Map(a)$ e $Map(b)$:

- Se $Map(a) = 0$ e $Map(b) \neq 0$, então o caractere $S(a)$ deverá ser mapeado para o mesmo vértice do caractere $S(b)$, ou seja, $Map(a) = Map(b)$;

- Se $Map(a) \neq 0$ e $Map(b) = 0$, então o caractere $S(b)$ deve ser mapeado para o vértice do caractere $S(a)$, ou seja, $Map(b) = Map(a)$;

- Se $Map(a) = 0$ e $Map(b) = 0$, então deveremos mapear estes dois caracteres para um novo vértice inexistente, o que poderá ser feito usando o número atual disponível para numerar um vértice, dado por na, e incrementando este valor para impedir que um outro vértice criado posteriormente por este processo seja numerado com o mesmo valor, ou seja, devemos fazer $Map(a) = Map(b) = na$ e $na = na + 1$;

- Se $Map(a) \neq 0$ e $Map(b) \neq 0$, então $S(a)$ e $S(b)$ deverão estar no mesmo vértice. Se $Map(a) = Map(b)$, $S(a)$ e $S(b)$ já estão no mesmo vértice, e nada deve ser feito. Agora, se $Map(a) \neq Map(b)$, os caracteres representados pelos vértices $Map(a)$ e $Map(b)$ deveriam estar num mesmo vértice. Com isso,

vamos juntar estes dois vértices num só, que poderá receber, por convenção, o menor dos números atribuídos aos vértices. Para que os caracteres mapeados ao vértice de maior número estejam corretamente mapeados, deveremos procurar no vetor Map por todos os caracteres que estejam mapeados para este vértice, e remapeá-los para o vértice com o menor número que representa agora o único vértice resultante desta fusão.

Para obtermos todos os vértices do grafo, basta repetirmos o processo descrito anteriormente para todas as relações de igualdade existentes na matriz, e após a análise das relações de igualdade, devemos mapear os caracteres da cadeia que não foram mapeados pelo processo descrito anteriormente, significando que estes caracteres não são iguais a nenhum outro caractere. Isso pode ser feito varrendo-se o vetor, da esquerda para a direita, procurando por um caractere na posição a de S, tal que $Map(a) = 0$, e neste caso, deveremos mapear este caractere para um novo vértice numerado pelo número na atualmente disponível para numerarmos um vértice, e depois deveremos novamente incrementar o valor de na para evitar que outros vértices sejam numerados por este mesmo valor. Deveremos repetir este processo para cada uma destas posições existentes neste vetor. Após o processo de finalização descrito acima, todos os caracteres da cadeia S estarão mapeados para algum vértice do grafo.

Após determinarmos todos os vértices do grafo, deveremos agora analisar as relações de desigualdade da matriz. Seja $c_{i,j}$ uma célula da matriz que indique uma relação de desigualdade, ou seja, seja $c_{i,j} = 2$. Se a e b forem os mesmos valores dados anteriormente, então se $Map(a) = Map(b)$ teremos uma inconsistência, pois tentaremos criar uma aresta de um vértice para si próprio. Caso contrário, criaremos uma aresta entre os vértices numerados por $Map(a)$ e $Map(b)$, se esta já não existir. Após repetir o processo acima para todas as relações de desigualdade, teremos o grafo de relações representando esta matriz de condições, caso nenhuma inconsistência tenha sido encontrada ao construirmos este grafo. Caso exista alguma inconsistência, não teremos um grafo, e o número de casos será igual a 0.

Na figura 6.17 mostramos como construir o grafo para a matriz C_j calculada, na figura 6.16, para o estado $S_j = (-1, 2, 1, 0, -1)$. Na parte (a) da figura, mostramos o processo de mapeamento, e na parte (b) desta mesma figura, mostramos o grafo resultante após a adição das arestas entre os vértices geradas pelas relações de desigualdade dadas pela matriz da figura 6.16. Na figura 6.18 mostramos um exemplo

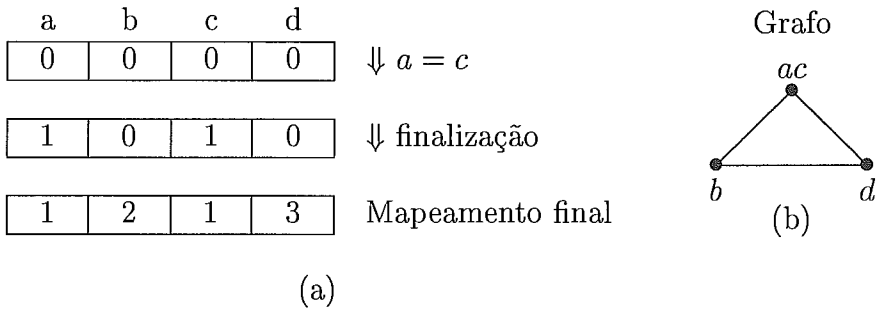


Figura 6.17: Exemplo da construção do grafo de relações, para a matriz de condições computada na figura 6.16. Na parte (a) desta figura, vemos que após a execução do mapeamento, teremos três vértices: ac, b e d. Na parte (b) desta figura, vemos o grafo resultante após a adição das arestas entre ac e b, ac e d, e b e d, provenientes das condições $b \neq c$, $a \neq d$ e $b \neq d$, respectivamente.

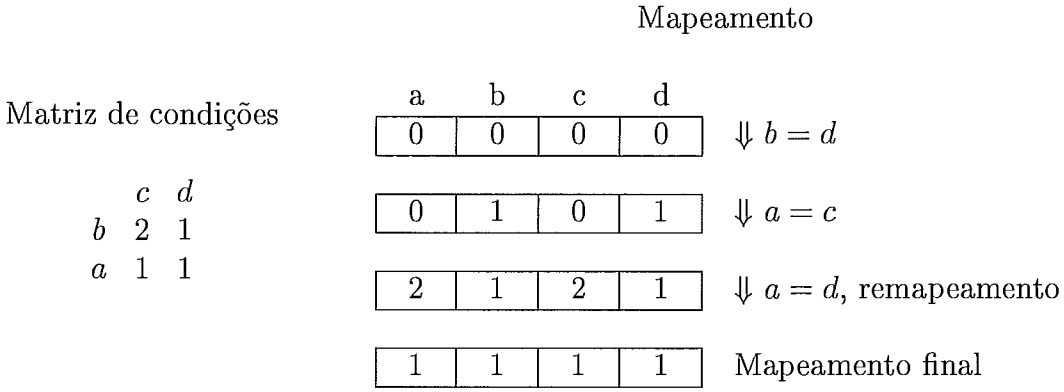


Figura 6.18: Nesta figura, tentamos montar o grafo de relações para a matriz de condições dada. Mas como o mapeamento mapeia todos os caracteres da cadeia S em um mesmo vértice, indicando que estes caracteres devem ser todos iguais, então não poderíamos ter nenhuma relação de desigualdade na matriz, mas como no nosso caso há a relação de desigualdade $b \neq c$, não existe o grafo para esta matriz, e o número de casos será igual a 0.

em que não há um grafo de relações devido à existência de uma inconsistência para o estado $S_j = (-1, 2, 2, 1, -1)$ do grafo da figura 6.14.

Agora vamos mostrar como calcular o número de casos, caso não tenham ocorrido inconsistências, através do cálculo do polinômio cromático. Isso será feito através do seguinte processo recursivo, descrito pelos passos dados a seguir:

1. Se o grafo não tiver arestas, então o número de casos será igual a σ^{N_v}, onde N_v é o número de vértices deste grafo.

2. Caso o grafo tenha arestas, vamos escolher qualquer uma destas arestas, digamos, uma aresta que liga os vértices numerados com a e b. Recursivamente calculamos o número de casos n_1 para um grafo gerado a partir do grafo original pela retirada desta aresta, e também recursivamente, o número de casos n_2 para o grafo obtido a partir do grafo original pela retirada desta aresta, e pela fusão dos vértices a e b num único vértice, não esquecendo de ajustar as arestas incidentes a estes vértices, após este processo de fusão. O número de casos para o grafo de entrada será igual a $n_1 - n_2$.

Na figura 6.19 mostramos como obter, usando o algoritmo recursivo descrito anteriormente, o número de casos para a matriz de condições do estado $S_j = (-1, 2, 1, 0, -1)$ dada na figura 6.16.

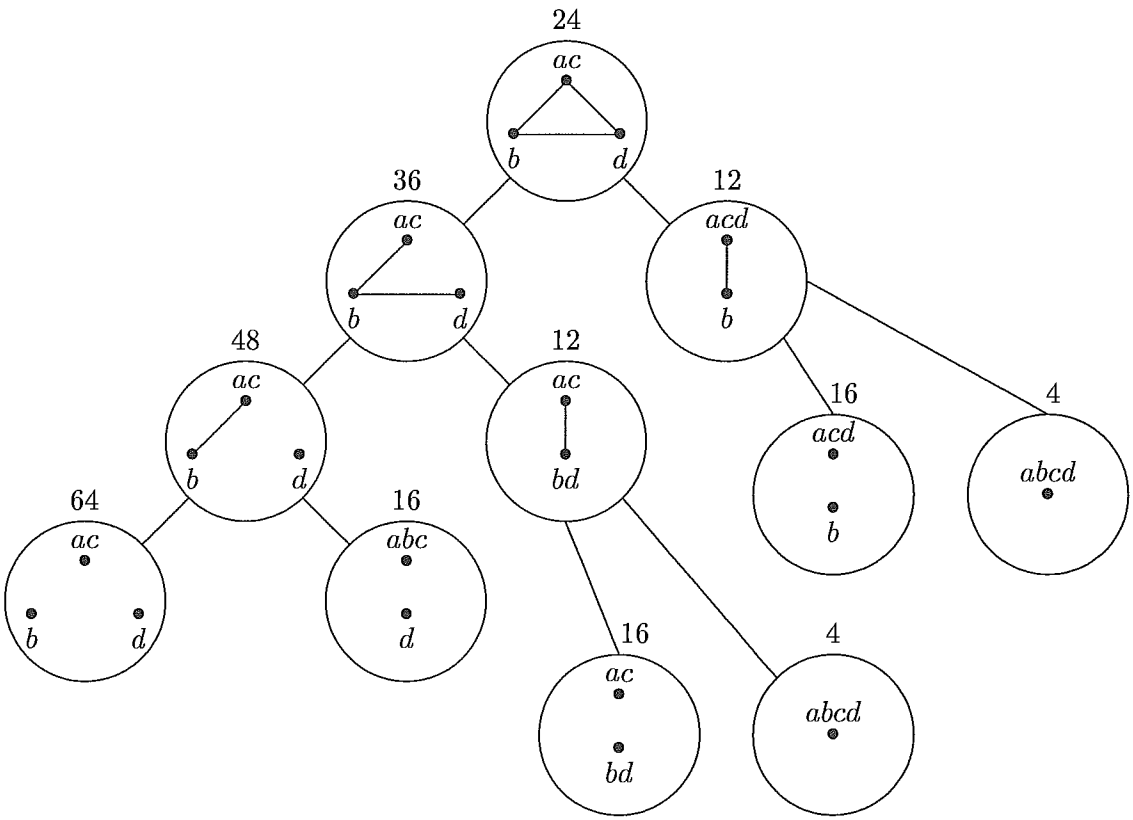


Figura 6.19: Nesta figura, mostramos como usar a idéia do cálculo de um polinômio cromático para calcular o número de casos para o grafo de relações dado na parte (b) da figura 6.17, através do algoritmo recursivo descrito, para um alfabeto com tamanho de $\sigma = 4$. Os números dados acima de cada um dos grafos parciais obtidos pelo processo recursivo dão o valor do polinômio cromático deste grafo para o tamanho do alfabeto dado.

Para calcularmos o número médio de passos, devemos saber, a partir do estado inicial, qual é o número de passos e a probabilidade de cada um dos caminhos do estado inicial até algum estado final. O número médio de passos será a soma dos produtos, para cada um destes caminhos, da probabilidade deste caminho vezes o número de passos gasto neste caminho em particular. Como nenhuma das otimizações descritas no capítulo anterior reduzem este número de passos, esta média deverá ser igual ao aplicarmos o nosso modelo em ambas as otimizações. Para calcularmos a área média basta que, ao calcularmos o comprimento de um caminho do estado inicial até algum dos estados finais, consideremos o tempo gasto ao passarmos de um estado S_i para um outro estado S_j, que será igual ao número de diagonais computadas ao gerarmos o estado S_j a partir do estado S_i. Como o número de diagonais computadas é igual ao número de diagonais ativas em S_i, e como o uso de faixas pela segunda otimização pode reduzir este número de diagonais, a área média para a segunda otimização em um dado exemplo deverá ser necessariamente menor ou igual a área média gerada pela primeira otimização para este mesmo exemplo. Podemos usar a mesma idéia para calcularmos estas médias a partir de um estado S_i não final genérico, ao invés do estado inicial. Se o estado S_i for um estado final, então, como não teremos mais computações, o número médio de passos e a área média serão iguais a 0.

Podemos calcular o número médio de passos e a área média através de um caminhamento em profundidade no grafo, em que visitamos todos os caminhos do estado inicial até o estado final. Este algoritmo recursivo irá nos possibilitar calcular estas médias sem precisarmos armazenar as probabilidades de transição de um estado S_i para um estado S_j, geradas pelos vários caminhos existentes do estado inicial até este estado S_i.

Este algoritmo recursivo será baseado na seguinte propriedade destas médias para qualquer vértice do nosso grafo: o número médio de passos de um estado não final S_i para qualquer um dos estados finais atingíveis a partir do estado S_i, para um dado caminho em particular do estado inicial até este estado S_i, será igual a soma, para cada um dos estados S_j gerados a partir do estado S_i^*, do produto da probabilidade de transição do estado S_i para o estado S_j vezes a soma de um mais o número médio de passos para este estado S_j. Devemos somar um antes de multiplicarmos pela probabilidade para contarmos o passo executado ao passarmos do estado S_i para o estado S_j.

A área média poderá ser calculada similarmente, mas ao invés de somarmos uma unidade para contabilizar a passagem do estado S_i para o estado S_j, devemos somar o número de diagonais ativas ao passarmos deste estado S_i para o estado S_j. O número de diagonais ativas será igual ao número de diagonais não invalidadas no estado S_i, limitadas por L_i e L_s, mais o número de diagonais que acabaram de ser inicializadas, ao computarmos o estado mínimo S_i^* gerado por este estado S_i. A propriedade descrita anteriormente somente deverá ser aplicada se a probabilidade de transição do estado S_i para o estado S_j for diferente de 0, pois se esta probabilidade for igual a zero, este estado não irá colaborar na soma do número médio de passos e na soma da área média, pois não poderemos atingir nenhum estado final passando por este estado S_j.

Antes de mostrarmos o algoritmo recursivo, iremos provar a propriedade descrita acima. Como cada um dos caminhos do estado S_i até algum estado final depende do estado S_j em que este caminho em particular começa, vamos agrupar na soma original, dada pela descrição ao calcularmos as médias a partir do estado inicial, todos os caminhos que passam por um certo estado S_j. Se S_j é um estado final, a propriedade está correta. Agora basta provarmos, para um estado não final S_j em particular, que a soma das médias dos caminhos que passam por este estado será igual a probabilidade de transição do estado S_i para este estado S_j vezes a soma do tempo gasto ao passarmos do estado S_i para o estado S_j com a média para este estado S_j. O tempo gasto na transição será ou igual a um, ou igual ao número de diagonais ativas, dependendo de estarmos calculando o número médio de passos ou a área média. Vamos supor que existam x caminhos do estado S_i até algum estado final passando por S_j, ou seja, x caminhos partindo do estado S_j. Vamos ainda supor que os comprimentos destes caminhos a partir do estado S_j são iguais a l_i, que as probabilidades destes caminhos a partir do estado S_j são dadas por P_i, $1 \leq i \leq x$, que o tempo gasto ao passarmos do estado S_i ao estado S_j é igual a t, e que a probabilidade de transição de S_i para S_j é igual a P. Então, a soma das médias dos caminhos que passam por este estado S_j será igual a:

$$\begin{aligned}&(t+l_1)\times P\times P_1+(t+l_2)\times P\times P_2+\cdots+(t+l_x)\times P\times P_x=\\&(\underbrace{(l_1\times P_1+l_2\times P_2+\cdots+l_x\times P_x)}_{\text{média para } s_j}+t\times(P_1+P_2+\cdots+P_x))\times P\end{aligned}$$

Como $P_1+P_2+\cdots+P_x=1$, pois deveremos sempre atingir um estado final a partir de um caminho válido, e como a primeira parcela da última soma dada acima

é igual a média para o estado S_j, provamos o que foi descrito acima para esta parcela da soma para o cálculo da média para o estado S_i. Com isso, podemos calcular esta soma usando a propriedade descrita anteriormente.

Vamos agora descrever o algoritmo recursivo que irá calcular o número médio de passos e a área média de um grafo gerado a partir de um exemplo particular, usando alguma das duas otimizações. Este algoritmo será composto pelos seguintes passos:

1. Para o estado inicial, como ainda não executamos nenhum passo do algoritmo, o número de casos será igual a σ^N e a matriz de condições será a matriz nula. Aplicamos, recursivamente, os passos de 2 a 4 descritos abaixo para calcular o número médio de passos e a área média para o estado inicial, o que dará as médias para calcularmos o palíndromo para o exemplo em particular descrito pelo grafo que foi construído.

2. Seja S_i o estado para o qual desejamos calcular o número médio de passos e a área média, seja C_i a matriz de condições necessária para atingirmos este estado a partir de algum caminho do estado inicial, seja x o número de casos possíveis para este caminho, e seja N_m o número médio de passos e A_m a área média para este estado S_i. Vamos inicializar ambas as médias com o valor 0. Se o estado S_i não é um estado final, deveremos executar os seguintes passos:

3. Para um dado estado S_j gerado a partir do estado mínimo S_i^*, executaremos os seguintes passos:

   - Determinar os limites L_i e L_s, que para a primeira otimização, poderão ser calculados determinando-se a faixa de diagonais já inicializadas em S_i, e através da correção desta faixa, se usarmos a segunda otimização, pelo método que já foi descrito anteriormente para determinar as faixas. Calcular L_i' e L_s' a partir destes limites, usando os seus valores para identificar se podemos inicializar mais diagonais.
   - Calcular a matriz de condições C_j a partir da matriz C_i, usando o procedimento descrito anteriormente.
   - Calcular o grafo de relações a partir da matriz C_j, se for possível a construção desta matriz, através do método de mapeamento.
   - Calcular o número de casos y usando o grafo de relações, se foi possível a construção deste grafo, através do cálculo do polinômio cromático. Caso

não seja possível construir o grafo ou a matriz, o número de casos será igual a 0.

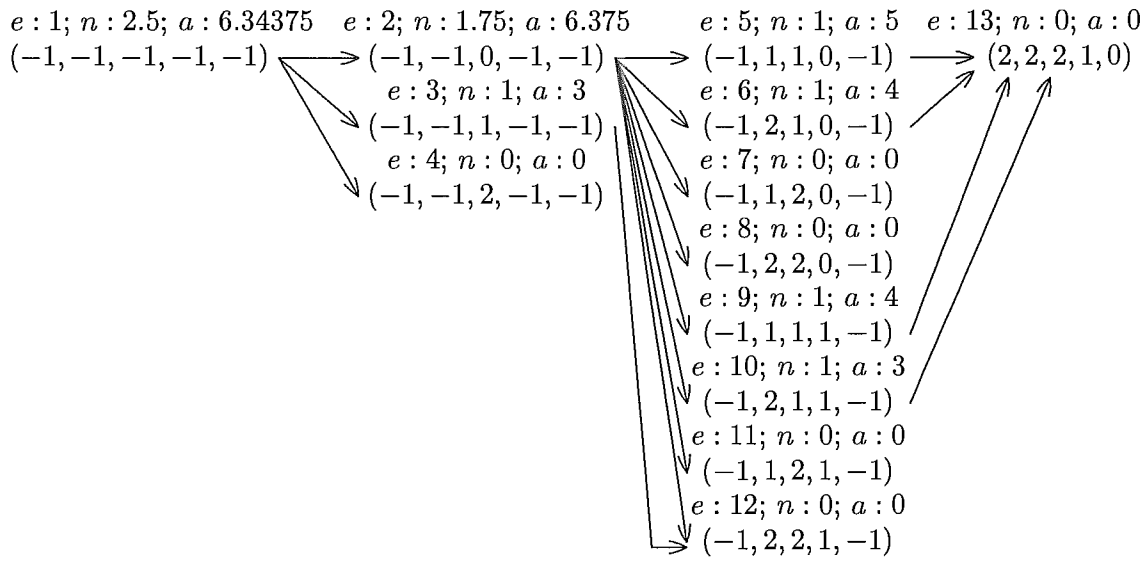
- Calcular a probabilidade de transição de S_i para S_j, dada por $P = y/x$.
- Se o número de casos for diferente de 0, calcular recursivamente o número médio de passos N'_m e a área média A'_m para o estado S_j, dada a matriz C_j e o número de casos y. Somar a N_m o valor dado por $(1 + N'_m) \times P$ e a A_m o valor dado por $(t + A'_m) \times P$, onde t dá o número de diagonais ativas ao calcularmos o estado S_j.

4. Repetir o processo acima para todos os estados S_j gerados a partir deste estado S_i.

Como um exemplo da aplicação do algoritmo recursivo descrito anteriormente, na figura 6.20 é descrito como é calculado o número médio de passos e a área média para cada um dos estados, para o grafo do exemplo construído na figura 6.14, usando este algoritmo. Após calcularmos as médias para o estado inicial $(-1, -1, -1, -1, -1)$, concluímos que o número médio de passos é igual a 2.5, e que a área média é igual a 6.34375. Na figura 6.21, descrevemos o mesmo processo para o grafo da figura 6.15, e descobrimos que neste caso, o número médio de passos é igual a 2.5, e que a área média é igual a 6.0625. Como os grafos das figuras 6.20 e 6.21 foram obtidos a partir do mesmo exemplo, mas aplicando-se a primeira otimização ao gerarmos o grafo da figura 6.20 e a segunda otimização ao gerarmos o grafo da figura 6.21, o número médio de passos foi o mesmo em ambos os casos, e como já era esperado, a área média ao usarmos a segunda otimização é menor do que se usarmos a primeira.

6.2.2 Análise da Precisão do Modelo

Nesta subseção iremos analisar os resultados retornados pelo modelo descrito na subseção anterior, mas como o número de palíndromos aproximados existentes em uma cadeia S depende do seu tamanho N, iremos analisar a precisão da área média e do número médio de passos para o cálculo de todos os palíndromos aproximados nesta cadeia. O número médio de passos e a área média para toda a cadeia S podem ser obtidos a partir da média aritimética destas médias para cada posição c particular desta cadeia S, onde podemos ter um palíndromo aproximado par e um palíndromo aproximado ímpar (com exceção da posição 1).



(a)

Probabilidades das transições entre os estados

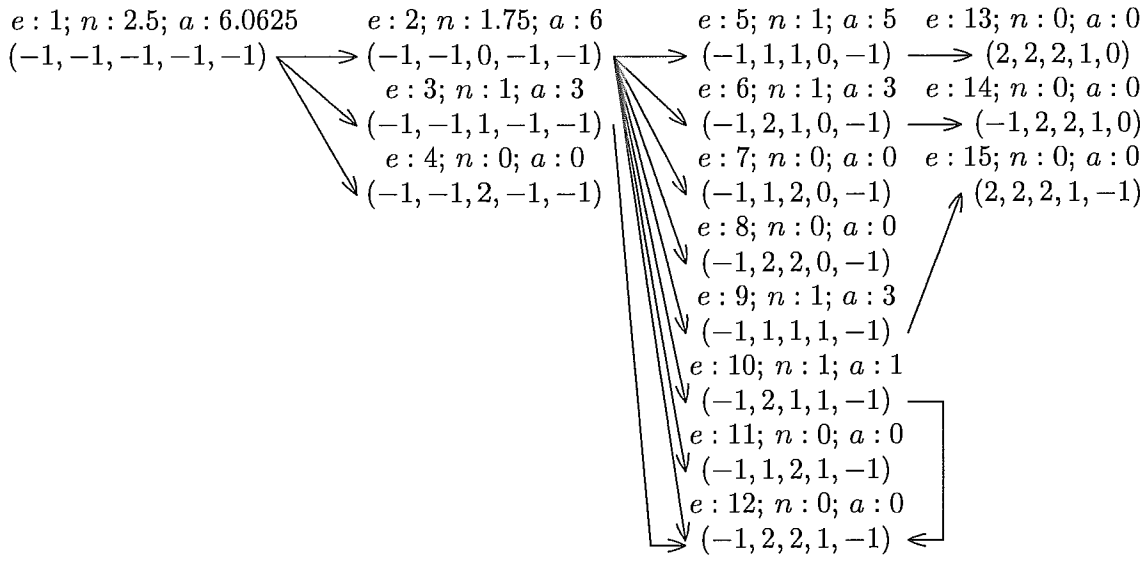
$1 \to 2 : 0.75$	$1 \to 3 : 0.1875$	$1 \to 4 : 0.0625$	$2 \to 5 : 0.4375$
$2 \to 6 : 0.125$	$2 \to 7 : 0.125$	$2 \to 8 : 0.0625$	$2 \to 9 : 0.125$
$2 \to 10 : 0.0625$	$2 \to 11 : 0.0625$	$2 \to 12 : 0$	$3 \to 12 : 1.0$
$5 \to 13 : 1.0$	$6 \to 13 : 1.0$	$9 \to 13 : 1.0$	$10 \to 13 : 1.0$

(b)

Figura 6.20: Nesta figura, mostramos como o algoritmo recursivo descrito anteriormente é usado para calcular o número médio de passos e a área média. Na parte (a) desta figura, mostramos, para cada um dos estados, um número identificando o estado dado por e, o número médio de passos para atingirmos algum estado final a partir deste estado dado por n, e a área média necessária para atingirmos algum estado final dada por a. Na parte (b) da figura, damos as probabilidades das transições entre os estados (vértices) deste grafo. Como vimos na figura 6.14, este grafo foi construído para o exemplo $N = 4$, $\sigma = 4$ e $k = 2$, para calcular um palíndromo par na posição 2 da cadeia, através da primeira otimização.

Para avaliarmos a precisão deste modelo probabilístico, executamos exemplos sobre este modelo para as duas otimizações, para N igual a 10, 20, 30, 40 e 50, com o número de erros máximos $k = 1$, e o tamanho do alfabeto σ igual a 1, 2, 4, 32, 128 e 256, e para $N = 10$, com o número de erros k igual a 1, 2, 3, 4 e 5, e com o tamanho do alfabeto σ igual a 1, 2, 4, 32, 128 e 256.

Executamos estes mesmos exemplos sobre os algoritmos que implementam as duas otimizações, computamos o número médio de passos e a área média da melhor conversão gerada pelos algoritmos que implementam estas duas otimizações,



(a)

Probabilidades das transições entre os estados

$1 \to 2 : 0.75$	$1 \to 3 : 0.1875$	$1 \to 4 : 0.0625$	$2 \to 5 : 0.4375$
$2 \to 6 : 0.125$	$2 \to 7 : 0.125$	$2 \to 8 : 0.0625$	$2 \to 9 : 0.125$
$2 \to 10 : 0.0625$	$2 \to 11 : 0.0625$	$2 \to 12 : 0$	$3 \to 12 : 1.0$
$5 \to 13 : 1.0$	$6 \to 14 : 1.0$	$9 \to 15 : 1.0$	$10 \to 12 : 1.0$

(b)

Figura 6.21: Nesta figura, mostramos como o algoritmo recursivo descrito anteriormente é usado para calcular o número médio de passos e a área média. Na parte (a) desta figura, mostramos, para cada um dos estados, um número identificando o estado dado por e, o número médio de passos para atingirmos algum estado final a partir deste estado dado por n, e a área média necessária para atingirmos algum estado final dada por a. Na parte (b) da figura, damos as probabilidades das transições entre os estados (vértices) deste grafo. Como vimos na figura 6.15, este grafo foi construído para o exemplo $N = 4$, $\sigma = 4$ e $k = 2$, para calcular um palíndromo par na posição 2 da cadeia, através da segunda otimização.

para calcular todos os palíndromos aproximados da cadeia S de tamanho igual a N, e comparamos os resultados reportados pela execução destes algoritmos com os mesmos resultados obtidos pela aplicação do modelo probabilístico, o que nos dará uma idéia da precisão do nosso modelo. O número médio de passos e a área média para toda a cadeia, na execução dos algoritmos que implementam as otimizações, serão calculados usando a mesma idéia do cálculo destas médias no modelo probabilístico, a partir do número médio de passos e da área média para computar cada palíndromo aproximado. O número médio de passos e a área média para ca-

da palíndromo serão calculados como a média do número de passos e da área ao determinarmos o palíndromo nesta posição para uma cadeia genérica S gerada de forma aleatória, para 10000 cadeias aleatórias diferentes. Podemos notar na prática que quanto mais se aproximar o número de cadeias aleatórias diferentes do número máximo de cadeias diferentes σ^N, onde N é o tamanho da cadeia e σ é o tamanho do alfabeto, mais próximas estarão as médias dadas pela execução do algoritmo daquelas indicadas pelo modelo probabilístico.

Como para o exemplo onde $N = 10$ e $\sigma = 2$ temos somente 1024 cadeias distintas, e como para $\sigma = 1$ temos somente uma cadeia em cada exemplo, calcularemos estes exemplos para 1024 cadeias e para uma cadeia, respectivamente, ao invés de 10000 cadeias, e nestes casos, o número médio de passos e a área média deverão ser exatamente os mesmos obtidos pelo modelo probabilístico, pois executamos os algoritmos para todas as possíveis cadeias existentes neste problema.

Na figura 6.22 é mostrada, para ambas as otimizações, o cálculo do número médio de passos pelo modelo probabilístico, para um número fixo de erros $k = 1$, para N igual a 10, 20, 30, 40 e 50, e para σ igual a 1, 2, 4, 32, 128 e 256, na parte (a) da figura, e pela execução do algoritmo sobre estes mesmos exemplos para 10000 cadeias aleatórias na parte (b) desta mesma figura. Na figura 6.23, mostramos as mesmas figuras descritas anteriormente, mas agora o tamanho da cadeia $N = 10$ é fixo, ao invés do número de erros k, que varia de 1 até 5. Além disso, como neste caso o número de passos totais é diferente para cada exemplo, mostramos as porcentagens de ganho das otimizações, ao invés destes valores. Nas figuras 6.24 e 6.25 mostramos os mesmos exemplos das figuras 6.22 e 6.23, respectivamente, mas agora analisamos a precisão do modelo ao calcular a área média ao invés do número médio de passos para a primeira otimização. A análise da precisão do modelo ao calcularmos a área média usando a segunda otimização é dada pela figura 6.26 para o primeiro exemplo, e pela figura 6.27 para o segundo exemplo.

Como podemos ver pela figura 6.22 para o primeiro exemplo descrito acima, a redução do número médio de passos em relação ao número total de passos não é muito significativa. Isso ocorre porque a única forma de reduzirmos o número de passos é atingir o ponto máximo da diagonal $d_m = m - n$, se existir, que contém o ponto (m, n), antes do final do último passo k, e isso requer que invalidemos esta diagonal antes deste passo. Como devemos invalidar a diagonal d_m, o aumento do tamanho do alfabeto irá reduzir a probabilidade de casamentos sobre as diagonais,

o que reduzirá a probabilidade dos caminhos do estado inicial até algum estado final que tenha esta diagonal invalidada, reduzindo assim o ganho em relação ao número de passos totais. Da mesma forma, como o aumento do tamanho da cadeia irá aumentar o número de pontos das diagonais, assim como irá aumentar o número de possíveis estados finais, e de caminhos do estado inicial até algum estado final, também teremos a redução da probabilidade destes caminhos, e portanto, dos ganhos das otimizações.

No gráfico do segundo exemplo executado, dado pela figura 6.23, vemos também que o aumento do tamanho do alfabeto reduz o ganho em relação ao número máximo de passos, devido à redução das probabilidades dos caminhos do estado inicial até algum estado final que terminem antes do último passo, mas também vemos que o aumento do número de erros aumenta o ganho do nosso algoritmo, e isso é devido ao aumento do número máximo de passos executados, o que aumenta o tamanho máximo do caminho do estado inicial até o estado final, dado por $k + 1$, e também aumenta o número de caminhos menores que terminam antes do último passo.

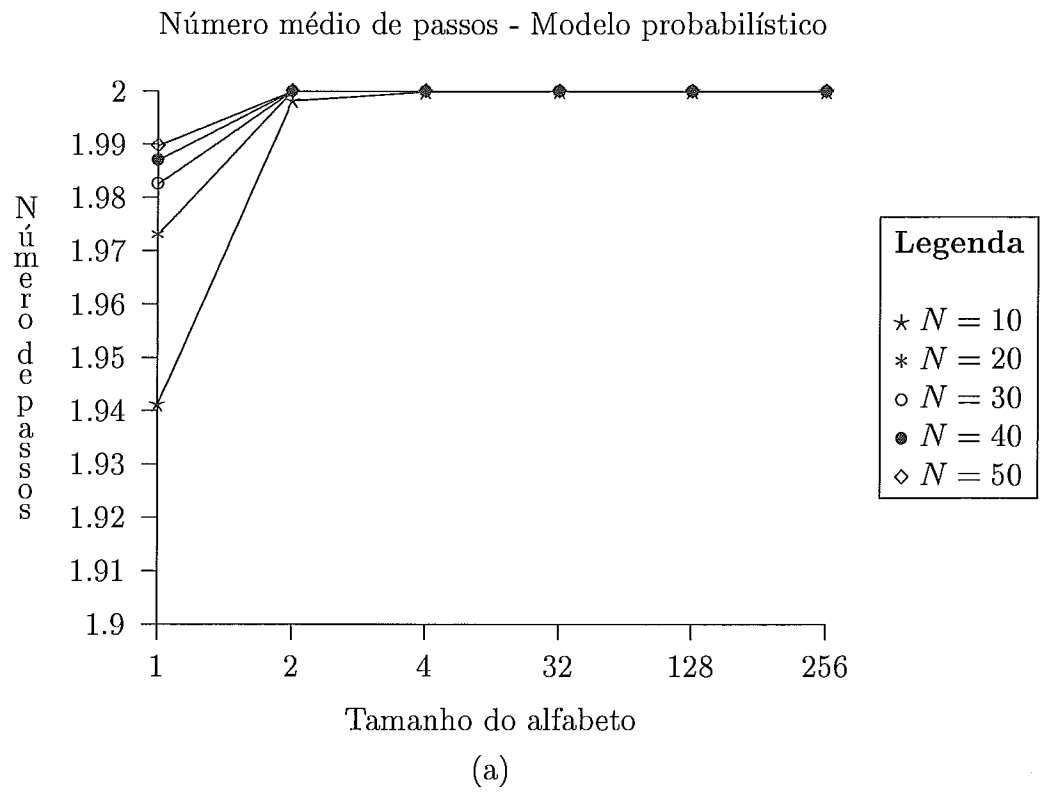
Nas figuras 6.24 e 6.26 temos os gráficos dos ganhos das áreas médias em relação as áreas totais para o primeiro exemplo, usando a primeira e a segunda otimização, respectivamente. Como podemos ver por estas figuras, há um aumento da área ao aumentarmos o tamanho do alfabeto e o tamanho da cadeia S, e isso ocorre tanto devido aos fatos descritos anteriormente ao analisarmos o número médio de passos, como devido ao fato do aumento do número de pontos das diagonais reduzir a probabilidade de invalidarmos uma diagonal, e portanto, a probabilidade de reduzirmos o tamanho de um passo, assim como o aumento do tamanho do alfabeto irá reduzir a probabilidade de casamentos, o que também reduzirá a probabilidade de redução da área dentro de um passo.

Também podemos ver por estas figuras que a segunda otimização produz melhores ganhos do que a primeira otimização, e que este ganho relativo é menor ao aumentarmos o tamanho da cadeia e o tamanho do alfabeto, devido à redução da probabilidade de invalidarmos uma diagonal, que irá limitar a definição de faixas pela segunda otimização.

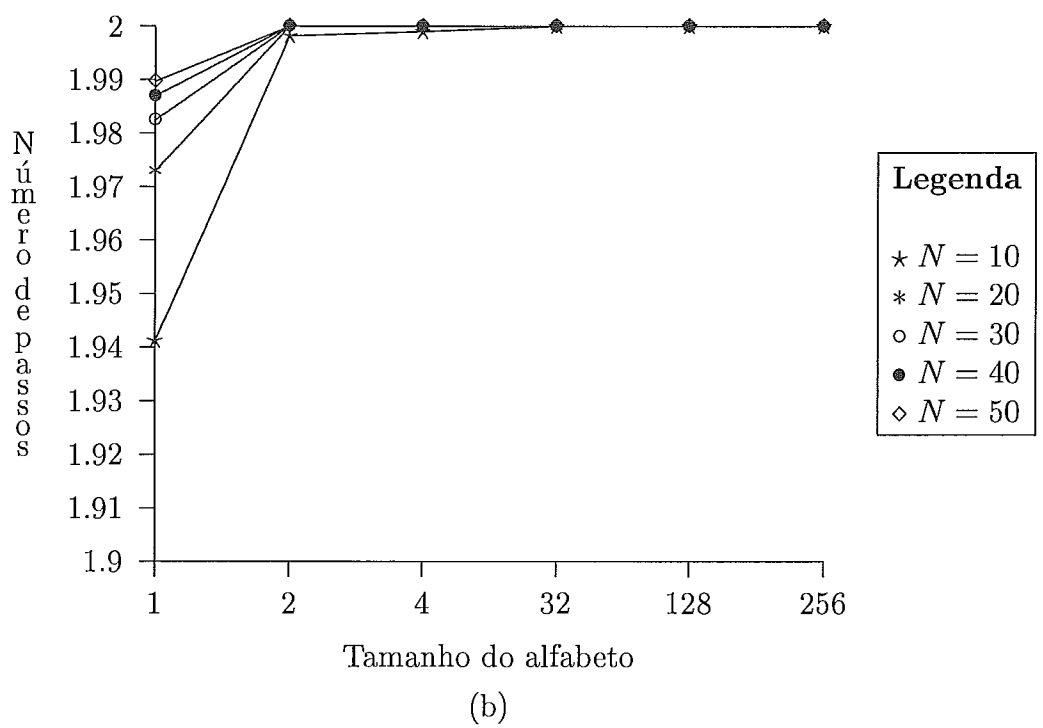
Nos gráficos das figuras 6.25 e 6.27, que dão os ganhos das áreas médias em relação as áreas totais para o segundo exemplo, usando as duas otimizações, vemos novamente que o aumento do número de erros irá aumentar os ganhos, pois neste caso além de o número de erros aumentar o número de passos máximos, ele irá

aumentar o tamanho máximo de um passo, o que aumentará, em conseqüência, as áreas totais. Nestes exemplos, também podemos ver que a segunda otimização gera melhores ganhos do que a primeira otimização. As reduções dos ganhos ocorrem devido aos mesmos fatores já descritos anteriormente.

Podemos notar, em todas as figuras dadas, que o ganho do número médio de passos e da área média é maior para o caso $\sigma = 1$, o que ocorre devido ao fato de a probabilidade de casamentos entre os caracteres ser igual a 1.0 neste caso, o que irá permitir termos as maiores reduções do número de passos e da área, pois as diagonais serão invalidadas no mesmo passo em que são inicializadas.

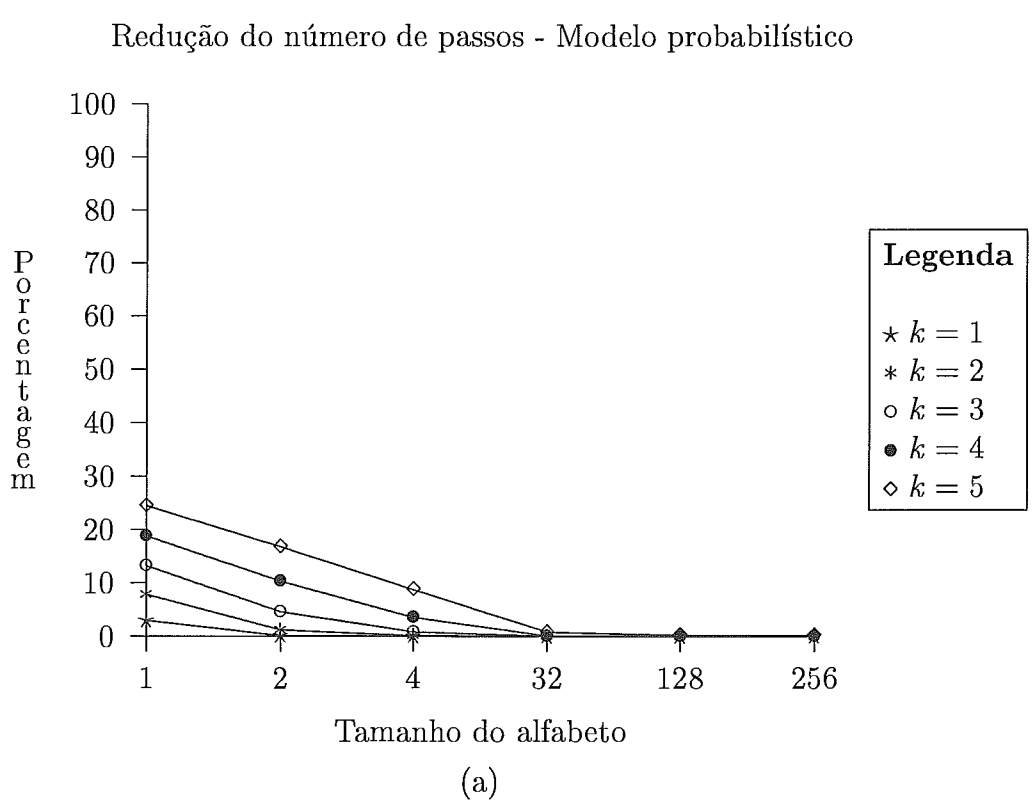


(a)

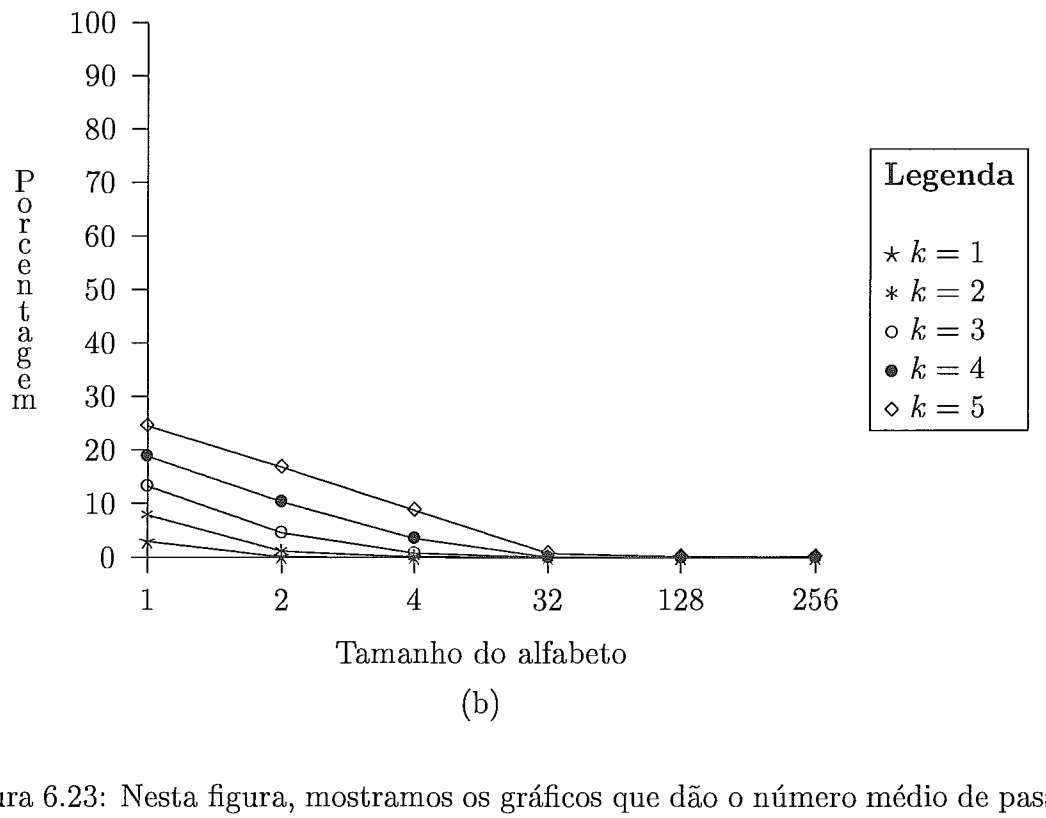


(b)

Figura 6.22: Nesta figura, mostramos os gráficos que dão o número médio de passos para o primeiro exemplo, onde k é fixo, para ambas as otimizações.

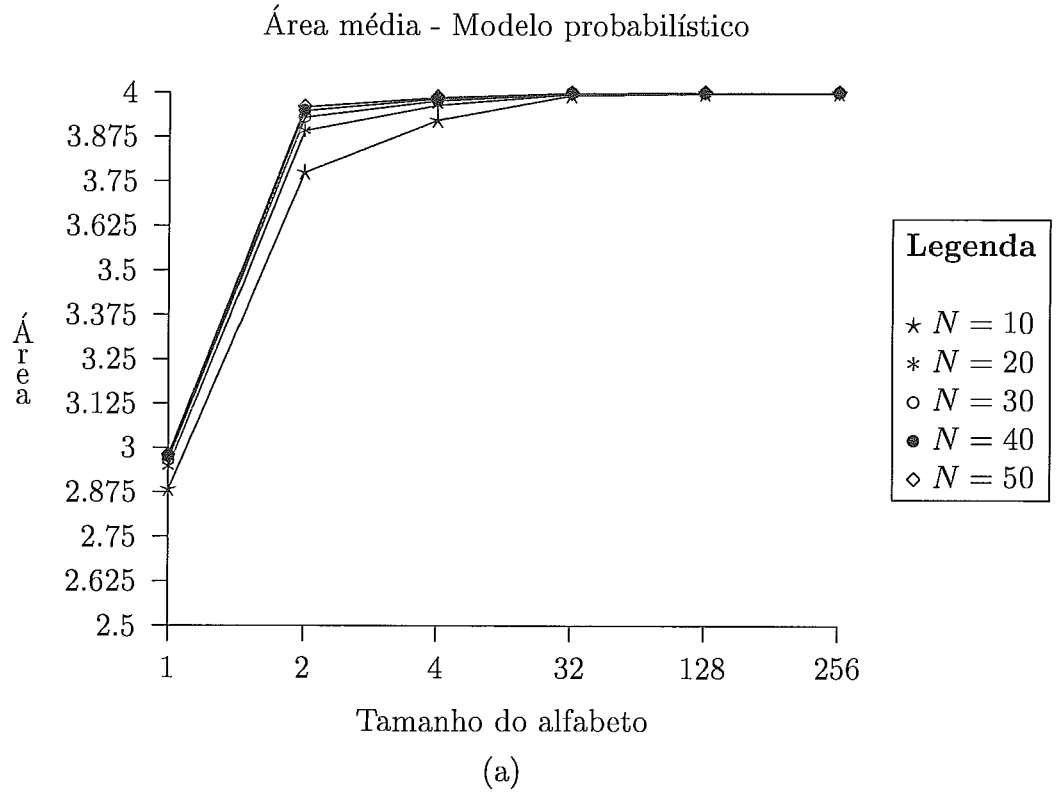


(a)



(b)

Figura 6.23: Nesta figura, mostramos os gráficos que dão o número médio de passos para o segundo exemplo, onde N é fixo, para ambas as otimizações.



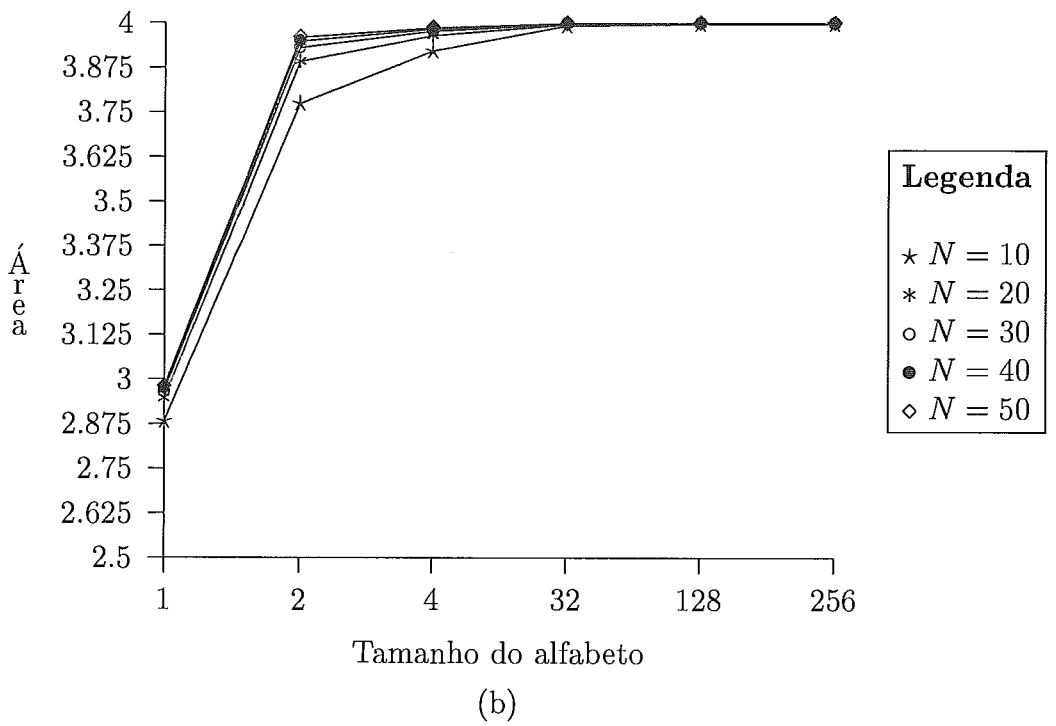
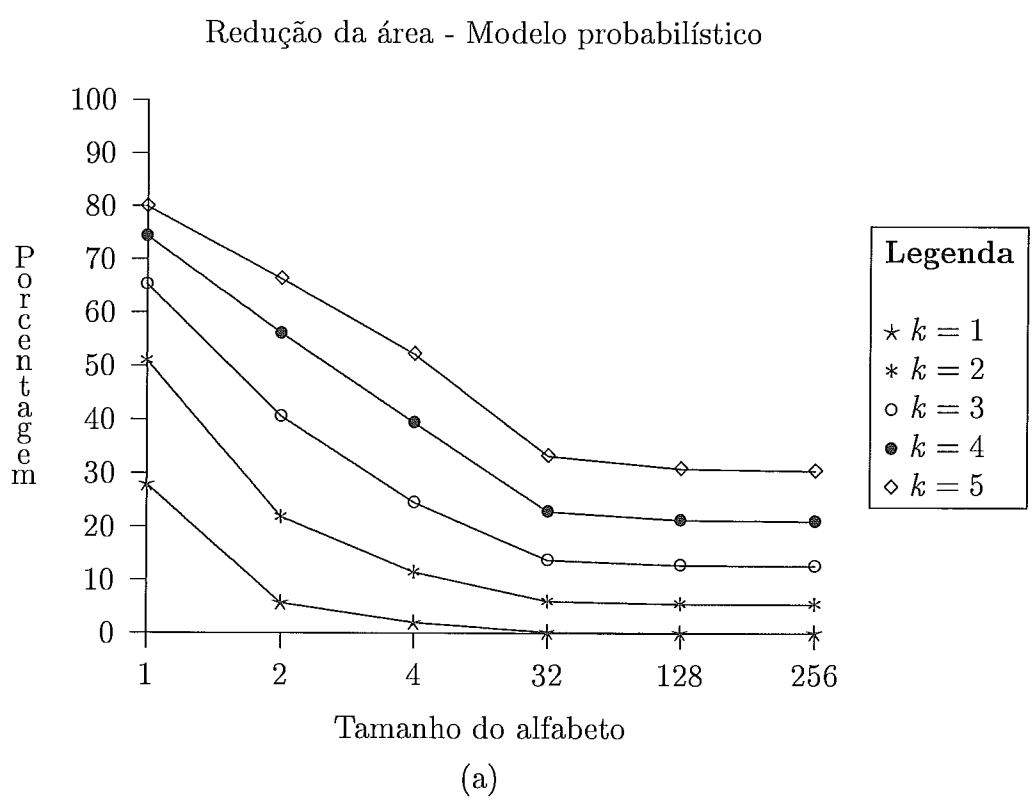
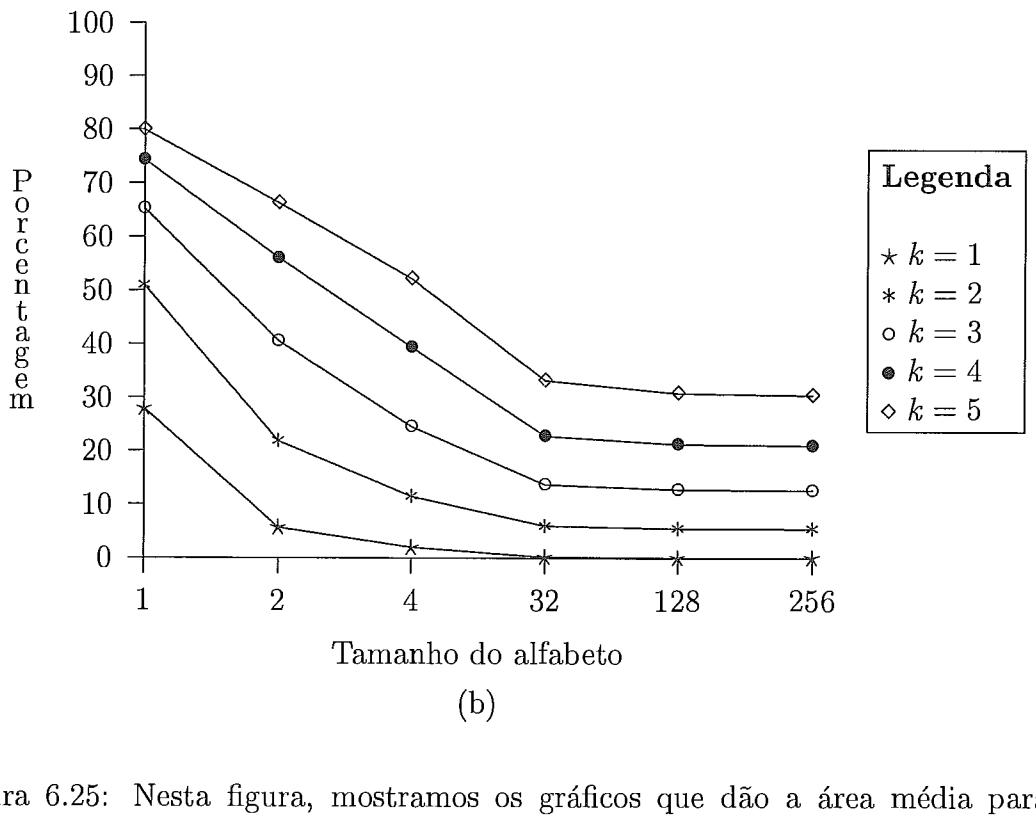


Figura 6.24: Nesta figura, mostramos os gráficos que dão a área média para o primeiro exemplo, onde k é fixo, para a primeira otimização.

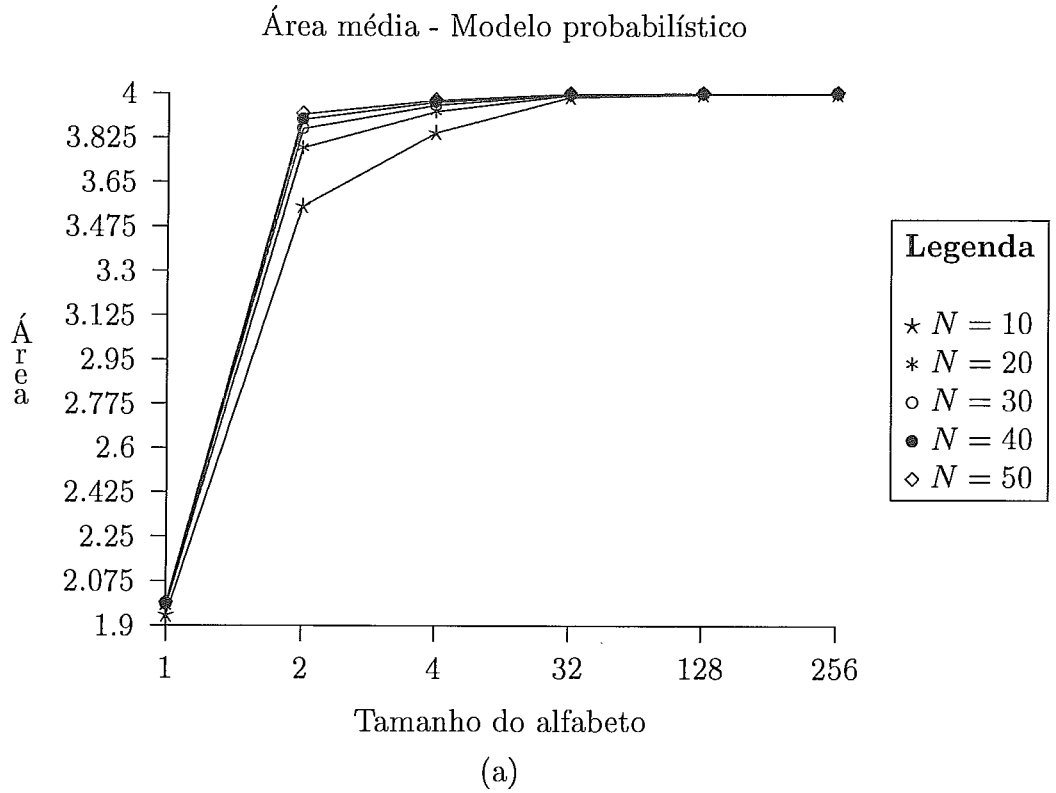


(a)

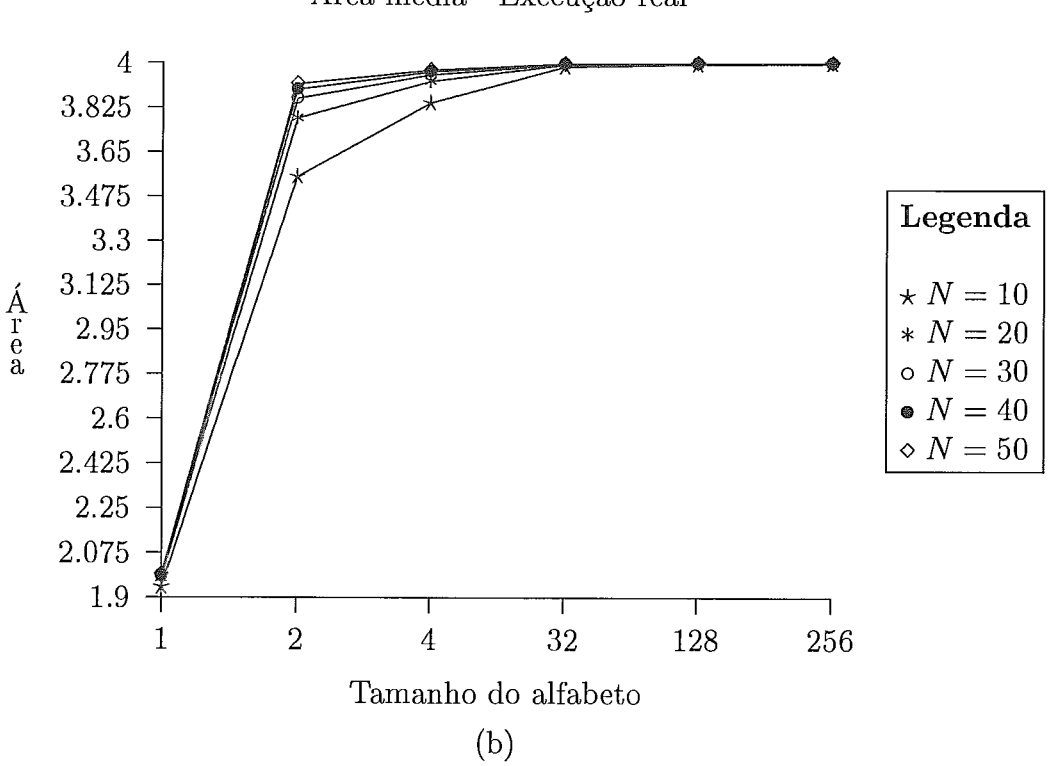


(b)

Figura 6.25: Nesta figura, mostramos os gráficos que dão a área média para o segundo exemplo, onde N é fixo, para a primeira otimização.

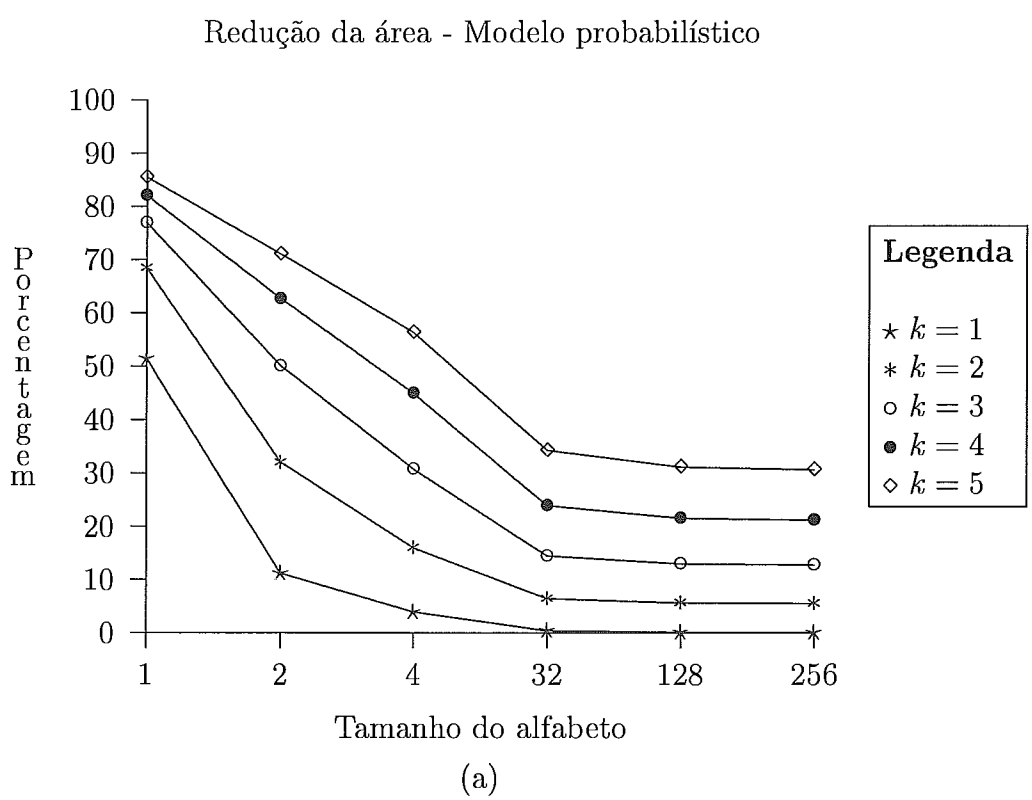


(a)

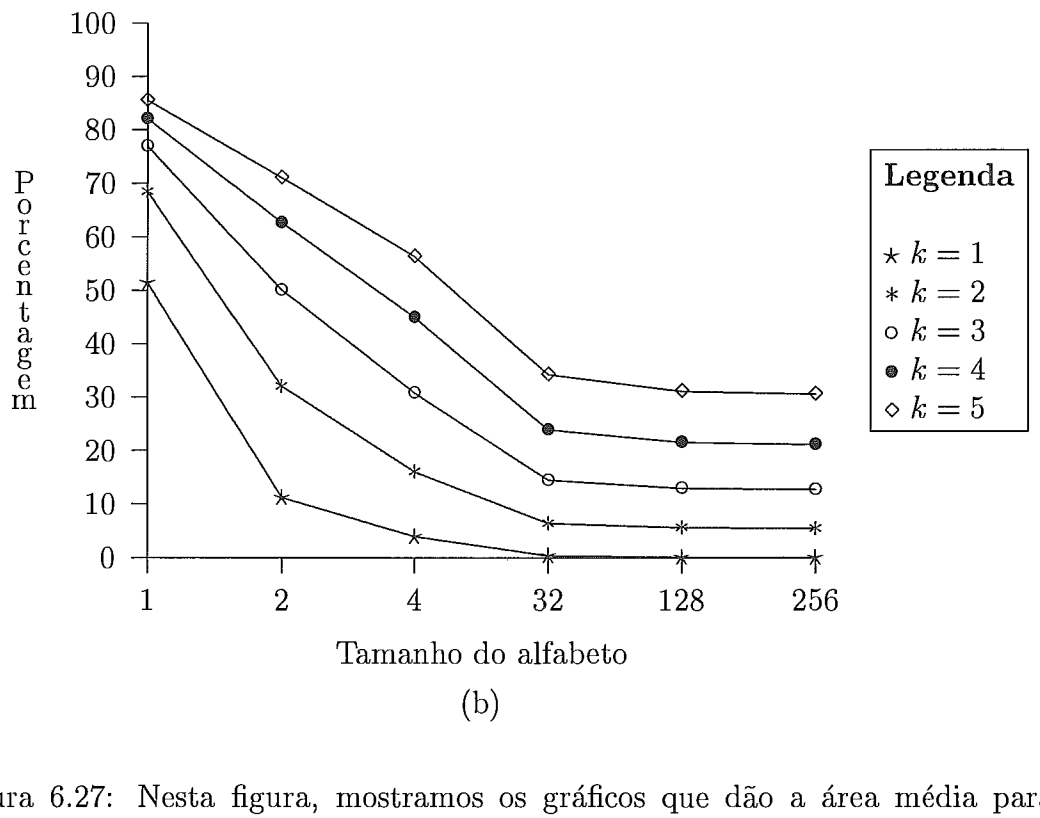


(b)

Figura 6.26: Nesta figura, mostramos os gráficos que dão a área média para o primeiro exemplo, onde k é fixo, para a segunda otimização.



(a)



(b)

Figura 6.27: Nesta figura, mostramos os gráficos que dão a área média para o segundo exemplo, onde N é fixo, para a segunda otimização.

Capítulo 7

Conclusão

Neste último capítulo, descreveremos o que foi concluído do estudo realizado nos capítulos anteriores, e o que ainda pode ser feito em relação ao problema proposto no futuro.

7.1 Conclusão

Nesta tese apresentamos um algoritmo para calcular todos os palíndromos aproximados em uma dada cadeia S. Nós definimos a noção do que é um palíndromo aproximado par e ímpar, pois não encontramos uma definição adequada para ser aplicada. A nossa definição de palíndromos aproximados foi baseada no problema de edição de cadeias devido tanto a este algoritmo permitir inserções e deleções de caracteres além de descasamentos (como na definição simples de palíndromos não exatos dada em [17, Cap. 9], como pela grande quantidade de algoritmos existentes propostos aos casos particulares deste problema geral. Foi concluído que o problema de k-diferenças era o mais adequado ao nosso caso, e com isso, limitamos o número de erros do palíndromo a no máximo k erros.

Após estudarmos os vários algoritmos existentes para os casos particulares do problema de edição de cadeias, decidimos que aplicar uma variação do algoritmo descrito em [17, 26] para o problema de casamento de cadeias aproximado seria a forma mais eficiente de implementar o nosso problema. Primeiro mostramos como aplicar este algoritmo ao problema de k-diferenças para depois adequá-lo ao nosso caso, gerando um algoritmo com tempo de execução de pior caso de $O(k^2)$ ao calcular um palíndromo de uma cadeia S de tamanho igual a N. Esse algoritmo gerou um algoritmo com tempo de execução total de $O(k^2N)$ para o cálculo de todos os palíndromos, que até o momento é o melhor algoritmo para este problema.

Ao analisarmos esta conversão, notamos que duas otimizações poderiam ser aplicadas ao nosso caso, baseadas no fato de que uma diagonal não será mais usada após atingirmos o seu último ponto. A primeira otimização simplesmente altera o modo de execução do algoritmo, usando uma lista que mantém somente as diagonais para as quais ainda não atingimos o seu último ponto. Esta otimização pode ser também aplicada ao problema de k-diferenças. A segunda otimização, além de retirar estas diagonais da execução, retirará outras diagonais que certamente não serão usadas para gerarmos um palíndromo aproximado em particular. Apesar de esta otimização ser baseada no nosso problema em particular, ela poderá ser aplicada ao problema de k-diferenças, pois a célula (m, n) usada neste problema tem a maior soma.

Apesar de as duas otimizações não reduzirem o tempo de execução de pior caso de $O(k^2N)$ ao calcularmos os palíndromos em uma cadeia S, elas reduzem bem o tempo na prática, como podemos ver pelos gráficos dados no apêndice B, pelos gráficos estudados no capítulo anterior, e pelos resultados obtidos no capítulo anterior ao calcularmos a área média através do modelo probabilístico que foi definido neste capítulo. Além disso, como podemos ver por estes mesmos gráficos, assim como pelos resultados retornados pelos modelos probabilísticos, a segunda otimização também melhora o tempo da primeira otimização, apesar deste melhoramento não ser tão significativo, pois uma boa otimização neste algoritmo dependerá da freqüência de casamentos, que deverá ser maior na medida em que reduzimos a faixa de diagonais a serem executadas.

7.2 Estudos Futuros

Nesta tese nós estudamos uma implementação seqüencial para o problema da busca de todos os palíndromos pares e ímpares em uma dada cadeia S de tamanho igual a N num tempo total de $O(k^2N)$, onde k é o número máximo de erros permitidos, através da adaptação de um algoritmo de k-diferenças que é baseado em árvores de sufixos e na busca do último ancestral comum de dois nós desta árvore. Depois de mostrarmos como adaptar este algoritmo ao nosso problema, apresentamos duas otimizações que apesar de não reduzirem o tempo de pior caso do nosso algoritmo, tem um bom ganho para exemplos práticos, como podemos ver pelos gráficos dados no apêndice B.

Então, uma possibilidade é verificar se existem mais otimizações que possam re-

duzir mais o tempo de execução, e até talvez o tempo de pior caso. Outra alternativa é aplicar algum algoritmo melhor do que a adaptação do problema de k-diferenças ao nosso problema, que tenha um tempo de execução total menor do que $O(k^2)$.

Uma outra possibilidade é a de estudar como podemos implementar as idéias descritas no capítulo 5 para calcularmos todos os palíndromos aproximados em uma cadeia S paralelamente de forma eficiente, e não da forma não ótima e simples descrita no final do capítulo 5. Um bom começo é visualizar o nosso problema sobre um grafo ao invés de uma matriz, e tentar adaptar os algoritmos descritos em [2], que são bem eficientes em modelos de computação PRAM, para o problema de edição de cadeias, que podem ser aplicados em todos os casos, inclusive nos casos em que os pesos da inserção, da deleção, e da substituição variam de acordo com os caracteres ou com as operações.

Também faltou calcularmos uma fórmula para o tempo de execução médio, dado ao calcularmos a área média usando o modelo probabilístico dado no capítulo 6, em função de σ, k e N, o que poderia ser feito através da aplicação do modelo probabilístico a um caso genérico.

Apêndice A

Tabelas com as Áreas Totais

Neste apêndice, damos as tabelas com as áreas totais para os palíndromos pares e ímpares, para todos os exemplos executados sobre os algoritmos que implementam a primeira e a segunda otimização. Lembremos de que os números de erros são baseados em uma certa porcentagem do tamanho N da cadeia S. Os nomes das cadeias exemplo foram dados segundo as convenções já definidas no capítulo 6. Na primeira seção, damos as tabelas com as áreas para o algoritmo que implementa a primeira otimização, e na última seção, damos as tabelas com as áreas para o algoritmo que implementa a segunda otimização.

A.1 Tabelas para a Primeira Otimização

Daremos agora as tabelas com os tempos de execução, representados pelas áreas totais, para todos os dez exemplos descritos no capítulo 6, para o tamanho da cadeia N igual a 50, 100, 500, 1000, 2500 e 5000. As tabelas são mostradas em ordem, das cadeias com menor tamanho para aquelas com maior tamanho. São mostradas primeiramente as tabelas com as áreas para calcular todos os palíndromos pares, seguida das tabelas com as áreas para calcular todos os palíndromos ímpares.

Arquivos/ Número de Erros	Área Usada - Palíndromos pares					
	1%	5%	10%	20%	40%	80%
dna-50	193	741	1582	4213	7916	8669
dnap1-50	196	469	711	1195	1667	2047
dnap2-50	195	736	1515	3434	4396	4780
dnap3-50	196	746	1567	4064	6329	6758
txt-50	196	760	1629	4779	12113	16384
txtp1-50	196	469	711	1195	1667	2047
txtp2-50	196	760	1634	4720	10810	12755
txtp3-50	196	758	1633	4814	12269	16672
igual	145	327	493	839	1245	1625
diferente	196	760	1644	4939	13629	22904
Área Total	196	776	1724	5599	18949	61049

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares					
	1%	5%	10%	20%	40%	80%
dna-50	191	733	1553	4079	7522	8207
dnap1-50	144	408	644	1110	1560	1920
dnap2-50	191	716	1458	3209	4039	4403
dnap3-50	192	728	1530	3905	5912	6305
txt-50	192	743	1589	4640	11647	15407
txtp1-50	144	408	644	1110	1560	1920
txtp2-50	192	744	1594	4558	10212	11875
txtp3-50	192	741	1597	4698	11832	15810
igual	144	322	484	818	1208	1568
diferente	192	744	1608	4818	13188	21688
Área Total	192	760	1688	5478	18508	59368

Arquivos/ Número de Erros	Área Usada - Palíndromos pares					
	1%	5%	10%	20%	40%	80%
dna-100	396	3377	10312	29264	49662	52498
dnap1-100	396	3356	9895	24996	32559	34119
dnap2-100	396	3380	10185	26865	38004	40197
dnap3-100	393	3352	10031	27601	42872	45301
txt-100	396	3433	10847	34168	87311	111903
txtp1-100	396	3371	10089	26709	40143	42209
txtp2-100	396	3441	10884	33917	82674	99066
txtp3-100	396	3429	10839	34422	88949	115625
igual	295	1043	1889	3279	4925	6485
diferente	396	3444	10989	35679	102459	173609
Área Total	396	3524	11649	40999	145099	478899

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares					
	1%	5%	10%	20%	40%	80%
dna-100	392	3345	10190	28854	48498	51275
dnap1-100	391	3298	9683	24215	31098	32618
dnap2-100	391	3334	10043	26265	36716	38801
dnap3-100	390	3318	9892	27117	41646	43974
txt-100	392	3399	10739	33696	85562	108463
txtp1-100	392	3339	9962	26253	39091	41063
txtp2-100	392	3399	10737	33373	80544	95725
txtp3-100	392	3389	10713	33923	86997	111951
igual	294	1034	1868	3238	4852	6372
diferente	392	3408	10868	35238	100778	168878
Área Total	392	3488	11528	40558	143418	472338

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	1%	5%	10%
dna-500	17764	307378	1045766
dnap1-500	17729	301690	993376
dnap2-500	17784	309709	1059998
dnap3-500	17713	307264	1042535
txt-500	17830	318792	1142512
txtp1-500	17844	318716	1140393
txtp2-500	17841	319034	1144932
txtp3-500	17835	317995	1138411
igual	5443	24223	45449
diferente	17844	321724	1172949
Área Total	17924	332124	1256249

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-500	3045526	4839498	5007176
dnap1-500	2561774	3236530	3313124
dnap2-500	3102118	4836344	4967039
dnap3-500	2939220	4189198	4270206
txt-500	3793307	9772480	12117031
txtp1-500	3785665	9546921	11523384
txtp2-500	3831671	9855658	12132942
txtp3-500	3788005	9521213	11451695
igual	80399	121965	161765
diferente	4090399	12160299	20740049
Área Total	4756999	17493499	58906499

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-500	17734	306533	1042865
dnap1-500	17693	301021	990801
dnap2-500	17753	309069	1057398
dnap3-500	17673	306552	1039508
txt-500	17791	318109	1139732
txtp1-500	17807	317941	1137358
txtp2-500	17804	318294	1142085
txtp3-500	17792	317154	1135565
igual	5434	24174	45348
diferente	17808	321048	1170348
Área Total	17888	331448	1253648

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-500	3033608	4808859	4973954
dnap1-500	2552144	3219140	3294479
dnap2-500	3092348	4810466	4939433
dnap3-500	2927973	4162767	4242804
txt-500	3781928	9730076	12042041
txtp1-500	3773505	9498672	11444163
txtp2-500	3820197	9810109	12053401
txtp3-500	3777348	9476677	11377303
igual	80198	121598	161198
diferente	4080198	12119898	20624398
Área Total	4746798	17453098	58745698

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	1%	5%	10%
dna-1000	119146	2344780	8039128
dnap1-1000	118952	2312945	7741971
dnap2-1000	118850	2330057	7932471
dnap3-1000	118867	2338389	8083669
txt-1000	119769	2442546	8910218
txtp1-1000	119776	2438207	8871284
txtp2-1000	119708	2444175	8931800
txtp3-1000	119807	2444140	8895111
igual	20789	95949	180899
diferente	119889	2473449	9190899
Área Total	120549	2556749	9857499

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-1000	22877434	33967647	34637040
dnap1-1000	19860822	24284698	24601101
dnap2-1000	21792173	29166207	29584827
dnap3-1000	22851192	32513509	33018874
txt-1000	29783875	73950835	88495862
txtp1-1000	29581301	73366442	87819778
txtp2-1000	30085791	77288098	95085562
txtp3-1000	29752031	74341369	88081790
igual	320799	487265	646865
diferente	32360799	96640599	164960099
Área Total	37693999	139306999	470292999

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-1000	119020	2341902	8030325
dnap1-1000	118849	2310275	7731449
dnap2-1000	118714	2327353	7922486
dnap3-1000	118763	2336432	8075671
txt-1000	119646	2439697	8898753
txtp1-1000	119645	2435235	8859597
txtp2-1000	119587	2441363	8920772
txtp3-1000	119674	2441217	8884430
igual	20768	95848	180698
diferente	119768	2470848	9180698
Área Total	120428	2554148	9847298

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-1000	22844026	33889210	34554210
dnap1-1000	19818142	24211589	24525709
dnap2-1000	21749737	29074768	29490331
dnap3-1000	22817775	32439465	32941656
txt-1000	29738215	73773159	88212487
txtp1-1000	29535244	73179782	87521306
txtp2-1000	30041648	77113470	94784638
txtp3-1000	29708407	74167495	87802984
igual	320398	486532	645732
diferente	32320398	96479798	164498798
Área Total	37653598	139146198	469651398

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	1%	5%	10%
dna-2500	1658702	35647550	124773813
dnap1-2500	1657023	35303892	120883789
dnap2-2500	1659663	35668624	124500073
dnap3-2500	1659710	35125357	120519567
txt-2500	1670464	37088124	136445080
txtp1-2500	1670381	37117154	136494442
txtp2-2500	1670387	37199686	137403240
txtp3-2500	1670628	37277387	138049739
igual	126223	596123	1127249
diferente	1673724	37721124	141814749
Área Total	1684124	39023124	152231249

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-2500	359571381	525847150	531700320
dnap1-2500	324198994	417168608	420832630
dnap2-2500	357396917	518093828	525041726
dnap3-2500	324838653	424196469	429080339
txt-2500	455866533	1106765334	1291457249
txtp1-2500	457454320	1136670296	1340200892
txtp2-2500	465110067	1196677215	1465356667
txtp3-2500	467942603	1223145310	1533505426
igual	2001999	3043165	4042165
diferente	502251999	1504001499	2568500249
Área Total	585584999	2170667499	7339332499

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-2500	1657839	35633417	124715034
dnap1-2500	1656495	35289878	120826345
dnap2-2500	1659023	35653059	124441155
dnap3-2500	1659192	35112207	120468229
txt-2500	1669707	37069606	136376004
txtp1-2500	1669619	37098847	136422806
txtp2-2500	1669638	37183394	137336250
txtp3-2500	1669829	37260676	137984498
igual	126174	595874	1126748
diferente	1673048	37705248	141751748
Área Total	1683448	39007248	152168248

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-2500	359337250	525251015	531083287
dnap1-2500	323970596	416728078	420379303
dnap2-2500	357151719	517521085	524444358
dnap3-2500	324631872	423798965	428670220
txt-2500	455601067	1105689461	1289820986
txtp1-2500	457165895	1135519102	1338370287
txtp2-2500	464842745	1195613685	1463516026
txtp3-2500	467680008	1222112139	1531647929
igual	2000998	3041332	4039332
diferente	502000998	1502999498	2565621998
Área Total	585333998	2169665498	7335328498

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	1%	5%	10%
dna-5000	12763572	282313258	992185739
dnap1-5000	12747378	280859971	976668190
dnap2-5000	12756350	282885259	995268545
dnap3-5000	12748259	283631706	1004036525
txt-5000	12847248	294837159	1090726142
txtp1-5000	12849698	295330534	1095600411
txtp2-5000	12824482	288624051	1041946503
txtp3-5000	12848038	294553421	1090649894
igual	499949	2379749	4504499
diferente	12877449	299317249	1129754499
Área Total	12960749	309733749	1213087499

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-5000	2881062190	4279870163	4333570495
dnap1-5000	2757391716	3832201538	3865898649
dnap2-5000	2906513458	4372006250	4432209523
dnap3-5000	2965046326	4624214246	4702334231
txt-5000	3680880448	9293928917	11220121713
txtp1-5000	3724723129	9688202240	12087056266
txtp2-5000	3265874206	6217824520	7094490436
txtp3-5000	3700506739	9472761490	11561521789
igual	8003999	12169665	16167665
diferente	4009003999	12016002999	20524000499
Área Total	4675669999	17349334999	58690664999

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-5000	12761000	282252776	991956304
dnap1-5000	12745151	280802076	976430825
dnap2-5000	12753913	282832111	995040074
dnap3-5000	12746222	283586680	1003836759
txt-5000	12844651	294771567	1090467759
txtp1-5000	12846803	295265173	1095339548
txtp2-5000	12822047	288569199	1041758732
txtp3-5000	12845368	294491426	1090393221
igual	499848	2379248	4503498
diferente	12874848	299254248	1129503498
Área Total	12958148	309670748	1212836498

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-5000	2880153479	4277653265	4331258961
dnap1-5000	2756427521	3830042573	3863683537
dnap2-5000	2905641085	4369781167	4429875720
dnap3-5000	2964268523	4622137088	4700146076
txt-5000	3679838782	9289866510	11213417084
txtp1-5000	3723685395	9684056328	12079621834
txtp2-5000	3265055165	6215236735	7091040052
txtp3-5000	3699487432	9468676218	11554507401
igual	8001998	12165998	16161998
diferente	4008001998	12011998998	20512493998
Área Total	4674667998	17345330998	58674656998

A.2 Tabelas para a Segunda Otimização

Daremos agora as tabelas com os tempos de execução, representados pelas áreas totais, para todos os dez exemplos descritos no capítulo 6, para o tamanho da cadeia N igual a 50, 100, 500, 1000, 2500 e 5000. As tabelas são mostradas em ordem, das cadeias com menor tamanho para aquelas com maior tamanho. São mostradas primeiramente as tabelas com as áreas para calcular todos os palíndromos pares, seguida das tabelas com as áreas para calcular todos os palíndromos ímpares.

Arquivos/ Número de Erros	Área Usada - Palíndromos pares					
	1%	5%	10%	20%	40%	80%
dna-50	190	733	1539	3982	7110	7863
dnap1-50	196	291	381	589	929	1309
dnap2-50	194	720	1416	2816	3474	3858
dnap3-50	196	739	1527	3743	5451	5876
txt-50	196	760	1611	4701	11791	15945
txtp1-50	196	291	381	589	929	1309
txtp2-50	196	760	1622	4606	10106	12039
txtp3-50	196	756	1627	4773	11810	16034
igual	97	191	281	489	829	1209
diferente	196	760	1644	4939	13629	22904
Área Total	196	776	1724	5599	18949	61049

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares					
	1%	5%	10%	20%	40%	80%
dna-50	190	730	1517	3858	6754	7439
dnap1-50	96	188	276	478	808	1168
dnap2-50	190	695	1341	2578	3158	3522
dnap3-50	192	716	1487	3592	5100	5491
txt-50	192	741	1574	4570	11347	15006
txtp1-50	96	188	276	478	808	1168
txtp2-50	192	744	1576	4435	9574	11233
txtp3-50	192	740	1591	4663	11385	15214
igual	96	188	276	478	808	1168
diferente	192	744	1608	4818	13188	21688
Área Total	192	760	1688	5478	18508	59368

Arquivos/ Número de Erros	Área Usada - Palíndromos pares					
	1%	5%	10%	20%	40%	80%
dna-100	396	3339	9994	27403	43618	46446
dnap1-100	396	3269	9342	21650	26736	28296
dnap2-100	396	3333	9840	23748	31993	34186
dnap3-100	390	3281	9714	25147	37390	39819
txt-100	396	3430	10779	33629	84380	107805
txtp1-100	396	3327	9657	23522	34135	36201
txtp2-100	396	3438	10846	33479	79428	95118
txtp3-100	396	3423	10795	34084	86149	111185
igual	197	581	1039	1879	3259	4819
diferente	396	3444	10989	35679	102459	173609
Área Total	396	3524	11649	40999	145099	478899

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares					
	1%	5%	10%	20%	40%	80%
dna-100	392	3311	9874	26960	42579	45347
dnap1-100	390	3205	9107	20776	25358	26878
dnap2-100	390	3287	9696	23122	30881	32966
dnap3-100	388	3236	9586	24643	36318	38646
txt-100	392	3395	10671	33176	82666	104480
txtp1-100	392	3290	9537	23067	33213	35185
txtp2-100	392	3394	10688	32914	77346	91911
txtp3-100	392	3379	10672	33586	84211	107629
igual	196	576	1028	1858	3218	4738
diferente	392	3408	10868	35238	100778	168878
Área Total	392	3488	11528	40558	143418	472338

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	1%	5%	10%
dna-500	17730	304523	1026901
dnap1-500	17665	296263	957657
dnap2-500	17749	306766	1037873
dnap3-500	17657	304026	1022294
txt-500	17824	318085	1136557
txtp1-500	17844	317974	1135613
txtp2-500	17838	318057	1140190
txtp3-500	17831	317107	1132796
igual	2981	12661	24199
diferente	17844	321724	1172949
Área Total	17924	332124	1256249

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-500	2909814	4492498	4660158
dnap1-500	2318405	2877763	2954357
dnap2-500	2967253	4460474	4590494
dnap3-500	2798630	3838573	3919541
txt-500	3754003	9526931	11779103
txtp1-500	3754931	9331417	11231325
txtp2-500	3796357	9624291	11821088
txtp3-500	3751947	9259684	11121170
igual	45399	80299	120099
diferente	4090399	12160299	20740049
Área Total	4756999	17493499	58906499

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-500	17697	303612	1023931
dnap1-500	17635	295599	955015
dnap2-500	17706	306124	1035308
dnap3-500	17606	303211	1019180
txt-500	17782	317397	1133655
txtp1-500	17802	317135	1132389
txtp2-500	17802	317294	1137419
txtp3-500	17781	316239	1129906
igual	2976	12636	24148
diferente	17808	321048	1170348
Área Total	17888	331448	1253648

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-500	2897774	4463669	4628746
dnap1-500	2308304	2861705	2937044
dnap2-500	2957024	4436650	4564948
dnap3-500	2787253	3814170	3894166
txt-500	3742856	9483733	11705552
txtp1-500	3742193	9281117	11151686
txtp2-500	3784872	9578618	11742961
txtp3-500	3741133	9215368	11048474
igual	45298	80098	119698
diferente	4080198	12119898	20624398
Área Total	4746798	17453098	58745698

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	1%	5%	10%
dna-1000	118873	2321786	7896187
dnap1-1000	118691	2282796	7526140
dnap2-1000	118424	2306116	7762926
dnap3-1000	118490	2315036	7935882
txt-1000	119739	2438034	8871786
txtp1-1000	119732	2432285	8831336
txtp2-1000	119637	2438931	8898469
txtp3-1000	119767	2438107	8859103
igual	10939	49699	95899
diferente	119889	2473449	9190899
Área Total	120549	2556749	9857499

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-1000	21913947	31648515	32317801
dnap1-1000	18308591	21982889	22299292
dnap2-1000	20613897	26917514	27336134
dnap3-1000	21832684	30131332	30636555
txt-1000	29507784	72014553	86223913
txtp1-1000	29315616	71611361	85948466
txtp2-1000	29834494	75442762	92595459
txtp3-1000	29467133	72321661	85688860
igual	180799	320599	480199
diferente	32360799	96640599	164960099
Área Total	37693999	139306999	470292999

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-1000	118784	2318938	7887551
dnap1-1000	118598	2280034	7515178
dnap2-1000	118280	2303448	7753419
dnap3-1000	118383	2313121	7927761
txt-1000	119609	2435194	8860325
txtp1-1000	119600	2429328	8819612
txtp2-1000	119512	2436084	8887381
txtp3-1000	119634	2435233	8848436
igual	10928	49648	95798
diferente	119768	2470848	9180698
Área Total	120428	2554148	9847298

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-1000	21881475	31576903	32241797
dnap1-1000	18264657	21914777	22228897
dnap2-1000	20571141	26833592	27249155
dnap3-1000	21800071	30063975	30566024
txt-1000	29462222	71837254	85947932
txtp1-1000	29269737	71429841	85656854
txtp2-1000	29789756	75267161	92300252
txtp3-1000	29422978	72144660	85415475
igual	180598	320198	479398
diferente	32320398	96479798	164498798
Área Total	37653598	139146198	469651398

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	1%	5%	10%
dna-2500	1655254	35425050	123116379
dnap1-2500	1653750	34956756	118014167
dnap2-2500	1656964	35436413	122764051
dnap3-2500	1656077	34794590	118170181
txt-2500	1669572	37009196	135870697
txtp1-2500	1669359	37040425	135958609
txtp2-2500	1669308	37133909	136975255
txtp3-2500	1669620	37209986	137611684
igual	64661	307061	595999
diferente	1673724	37721124	141814749
Área Total	1684124	39023124	152231249

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-2500	347750727	495231049	501082402
dnap1-2500	300979723	378843006	382507028
dnap2-2500	344449591	487335126	494282567
dnap3-2500	307836699	393184144	398064582
txt-2500	452002085	1078474531	1259891537
txtp1-2500	453524440	1107041723	1304974921
txtp2-2500	461859713	1172625883	1433226149
txtp3-2500	464924897	1199253018	1499929660
igual	1126999	2001499	3000499
diferente	502251999	1504001499	2568500249
Área Total	585584999	2170667499	7339332499

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-2500	1654375	35411665	123058282
dnap1-2500	1653291	34941782	117956475
dnap2-2500	1656348	35420920	122704139
dnap3-2500	1655531	34781942	118119454
txt-2500	1668821	36990619	135800785
txtp1-2500	1668608	37021790	135886544
txtp2-2500	1668571	37117314	136907944
txtp3-2500	1668829	37193580	137546142
igual	64636	306936	595748
diferente	1673048	37705248	141751748
Área Total	1683448	39007248	152168248

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-2500	347511926	494672990	500503453
dnap1-2500	300751907	378443035	382094260
dnap2-2500	344204293	486791589	493714408
dnap3-2500	307633795	392825049	397692950
txt-2500	451734001	1077410557	1258293560
txtp1-2500	453234592	1105890462	1303191906
txtp2-2500	461592676	1171564018	1431425104
txtp3-2500	464664046	1198211132	1498107753
igual	1126498	2000498	2998498
diferente	502000998	1502999498	2565621998
Área Total	585333998	2169665498	7335328498

Arquivos/ Número de Erros	Area Usada - Palíndromos pares		
	1%	5%	10%
dna-5000	12744894	280679689	980873546
dnap1-5000	12728682	279076081	964206259
dnap2-5000	12737701	281372494	983739305
dnap3-5000	12728657	282244970	992497064
txt-5000	12842513	294355515	1087488735
txtp1-5000	12843924	294913995	1092453422
txtp2-5000	12805837	285653546	1023411221
txtp3-5000	12842771	294129792	1087346796
igual	253699	1223499	2379499
diferente	12877449	299317249	1129754499
Área Total	12960749	309733749	1213087499

Arquivos/ Número de Erros	Área Usada - Palíndromos pares		
	20%	40%	80%
dna-5000	2797731365	4054699271	4108396434
dnap1-5000	2668095412	3626457881	3660151266
dnap2-5000	2824144809	4141929853	4202128601
dnap3-5000	2881353356	4377308871	4455383660
txt-5000	3656509429	9105728061	10984851958
txtp1-5000	3701877643	9519767480	11860846111
txtp2-5000	3109182671	5688019569	6564677157
txtp3-5000	3676411449	9290487866	11330493053
igual	4503999	8002999	12000999
diferente	4009003999	12016002999	20524000499
Area Total	4675669999	17349334999	58690664999

Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	1%	5%	10%
dna-5000	12742259	280620833	980646521
dnap1-5000	12726422	279019006	963970116
dnap2-5000	12735429	281321024	983515391
dnap3-5000	12726612	282201908	992300222
txt-5000	12839908	294290581	1087228387
txtp1-5000	12841035	294847036	1092190957
txtp2-5000	12803532	285601192	1023224789
txtp3-5000	12840062	294068142	1087091560
igual	253648	1223248	2378998
diferente	12874848	299254248	1129503498
Área Total	12958148	309670748	1212836498

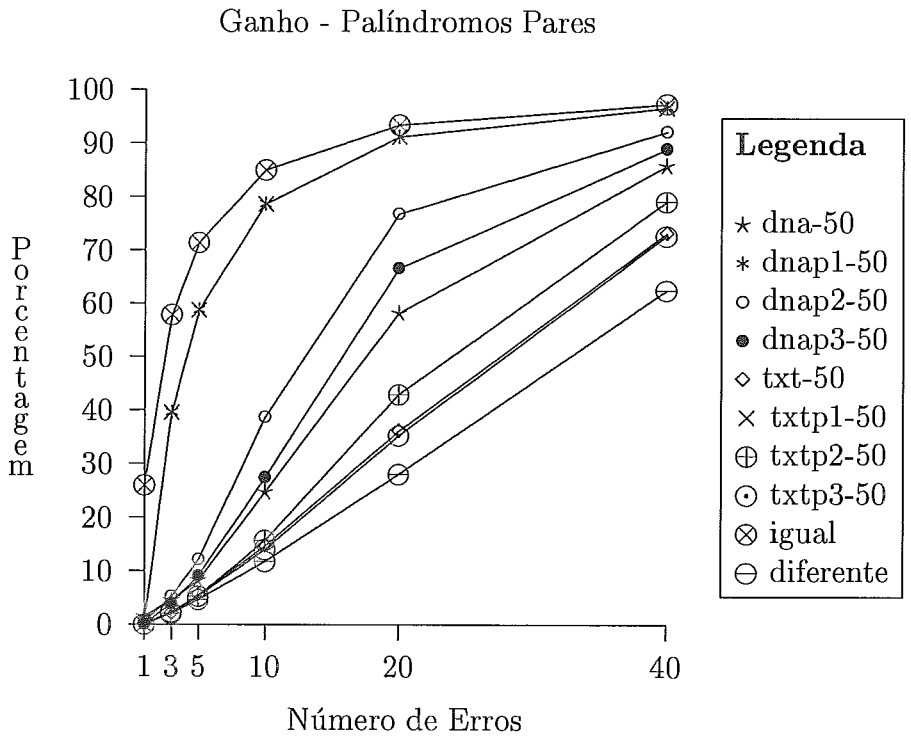
Arquivos/ Número de Erros	Área Usada - Palíndromos ímpares		
	20%	40%	80%
dna-5000	2796841214	4052632399	4106234946
dnap1-5000	2667139424	3624458202	3658095450
dnap2-5000	2823276214	4139828618	4199918653
dnap3-5000	2880603083	4375403827	4453367396
txt-5000	3655469541	9101676502	10978276719
txtp1-5000	3700835254	9515607471	11853534461
txtp2-5000	3108407913	5685751355	6561546353
txtp3-5000	3675392849	9286414773	11323618359
igual	4502998	8000998	11996998
diferente	4008001998	12011998998	20512493998
Área Total	4674667998	17345330998	58674656998

Apêndice B

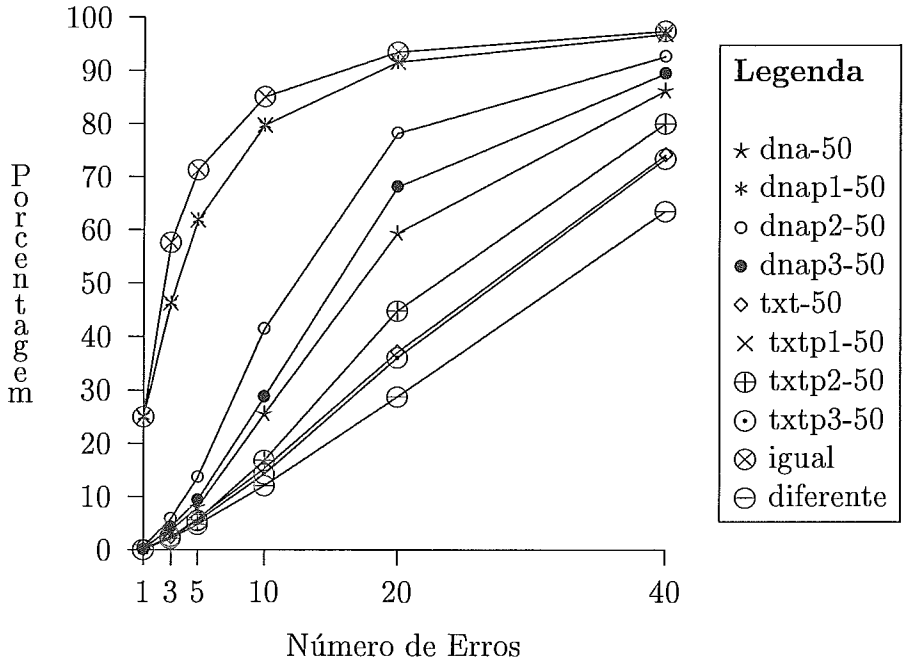
Gráficos dos Ganhos

Neste apêndice, damos todos os gráficos com os ganhos do algoritmo que implementa a primeira otimização sobre a área máxima, na primeira seção, os gráficos com os mesmos ganhos para o algoritmo que implementa a segunda otimização na segunda seção, e os gráficos dos ganhos do algoritmo que implementa a segunda otimização sobre as áreas do algoritmo que implementa a primeira otimização na última seção. Os exemplos são os mesmos descritos no capítulo 6, e os números de erros são baseados em porcentagens sobre os tamanhos das cadeias, que serão iguais a 50, 100, 500, 1000, 2500 e 5000.

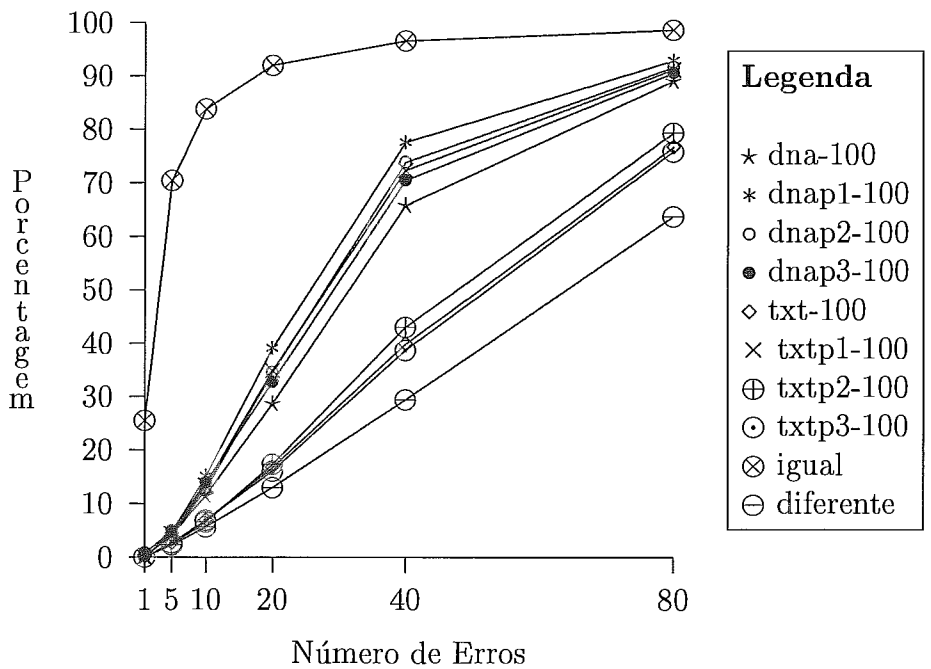
B.1 Ganhos ao Usarmos a Primeira Otimização



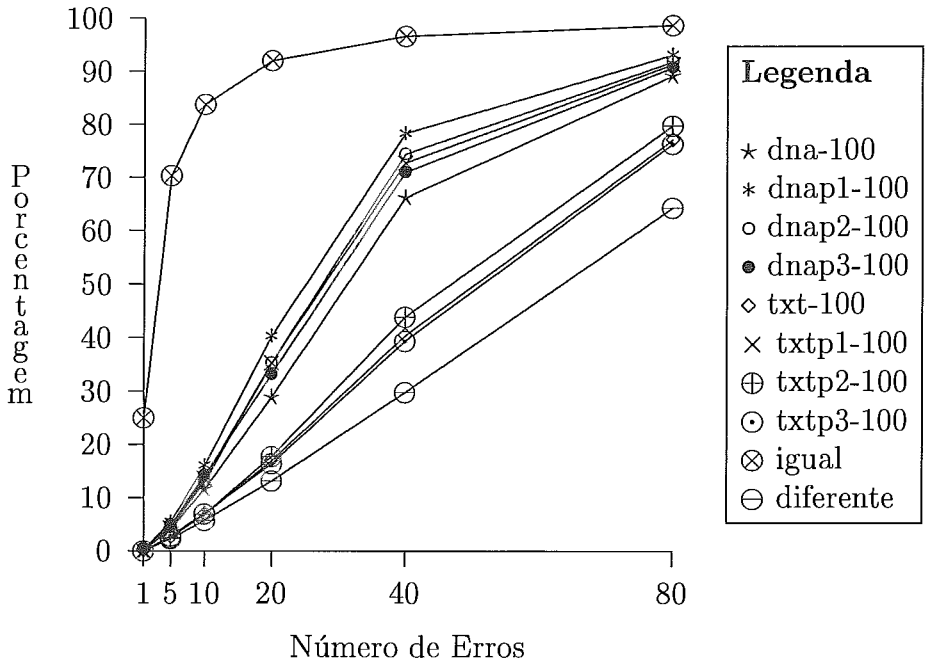
Ganho - Palíndromos Ímpares



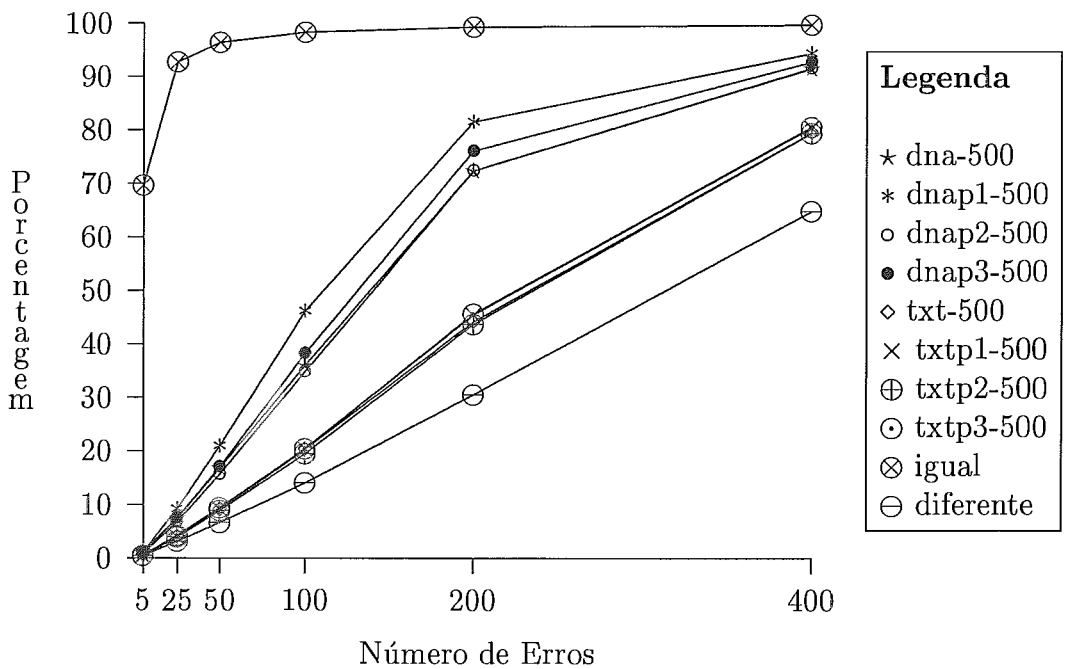
Ganho - Palíndromos Pares



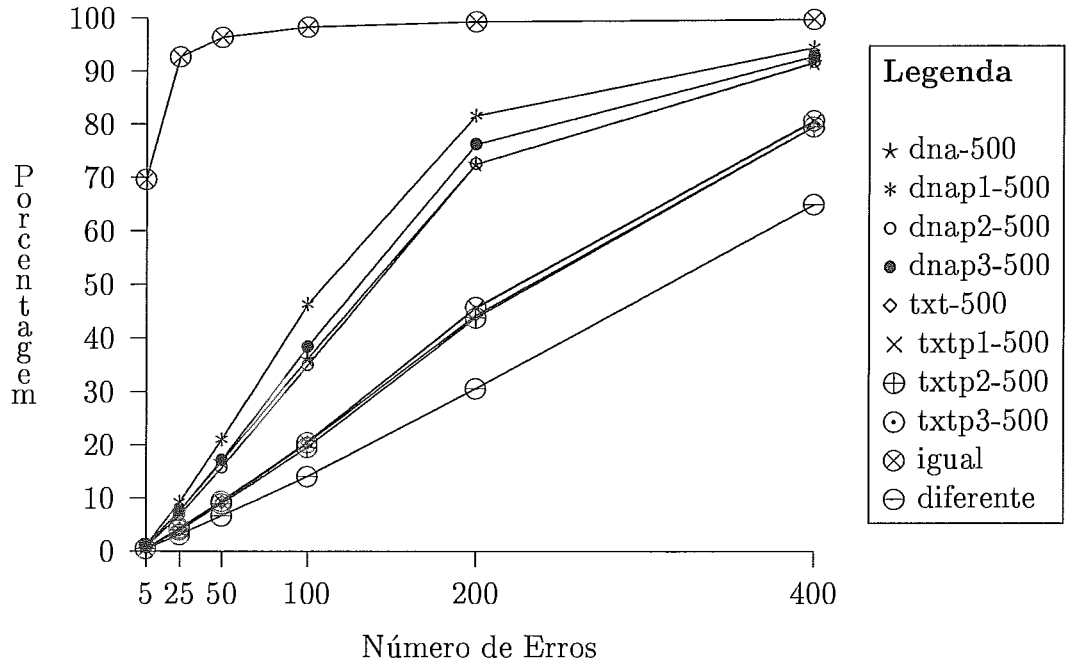
Ganho - Palíndromos Ímpares



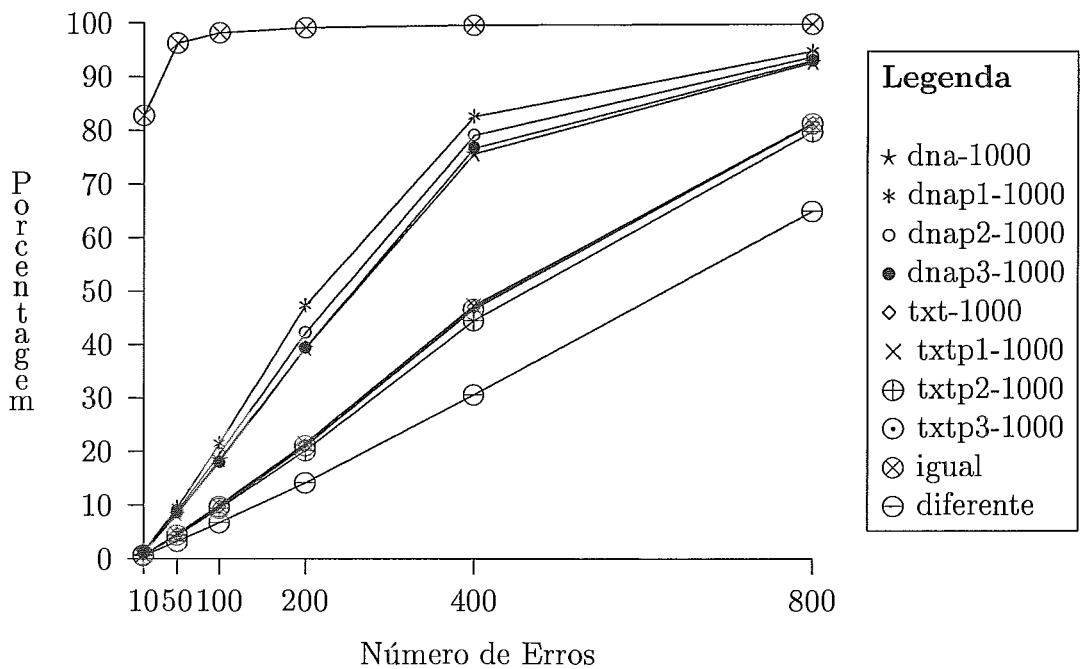
Ganho - Palíndromos Pares



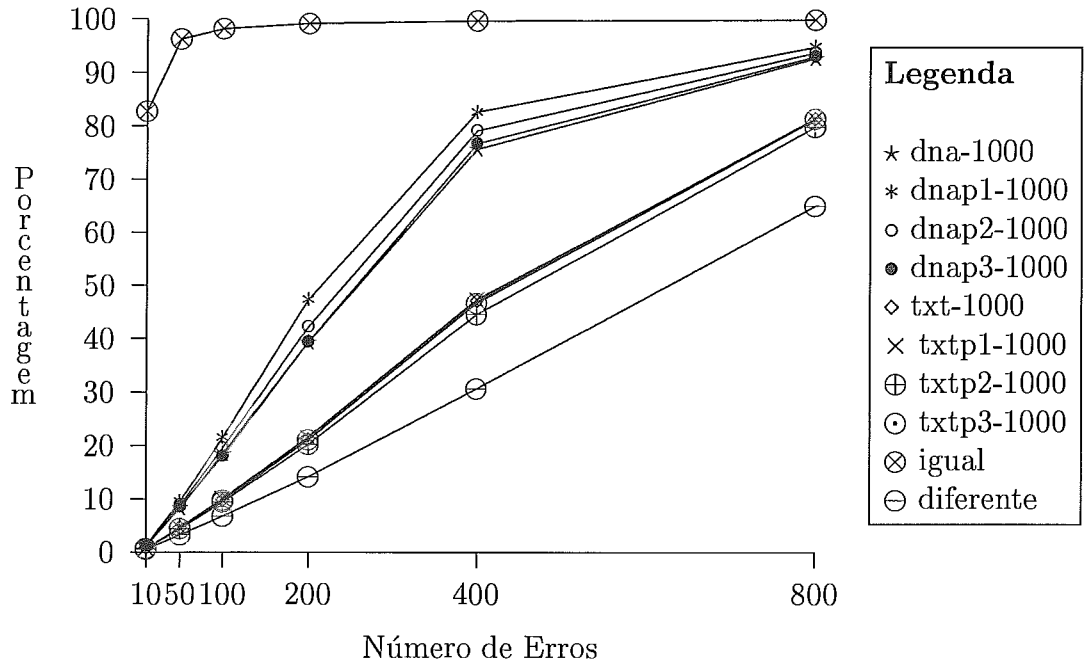
Ganho - Palíndromos Ímpares



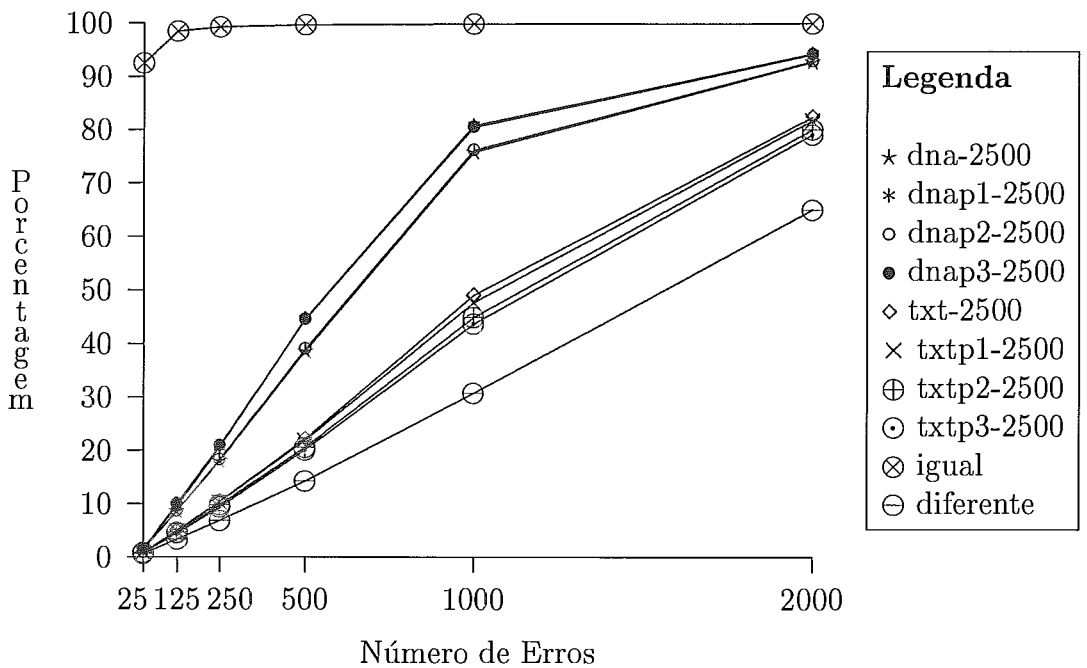
Ganho - Palíndromos Pares



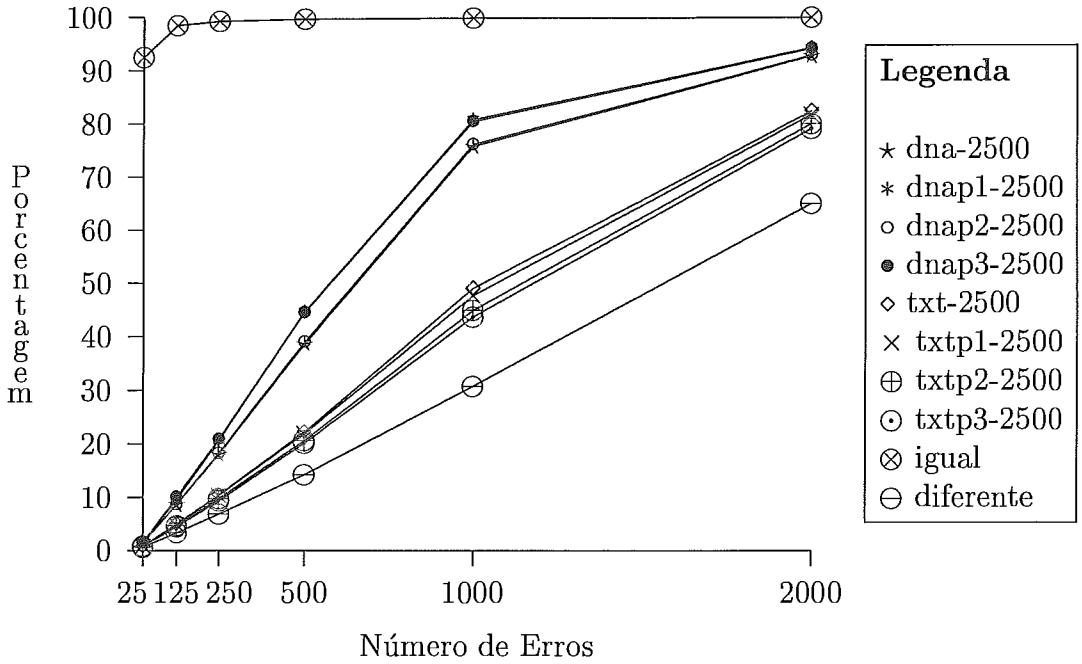
Ganho - Palíndromos Ímpares



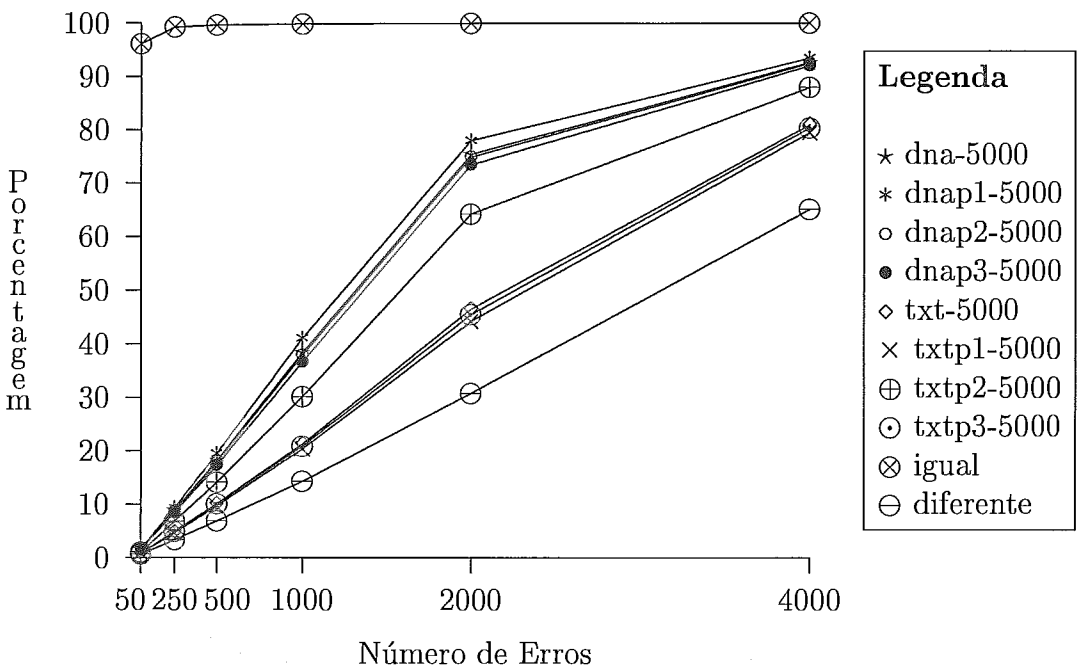
Ganho - Palíndromos Pares

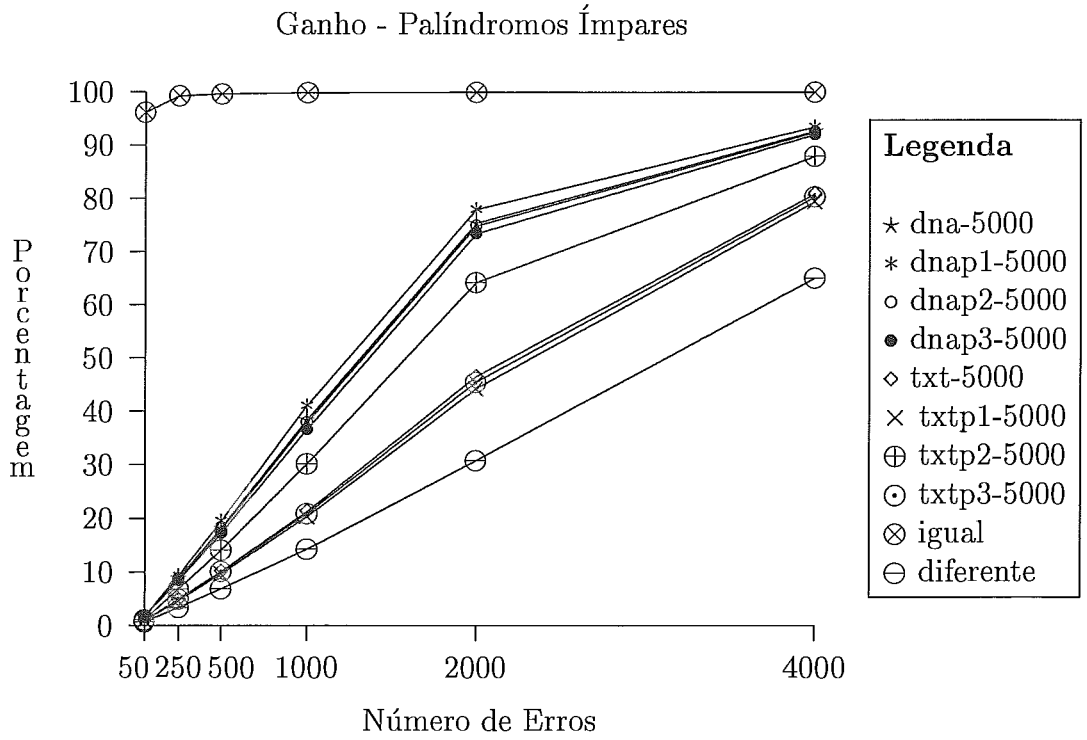


Ganho - Palíndromos Ímpares

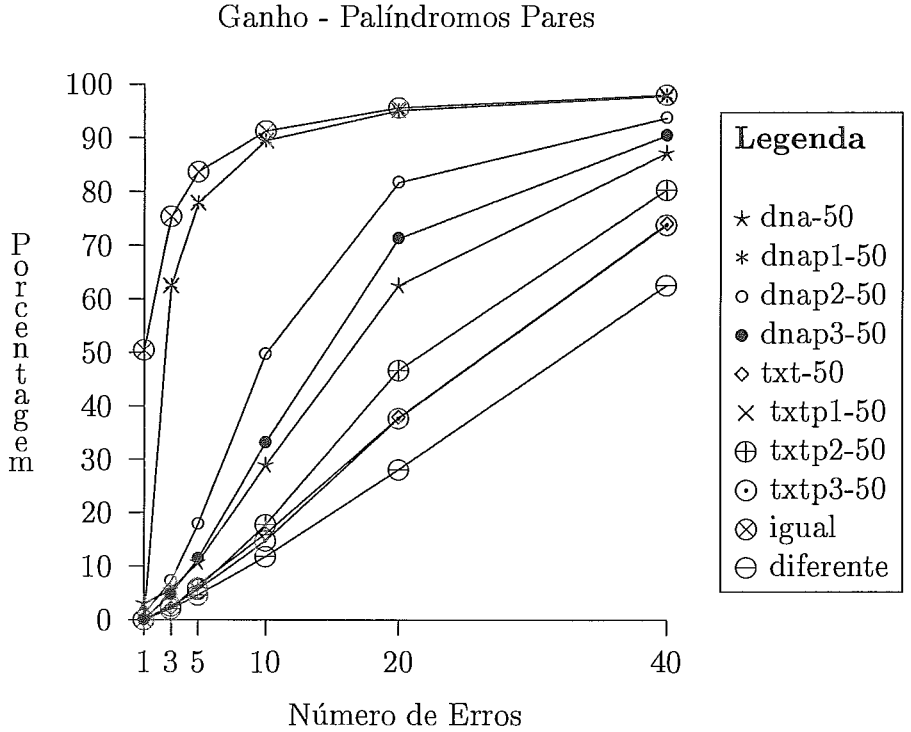


Ganho - Palíndromos Pares

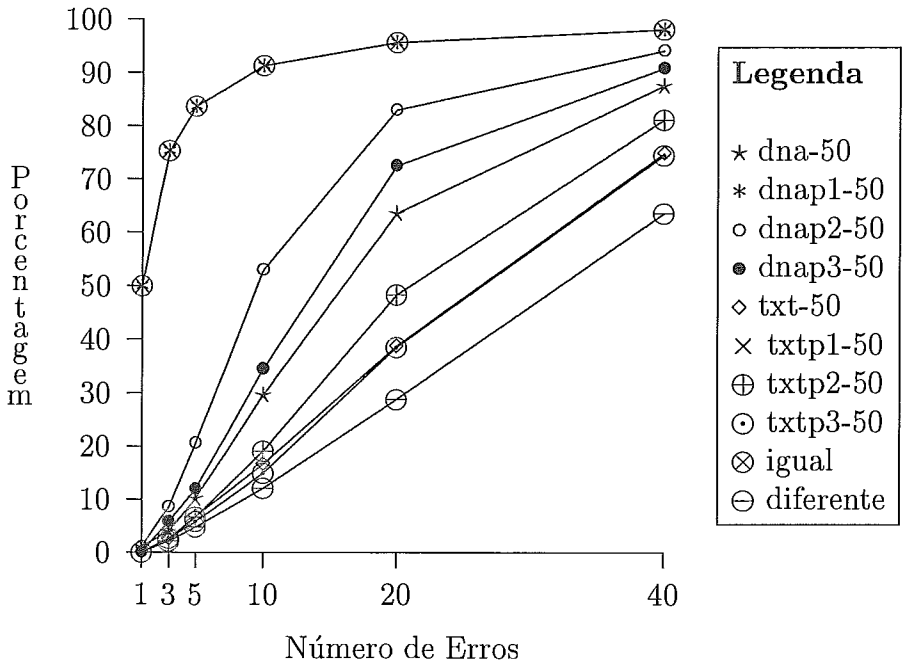




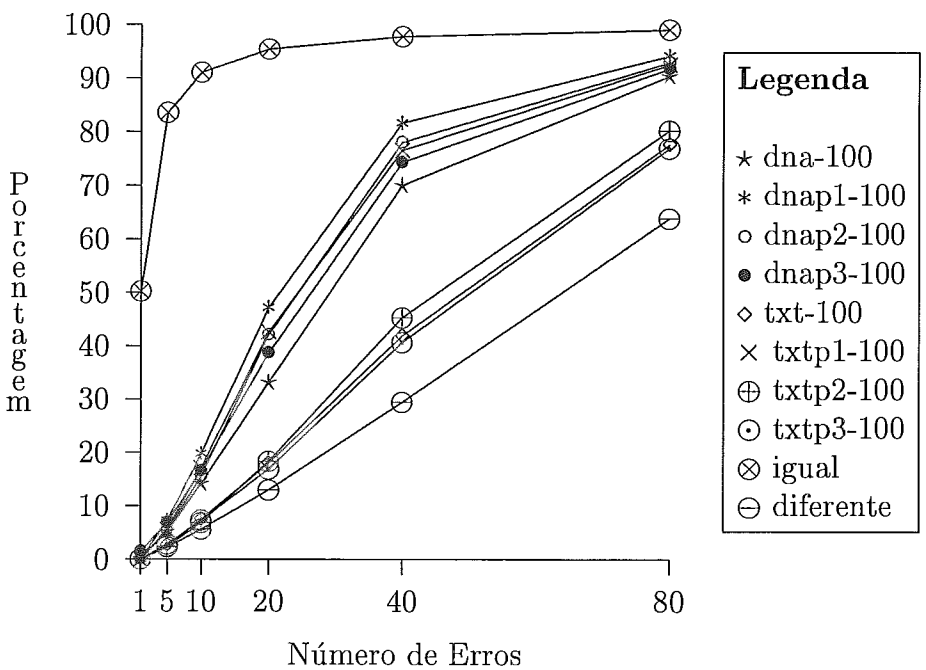
B.2 Ganhos ao Usarmos a Segunda Otimização



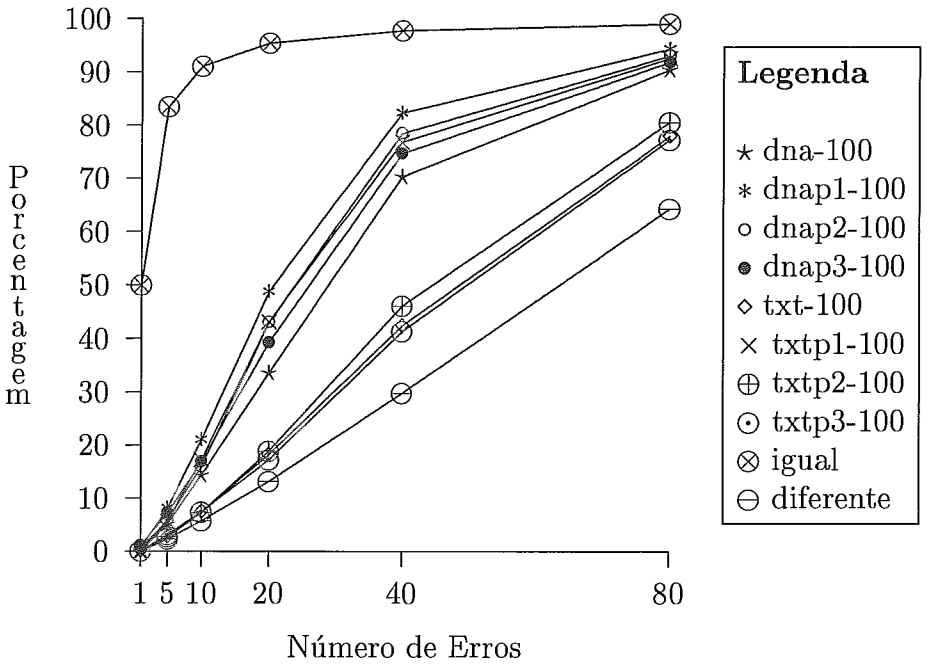
Ganho - Palíndromos Ímpares



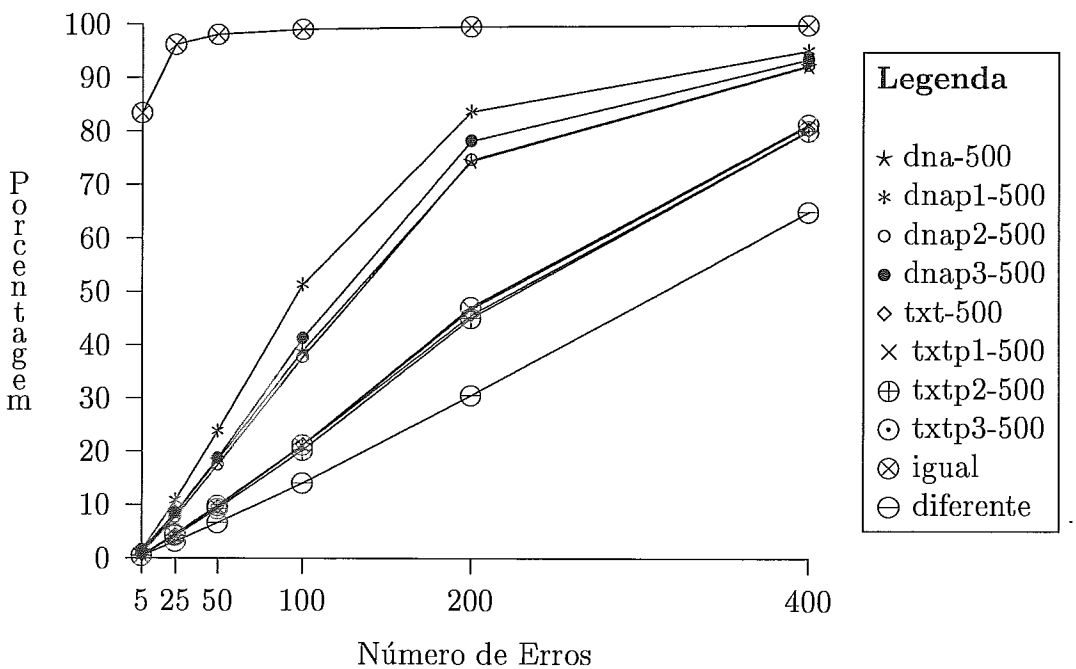
Ganho - Palíndromos Pares



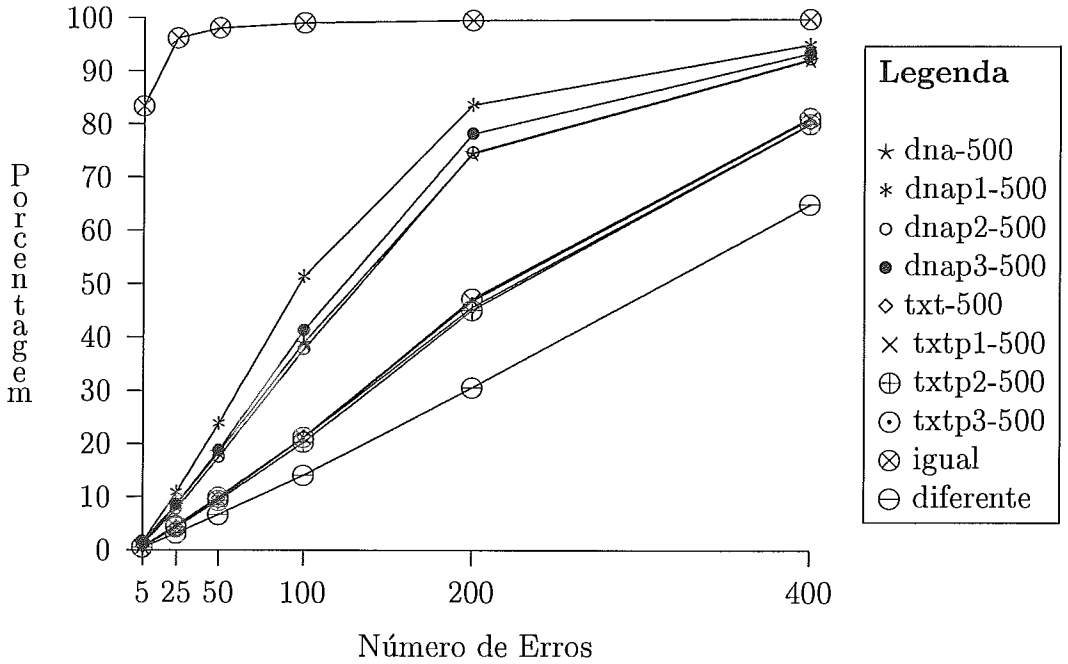
Ganho - Palíndromos Ímpares



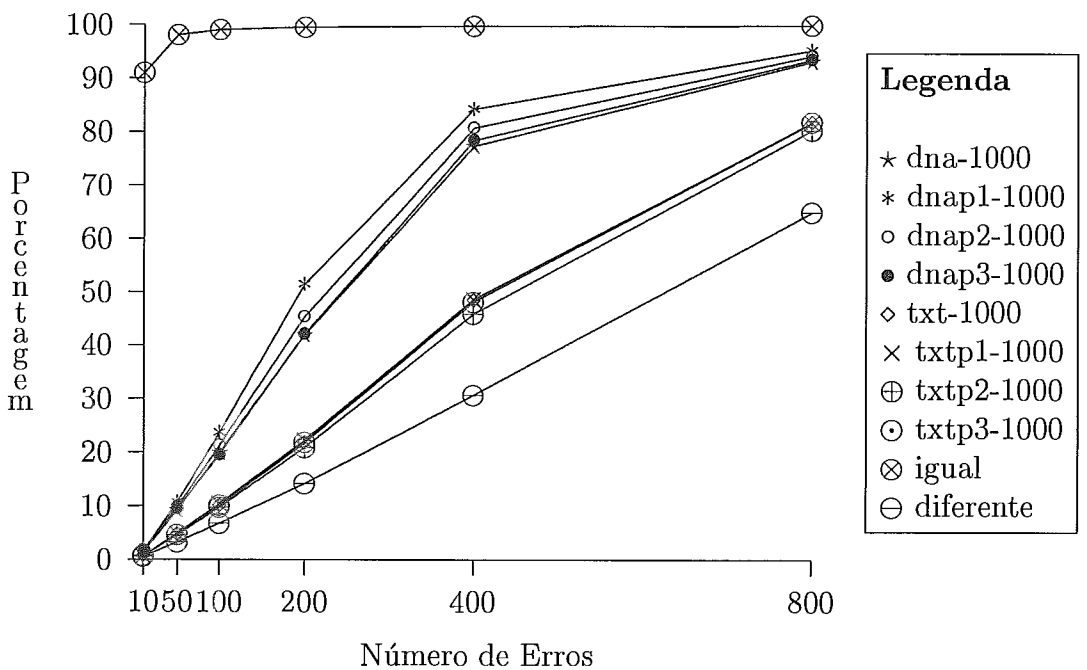
Ganho - Palíndromos Pares



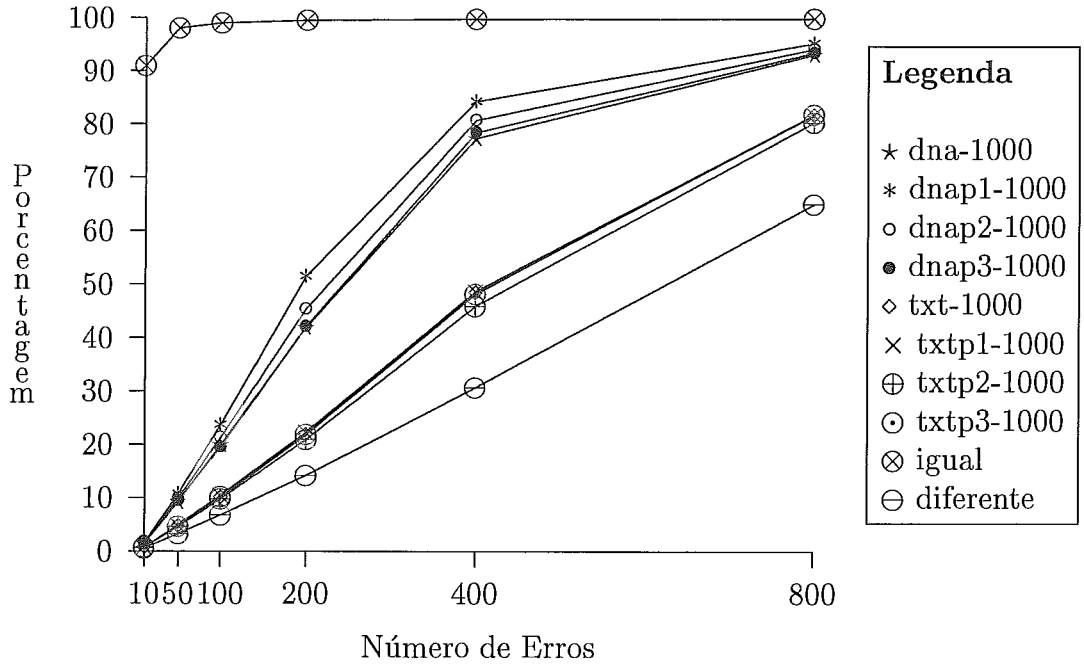
Ganho - Palíndromos Ímpares



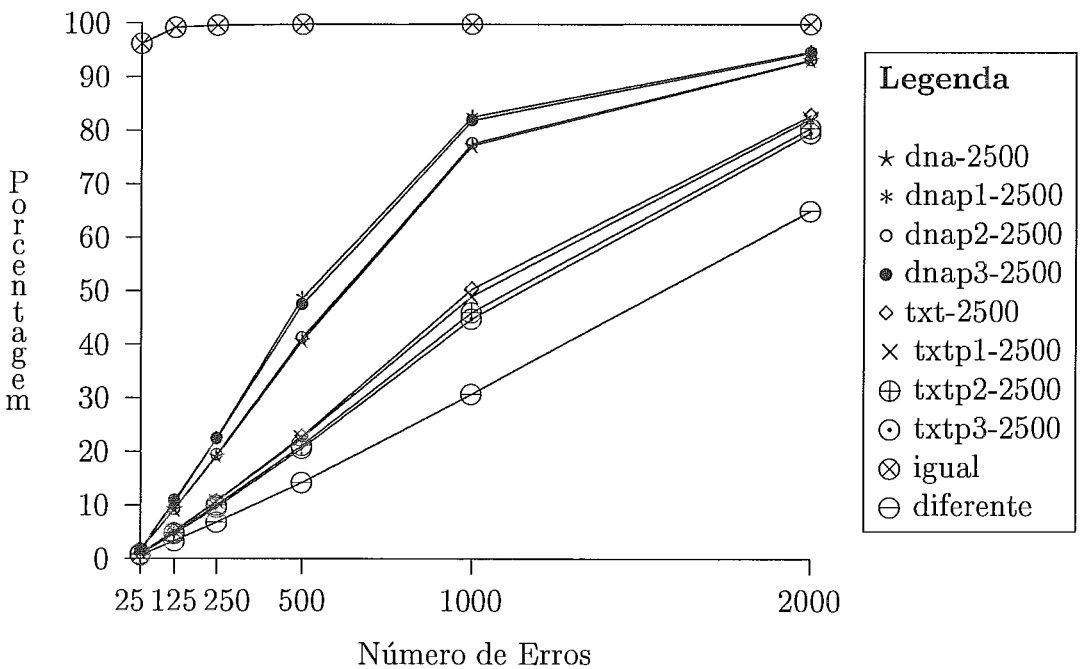
Ganho - Palíndromos Pares



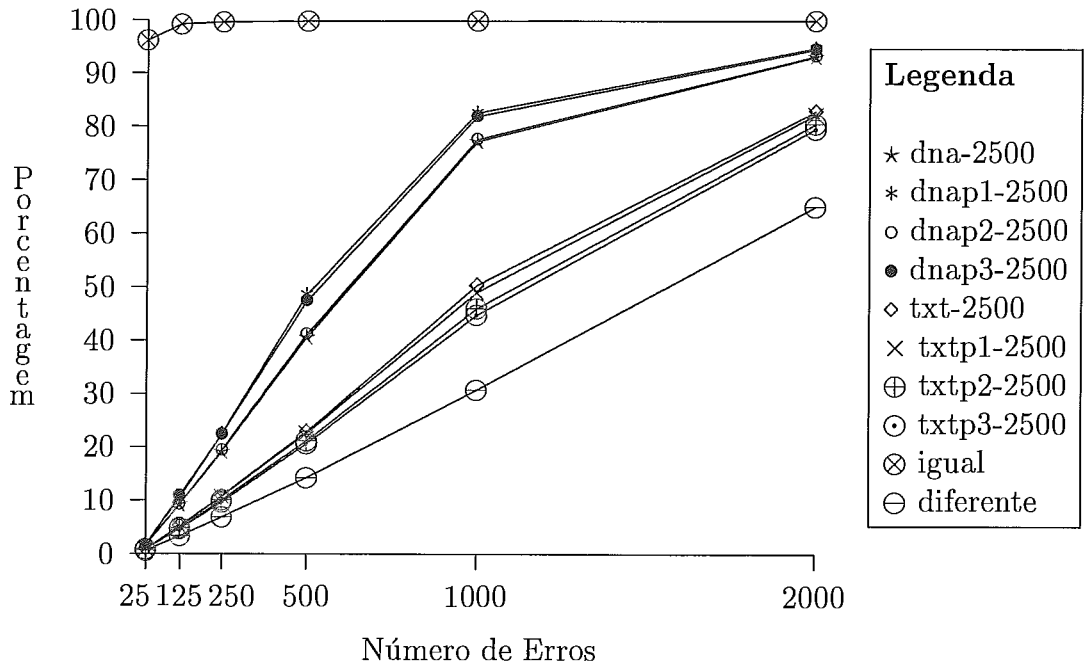
Ganho - Palíndromos Ímpares



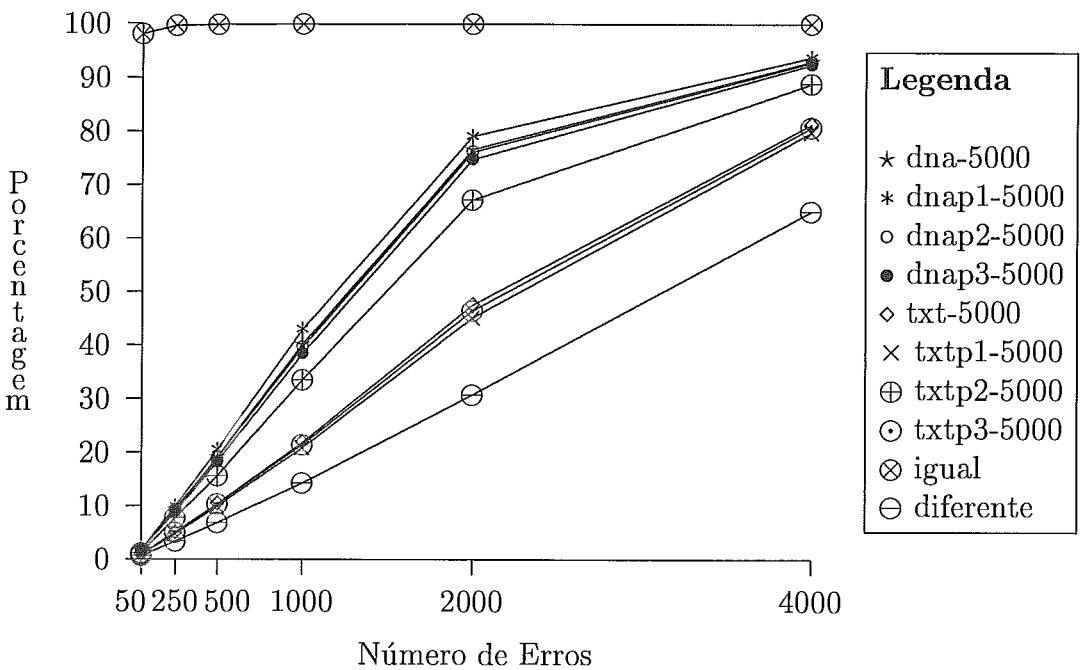
Ganho - Palíndromos Pares

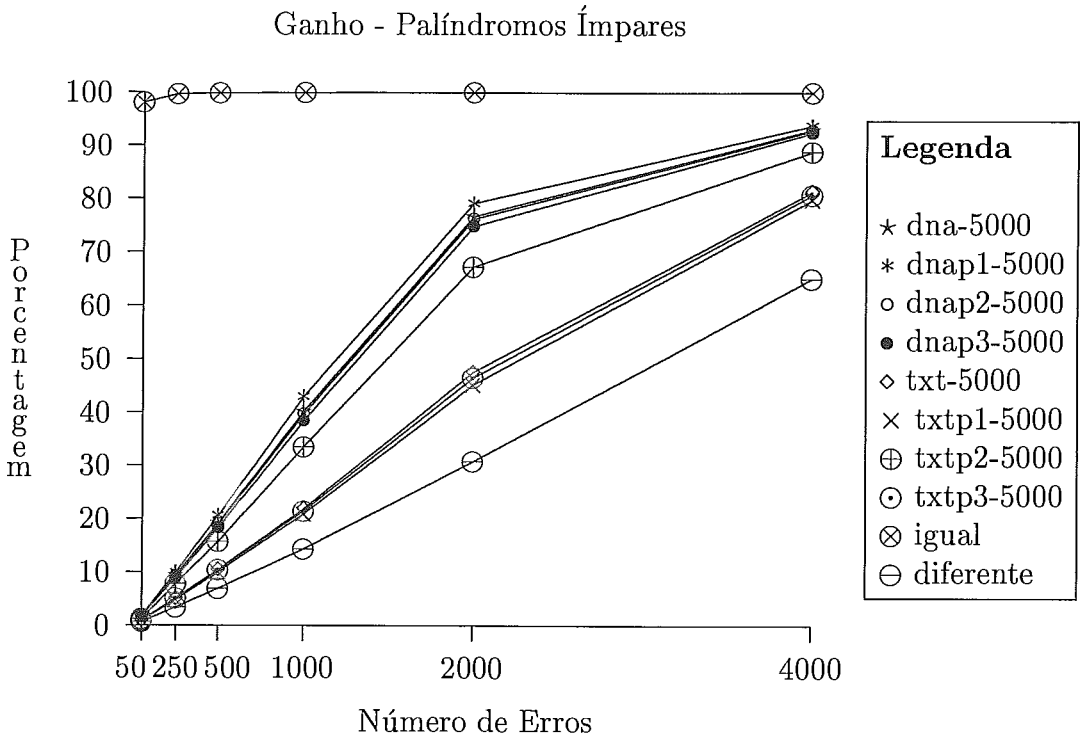


Ganho - Palíndromos Ímpares

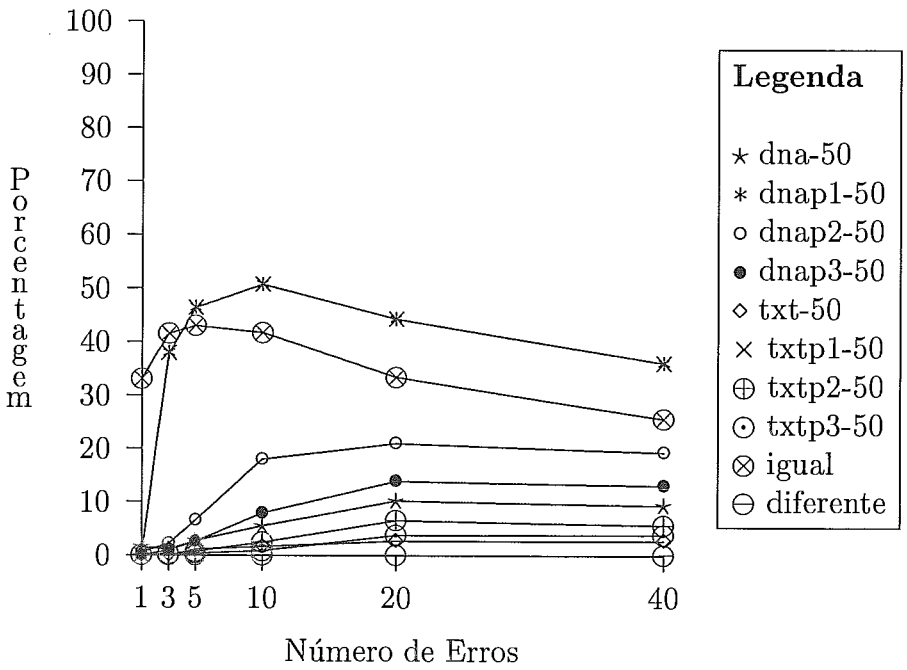


Ganho - Palíndromos Pares

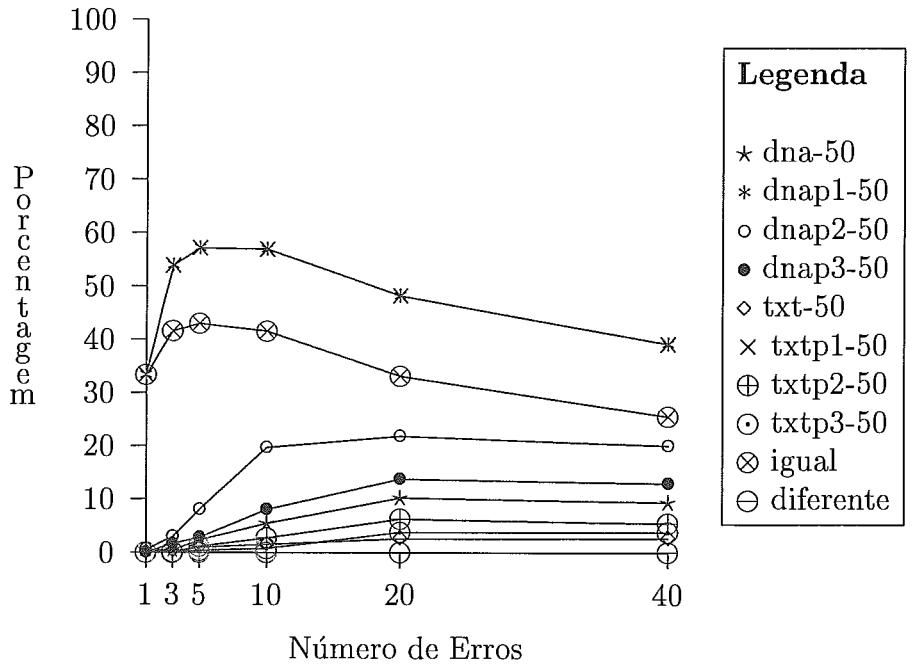




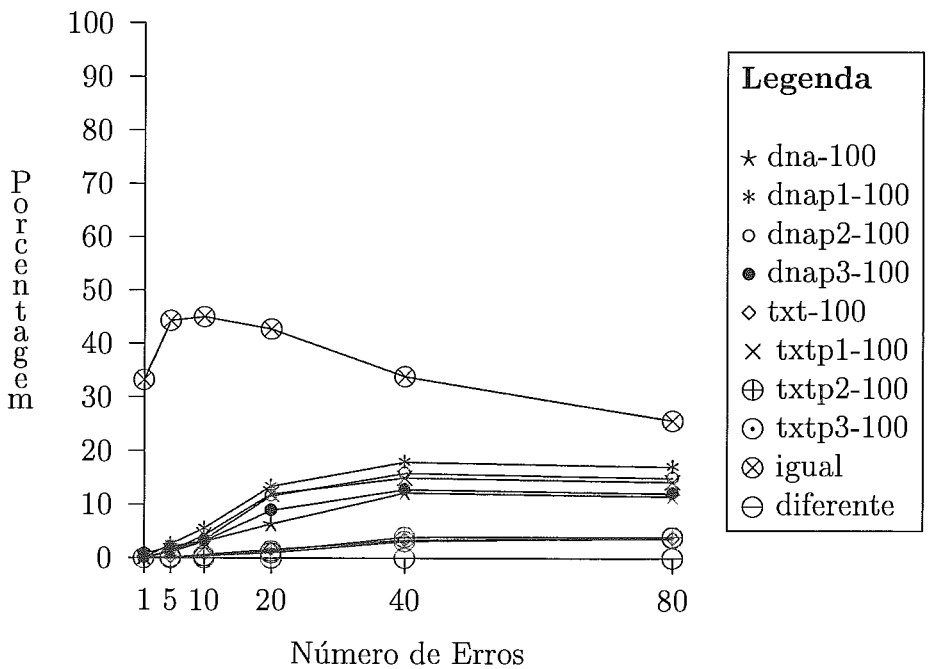
B.3 Ganhos da Segunda Otimização sobre a Primeira



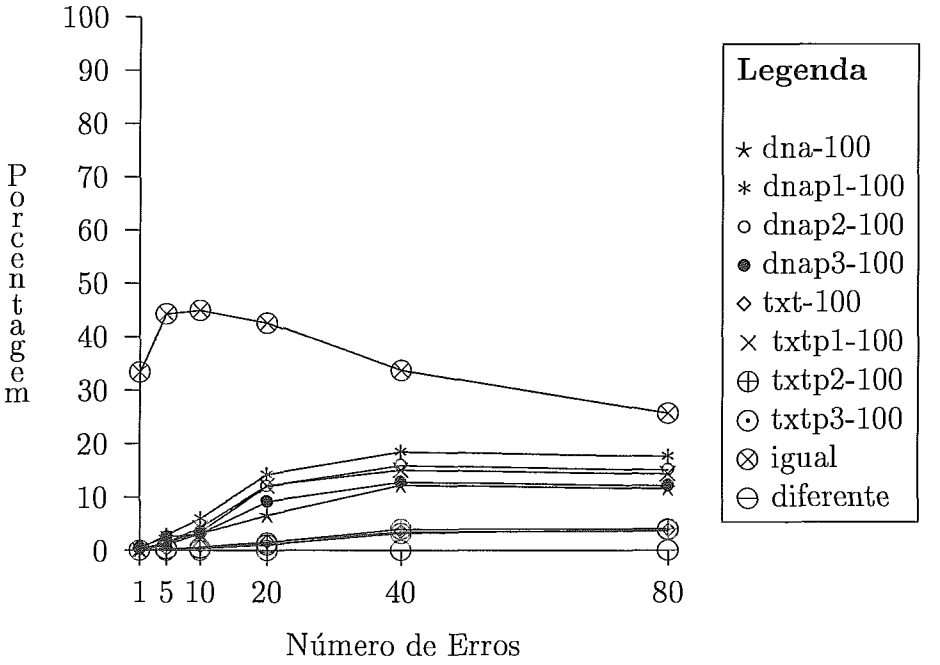
Ganho - Palíndromos Ímpares



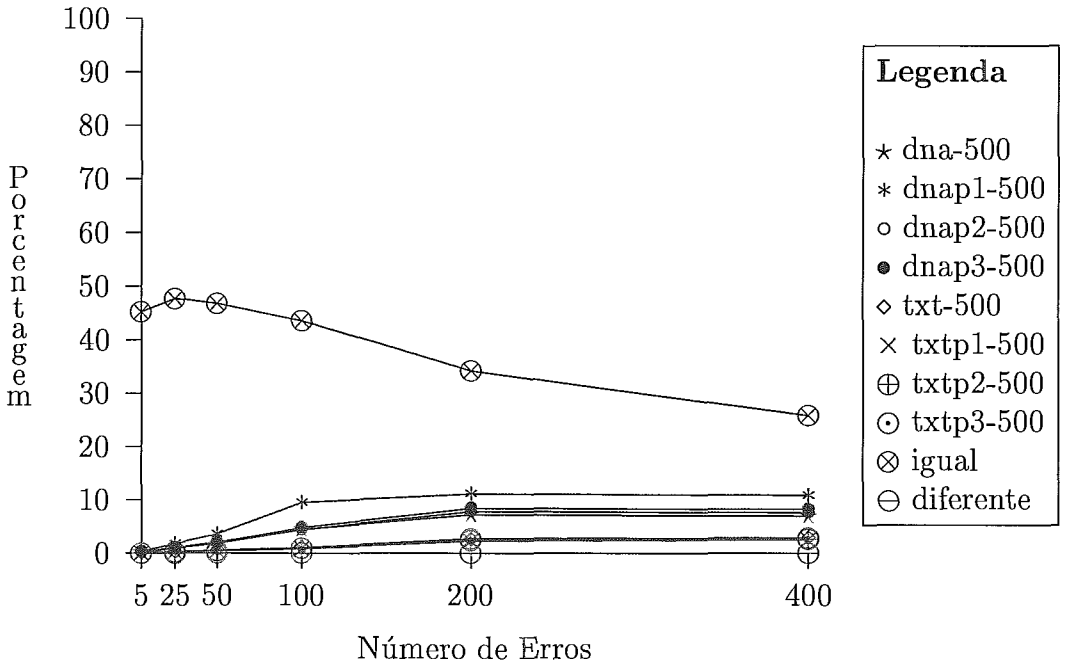
Ganho - Palíndromos Pares



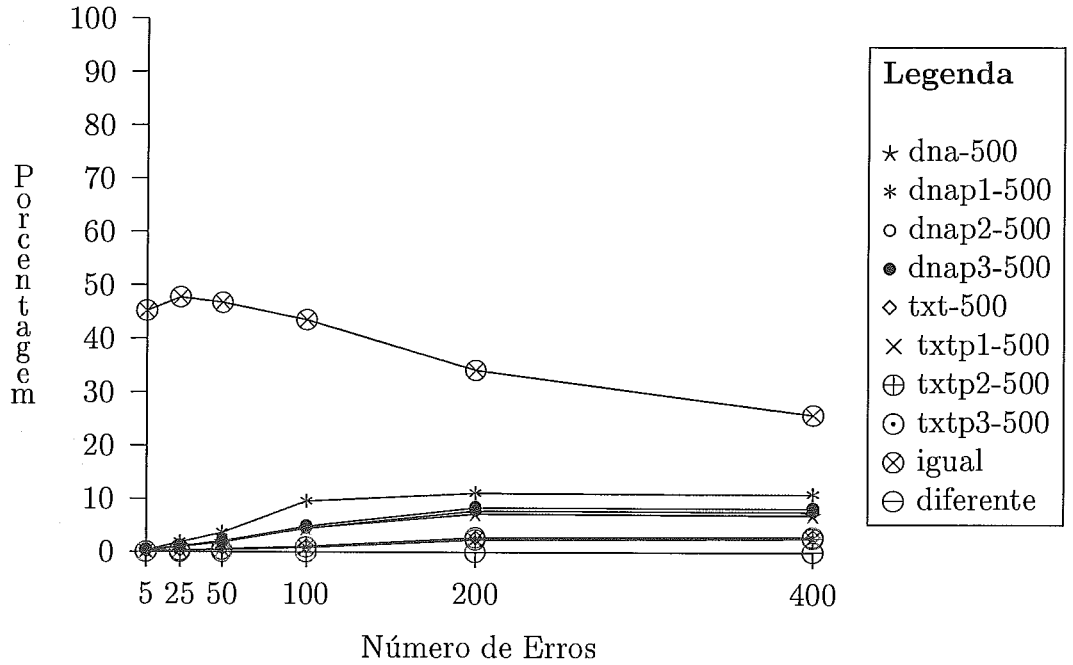
Ganho - Palíndromos Ímpares



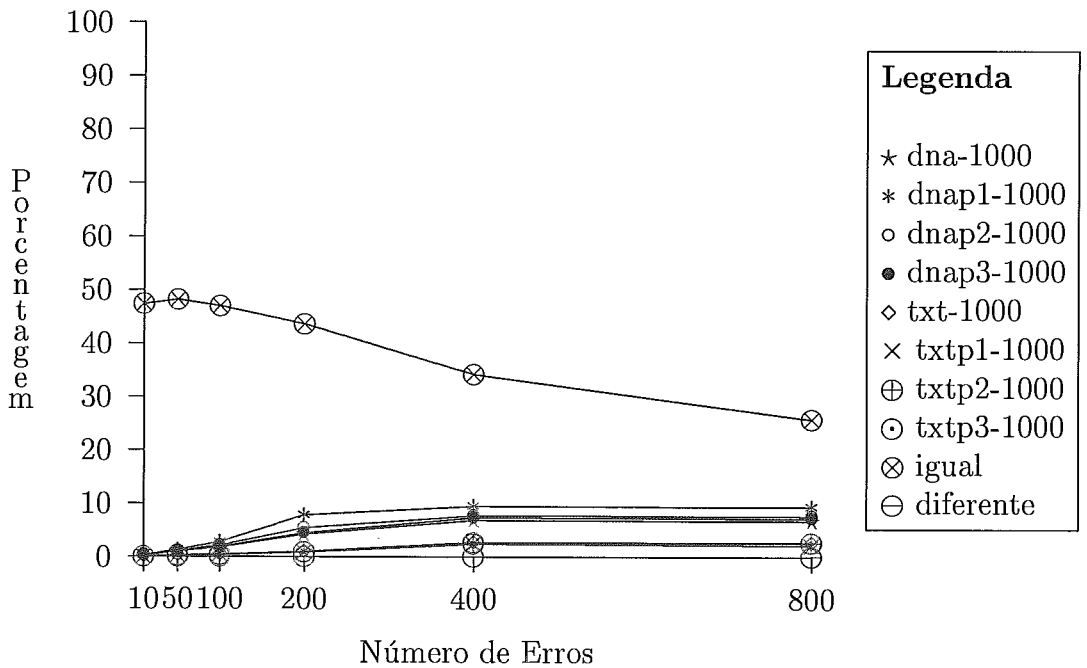
Ganho - Palíndromos Pares



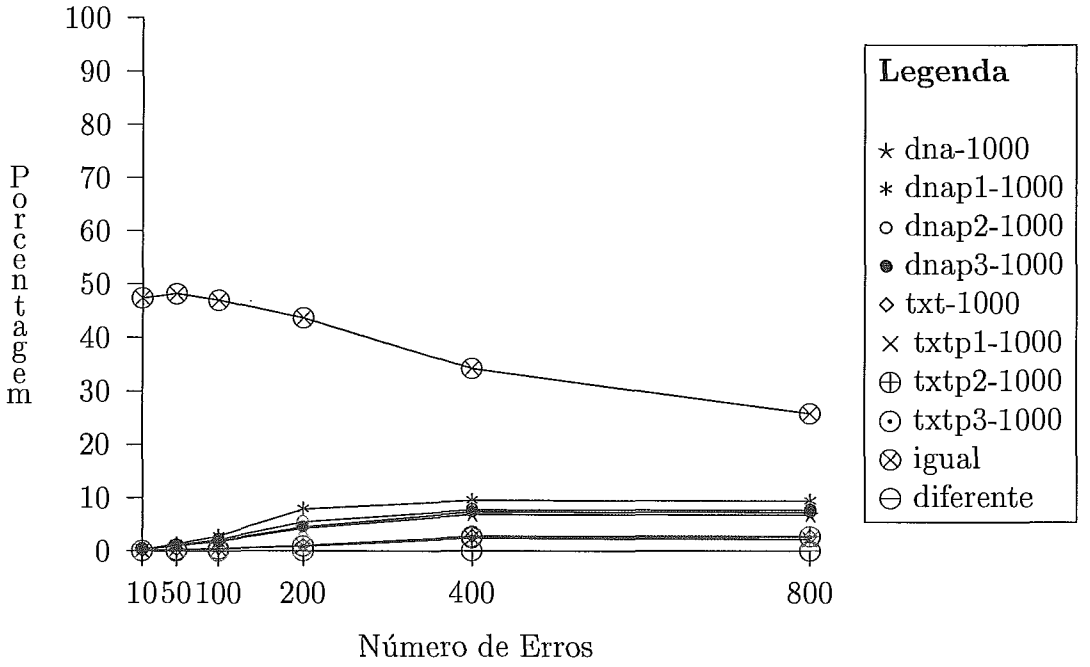
Ganho - Palíndromos Ímpares



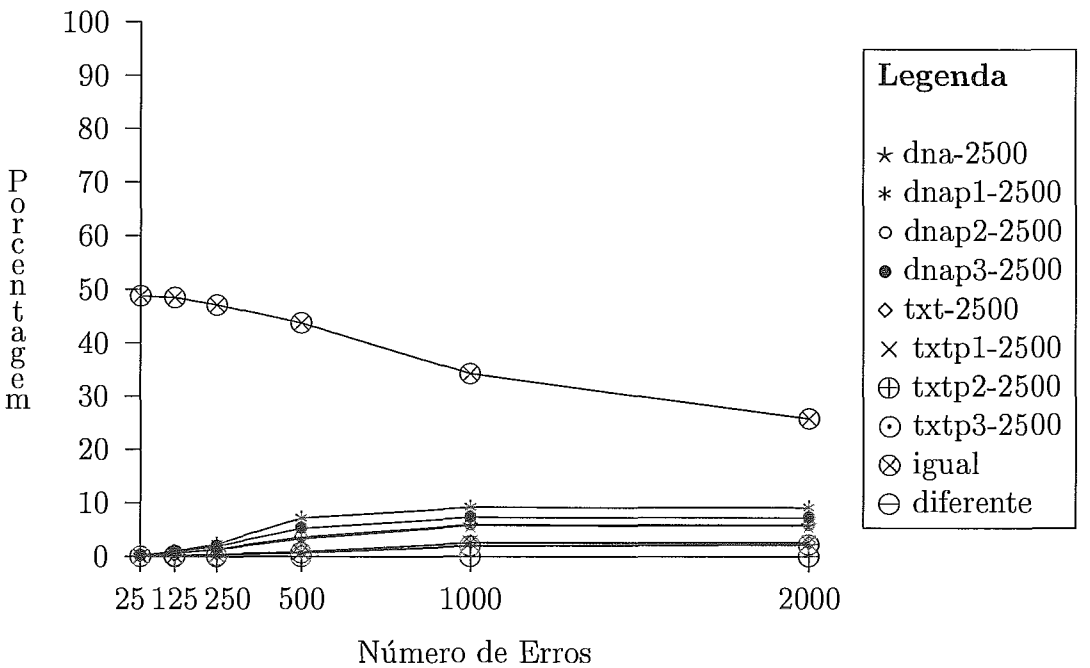
Ganho - Palíndromos Pares



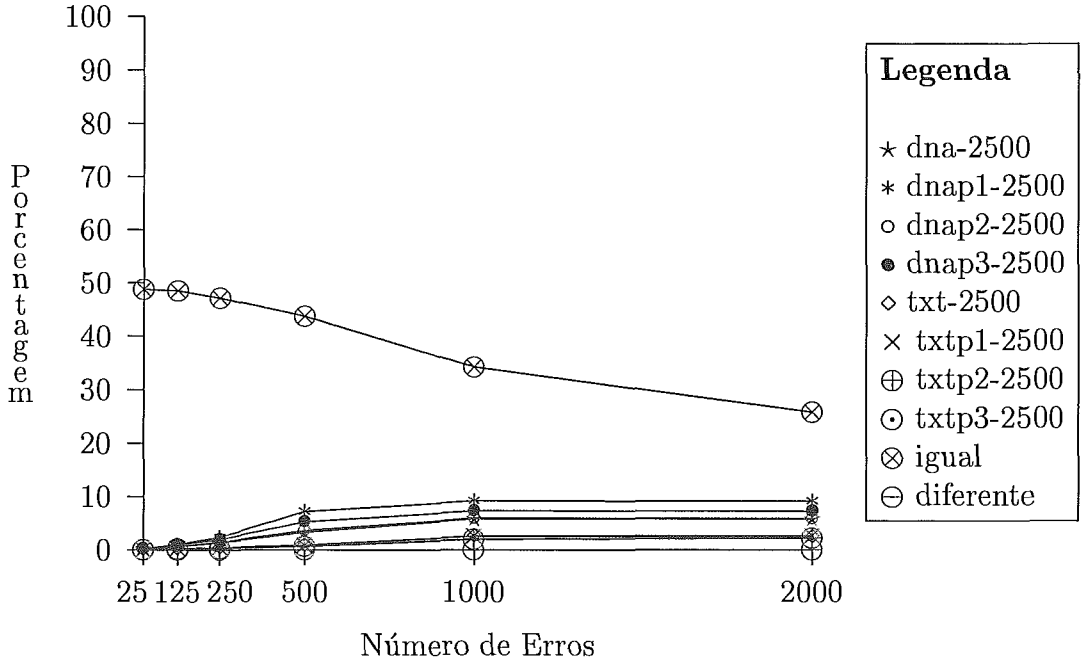
Ganho - Palíndromos Ímpares



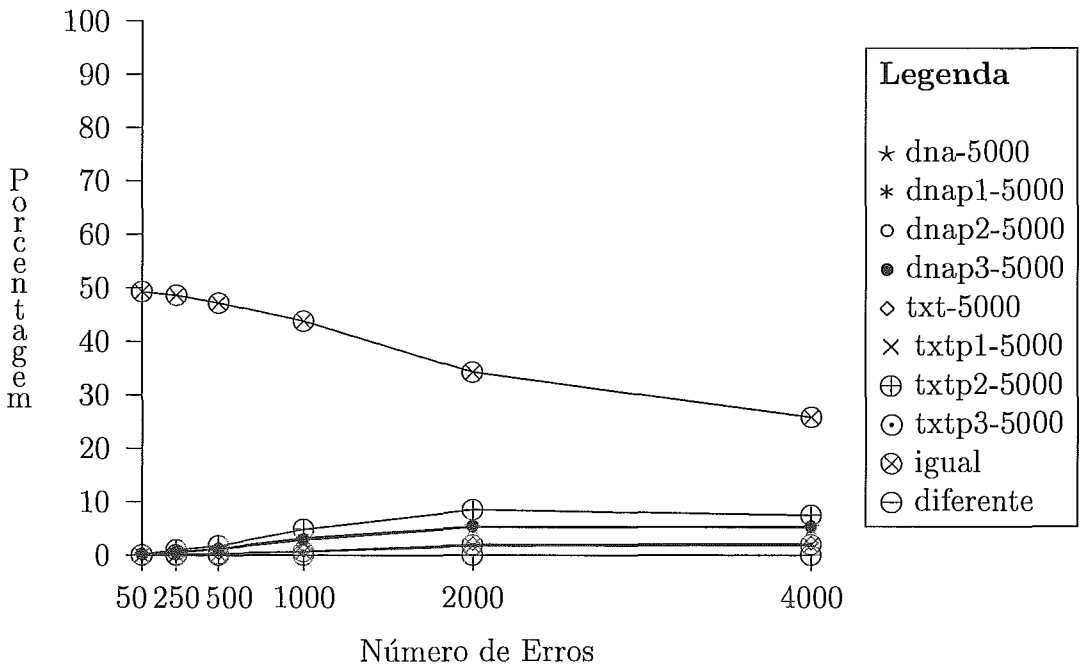
Ganho - Palíndromos Pares



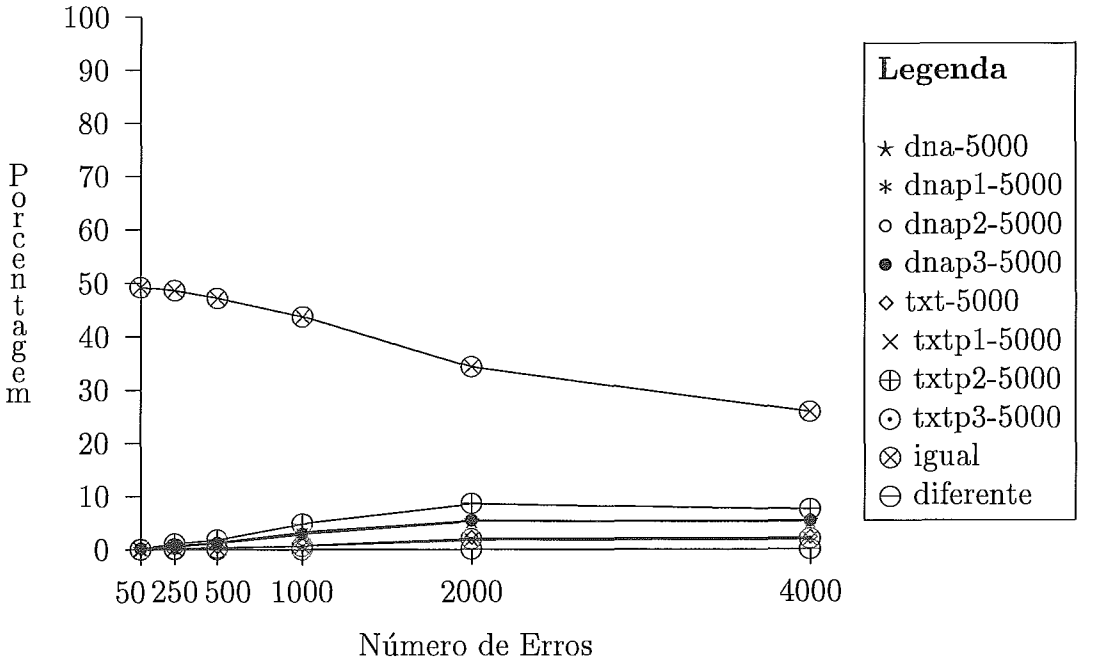
Ganho - Palíndromos Ímpares



Ganho - Palíndromos Pares



Ganho - Palíndromos Ímpares



Referências Bibliográficas

[1] S. Altschul,"Amino Acid Substitution Matrices from an Information Theoretic Perspective", *J. Mol. Biol.*, v. 219, pp. 555-565, 1991.

[2] Alberto Apostolico, Mikhail J. Atallah, Lawrence L. Larmore et al, "Efficient Parallel Algorithms for String Editing and Related Problems", *SIAM J. Comput.*, v. 19, n. 5, pp. 968-988, October 1990.

[3] Alberto Apostolico, Dany Breslauer and Zvi Galil, "Optimal Parallel Algorithms for Periods, Palindromes and Squares", In: *Proc. 19th Internat. Colloq. on Automata, Languages, and Programming*, Lecture Notes in Computer Science, v. 623, pp. 296-307, Springer, Berlin, 1992.

[4] Alberto Apostolico, Dany Breslauer and Zvi Galil, "Parallel Detection of All Palindromes in a String", *Theoretical Computer Science*, v. 141, pp. 163-173, 1995.

[5] A. Apostolico, C. Iliopoulos, G. M. Landau et al., "Parallel Construction of a Suffix Tree with Applications", *Algorithmica*, v. 3, pp. 347-365, 1988.

[6] V. L. Arlazarov, E. A. Dinic, M. A. Kronrod et al., "On Economic Construction of the Transitive Closure of a Directed Graph", *Dokl.Acad. Nauk SSSR*, v. 194, pp. 487-488, 1970.

[7] Ricardo Baeza-Yates and Gonzalo Navarro, *Faster Approximate String Matching*, Department of Computer Science, University of Chile, 1998.

[8] A. A. Bertossi, F. Luccio, L. Pagli et al., "A Parallel Solution to the Approximate String Matching Problem", *The Computer Journal*, v. 35, n. 5, pp. 524-526, 1992.

[9] J. A. Bondy e U. S. R. Murty, *Graph Theory with Applications*, New York, NY, North-Holland, 1976.

[10] Dany Breslauer and Zvi Galil, "Finding All Periods and Initial Palindromes of a String in Parallel", *Algorithmica*, v. 14, pp. 355-366, 1995.

[11] W. I. Chang and J. Lampe, "Theoretical and Empirical Comparions of Approximate String Matching Algorithms", In: *Proc. 3rd Symp. on Combinatorial Pattern Matching*, Springer LNCS, v. 644, pp. 175-184, 1992.

[12] Richard Cole and Ramesh Hariharan, "Approximate String Matching: A Simpler Faster Algorithm", In: *Proceedings of the Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 463-472, San Francisco, California, 25-27 January 1998.

[13] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, Cambridge, MA, MIT Press and McGraw Hill, 1992.

[14] M. O. Dayhoff, R. M. Schwartz, and B. C. Orcutt, "A Model of Evolutionary Change in Proteins", *Atlas of Protein Sequence and Structure*, v. 5, pp. 345-352, 1978.

[15] J. W. Fickett, "Fast Optimal Alignment", *Nucl. Acids Res.*, v. 12, pp. 175-180, 1984.

[16] Z. Galil, Optimal "Parallel Algorithms for String Matching", *Inform. and Control*, v. 67, pp. 144-157, 1985.

[17] Dan Gusfield, *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*, 1 ed., Cambridge University Press, 1997.

[18] D. Harel and R. E. Tarjan, "Fast Algorithms for Finding Nearest Commom Ancestors", *SIAM J. Comput.*, v. 13, pp. 338-355, 1984.

[19] S. Henikoff and J. G. Henikoff, "Amino Acid Substitution Matrices from Protein Blocks", *Proc. Natl. Academy Science*, v. 89, n. 10, pp. 915-919, 1992.

[20] T. H. Jukes and C. R. Cantor, "Evolution of Protein Molecules", In: H. N. Munro (ed), *Mammalian Protein Metabolism*, New York, Academic Press, pp. 21-132, 1969.

[21] J. Jurka. "Origin and Evolution of Alu Repetitive Elements", In: R. J. Maraia (ed), *The Impact of Short Interspersed Elements (SINEs) on the Host Genome*, R. G. Landes, New York, pp. 25-41, 1995.

[22] S. Karlin and S. F. Altschul, "Methods for Assessing the Statistical Significance of Molecular Sequence Features by Using General Scoring Schemes", *Proc. Natl. Academy Science*, v. 87, pp. 2264-2268, 1990.

[23] D. E. Knuth, J. H. Morris and V.R. Pratt, "Fast Pattern Matching in Strings", *SIAM J. Comput.*, v. 6, pp. 322-350, 1977.

[24] G. M. Landau, B. Schieber and U. Vishkin, "Parallel Construction of a Suffix Tree", In: *Proceedings 14th ICALP*, Lecture Notes in Computer Science, v. 267, Springer-Verlag, New York/Berlin, pp. 314-325, 1987.

[25] G. M. Landau and U. Vishkin, "Efficient String Matching with k Mismatches", *Theor. Comp. Sci.*, v. 43, pp. 239-249, 1986.

[26] G. M. Landau and U. Vishkin, "Fast Parallel and Serial Approximate String Matching", *Journal of Algorithms*, pp. 157-169, 1989.

[27] G. M. Landau and U. Vishkin, "Introducing Efficient Parallelism into Approximate String Matching and a New Serial Algorithm", In: *Proc. of the ACM Symp. on the Theory of Computing*, pp. 220-230, 1986.

[28] G. M. Landau, U. Vishkin, and R. Nussinov, "Locating Alignments with k Differences for Nucleotide and Amino Acid Sequences", *Comp. Appl. Biosciences*, v. 4, pp. 12-24, 1988.

[29] *Grande Enciclopédia Larousse Cultural*, v. 18, Nova Cultural Ltda, 1998.

[30] V. I. Levenstein, "Binary Codes Capable of Correcting Insertions and Reversals", *Sov. Phys. Dokl.*, v. 10, pp. 707-710, 1966.

[31] Glenn Manacher, "A New Linear-Time On-Line Algorithm for Finding the Smallest Initial Palindrome of a String", *Journal of the ACM*, v. 22, n. 3, pp. 346-351, July 1975.

[32] E. M. McCreight, "A Space-Economical Suffix Tree Construction Algorithm", *J. ACM*, v. 23, pp. 262-272, 1976.

[33] E. W. Myers, "An $O(nd)$ Difference Algorithm and its Variations", *Algorithmica*, v. 1, pp. 251-266, 1986.

[34] D. Sankoff and J. Kruskal (eds.), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparion*, Reading, MA, Addison-Wesley, 1983.

[35] B. Schieber and U. Vishkin, "On Finding Lowest Commom Ancestors: Simplifications and Parallelization", *SIAM J. Comput.*, v. 17, n. 6, pp. 1253-1262, 1988.

[36] Peter H. Sellers, "The Theory and Computation of Evolutionary Distances: Pattern Recognition", *Journal of Algorithms*, v. 1, pp. 359-373, 1980.

[37] G. A. Stephen, *String Searching Algorithms*, Singapore, World Scientific, 1994.

[38] Jayme Luiz Szwarcfiter e Lilian Markenzon, *Estruturas de Dados e seus Algoritmos*, Editora LTC, 1994.

[39] E. Ukkonen, "Algorithms for Approximate String Matching", *Information Control*, v. 64, pp. 100-118, 1985.

[40] Esko Ukkonen, "Finding Approximate Patterns in Strings", *Journal of Algorithms*, v. 6, pp. 132-137, 1985.

[41] E. Ukkonen, "On-Line Construction of Suffix-Trees", *Algorithmica*, v. 14, pp. 249-260, 1995.

[42] P. Weiner, "Linear Pattern Matching Algorithms", In: *Proc. of the 14th IEEE Symp. on Switching and Automata Theory*, pp. 1-11, 1973.

[43] N. Wirth, *Algorithms and Data Structures*, Englewood Cliffs, N. J., Prentice Hall, 1986.

[44] Sun Wu and Udi Manber, "Fast Text Searching Allowing Erros", *Communications of the ACM*, v. 35, n. 10, pp. 83-91, October 1992.